

17th Symposium on Experimental Algorithms

SEA 2018, June 27–29, 2018, L'Aquila, Italy

Edited by

Gianlorenzo D'Angelo



Editor

Gianlorenzo D'Angelo
Gran Sasso Science Institute (GSSI)
L'Aquila, Italy
gianlorenzo.dangelo@gssi.it

ACM Classification 2012

Theory of computation → Design and analysis of algorithms

ISBN 978-3-95977-070-5

Published online and open access by

Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at <http://www.dagstuhl.de/dagpub/978-3-95977-070-5>.

Publication date

June, 2018

Bibliographic information published by the Deutsche Nationalbibliothek

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

License

This work is licensed under a Creative Commons Attribution 3.0 Unported license (CC-BY 3.0): <http://creativecommons.org/licenses/by/3.0/legalcode>.



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

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

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/LIPIcs.SEA.2018

ISBN 978-3-95977-070-5

ISSN 1868-8969

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

LIPICs – Leibniz International Proceedings in Informatics

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

Editorial Board

- Luca Aceto (*Chair*, Gran Sasso Science Institute and Reykjavik University)
- Susanne Albers (TU München)
- Chris Hankin (Imperial College London)
- Deepak Kapur (University of New Mexico)
- Michael Mitzenmacher (Harvard University)
- Madhavan Mukund (Chennai Mathematical Institute)
- Anca Muscholl (University Bordeaux)
- Catuscia Palamidessi (INRIA)
- Raimund Seidel (Saarland University and Schloss Dagstuhl – Leibniz-Zentrum für Informatik)
- Thomas Schwentick (TU Dortmund)
- Reinhard Wilhelm (Saarland University)

ISSN 1868-8969

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

■ Contents

Preface	
<i>Gianlorenzo D'Angelo</i>	0:ix
Regular Papers	
Network Flow-Based Refinement for Multilevel Hypergraph Partitioning	
<i>Tobias Heuer, Peter Sanders, and Sebastian Schlag</i>	1:1–1:20
Aggregative Coarsening for Multilevel Hypergraph Partitioning	
<i>Ruslan Shaydulin and Ilya Safro</i>	2:1–2:15
Memetic Graph Clustering	
<i>Sonja Biedermann, Monika Henzinger, Christian Schulz, and Bernhard Schuster</i> .	3:1–3:15
ILP-based Local Search for Graph Partitioning	
<i>Alexandra Henzinger, Alexander Noe, and Christian Schulz</i>	4:1–4:15
Decision Diagrams for Solving a Job Scheduling Problem Under Precedence Constraints	
<i>Kosuke Matsumoto, Kohei Hatano, and Eiji Takimoto</i>	5:1–5:12
Speeding up Dualization in the Fredman-Khachiyan Algorithm B	
<i>Nafiseh Sedaghat, Tamon Stephen, and Leonid Chindelevitch</i>	6:1–6:13
An Ambiguous Coding Scheme for Selective Encryption of High Entropy Volumes	
<i>M. Oğuzhan Külekci</i>	7:1–7:13
A $\frac{3}{2}$ -Approximation Algorithm for the Student-Project Allocation Problem	
<i>Frances Cooper and David Manlove</i>	8:1–8:13
How Good Are Popular Matchings?	
<i>Krishnapriya A M, Meghana Nasre, Prajakta Nimbhorkar, and Amit Rawat</i>	9:1–9:14
Evaluating and Tuning n -fold Integer Programming	
<i>Kateřina Altmanova, Duřan Knop, and Martin Koutecky</i>	10:1–10:14
A Computational Investigation on the Strength of Dantzig-Wolfe Reformulations	
<i>Michael Bastubbe, Marco E. Lubbecke, and Jonas T. Witt</i>	11:1–11:12
Experimental Evaluation of Parameterized Algorithms for Feedback Vertex Set	
<i>Krzysztof Kiljan and Marcin Pilipczuk</i>	12:1–12:12
An Efficient Local Search for the Minimum Independent Dominating Set Problem	
<i>Kazuya Haraguchi</i>	13:1–13:13
Empirical Evaluation of Approximation Algorithms for Generalized Graph Coloring and Uniform Quasi-Wideness	
<i>Wojciech Nadara, Marcin Pilipczuk, Roman Rabinovich, Felix Reidl, and Sebastian Siebertz</i>	14:1–14:16



Multi-Level Steiner Trees <i>Reyan Ahmed, Patrizio Angelini, Faryad Darabi Sahneh, Alon Efrat, David Glickenstein, Martin Gronemann, Niklas Heinsohn, Stephen G. Kobourov, Richard Spence, Joseph Watkins, and Alexander Wolff</i>	15:1–15:14
Dictionary Matching in Elastic-Degenerate Texts with Applications in Searching VCF Files On-line <i>Solon P. Pissis and Ahmad Retha</i>	16:1–16:14
Fast matching statistics in small space <i>Djamal Belazzougui, Fabio Cunial, and Olmert Denas</i>	17:1–17:14
Practical lower and upper bounds for the Shortest Linear Superstring <i>Bastien Cazaux, Samuel Juhel, and Eric Rivals</i>	18:1–18:14
Experimental Study of Compressed Stack Algorithms in Limited Memory Environments <i>Jean-François Baffier, Yago Diez, and Matias Korman</i>	19:1–19:13
Restructuring Expression Dags for Efficient Parallelization <i>Martin Wilhelm</i>	20:1–20:13
Enumerating Graph Partitions Without Too Small Connected Components Using Zero-suppressed Binary and Ternary Decision Diagrams <i>Yu Nakahata, Jun Kawahara, and Shoji Kasahara</i>	21:1–21:13
Exact Algorithms for the Maximum Planar Subgraph Problem: New Models and Experiments <i>Markus Chimani, Ivo Hedtke, and Tilo Wiedera</i>	22:1–22:14
A Linear-Time Algorithm for Finding Induced Planar Subgraphs <i>Shixun Huang, Zhifeng Bao, J. Shane Culpepper, Ping Zhang, and Bang Zhang</i> ..	23:1–23:15
Fast Spherical Drawing of Triangulations: An Experimental Study of Graph Drawing Tools <i>Luca Castelli Aleardi, Gaspard Denis, and Éric Fusy</i>	24:1–24:14
Fleet Management for Autonomous Vehicles Using Multicommodity Coupled Flows in Time-Expanded Networks <i>Sahar Bsaybes, Alain Quilliot, and Annegret K. Wagler</i>	25:1–25:14
The Steiner Multi Cycle Problem with Applications to a Collaborative Truckload Problem <i>Vinicius N. G. Pereira, Mário César San Felice, Pedro Henrique D. B. Hokama, and Eduardo C. Xavier</i>	26:1–26:13
Real-Time Traffic Assignment Using Fast Queries in Customizable Contraction Hierarchies <i>Valentin Buchhold, Peter Sanders, and Dorothea Wagner</i>	27:1–27:15
Engineering Motif Search for Large Motifs <i>Petteri Kaski, Juho Lauri, and Suhas Thejaswi</i>	28:1–28:19
Finding Hamiltonian Cycle in Graphs of Bounded Treewidth: Experimental Evaluation <i>Michał Ziobro and Marcin Pilipczuk</i>	29:1–29:14

Isomorphism Test for Digraphs with Weighted Edges
Adolfo Piperno 30:1–30:13

■ Preface

This volume contains papers presented at the 17th International Symposium on Experimental Algorithms (SEA 2018), held June 27–29, 2018, in L’Aquila, Italy.

Since 2002, the series of SEA symposia (previously known as Workshop on Experimental Algorithms, WEA) bring together specialists and young researchers working in experimental algorithms and algorithm engineering, encouraging high-quality research in the area. Previous WEA and SEA meetings have been held in Latvia, Switzerland, Brazil, Greece, Spain, Italy, USA, Germany, Denmark, France, Russia, and UK.

We solicited papers in the broad area of design, analysis, and experimental evaluation and engineering of algorithms, as well as of combinatorial optimization and its applications. In response to the call for papers, we received 70 submissions, with the Program Committee deciding to accept 30 papers. Each submission was reviewed by at least three program committee members with the help of several external reviewers. Papers were submitted and reviewed using the EasyChair online system. Authors of accepted papers come from 20 countries, across five continents.

In addition to the accepted contributions, the program also included three invited lectures by Dorothea Wagner (KIT), Giuseppe Italiano (University of Rome Tor Vergata), and Simon J. Puglisi (University of Helsinki).

We would like to thank all the authors who responded to the call for papers, the invited speakers, the members of the PC, the external reviewers, and the members of the Organizing Committee. We also thank the SEA steering committee for giving us the opportunity to host SEA 2018.

L’Aquila
June 2018

Gianlorenzo D’Angelo



Program committee

Ittai Abraham	VMware Research (Israel)
Martin Aumüller	IT University of Copenhagen (Denmark)
Vincenzo Bonifaci	IASI-CNR (Italy)
David Coudert	Université Côte d'Azur, Inria, CNRS, I3S (France)
Maxime Crochemore	Université Paris-Est (France), and King's College London (UK)
Gianlorenzo D'Angelo (chair)	Gran Sasso Science Institute (Italy)
Mattia D'Emidio	University of L'Aquila (Italy) & Gran Sasso Science Institute (Italy)
Simone Faro	University of Catania (Italy)
Paola Festa	University of Naples Federico II (Italy)
Daniele Frigioni	University of L'Aquila (Italy)
Loukas Georgiadis	University of Ioannina (Greece)
Ralf Klasing	CNRS - LaBRI - Université de Bordeaux (France)
Arie Koster	RWTH Aachen University (Germany)
Giuseppe Liotta	University of Perugia (Italy)
Andrea Lodi	École Polytechnique de Montréal (Canada)
Andrea Marino	University of Pisa (Italy)
Henning Meyerhenke	University of Cologne (Germany)
Matúš Mihalák	Maastricht University (The Netherlands)
Nicolas Nisse	Université Côte d'Azur, INRIA, CNRS, I3S (France)
Mauricio G. C. Resende	Amazon.com Inc. (USA)
Marie-France Sagot	INRIA (France)
Stefan Schmid	University of Vienna (Austria)
Anita Schöbel	University of Göttingen (Germany)
Sabine Storandt	University Würzburg (Germany)
Suresh Venkatasubramanian	University of Utah (USA)
Renato F. Werneck	Amazon.com Inc. (USA)
Norbert Zeh	Dalhousie University (Canada)

External reviewers

Alessio Arleo	Chia-Wei Lee
Chen Avin	Stefano Leucci
Stephan Beyer	Margaux Luck
Carla Binucci	David Manlove
Nicolas Bousquet	Carlo Mannino
Laurent Bulteau	Shuichi Miyazaki
Margarida Carvalho	Shima Moghtasedi
Li-Hsuan Chen	Fabrizio Montecchiani
Serafino Cicerone	Wojciech Mula
Andre Augusto Cire	Wolfgang Mulzer
Alessio Conte	Fionn Murtagh
Graham Cormode	Alfredo Navarra
Ágnes Cseh	Viet Hung Nguyen
Palash Dey	Shmuel Onn
Emilio Di Giacomo	Katarzyna Paluch
Gabriele Di Stefano	Charis Papadopoulos
Riccardo Dondi	Maurizio Patrignani
Ingo van Duijn	Julius Pätzold
Khaled Elbassioni	Giulio Ermanno Pibiri
Carlos Eduardo Ferreira	Maria Predari
Guillaume Fertin	Mathieu Raffinot
Daniel Fleischman	Michael Rice
Luca Forlizzi	Giovanni Rinaldi
Klaus-Tycho Förster	Stéphane Robin
Travis Gagie	Philine Schiewe
Konstantinos Giannis	Lorenzo Severini
Alexander Golovnev	Georgios Stamoulis
Szymon Grabowski	Mathieu Tanneau
Luca Grilli	Charilaos Tzovas
Alexander van der Grinten	Yllka Velaj
Jan Holub	Luca Versari
Chien-Chung Huang	Stéphane Vialette
Ling-Ju Hung	Fábio Henrique Viduani Martinez
Aikaterini Karanasiou	Armin Weiß
Christian Laforest	Sebastian Wild
Luigi Laura	Yu Yokoi
Thierry Lecroq	

Local organizing committee

Gianlorenzo D'Angelo (chair)
 Mattia D'Emidio
 Michele Flammioni
 Ludovico Iovino

Network Flow-Based Refinement for Multilevel Hypergraph Partitioning

Tobias Heuer

Karlsruhe Institute of Technology, Germany
tobias.heuer@gmx.net

Peter Sanders

Karlsruhe Institute of Technology, Germany
sanders@kit.edu

Sebastian Schlag

Karlsruhe Institute of Technology, Germany
sebastian.schlag@kit.edu

Abstract

We present a refinement framework for multilevel hypergraph partitioning that uses max-flow computations on pairs of blocks to improve the solution quality of a k -way partition. The framework generalizes the flow-based improvement algorithm of KaFFPa from graphs to hypergraphs and is integrated into the hypergraph partitioner KaHyPar. By reducing the size of hypergraph flow networks, improving the flow model used in KaFFPa, and developing techniques to improve the running time of our algorithm, we obtain a partitioner that computes the best solutions for a wide range of benchmark hypergraphs from different application areas while still having a running time comparable to that of hMetis.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms

Keywords and phrases Multilevel Hypergraph Partitioning, Network Flows, Refinement

Digital Object Identifier 10.4230/LIPIcs.SEA.2018.1

Related Version A full version of the paper is available at <https://arxiv.org/abs/1802.03587>.

1 Introduction

Given an undirected hypergraph $H = (V, E)$, the k -way hypergraph partitioning problem is to partition the vertex set into k disjoint blocks of bounded size (at most $1 + \varepsilon$ times the average block size) such that an objective function involving the cut hyperedges is minimized. Hypergraph partitioning (HGP) has many important applications in practice such as scientific computing [10] or VLSI design [40]. Particularly VLSI design is a field where small improvements can lead to significant savings [53].

It is well known that HGP is NP-hard [35], which is why practical applications mostly use heuristic *multilevel* algorithms [9, 11, 22, 23]. These algorithms successively *contract* the hypergraph to obtain a hierarchy of smaller, structurally similar hypergraphs. After applying an *initial partitioning* algorithm to the smallest hypergraph, contraction is undone and, at each level, a *local search* method is used to improve the partitioning induced by the coarser level. All state-of-the-art HGP algorithms [2, 4, 6, 14, 25, 29, 30, 31, 45, 48, 49, 51] either use variations of the Kernighan-Lin (KL) [32, 46] or the Fiduccia-Mattheyses (FM) heuristic [17, 43], or simpler greedy algorithms [30, 31] for local search. These heuristics move vertices between blocks in descending order of improvements in the optimization objective



© Tobias Heuer, Peter Sanders, and Sebastian Schlag;
licensed under Creative Commons License CC-BY

17th International Symposium on Experimental Algorithms (SEA 2018).

Editor: Gianlorenzo D'Angelo; Article No. 1; pp. 1:1–1:19

Leibniz International Proceedings in Informatics



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

(gain) and are known to be prone to get stuck in local optima when used directly on the input hypergraph [31]. The multilevel paradigm helps to some extent, since it allows a more global view on the problem on the coarse levels and a very fine-grained view on the fine levels of the hierarchy. However, the performance of move-based approaches degrades for hypergraphs with large hyperedges. In these cases, it is difficult to find meaningful vertex moves that improve the solution quality because large hyperedges are likely to have many vertices in multiple blocks [50]. Thus the gain of moving a single vertex to another block is likely to be *zero* [38].

While finding *balanced* minimum cuts in hypergraphs is NP-hard, a minimum cut separating two vertices can be found in polynomial time using network flow algorithms and the max-flow min-cut theorem [19]. Flow algorithms find an optimal min-cut and do not suffer the drawbacks of move-based approaches. However, they were long overlooked as heuristics for balanced partitioning due to their high complexity [37, 54]. Sanders and Schulz [44] presented a max-flow-based improvement algorithm for graph partitioning which is integrated into the multilevel partitioner KaFFPa and computes high quality solutions.

Outline and Contribution. Motivated by the results of Sanders and Schulz [44], we generalize the max-flow min-cut refinement framework of KaFFPa from graphs to hypergraphs. After introducing basic notation and giving a brief overview of related work and the techniques used in KaFFPa in Section 2, we explain how hypergraphs are transformed into flow networks and present a technique to reduce the size of the resulting hypergraph flow network in Section 3.1. In Section 3.2 we then show how this network can be used to construct a flow problem such that the min-cut induced by a max-flow computation between a pair of blocks improves the solution quality of a k -way partition. We furthermore identify shortcomings of the KaFFPa approach that restrict the search space of feasible solutions significantly and introduce an advanced model that overcomes these limitations by exploiting the structure of hypergraph flow networks. We implemented our algorithm in the open source HGP framework KaHyPar and therefore briefly discuss implementation details and techniques to improve the running time in Section 3.3. Extensive experiments presented in Section 4 demonstrate that our flow model yields better solutions than the KaFFPa approach for both hypergraphs *and* graphs. We furthermore show that using pairwise flow-based refinement significantly improves partitioning quality. The resulting hypergraph partitioner, KaHyPar-MF, performs better than *all* competing algorithms on *all* instance classes and still has a running time comparable to that of hMetis. On a large benchmark set consisting of 3222 instances from various application domains, KaHyPar-MF computes the best partitions in 2427 cases.

2 Preliminaries

2.1 Notation and Definitions

An *undirected hypergraph* $H = (V, E, c, \omega)$ is defined as a set of n vertices V and a set of m hyperedges/nets E with vertex weights $c : V \rightarrow \mathbb{R}_{>0}$ and net weights $\omega : E \rightarrow \mathbb{R}_{>0}$, where each net e is a subset of the vertex set V (i.e., $e \subseteq V$). The vertices of a net are called *pins*. We extend c and ω to sets, i.e., $c(U) := \sum_{v \in U} c(v)$ and $\omega(F) := \sum_{e \in F} \omega(e)$. A vertex v is *incident* to a net e if $v \in e$. $I(v)$ denotes the set of all incident nets of v . The *degree* of a vertex v is $d(v) := |I(v)|$. The *size* $|e|$ of a net e is the number of its pins. Given a subset $V' \subset V$, the *subhypergraph* $H_{V'}$ is defined as $H_{V'} := (V', \{e \cap V' \mid e \in E : e \cap V' \neq \emptyset\})$.

A k -way *partition* of a hypergraph H is a partition of its vertex set into k *blocks* $\Pi = \{V_1, \dots, V_k\}$ such that $\bigcup_{i=1}^k V_i = V$, $V_i \neq \emptyset$ for $1 \leq i \leq k$, and $V_i \cap V_j = \emptyset$ for $i \neq j$.

We call a k -way partition Π ε -balanced if each block $V_i \in \Pi$ satisfies the *balance constraint*: $c(V_i) \leq L_{\max} := (1 + \varepsilon) \lceil \frac{c(V)}{k} \rceil$ for some parameter ε . For each net e , $\Lambda(e) := \{V_i \mid V_i \cap e \neq \emptyset\}$ denotes the *connectivity set* of e . The *connectivity* of a net e is $\lambda(e) := |\Lambda(e)|$. A net is called *cut net* if $\lambda(e) > 1$. Given a k -way partition Π of H , the *quotient graph* $Q := (\Pi, \{(V_i, V_j) \mid \exists e \in E : \{V_i, V_j\} \subseteq \Lambda(e)\})$ contains an edge between each pair of adjacent blocks. The *k -way hypergraph partitioning problem* is to find an ε -balanced k -way partition Π of a hypergraph H that minimizes an objective function over the cut nets for some ε . The most commonly used cost functions are the *cut-net* metric $\text{cut}(\Pi) := \sum_{e \in E'} \omega(e)$ and the *connectivity* metric $(\lambda - 1)(\Pi) := \sum_{e \in E'} (\lambda(e) - 1) \omega(e)$ [1], where E' is the set of all cut nets [15]. In this paper, we use the $(\lambda - 1)$ -metric. Optimizing both objective functions is known to be NP-hard [35]. In the following, we use *nodes* and *edges* when referring to graphs and *vertices* and *nets* when referring to hypergraphs. Hypergraphs can be represented as *bipartite* graphs [27]: In the *star-expansion* $G_*(V \cup E, F)$ the vertices and nets of H form the node set and for each net $e \in I(v)$, we add an edge (e, v) to G_* . The edge set F is thus defined as $F := \{(e, v) \mid e \in E, v \in e\}$. Each net in E therefore corresponds to a *star* in G_* .

Let $G = (V, E, c, \omega)$ be a weighted directed graph. We use the same notation to refer to node weights c , edge weights ω , and node degrees $d(v)$. Furthermore $\Gamma(u) := \{v \mid (u, v) \in E\}$ denotes the neighbors of node u . A *path* $P = \langle v_1, \dots, v_k \rangle$ is a sequence of nodes, such that each pair of consecutive nodes is connected by an edge. A *strongly connected component* $C \subseteq V$ is a set of nodes such that for each $u, v \in C$ there exists a path from u to v . A *topological ordering* is a linear ordering \prec of V such that every directed edge $(u, v) \in E$ implies $u \prec v$ in the ordering. A set of nodes $B \subseteq V$ is called a *closed set* iff there are no outgoing edges leaving B , i.e., if the conditions $u \in B$ and $(u, v) \in E$ imply $v \in B$. A subset $S \subset V$ is called a *node separator* if its removal divides G into two disconnected components. Given a subset $V' \subset V$, the induced subgraph $G[V']$ is defined as $G[V'] := (V', \{(u, v) \in E \mid u, v \in V'\})$.

A flow network $\mathcal{N} = (\mathcal{V}, \mathcal{E}, c)$ is a directed graph with two distinguished nodes s and t in which each edge $e \in \mathcal{E}$ has a capacity $c(e) \geq 0$. An (s, t) -flow (or flow) is a function $f : \mathcal{V} \times \mathcal{V} \rightarrow \mathbb{R}$ that satisfies the *capacity constraint* $\forall u, v \in \mathcal{V} : f(u, v) \leq c(u, v)$, the *skew symmetry constraint* $\forall u, v \in \mathcal{V} \times \mathcal{V} : f(u, v) = -f(v, u)$, and the *flow conservation constraint* $\forall u \in \mathcal{V} \setminus \{s, t\} : \sum_{v \in \mathcal{V}} f(u, v) = 0$. The value of a flow $|f| := \sum_{v \in \mathcal{V}} f(s, v)$ is defined as the total amount of flow transferred from s to t . The *residual capacity* is defined as $r_f(u, v) = c(u, v) - f(u, v)$. Given a flow f , $\mathcal{N}_f = (\mathcal{V}, \mathcal{E}_f, r_f)$ with $\mathcal{E}_f = \{(u, v) \in \mathcal{V} \times \mathcal{V} \mid r_f(u, v) > 0\}$ is the *residual network*. An (s, t) -cut (or cut) is a bipartition $(\mathcal{S}, \mathcal{V} \setminus \mathcal{S})$ of a flow network \mathcal{N} with $s \in \mathcal{S} \subset \mathcal{V}$ and $t \in \mathcal{V} \setminus \mathcal{S}$. The capacity of an (s, t) -cut is defined as $\sum_{e \in \mathcal{E}'} c(e)$, where $\mathcal{E}' = \{(u, v) \in \mathcal{E} : u \in \mathcal{S}, v \in \mathcal{V} \setminus \mathcal{S}\}$. The max-flow min-cut theorem states that the value $|f|$ of a maximum flow f is equal to the capacity of a minimum cut separating s and t [19].

2.2 Related Work

Hypergraph Partitioning. HGP has evolved into a broad research area since the 1990s. We refer to existing literature [5, 7, 40, 47] for an extensive overview. Well-known multilevel HGP software packages with certain distinguishing characteristics include PaToH [10] (originating from scientific computing), hMetis [30, 31] (originating from VLSI design), KaHyPar [2, 25, 45] (general purpose, n -level), Mondriaan [51] (sparse matrix partitioning), MLPart [4] (circuit partitioning), Zoltan [14], Parkway [48] and SHP [29] (distributed), UMPa [49] (directed hypergraph model, multi-objective), and kPaToH (multiple constraints, fixed vertices) [6]. All of these tools use move-based local search algorithms in the refinement phase.

Flows on Hypergraphs. While flow-based approaches have not yet been considered as refinement algorithms for multilevel HGP, several works deal with flow-based hypergraph min-cut computation. The problem of finding minimum (s, t) -cuts in hypergraphs was first considered by Lawler [33], who showed that it can be reduced to computing maximum flows in directed graphs. Hu and Moerder [27] present an augmenting path algorithm to compute a minimum-weight node separator on the star-expansion of the hypergraph. Their node-capacitated network can also be transformed into an edge-capacitated network using a transformation due to Lawler [34]. Yang and Wong [54] use repeated, incremental max-flow min-cut computations on the Lawler network [33] to find ε -balanced hypergraph bipartitions. Solution quality and running time of this algorithm are improved by Lillis and Cheng [36] by introducing advanced heuristics to select source and sink nodes. Furthermore, they present a preflow-based [20] min-cut algorithm that implicitly operates on the star-expanded hypergraph. Pistorius and Minoux [42] generalize the algorithm of Edmonds and Karp [16] to hypergraphs by labeling both vertices and nets. Liu and Wong [37] simplify Lawler’s hypergraph flow network [33] by explicitly distinguishing between graph edges and hyperedges with three or more pins. This approach significantly reduces the size of flow networks derived from VLSI hypergraphs, since most of the nets in a circuit are graph edges. Note that the above-mentioned approaches to model hypergraphs as flow networks for max-flow min-cut computations do not contradict the negative results of Ihler et al. [28], who show that, *in general*, there does not exist an edge-weighted graph $G = (V, E)$ that correctly represents the min-cut properties of the corresponding hypergraph $H = (V, E)$.

KaHyPar. Since our algorithm is integrated into the KaHyPar framework, we briefly review its core components. While traditional multilevel HGP algorithms contract matchings or clusterings and therefore work with a coarsening hierarchy of $\mathcal{O}(\log n)$ levels, KaHyPar instantiates the multilevel paradigm in the extreme n -level version, removing only a *single* vertex between two levels. After coarsening, a portfolio of simple algorithms is used to create an initial partition of the coarsest hypergraph. During uncoarsening, strong localized local search heuristics based on the FM algorithm [17, 43] are used to refine the solution. Our work builds on KaHyPar-CA [25], which is a direct k -way partitioning algorithm for optimizing the $(\lambda - 1)$ -metric. It uses an improved coarsening scheme that incorporates global information about the community structure of the hypergraph into the coarsening process.

2.3 The Flow-Based Improvement Framework of KaFFPa

We discuss the framework of Sanders and Schulz [44] in greater detail, since our work makes use of the techniques proposed by the authors. For simplicity, we assume $k = 2$. The techniques can be applied on a k -way partition by repeatedly executing the algorithm on pairs of adjacent blocks. To schedule these refinements, the authors propose an *active block scheduling* algorithm, which schedules blocks adjacent in the quotient graph Q as long as their participation in a pairwise refinement step improves solution quality or imbalance.

An ε -balanced bipartition of a graph $G = (V, E, c, \omega)$ is improved with flow computations as follows. The basic idea is to construct a flow network \mathcal{N} based on the induced subgraph $G[B]$, where $B \subseteq V$ is a set of nodes around the cut of G . The size of B is controlled by an imbalance factor $\varepsilon' := \alpha\varepsilon$, where α is a scaling parameter that is chosen adaptively depending on the result of the min-cut computation. If the heuristic found an ε -balanced partition using ε' , the cut is accepted and α is increased to $\min(2\alpha, \alpha')$ where α' is a predefined upper bound. Otherwise it is decreased to $\max(\frac{\alpha}{2}, 1)$. This scheme continues until a maximal number of rounds is reached or a feasible partition that did not improve the cut is found.

In each round, the corridor $B := B_1 \cup B_2$ is constructed by performing two breadth-first searches (BFS). The first BFS is done in the induced subgraph $G[V_1]$. It is initialized with the boundary nodes of V_1 and stops if $c(B_1)$ would exceed $(1 + \varepsilon') \lceil \frac{c(V_1)}{2} \rceil - c(V_2)$. The second BFS constructs B_2 in an analogous fashion using $G[V_2]$. Let $\delta B := \{u \in B \mid \exists (u, v) \in E : v \notin B\}$ be the border of B . Then \mathcal{N} is constructed by connecting all border nodes $\delta B \cap V_1$ of $G[B]$ to the source s and all border nodes $\delta B \cap V_2$ to the sink t using directed edges with an edge weight of ∞ . By connecting s and t to the respective border nodes, it is ensured that edges incident to border nodes, but not contained in $G[B]$, cannot become cut edges. For $\alpha = 1$, the size of B thus ensures that the flow network \mathcal{N} has the *cut property*, i.e., each (s, t) -min-cut in \mathcal{N} yields an ε -balanced partition of G with a possibly smaller cut. For larger values of α , this does not have to be the case.

After computing a max-flow in \mathcal{N} , the algorithm tries to find a min-cut with better balance. This is done by exploiting the fact that *one* (s, t) -max-flow contains information about *all* (s, t) -min-cuts [41]. More precisely, the algorithm uses the 1–1 correspondence between (s, t) -min-cuts and closed sets containing s in the Picard-Queyranne-DAG $D_{s,t}$ of the residual graph \mathcal{N}_f [41]. First, $D_{s,t}$ is constructed by contracting each strongly connected component of the residual graph. Then the following heuristic (called most balanced minimum cuts) is repeated several times using different random seeds. Closed sets containing s are computed by sweeping through the nodes of $D_{s,t}$ in reverse topological order (e.g. computed using a randomized DFS). Each closed set induces a differently balanced min-cut and the one with the best balance (with respect to the original balance constraint) is used as bipartition.

3 Hypergraph Max-Flow Min-Cut Refinement

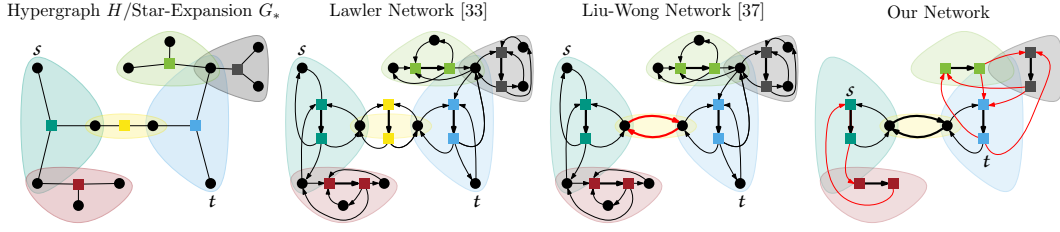
In the following, we generalize the KaFFPa algorithm to hypergraph partitioning. In Section 3.1 we first show how hypergraph flow networks \mathcal{N} are constructed in general and introduce a technique to reduce their size by removing *low-degree* vertices. Given a k -way partition $\Pi_k = \{V_1, \dots, V_k\}$ of a hypergraph H , a pair of blocks (V_i, V_j) adjacent in the quotient graph Q , and a corridor B , Section 3.2 then explains how \mathcal{N} is used to build a flow problem \mathcal{F} based on the subhypergraph $H_B = (V_B, E_B)$. \mathcal{F} is constructed such that an (s, t) -max-flow computation optimizes the *cut* metric of the bipartition $\Pi_2 = (V_i, V_j)$ of H_B and thus improves the $(\lambda - 1)$ -metric in H . Section 3.3 then discusses the integration into KaHyPar and introduces several techniques to speed up flow-based refinement. A pseudocode description of the entire flow-based refinement framework is given in Appendix A.

3.1 Hypergraph Flow Networks

From Hypergraphs to Flow Networks. Given a hypergraph $H = (V, E, c, \omega)$ and two distinct vertices s and t , we first reduce the problem of finding an (s, t) -min-cut in H to the problem of finding a minimum-weight (s, t) -node-separator in the star-expansion G_* , where each star-node e has weight $c(e) = \omega(e)$ and all other nodes v have a weight of infinity [27].

This network is then transformed into the *edge-capacitated* flow network $\mathcal{N} = (\mathcal{V}, \mathcal{E}, c)$ of Lawler [33] as follows: \mathcal{V} contains all non-star nodes v . For each star-node e , add two *bridging* nodes e' and e'' to \mathcal{V} and a *bridging* edge (e', e'') with capacity $c(e', e'') = c(e)$ to \mathcal{E} . For each neighbor $u \in \Gamma(e)$, add two edges (u, e') and (e'', u) with infinite capacity to \mathcal{E} .

The size of this network can be reduced by distinguishing between star-nodes corresponding to multi-pin nets and those corresponding to two-pin nets in H . In the flow network of Liu and Wong [37] the former are transformed as described above, while the latter (i.e., star-nodes e with $|\Gamma(e)| = |\{u, v\}| = 2$) are replaced with two edges (u, v) and (v, u) with



■ **Figure 1** Unweighted hypergraph H with overlaid star-expansion G_* and illustration of the corresponding hypergraph flow networks. Thin, directed edges have infinite capacity, thick edges have unit capacity. Differences between the networks are highlighted in red: Special treatment of two-pin nets in the network of Liu and Wong [37], removal of low degree vertices in our network.

capacity $c(e)$. For each such star-node, this decreases the network size by two nodes and three edges. Figure 1 shows both networks as well as ours, which we describe in the following.

Removing Low Degree Vertices. We further decrease the network size by using the observation that the (s, t) -node-separator in G_* has to be a subset of the star-nodes, since all other nodes have infinite capacity. Thus it is possible to replace *any* infinite-capacity node by adding a clique between all adjacent star-nodes without affecting the separator. The key observation now is that an infinite-capacity node v with degree $d(v)$ induces $2d(v)$ edges in the Lawler network [33], while a clique between star-nodes induces $d(v)(d(v) - 1)$ edges. For non-star nodes v with $d(v) \leq 3$, it therefore holds that $d(v)(d(v) - 1) \leq 2d(v)$. We therefore remove all infinite-capacity nodes v corresponding to hypernodes with $d(v) \leq 3$ that are *not* incident to any two-pin nets by adding a clique between all star-nodes $\Gamma(v)$. In case v was either source or sink, we create a multi-source multi-sink problem by adding the star-nodes $\Gamma(v)$ to the set of sources resp. sinks [18]. We then apply transformation of Liu and Wong.

Reconstructing Min-Cuts. After computing an (s, t) -max-flow f in the Lawler or Liu-Wong network, an (s, t) -min-cut of H can then be computed by a BFS in the residual graph \mathcal{N}_f starting from s [33]. Let \mathcal{S} be the set of nodes corresponding to vertices of H reached by the BFS. Then $(\mathcal{S}, V \setminus \mathcal{S})$ is an (s, t) -min-cut. Since our network does not contain low degree vertices, we use the following lemma to compute an (s, t) -min-cut of H :

► **Lemma 1.** *Let f be a maximum (s, t) -flow in the Lawler network $\mathcal{N} = (\mathcal{V}, \mathcal{E})$ of a hypergraph $H = (V, E)$ and $(\mathcal{S}, \mathcal{V} \setminus \mathcal{S})$ be the corresponding (s, t) -min-cut in \mathcal{N} . Then for each node $v \in \mathcal{S} \cap V$, the residual graph $\mathcal{N}_f = (\mathcal{V}_f, \mathcal{E}_f)$ contains at least one path $\langle s, \dots, e'' \rangle$ to a bridging node e'' of a net $e \in \mathcal{I}(v)$.*

Proof. The proof can be found in Appendix B. ◀

Thus $(A, V \setminus A)$ is an (s, t) -min-cut of H , where $A := \{v \in e \mid \exists e \in E : \langle s, \dots, e'' \rangle \text{ in } \mathcal{N}_f\}$. Furthermore this allows us to employ the most balanced minimum cut heuristic as described in Section 2.3. By the definition of closed sets it follows that if a bridging node e'' is contained in a closed set C , then all nodes $v \in \Gamma(e'')$ (which correspond to vertices of H) are also contained in C . Thus we can use the respective bridging nodes e'' as representatives of removed low degree vertices.

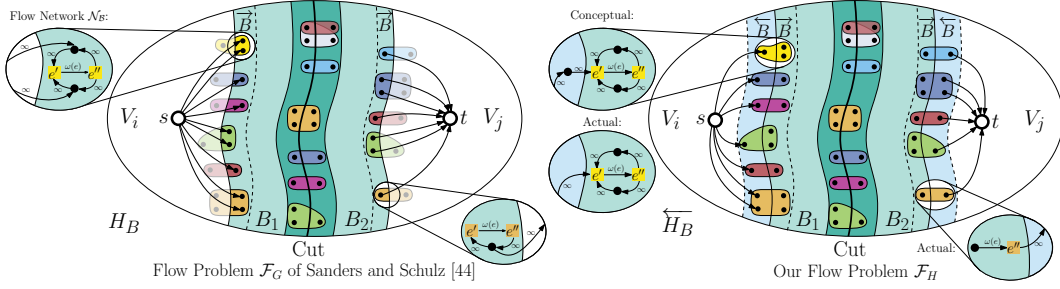
3.2 Constructing the Hypergraph Flow Problem

Let $H_B = (V_B, E_B)$ be the subhypergraph of $H = (V, E)$ that is induced by a corridor B computed in the bipartition $\Pi_2 = (V_i, V_j)$. In the following, we distinguish between the set of *internal* border nodes $\vec{B} := \{v \in V_B \mid \exists e \in E : \{u, v\} \subseteq e \wedge u \notin V_B\}$ and the set of *external* border nodes $\overleftarrow{B} := \{u \notin V_B \mid \exists e \in E : \{u, v\} \subseteq e \wedge v \in V_B\}$. Similarly, we distinguish between *external* nets ($e \cap V_B = \emptyset$) with no pins inside H_B , *internal* nets ($e \cap V_B = e$) with all pins inside H_B , and the set of *border* nets $\overleftrightarrow{E}_B := \{e \in E \mid e \in \mathcal{I}(\vec{B}) \cap \mathcal{I}(\overleftarrow{B})\}$.

A hypergraph flow problem consists of a flow network $\mathcal{N}_B = (\mathcal{V}_B, \mathcal{E}_B)$ derived from H_B and two *additional* nodes s and t that are connected to some nodes $v \in \mathcal{V}_B$. It has the cut property if the max-flow induced min-cut bipartition Π_f of H_B does not increase the $(\lambda - 1)$ -metric in H . Thus it has to hold that $\text{cut}(\Pi_f) \leq \text{cut}(\Pi_2)$. While external nets are not affected by a max-flow computation, the max-flow min-cut theorem [19] ensures the cut property for all internal nets. Border nets however require special attention. Since a border net e is only *partially* contained in H_B and \mathcal{N}_B , it will remain connected to the blocks of its external border nodes in H . In case external border nodes connect e to both V_i and V_j , it will remain a cut net in H even if it is removed from the cut-set in Π_f . It is therefore necessary to “encode” information about external border nodes into the flow problem.

The KaFFPa Model and its Limitations. In KaFFPa, this is done by directly connecting internal border nodes \vec{B} to s and t . This approach can also be used for hypergraphs. In the hypergraph flow problem \mathcal{F}_G , the source s is connected to all nodes $\mathcal{S} = \vec{B} \cap V_i$ and all nodes $\mathcal{T} = \vec{B} \cap V_j$ are connected to t using directed edges with infinite capacity. While this ensures that \mathcal{F}_G has the cut property, it unnecessarily *restricts* the search space. Since all internal border nodes \vec{B} are connected to either s or t , *every* min-cut $(S, V_B \setminus S)$ will have $\mathcal{S} \subseteq S$ and $\mathcal{T} \subseteq V_B \setminus S$. The KaFFPa model therefore prevents *all* min-cuts in which any non-cut border net (i.e., $e \in \overleftrightarrow{E}_B$ with $\lambda(e) = 1$) becomes part of the cut-set. This restricts the space of possible solutions, since corridor B was computed such that *even* a min-cut along either side of the border would result in a feasible cut in H_B . Thus, ideally, *all* vertices $v \in B$ should be able to change their block as result of an (s, t) -max-flow computation on \mathcal{F}_G – not only vertices $v \in B \setminus \vec{B}$. This limitation becomes increasingly relevant for hypergraphs with large nets as well as for partitioning problems with small imbalance ε , since large nets are likely to be only partially contained in H_B and tight balance constraints enforce small B -corridors. While the former is a problem only for HGP, the latter also applies to GP.

A more flexible Model. We exploit the structure of hypergraph flow networks such that (s, t) -max-flow computations can also cut through non-cut border nets. Instead of directly connecting s and t to internal border nodes \vec{B} and thus preventing all min-cuts in which these nodes switch blocks, we conceptually extend H_B to contain all external border nodes \overleftarrow{B} and all border nets \overleftrightarrow{E}_B . The resulting hypergraph is $\overleftarrow{H}_B = (V_B \cup \overleftarrow{B}, \{e \in E \mid e \cap V_B \neq \emptyset\})$. The key insight now is that by using the flow network of \overleftarrow{H}_B and connecting s resp. t to the *external* border nodes $\overleftarrow{B} \cap V_i$ resp. $\overleftarrow{B} \cap V_j$, we get a flow problem that does not lock *any* node $v \in V_B$ in its block, since none of them is directly connected to either s or t . Due to the max-flow min-cut theorem [19], this flow problem has the cut property, since all border nets of H_B are now internal nets and all external border nodes \overleftarrow{B} are locked inside their block. However, it is not necessary to use \overleftarrow{H}_B instead of H_B to achieve this result. For all $v \in \overleftarrow{B}$ the flow network of \overleftarrow{H}_B contains paths $\langle s, v, e' \rangle$ and $\langle e'', v, t \rangle$ that only involve infinite-capacity edges. Therefore we can remove all nodes $v \in \overleftarrow{B}$ by directly connecting s and t to the



■ **Figure 2** Comparison of the KaFFPa flow problem \mathcal{F}_G and our flow problem \mathcal{F}_H . For clarity the zoomed in view is based on the Lawler network.

corresponding *bridging nodes* e', e'' via infinite-capacity edges without affecting the maximal flow [18]. More precisely, in the flow problem \mathcal{F}_H , we connect s to all bridging nodes e' corresponding to border nets $e \in \overrightarrow{E}_B : e \subset \overrightarrow{B} \cap V_i$ and all bridging nodes e'' corresponding to border nets $e \in \overleftarrow{E}_B : e \subset \overleftarrow{B} \cap V_j$ to t using directed, infinite-capacity edges.

Single-Pin Border Nets. Furthermore, we model border nets with $|e \cap \overrightarrow{B}| = 1$ more efficiently. For such a net e , the flow problem contains paths of the form $\langle s, e', e'', v \rangle$ or $\langle v, e', e'', t \rangle$ which can be replaced by paths of the form $\langle s, e', v \rangle$ or $\langle v, e'', t \rangle$ with $c(e', v) = \omega(e)$ resp. $c(v, e'') = \omega(e)$. In both cases we can thus remove one bridging node and two infinite-capacity edges. A comparison of \mathcal{F}_G and \mathcal{F}_H is shown in Figure 2.

3.3 Implementation Details

Since KaHyPar is an n -level partitioner, its FM-based local search algorithms are executed each time a vertex is uncontracted. To prevent expensive recalculations, it therefore uses a cache to maintain the gain values of FM moves throughout the n -level hierarchy [2]. In order to combine our algorithm with FM local search, we not only perform the moves induced by the max-flow min-cut computation but also update the FM gain cache accordingly. Since it is not feasible to execute our algorithm on every level of the n -level hierarchy, we use an exponentially spaced approach that performs flow-based refinements after uncontracting $i = 2^j$ vertices for $j \in \mathbb{N}_+$. This way, the algorithm is executed more often on smaller flow problems than on larger ones. To further improve the running time, we introduce the following speedup techniques: We modify active block scheduling such that after the first round the algorithm is only executed on a pair of blocks if at least one execution using these blocks lead to an improvement on previous levels (S1). We skip flow-based refinement if the cut between two adjacent blocks is less than ten on all levels except the finest (S2). We stop resizing the B -corridor if the current cut did not improve the previously best solution (S3).

4 Experimental Evaluation

We implemented the max-flow min-cut refinement algorithm in KaHyPar. Our implementation is available from <http://www.kahypar.org>. The code is written in C++ and compiled using g++-5.2 with flags `-O3 -march=native`. We refer to our new algorithm as KaHyPar-MF.

Instances. All experiments use the benchmark set of Heuer and Schlag [25], which contains 488 hypergraphs derived from four benchmark sets: the ISPD98 VLSI Circuit Benchmark Suite [3], the DAC 2012 Routability-Driven Placement Contest [52], the SuiteSparse Matrix Collection [13], and the international SAT Competition 2014 [8]. All hypergraphs have unit vertex and net weights. The full benchmark set is referred to as set A. We furthermore use the representative subset of 165 hypergraphs proposed in [25] (set B) and a smaller subset consisting of 25 hypergraphs (set C). Furthermore we use 15 graphs from [39] to compare our flow model \mathcal{F}_H to the KaFFPa model \mathcal{F}_G (set D). Unless mentioned otherwise, all (hyper-)graphs are partitioned into $k \in \{2, 4, 8, 16, 32, 64, 128\}$ blocks with $\varepsilon = 0.03$. For each value of k , a k -way partition is considered to be *one* test instance, resulting in a total of 105 instances for set D, 175 instances for set C, 1155 instances for set B and 3416 instances for set A. An overview about the different benchmark sets is given in Appendix C.

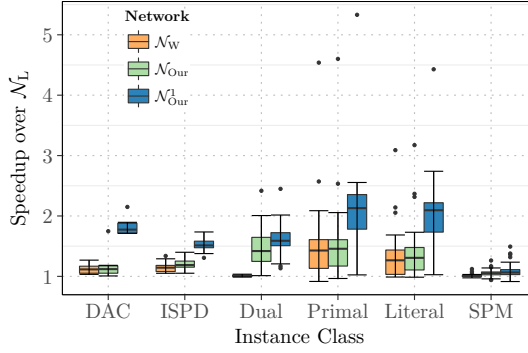
System and Methodology. We compare KaHyPar-MF to KaHyPar-CA, the k -way (hMetis-K) and recursive bisection variants (hMetis-R) of hMetis 2.0 (p1) [30, 31], and PaToH 3.2 [10], as these provide the best solution quality [2, 25]. The results of these tools are available online¹ and report the *arithmetic mean* of the computed cut and running time as well as the best cut found for ten repetitions with different seeds per instance. Since PaToH ignores the seed if configured to use the quality preset, the results contain one run of the quality (PaToH-Q) and ten runs using the default preset (PaToH-D). Each partitioner had a time limit of eight hours per instance. We use the same number of repetitions and the same time limit for our experiments². Furthermore the experiments are performed on the same machine, which runs RHEL 7.2 and consists of two Intel Xeon E5-2670 Octa-Core processors clocked at 2.6 GHz, 64 GB main memory, 20 MB L3- and 8x256 KB L2-Cache. When averaging over different instances we use the *geometric mean* in order to give every instance a comparable influence on the final result. To compare the algorithms in terms of solution quality, we use *improvement plots*. For each algorithm, these plots relate the minimum connectivity of KaHyPar-MF to the minimum connectivity of the algorithm on a *per-instance* basis. For each algorithm, these ratios are sorted in decreasing order. The plots use a cube root scale for the y-axis to reduce right skewness [12] and show the improvement of KaHyPar-MF in percent (i.e., $1 - (\text{KaHyPar-MF}/\text{algorithm})$). A value below zero indicates that KaHyPar-MF performed worse than the corresponding algorithm, while a value above zero indicates that KaHyPar-MF performed better than the algorithm in question. A value of zero implies that the partitions of both algorithms had the same solution quality. Values above one correspond to infeasible solutions that violated the balance constraint. To include instances with a connectivity of zero into the results, we set these values to *one* for ratio computations.

4.1 Evaluating Flow Networks and Models

Flow Networks. To analyze the effects of the different flow networks we compute five bipartitions for each hypergraph of set B with KaHyPar-CA. These are then used to generate flow networks for a corridor of size $|B| = 25\,000$ vertices around the cut, from which we create flow problems \mathcal{F}_H . Due to space constraints, we only report how the reductions in network size translate to improved running times of the max-flow algorithm and refer to the full version of the paper [26] for a detailed discussion. We use the highly tuned IBFS

¹ <http://algo2.iti.kit.edu/schlag/sea2017/>

² Detailed per-instance results can be found online: <http://algo2.iti.kit.edu/schlag/sea2018/>



■ **Figure 3** Speedup of the IBFS [21] max-flow algorithm over the execution on \mathcal{N}_L .

■ **Table 1** Comparing the KaFFPa flow model \mathcal{F}_G with our model \mathcal{F}_H as described in Section 3.2. The table shows the average improvement of \mathcal{F}_H over \mathcal{F}_G (in [%]) on benchmark sets C and D.

α'	Hypergraphs (Set C)			Graphs (Set D)		
	$\varepsilon = 1\%$	$\varepsilon = 3\%$	$\varepsilon = 5\%$	$\varepsilon = 1\%$	$\varepsilon = 3\%$	$\varepsilon = 5\%$
1	7.7	8.1	7.6	11.7	11.3	10.5
2	7.9	6.6	4.8	11.0	9.1	7.8
4	6.9	3.9	2.7	9.9	7.3	5.4
8	5.1	2.3	1.5	8.6	5.3	3.9
16	3.4	1.3	1.2	7.0	4.1	3.5

algorithm [21] to compute min-cuts, which performed best in preliminary experiments [24, 26]. Figure 3 compares the speedups of IBFS when executed on the Liu-Wong network \mathcal{N}_W , our network \mathcal{N}_{Our} , and \mathcal{N}_{Our}^1 to the execution on the Lawler network \mathcal{N}_L . \mathcal{N}_{Our}^1 exploits the fact that our flow problems are based on subhypergraphs H_B by additionally modeling single-pin border nets more efficiently as described in Section 3.2. We see that the algorithm benefits from improved network models on all instance classes except SPM. These instances have high average vertex degrees and large average net sizes, in which case the optimizations only have a very limited effect since they target small nets and low degree vertices. While \mathcal{N}_W speeds up computation for Primal and Literal instances, \mathcal{N}_{Our} additionally leads to a speedup for Dual instances. Using \mathcal{N}_{Our}^1 , which combines these techniques with efficient border net modeling results in an average speedup between 1.52 and 2.21 (except for SPM instances). Therefore we use \mathcal{N}_{Our}^1 in all following experiments.

Flow Models. We now compare the flow model \mathcal{F}_G of KaFFPa to our advanced model \mathcal{F}_H described in Section 3.2. The experiments summarized in Table 1 were performed using benchmark sets C and D. To focus on the impact of the models on solution quality, we deactivated KaHyPar’s FM local search algorithms and only use flow-based refinement without the most balanced minimum cut heuristic. The results confirm our hypothesis that \mathcal{F}_G restricts the search space of possible solutions. For *all* flow problem sizes and all imbalances tested, \mathcal{F}_H yields better solution quality. As expected, the effects are most pronounced for small flow problems and small imbalances where many vertices are likely to be border nodes. Since these nodes are locked inside their respective block in \mathcal{F}_G , they prevent all non-cut border nets from becoming part of the cut-set. Our model, on the other hand, allows *all* min-cuts that yield a feasible solution for the original partitioning problem. The fact that this effect *also* occurs for the graphs of set D indicates that our model can also be effective for traditional graph partitioning. In all following experiments, we use \mathcal{F}_H .

■ **Table 2** Quality and running times for different framework configurations and increasing α' . Column Avg[%] reports the quality improvement relative to the reference configuration (-F,-M,+FM).

α'	(+F,-M,-FM)		(+F,+M,-FM)		(+F,-M,+FM)		(+F,+M,+FM)		CONSTANT128	
	Avg[%]	$t[s]$	Avg[%]	$t[s]$	Avg[%]	$t[s]$	Avg[%]	$t[s]$	Avg[%]	$t[s]$
1	-6.09	12.9	-5.60	13.4	0.25	15.0	0.23	15.2	0.54	67.9
2	-3.19	15.8	-2.07	16.7	0.59	17.0	0.73	17.5	1.11	140.2
4	-1.82	20.4	-0.19	21.9	0.90	20.4	1.21	21.5	1.66	269.6
8	-0.85	28.5	0.98	30.7	1.24	26.5	1.71	28.7	2.20	512.1
16	-0.19	43.3	1.75	46.7	1.60	37.5	2.21	41.3	2.76	973.9
Ref.	(-F,-M,+FM)		6373.88		13.7					

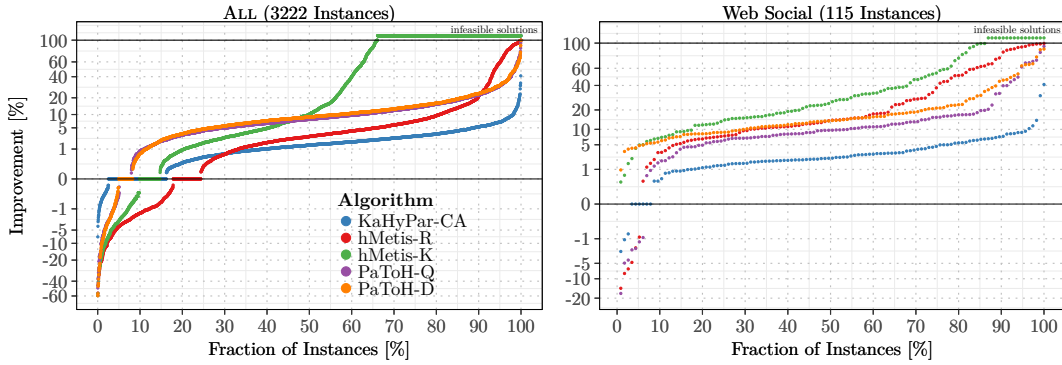
■ **Table 3** Comparison of quality improvement and running times using speedup heuristics. Column $t_{\text{flow}}[s]$ refers to the running time of flow-based refinement, $t[s]$ to the total partitioning time.

Configuration	Avg	Min	$t_{\text{flow}}[s]$	$t[s]$
KaHyPar-CA	7077.20	6820.17	-	29.3
KaHyPar-MF	2.47 %	2.12 %	43.0	72.3
KaHyPar-MF _(S1)	2.41 %	2.06 %	33.9	63.2
KaHyPar-MF _(S1,S2)	2.40 %	2.05 %	28.5	57.8
KaHyPar-MF _(S1,S2,S3)	2.41 %	2.06 %	21.2	50.5

4.2 Configuring the Algorithm

We now evaluate different configurations of our refinement framework on set C. In the following, KaHyPar-CA [25] is used as a reference and referred to as (-F,-M,+FM), since it neither uses (F)lows nor the (M)ost balanced minimum cut heuristic and only relies on the (FM) algorithm for local search. This basic configuration is then successively extended with specific components. The results of our experiments are summarized in Table 2 for increasing scaling parameter α' . In configuration CONSTANT128 all components are enabled (+F,+M,+FM) and flow-based refinements are performed every 128 uncontractions. It is used as a reference point for the quality achievable using flow-based refinement.

The results indicate that only using flow-based refinement (+F,-M,-FM) is inferior to FM local search in regard to both running time and solution quality. Although the quality improves with increasing flow problem size (i.e., increasing α'), the average connectivity is still worse than the reference configuration. Enabling the most balanced minimum cut heuristic improves partitioning quality. Configuration (+F,+M,-FM) performs better than the basic configuration for $\alpha' \geq 8$. By combining flows with the FM algorithm (+F,-M,+FM) we get a configuration that improves upon the baseline even for small flow problems. However, comparing this variant with (+F,+M,-FM) for $\alpha' = 16$, we see that using large flow problems together with the most balanced minimum cut heuristic yields solutions of comparable quality. Enabling all components (+F,+M,+FM) and using large flow problems performs best. Furthermore we see that enabling FM local search slightly improves the running time for $\alpha' \geq 8$. This can be explained by the fact that the FM algorithm already produces good cuts between the blocks such that fewer rounds of pairwise flow refinements are necessary to improve the solution. Comparing configuration (+F,+M,+FM) with CONSTANT128 shows that performing flows more often further improves solution quality at the cost of slowing down the algorithm by more than an order of magnitude. In all further experiments, we therefore use configuration (+F,+M,+FM) with $\alpha' = 16$ for KaHyPar-MF. This configuration also performed best in the effectiveness tests presented in Appendix D. While this configuration performs better than KaHyPar-CA, its running time is still more than a factor of 3 higher.



■ **Figure 4** Improvement plots comparing KaHyPar-MF with KaHyPar-CA and other partitioners.

We therefore perform additional experiments on set B and successively enable the speedup techniques described in Section 3.3. The results are summarized in Table 3. Only executing pairwise flow refinements on blocks that lead to an improvement on previous levels (S1) reduces the running time of flow-based refinement by a factor of 1.27, while skipping flows in case of small cuts (S2) results in a further speedup of 1.19. By additionally stopping the resizing of the flow problem as early as possible (S3), we decrease the running time of flow-based improvement by a factor of 2 in total, while still computing solutions of comparable quality. Thus in the comparisons with other systems, all heuristics are enabled.

4.3 Comparison with other Systems

We now compare KaHyPar-MF to the state-of-the-art HGP systems on the full benchmark set. The following comparison is based on 3222 instances, since we exclude the same 194 out of 3416 instances that were already excluded in [25]. As can be seen in Figure 4 (left), KaHyPar-MF outperforms all other algorithms. Comparing the best solutions of KaHyPar-MF to each partitioner individually across all instances, it produced *better* partitions than PaToH-Q, PaToH-D, hMetis-K, KaHyPar-CA, hMetis-R for 92.1%, 91.7%, 85.3%, 83.8%, and 75.6% of the instances, respectively. Comparing the best solutions of all systems simultaneously, KaHyPar-MF produced the best partitions for 2427 of the 3222 instances. As can be seen in Figure 4 (right), the improvement is most pronounced for hypergraphs derived from matrices of web graphs and social networks³, which are difficult to partition due to skewed degree and net size distributions. With 55.7s, the average running time of KaHyPar-MF is less than a factor of two slower than KaHyPar-CA, which took 31.1s. The average running times of hMetis-R, hMetis-K, PaToH-Q and PaToH-D were 79.2s, 57.9s, 5.9s and 1.2s, respectively. A detailed comparison of running times and solution quality can be found in Appendix E.

5 Conclusion

We generalize KaFFPa’s flow-based refinement framework [44] from graph to hypergraph partitioning. By removing low degree hypernodes and exploiting the fact that our flow problems are built on subhypergraphs, we reduce the size of hypergraph flow networks.

³ Based on the following matrices: `webbase-1M`, `ca-CondMat`, `soc-sign-epinions`, `wb-edu`, `IMDB`, `as-22july06`, `as-caida`, `astro-ph`, `HEP-th`, `Oregon-1`, `Reuters911`, `PGPgiantcompo`, `NotreDame_www`, `NotreDame_actors`, `p2p-Gnutella25`, `Stanford`, `cnr-2000`.

Furthermore we identify shortcomings of the KaFFPa [44] approach that restrict feasible solutions and introduce an advanced model that overcomes these limitations by utilizing the structure of hypergraph flow networks. Lastly, we present techniques to improve the running time of the framework by a factor of 2 without affecting solution quality. The resulting hypergraph partitioner KaHyPar-MF performs better than all competing algorithms and has a running time comparable to that of hMetis. Since our flow model yields better solutions for both hypergraphs *and* graphs than the KaFFPa approach, future work includes the integration of our flow model into KaFFPa and the evaluation in the context of graph partitioning. We also plan to extend our framework to optimize other objectives such as cut.

References

- 1 P. Agrawal, B. Narendran, and N. Shivakumar. Multi-way partitioning of VLSI circuits. In *9th International Conference on VLSI Design*, pages 393–399, 1996.
- 2 Y. Akhremtsev, T. Heuer, P. Sanders, and S. Schlag. Engineering a direct k -way Hypergraph Partitioning Algorithm. In *19th Workshop on Algorithm Engineering and Experiments, (ALENEX)*, pages 28–42, 2017.
- 3 C. J. Alpert. The ISPD98 Circuit Benchmark Suite. In *International Symposium on Physical Design, (ISPD)*, pages 80–85, 1998.
- 4 C. J. Alpert, J.-H. Huang, and A. B. Kahng. Multilevel Circuit Partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(8):655–667, 1998.
- 5 C. J. Alpert and A. B. Kahng. Recent Directions in Netlist Partitioning: a Survey. *Integration, the VLSI Journal*, 19(1–2):1–81, 1995.
- 6 C. Aykanat, B. B. Cambazoglu, and B. Uçar. Multi-level Direct K-way Hypergraph Partitioning with Multiple Constraints and Fixed Vertices. *Journal of Parallel and Distributed Computing*, 68(5):609–625, 2008.
- 7 D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, editors. *Proc. Graph Partitioning and Graph Clustering - 10th DIMACS Implementation Challenge Workshop*, volume 588 of *Contemporary Mathematics*. AMS, 2013. doi:10.1090/conm/588.
- 8 A. Belov, D. Diepold, M. Heule, and M. Järvisalo. The SAT Competition 2014. <http://www.satcompetition.org/2014/>, 2014.
- 9 T. N. Bui and C. Jones. A Heuristic for Reducing Fill-In in Sparse Matrix Factorization. In *SIAM Conference on Parallel Processing for Scientific Computing*, pages 445–452, 1993.
- 10 Ü. V. Catalyürek and C. Aykanat. Hypergraph-Partitioning-Based Decomposition for Parallel Sparse-Matrix Vector Multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, Jul 1999.
- 11 J. Cong and M. Smith. A Parallel Bottom-up Clustering Algorithm with Applications to Circuit Partitioning in VLSI Design. In *30th Conference on Design Automation, (DAC)*, pages 755–760, 1993.
- 12 N. J. Cox. Stata tip 96: Cube roots. *Stata Journal*, 11(1):149–154(6), 2011. URL: <http://www.stata-journal.com/article.html?article=st0223>.
- 13 T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, 38(1):1:1–1:25, 2011.
- 14 K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and Ü. V. Catalyürek. Parallel Hypergraph Partitioning for Scientific Computing. In *20th International Conference on Parallel and Distributed Processing, (IPDPS)*, pages 124–124. IEEE, 2006.
- 15 W.E. Donath. Logic partitioning. *Physical Design Automation of VLSI Systems*, pages 65–86, 1988.
- 16 J. Edmonds and R. M. Karp. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *Journal of the ACM*, 19(2):248–264, 1972.

- 17 C. Fiduccia and R. Mattheyses. A Linear Time Heuristic for Improving Network Partitions. In *19th ACM/IEEE Design Automation Conf.*, pages 175–181, 1982.
- 18 D. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.
- 19 Lester R Ford and Delbert R Fulkerson. Maximal Flow through a Network. *Canadian Journal of Mathematics*, 8(3):399–404, 1956.
- 20 A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35(4):921–940, 1988.
- 21 Andrew Goldberg, Sagi Hed, Haim Kaplan, Robert Tarjan, and Renato Werneck. Maximum Flows by Incremental Breadth-First Search. *19th European Symposium on Algorithms, (ESA)*, pages 457–468, 2011.
- 22 S. Hauck and G. Borriello. An Evaluation of Bipartitioning Techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(8):849–866, 1997.
- 23 B. Hendrickson and R. Leland. A Multi-Level Algorithm For Partitioning Graphs. *1995 Conference on Supercomputing, (SC)*, 0:28, 1995.
- 24 T. Heuer. High Quality Hypergraph Partitioning via Max-Flow-Min-Cut Computations. Master’s thesis, KIT, 2018.
- 25 T. Heuer and S. Schlag. Improving Coarsening Schemes for Hypergraph Partitioning by Exploiting Community Structure. In *16th International Symposium on Experimental Algorithms, (SEA)*, page 21:1–21:19, 2017.
- 26 Heuer, T. and Sanders, P. and Schlag, S. Network Flow-Based Refinement for Multilevel Hypergraph Partitioning, 2018. [arXiv:1802.03587](https://arxiv.org/abs/1802.03587).
- 27 T. C. Hu and K. Moerder. Multiterminal Flows in a Hypergraph. In T.C. Hu and E.S. Kuh, editors, *VLSI Circuit Layout: Theory and Design*, chapter 3, pages 87–93. IEEE Press, 1985.
- 28 E. Ihler, D. Wagner, and F. Wagner. Modeling Hypergraphs by Graphs with the Same Mincut Properties. *Information Processing Letters*, 45(4):171–175, 1993. doi:10.1016/0020-0190(93)90115-P.
- 29 I. Kabiljo, B. Karrer, M. Pundir, S. Pupyrev, A. Shalita, Y. Akhremtsev, and Presta. A. Social Hash Partitioner: A Scalable Distributed Hypergraph Partitioner. *PVLDB*, 10(11):1418–1429, 2017.
- 30 G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel Hypergraph Partitioning: Applications in VLSI Domain. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(1):69–79, 1999.
- 31 G. Karypis and V. Kumar. Multilevel K -way Hypergraph Partitioning. In *36th Design Automation Conference, (DAC)*, pages 343–348. ACM, 1999.
- 32 B. W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell System Technical Journal*, 49(2):291–307, Feb 1970.
- 33 E. Lawler. Cutsets and Partitions of Hypergraphs. *Networks*, 3(3):275–285, 1973.
- 34 E. Lawler. *Combinatorial Optimization : Networks and Matroids*. Holt, Rinehart, and Winston, 1976.
- 35 T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. John Wiley & Sons, Inc., 1990.
- 36 J. Li, J. Lillis, and C. K. Cheng. Linear decomposition algorithm for VLSI design applications. In *1995 International Conference on Computer Aided Design, (ICCAD)*, pages 223–228, Nov 1995.
- 37 H. Liu and D. F. Wong. Network-Flow-Based Multiway Partitioning with Area and Pin Constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(1):50–59, Jan 1998.

- 38 Z. Mann and P. Papp. Formula partitioning revisited. In Daniel Le Berre, editor, *POS-14. Fifth Pragmatics of SAT workshop*, volume 27 of *EPiC Series in Computing*, pages 41–56. EasyChair, 2014.
- 39 H. Meyerhenke, P. Sanders, and C. Schulz. Partitioning Complex Networks via Size-Constrained Clustering. In *13th International Symposium on Experimental Algorithms, (SEA)*, pages 351–363, 2014.
- 40 D. A. Papa and I. L. Markov. Hypergraph Partitioning and Clustering. In T. F. Gonzalez, editor, *Handbook of Approximation Algorithms and Metaheuristics*. Chapman and Hall/CRC, 2007. doi:10.1201/9781420010749.
- 41 Jean-Claude Picard and Maurice Queyranne. On the Structure of all Minimum Cuts in a Network and Applications. *Combinatorial Optimization II*, pages 8–16, 1980.
- 42 Joachim Pistorius and Michel Minoux. An Improved Direct Labeling Method for the Max-Flow Min-Cut Computation in Large Hypergraphs and Applications. *International Transactions in Operational Research*, 10(1):1–11, 2003.
- 43 L. A. Sanchis. Multiple-way Network Partitioning. *IEEE Transactions on Computers*, 38(1):62–81, 1989.
- 44 P. Sanders and C. Schulz. Engineering Multilevel Graph Partitioning Algorithms. In *19th European Symposium on Algorithms, (ESA)*, volume 6942 of *LNCS*, pages 469–480. Springer, 2011.
- 45 S. Schlag, V. Henne, T. Heuer, H. Meyerhenke, P. Sanders, and C. Schulz. k -way Hypergraph Partitioning via n -Level Recursive Bisection. In *18th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 53–67, 2016.
- 46 D. G. Schweikert and B. W. Kernighan. A Proper Model for the Partitioning of Electrical Circuits. In *9th Design Automation Workshop, (DAC)*, pages 57–62. ACM, 1972.
- 47 A. Trifunovic. *Parallel Algorithms for Hypergraph Partitioning*. PhD thesis, University of London, 2006.
- 48 A. Trifunović and W. J. Knottenbelt. Parallel Multilevel Algorithms for Hypergraph Partitioning. *Journal of Parallel and Distributed Computing*, 68(5):563–581, 2008.
- 49 Ü. V. Çatalyürek and M. Deveci and K. Kaya and B. Uçar. UMPa: A multi-objective, multi-level partitioner for communication minimization. In Bader et al. [7], pages 53–66. doi:10.1090/conm/588.
- 50 B. Uçar and C. Aykanat. Encapsulating Multiple Communication-Cost Metrics in Partitioning Sparse Rectangular Matrices for Parallel Matrix-Vector Multiplies. *SIAM Journal on Scientific Computing*, 25(6):1837–1859, 2004.
- 51 B. Vastenhouw and R. H. Bisseling. A Two-Dimensional Data Distribution Method for Parallel Sparse Matrix-Vector Multiplication. *SIAM Review*, 47(1):67–95, 2005.
- 52 N. Viswanathan, C. Alpert, C. Sze, Z. Li, and Y. Wei. The DAC 2012 Routability-driven Placement Contest and Benchmark Suite. In *49th Annual Design Automation Conference, (DAC)*, pages 774–782. ACM, 2012.
- 53 S. Wichlund. On multilevel circuit partitioning. In *International Conference on Computer-Aided Design, (ICCAD)*, ICCAD, pages 505–511. ACM, 1998.
- 54 H. H. Yang and D. F. Wong. Efficient Network Flow Based Min-Cut Balanced Partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(12):1533–1540, 1996.

A Framework Pseudocode

Algorithm 1: Flow-Based Refinement.

Input: Hypergraph H , k -way partition $\Pi_k = \{V_1, \dots, V_k\}$, imbalance parameter ε .

Algorithm MaxFlowMinCutRefinement(H, Π_k)

```

 $Q := \text{QuotientGraph}(H, \Pi_k)$ 
while  $\exists$  active blocks  $\in Q$  do // in the beginning all blocks are active
  foreach  $\{(V_i, V_j) \in Q \mid V_i \vee V_j \text{ is active}\}$  do // choose a pair of blocks
     $\Pi_{\text{old}} = \Pi_{\text{best}} := \{V_i, V_j\} \subseteq \Pi_k$  // extract bipartition to be improved
     $\varepsilon_{\text{old}} = \varepsilon_{\text{best}} := \text{imbalance}(\Pi_k)$  // imbalance of current  $k$ -way partition
     $\alpha := \alpha'$  // use large  $B$ -corridor for first iteration
    do // adaptive flow iterations
       $B := \text{computeB-Corridor}(H, \Pi_{\text{best}}, \alpha\varepsilon)$  // as described in Section 2.3
       $H_B := \text{SubHypergraph}(H, B)$  // create  $B$ -induced subhypergraph
       $\mathcal{N}_B := \text{FlowNetwork}(H_B)$  // as described in Section 3.1
       $\mathcal{F} := \text{FlowProblem}(\mathcal{N}_B)$  // as described in Section 3.2
       $f := \text{maxFlow}(\mathcal{F})$  // compute maximum flow on  $\mathcal{F}$ 
       $\Pi_f := \text{mostBalancedMinCut}(f, \mathcal{F})$  // as in Section 2.3 & 3.1
       $\varepsilon_f := \text{imbalance}(\Pi_f \cup \Pi_k \setminus \Pi_{\text{old}})$  // imbalance of new  $k$ -way partition
      if  $(\text{cut}(\Pi_f) < \text{cut}(\Pi_{\text{best}}) \wedge \varepsilon_f \leq \varepsilon) \vee \varepsilon_f < \varepsilon_{\text{best}}$  then // found improvement
         $\alpha := \min(2\alpha, \alpha')$ ,  $\Pi_{\text{best}} := \Pi_f$ ,  $\varepsilon_{\text{best}} := \varepsilon_f$  // update  $\alpha$ ,  $\Pi_{\text{best}}$ ,  $\varepsilon_{\text{best}}$ 
      else  $\alpha := \frac{\alpha}{2}$  // decrease size of  $B$ -corridor in next iteration
    while  $\alpha \geq 1$ 
    if  $\Pi_{\text{best}} \neq \Pi_{\text{old}}$  then // improvement found
       $\Pi_k := \Pi_{\text{best}} \cup \Pi_k \setminus \Pi_{\text{old}}$  // replace  $\Pi_{\text{old}}$  with  $\Pi_{\text{best}}$ 
      activateForNextRound( $V_i, V_j$ ) // reactivate blocks for next round
  return  $\Pi_k$ 

```

Output: improved ε -balanced k -way partition $\Pi_k = \{V_1, \dots, V_k\}$

B Proof of Lemma 1

► **Lemma 1.** Let f be a maximum (s, t) -flow in the Lawler network $\mathcal{N} = (\mathcal{V}, \mathcal{E})$ of a hypergraph $H = (V, E)$ and $(\mathcal{S}, \mathcal{V} \setminus \mathcal{S})$ be the corresponding (s, t) -min-cut in \mathcal{N} . Then for each node $v \in \mathcal{S} \cap V$, the residual graph $\mathcal{N}_f = (\mathcal{V}_f, \mathcal{E}_f)$ contains at least one path $\langle s, \dots, e'' \rangle$ to a bridging node e'' of a net $e \in I(v)$.

Proof. Since $v \in \mathcal{S}$, there has to be some path $s \rightsquigarrow v$ in \mathcal{N}_f . By definition of the flow network, this path can either be of the form $P_1 = \langle s, \dots, e'', v \rangle$ or $P_2 = \langle s, \dots, e', v \rangle$ for some bridging nodes e', e'' corresponding to nets $e \in I(v)$. In the former case we are done, since $e'' \in P_1$. In the latter case the existence of edge $(e', v) \in \mathcal{E}_f$ implies that there is a positive flow $f(v, e') > 0$ over edge $(v, e') \in \mathcal{E}$. Due to flow conservation, there exists at least one edge $(e'', v) \in \mathcal{E}$ with $f(e'', v) > 0$, which implies that $(v, e'') \in \mathcal{E}_f$. Thus we can extend the path P_2 to $\langle s, \dots, e', v, e'' \rangle$. ◀

C Overview and Properties of Benchmark Sets

Sparse matrices (SPM) are translated into hypergraphs using the row-net model [10], i.e., each row is treated as a net and each column as a vertex. SAT instances are converted to three different representations: For literal hypergraphs, each boolean *literal* is mapped to one vertex and each clause constitutes a net [40], while in the *primal* model each variable is represented by a vertex and each clause is represented by a net. In the *dual* model the opposite is the case [38].

■ **Table 4** Overview about different benchmark sets. Set B and set C are subsets of set A.

	Source	#	DAC	ISPD98	Primal	Dual	Literal	SPM	Graphs
Set A	[25]	477	10	18	92	92	92	184	-
Set B	[25]	165	5	10	30	30	30	60	-
Set C	new	25	-	5	5	5	5	5	-
Set D	[39]	15	-	-	-	-	-	-	15

■ **Table 5** Basic properties of set C. The number of pins is denoted with p .

Class	Hypergraph	n	m	p
ISPD	ibm06	32 498	34 826	128 182
	ibm07	45 926	48 117	175 639
	ibm08	51 309	50 513	204 890
	ibm09	53 395	60 902	222 088
	ibm10	69 429	75 196	297 567
Dual	6s9	100 384	34 317	234 228
	6s133	140 968	48 215	328 924
	6s153	245 440	85 646	572 692
	dated-10-11-u	629 461	141 860	1 429 872
	dated-10-17-u	1 070 757	229 544	2 471 122
Literal	6s133	96 430	140 968	328 924
	6s153	171 292	245 440	572 692
	aaai10-planning-ipc5	107 838	308 235	690 466
	dated-10-11-u	283 720	629 461	1 429 872
	atco_enc2_opt1_05_21	112 732	526 872	2 097 393
Primal	6s153	85 646	245 440	572 692
	aaai10-planning-ipc5	53 919	308 235	690 466
	hwmcc10-timeframe	163 622	488 120	1 138 944
	dated-10-11-u	141 860	629 461	1 429 872
	atco_enc2_opt1_05_21	56 533	526 872	2 097 393
SPM	mult_dcop_01	25 187	25 187	193 276
	vibrobox	12 328	12 328	342 828
	RFdevice	74 104	74 104	365 580
	mixtank_new	29 957	29 957	1 995 041
	laminar_duct3D	67 173	67 173	3 833 077

■ **Table 6** Basic properties of the graph instances.

Graph	n	m
p2p-Gnutella04	6 405	29 215
wordassociation-2011	10 617	63 788
PGPgiantcompo	10 680	24 316
email-EuAll	16 805	60 260
as-22july06	22 963	48 436
soc-Slashdot0902	28 550	379 445
loc-brightkite	56 739	212 945
enron	69 244	254 449
loc-gowalla	196 591	950 327
coAuthorsCiteseer	227 320	814 134
wiki-Talk	232 314	≈1.5M
citationCiteseer	268 495	≈1.2M
coAuthorsDBLP	299 067	977 676
cnr-2000	325 557	≈2.7M
web-Google	356 648	≈2.1M

■ **Table 7** Results of the effectiveness test for different configurations of our flow-based refinement framework for increasing α' . The quality in column Avg[%] is relative to the baseline configuration.

Config.	(+F,-M,-FM)	(+F,+M,-FM)	(+F,-M,+FM)	(+F,+M,+FM)
α'	Avg[%]	Avg[%]	Avg[%]	Avg[%]
1	-6.06	-5.52	0.23	0.24
2	-3.15	-2.06	0.55	0.73
4	-1.89	-0.19	0.86	1.20
8	-0.87	0.96	1.20	1.69
16	-0.29	1.66	1.52	2.17
Ref.	(-F,-M,+FM)	6377.15		

D Effectiveness Tests

We give each configuration the *same* time to compute a partition. For each instance (H, k) , we execute each configuration once and note the *largest* running time $t_{H,k}$. Then each configuration gets time $3t_{H,k}$ to compute a partition (i.e., we take the best partition out of several repeated runs). Whenever a new run of a partition would exceed the largest running time, we perform the next run with a certain probability such that the expected running time is $3t_{H,k}$. The results of this procedure, which was initially proposed in [44], are presented in Table 7. Combinations of flows and FM local search perform better than repeated executions of the baseline configuration, with (+F,+M,-FM) and $\alpha' = 16$ performing best.

E Comparison of Running Times and Solution Quality

■ **Table 8** Comparing the average running times of KaHyPar-MF with KaHyPar-CA and other hypergraph partitioners for different benchmark sets (top) and different values of k (bottom).

Algorithm	Running Time $t[s]$							
	All	DAC	ISPD98	Primal	Literal	Dual	SPM	WebSoc
KaHyPar-MF	55.67	504.27	20.83	61.78	119.51	97.22	27.40	110.15
KaHyPar-CA	31.05	368.97	12.35	32.91	64.65	68.27	13.91	67.14
hMetis-R	79.23	446.36	29.03	66.25	142.12	200.36	41.79	89.69
hMetis-K	57.86	240.92	23.18	44.23	94.89	125.55	35.95	111.95
PaToH-Q	5.89	28.34	1.89	6.90	9.24	10.57	3.42	4.71
PaToH-D	1.22	6.45	0.35	1.12	1.58	2.87	0.77	0.88
	$k = 2$	$k = 4$	$k = 8$	$k = 16$	$k = 32$	$k = 64$	$k = 128$	
KaHyPar-MF	19.75	32.89	47.52	60.38	78.51	100.34	119.15	
KaHyPar-CA	12.68	17.16	23.88	31.01	41.69	57.35	76.61	
hMetis-R	27.87	51.59	74.74	91.09	109.13	128.66	149.34	
hMetis-K	25.47	32.27	42.50	53.41	74.00	109.12	152.92	
PaToH-Q	1.93	3.61	5.44	7.01	8.40	10.06	11.44	
PaToH-D	0.43	0.77	1.12	1.42	1.71	2.02	2.29	

■ **Table 9** Comparing the best solutions of KaHyPar-MF with the best results of KaHyPar-CA and other partitioners for different benchmark sets (top) and different values of k (bottom). All values correspond to the quality improvement of KaHyPar-MF relative to the respective partitioner (in %).


Algorithm	Min. $(\lambda - 1)$							
	All	DAC	ISPD98	Primal	Literal	Dual	SPM	WebSoc
KaHyPar-MF	7542.88	16828.15	5511.40	15236.13	15197.60	2927.42	6010.05	7478.06
KaHyPar-CA	2.22 %	2.80 %	1.92 %	1.85 %	2.46 %	3.33 %	1.74 %	3.91 %
hMetis-R	14.40 %	4.75 %	2.76 %	3.88 %	4.17 %	31.20 %	16.37 %	41.92 %
hMetis-K	12.92 %	7.77 %	2.17 %	4.78 %	6.91 %	21.51 %	16.23 %	40.45 %
PaToH-Q	11.48 %	15.24 %	9.53 %	14.36 %	14.98 %	11.44 %	8.36 %	18.45 %
PaToH-D	12.06 %	15.57 %	10.90 %	12.47 %	15.17 %	13.64 %	9.45 %	23.04 %
	$k = 2$	$k = 4$	$k = 8$	$k = 16$	$k = 32$	$k = 64$	$k = 128$	
KaHyPar-MF	1005.76	2985.22	5805.19	9097.31	14352.34	21537.33	31312.48	
KaHyPar-CA	1.71 %	2.16 %	2.51 %	2.51 %	2.45 %	2.16 %	2.05 %	
hMetis-R	22.25 %	17.62 %	15.63 %	14.29 %	11.94 %	9.80 %	8.01 %	
hMetis-K	21.82 %	13.66 %	12.76 %	13.49 %	10.62 %	9.18 %	7.81 %	
PaToH-Q	14.92 %	12.60 %	11.81 %	11.66 %	10.66 %	9.77 %	8.63 %	
PaToH-D	8.54 %	10.41 %	13.64 %	14.50 %	12.70 %	12.66 %	11.89 %	

Aggregative Coarsening for Multilevel Hypergraph Partitioning

Ruslan Shaydulin

School of Computing, Clemson University, Clemson, SC


rshaydu@g.clemson.edu

 <https://orcid.org/0000-0002-8657-2848>

Ilya Safro

School of Computing, Clemson University, Clemson, SC

isafro@g.clemson.edu

 <https://orcid.org/0000-0001-6284-7408>

Abstract

Algorithms for many hypergraph problems, including partitioning, utilize multilevel frameworks to achieve a good trade-off between the performance and the quality of results. In this paper we introduce two novel aggregative coarsening schemes and incorporate them within state-of-the-art hypergraph partitioner Zoltan. Our coarsening schemes are inspired by the algebraic multigrid and stable matching approaches. We demonstrate the effectiveness of the developed schemes as a part of multilevel hypergraph partitioning framework on a wide range of problems.

2012 ACM Subject Classification Mathematics of computing → Hypergraphs, Mathematics of computing → Graph algorithms, Mathematics of computing → Matchings and factors

Keywords and phrases hypergraph partitioning, multilevel algorithms, coarsening, matching, combinatorial scientific computing

Digital Object Identifier 10.4230/LIPIcs.SEA.2018.2

Related Version A full version of the paper is available at [51], <https://arxiv.org/abs/1802.09610>.

Funding This material is based upon work supported by the National Science Foundation under Grant No. 1522751.

1 Introduction

Hypergraph is a generalization of graph. Whereas in a graph each edge connects only two vertices, in a hypergraph a hyperedge can connect an arbitrary number of vertices. In many cases this allows hypergraph to better capture the underlying structure of the problem. In graph partitioning (GP), the goal is to split the vertex set of a graph into approximately even parts while minimizing the number of the edges in a cut [8]. Hypergraph partitioning problem (HGP) extends it to hypergraphs. Hypergraph partitioning has many applications in fields ranging from VLSI design [30] to parallel matrix multiplication [10] to classification [55] to optimizing distributed systems [33, 13], among others [16, 28].

Hypergraph partitioning is NP-hard [20] and relies on heuristics in practice. Many state-of-the-art graph and hypergraph partitioners utilize the multilevel approach [8]. In multilevel methods, the original problem is iteratively coarsened by creating a hierarchy of smaller problems, until it becomes small enough to be solved. Then the coarsest problem is solved and the solution is iteratively projected onto finer levels and refined. Multilevel algorithms



© Ruslan Shaydulin and Ilya Safro;

licensed under Creative Commons License CC-BY

17th International Symposium on Experimental Algorithms (SEA 2018).

Editor: Gianlorenzo D'Angelo; Article No. 2; pp. 2:1–2:15

Leibniz International Proceedings in Informatics



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

for HGP are typically generalizations of multilevel algorithms for graph partitioning, those in turn drawing inspiration from multigrid and other multiscale optimization techniques [6]. Hypergraph partitioning is less well-studied than graph partitioning [13] and there is a relative lack of advanced coarsening schemes compared to GP.

Our *main contribution* are two novel aggregative coarsening schemes for HGP that are inspired by algebraic multigrid and stable matching methods. We expand and build on the insights from an unfinished attempt to build a coarsening scheme for HGP using algebraic multigrid ideas, published in Sandia Labs Summer Reports [7]. At each coarsening level we split the set of vertices into the set of seeds and the set of non-seeds. Each seed becomes a center of an aggregate which will, in turn, create a node at the next, coarser level. Aggregation rules are established to specify which aggregate a non-seed joins. We investigate two approaches to establishing aggregation rules. One approach is algebraic multigrid-based generalization of an inner-product matching similar to the matching scheme used in Zoltan[11] and PaToH[10]. Another approach is inspired by stable matching. Both approaches take advantage of the algebraic distance on hypergraphs when making coarsening decisions. Algebraic distance is a vertex similarity measure that extends simple measures such as hyperedge weights to better capture the structure of the hypergraphs [50]. While we outperform existing solvers on many instances, it is clear that final performance of HGP solvers heavily depends on the refinement. It is not the goal of this paper to outperform all existing HGP solvers. Instead, we would like to demonstrate that given similar uncoarsening schemes, the proposed coarsening schemes are at least as beneficial as traditional matching-based approaches.

2 Preliminaries

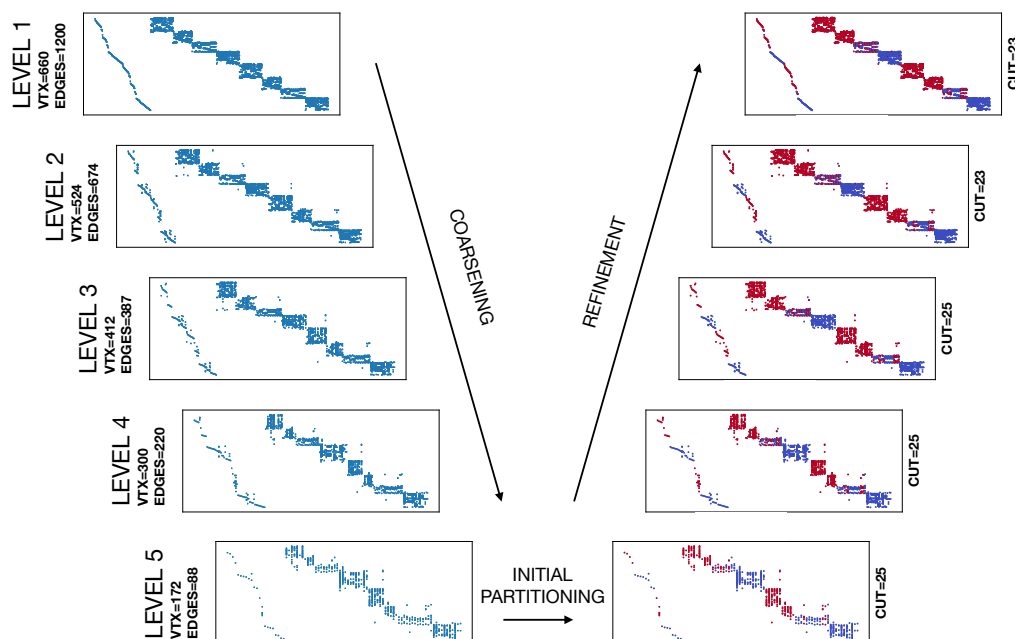
A hypergraph is an ordered pair of sets (V, E) , where V is the set of vertices and E is the set of hyperedges. Each hyperedge $e \in E$ is a nonempty subset of V . In this paper we make use of a graph representation of a hypergraph called “star expansion”. Star expansion graph (V', E') of a hypergraph (V, E) is an undirected bipartite graph with hypergraph vertices V on one side, hyperedges E on another and edges connecting hyperedges with the vertices they contain. Concretely, $V' = V \cup E$, $E' = \{(v, e) \mid e \in E, v \in e \subset V\}$. We will be referring to hyperedges as simply edges where it does not cause confusion. Both vertices and edges of the hypergraph are positively weighted. By $w(v)$ and $w(e)$, we denote weighting functions for nodes and edges, where $v \in V$ and $e \in E$. For both nodes and edges, a weight of zero practically means that corresponding nodes or edges do not exist (or do not affect the optimization and solution).

2.1 Hypergraph partitioning

In hypergraph k -partitioning the goal is to split the set of vertices V into k disjoint subsets (V_1, \dots, V_k) such that a metric on the cut is minimized subject to imbalance constraint. Here the cut is defined as the set of edges that span more than one partition, i.e.,

$$E_{\text{cut}} = \{e \in E \mid \exists i \neq j \text{ and } k \neq l \text{ for which } v_i, v_j \in e, v_i \in V_k \text{ and } v_j \in V_l\}. \quad (1)$$

There are multiple ways to define imbalance constraint. We will follow the definition used by the developers of state-of-the-art hypergraph partitioner Zoltan [15]. The imbalance is therefore defined as the ratio between the total weight of vertices in the largest partition



■ **Figure 1** Multilevel partitioning of a hypergraph constructed from LPnetlib/lp_scfxm2 matrix from SuiteSparse Matrix Collection [14] using row-net model: each column becomes a vertex and each row becomes a hyperedge. On the left side of the “V” the hypergraph (represented here as the sparsity pattern of the underlying matrix) is iteratively coarsened. At the bottom of the “V” the hypergraph is partitioned into two parts. This is represented by coloring the columns corresponding to vertices from one part into blue and another into red. On the right side of the “V” the hypergraph is uncoarsened and the partitioning is refined.

and the average sum of weights of vertices over all partitions. We define the imbalance as

$$imbal = \frac{\sum_{v \in V_{max}} w(v)}{\frac{1}{k} \sum_{v \in V} w(v)}, \quad (2)$$

where V_{max} is the largest partition by weight (i.e., $\sum_{v \in V_{max}} w(v) = \max_i (\sum_{v \in V_i} w(v))$). Imbalance constraint imposes a limit on the value of $imbal$, e.g., imbalance of 5% means $imbal < 1.05$. The cut metric used in this paper is simply total weight of the cut edges, namely, $\sum_{e \in E_{cut}} w(e)$.

2.2 Multilevel method

The main objective of multilevel methods is to construct a hierarchy of problems, each approximating the original problem but with fewer degrees of freedom. This is achieved by introducing a chain of successive restrictions of the problem domain into low-dimensional or smaller-size domains (coarsening) and solving the coarse problems in them using local processing (uncoarsening) [41]. The coarsening-uncoarsening pipeline is often referred to as V-cycle. The multilevel frameworks combine solutions obtained by the local processing at different levels of coarseness into one global solution. Typically, for combinatorial optimization problems, the multilevel algorithms are suboptimal metaheuristics [53] that incorporate other methods as refinement at all levels of coarseness. Except partitioning, examples can be found in linear ordering [27, 43, 44, 46], clustering and community detection [42, 5], and traveling

salesman problems [54]. In (H)GP, these algorithms were initially introduced to speed up existing algorithms [4] but later proved to improve the quality of the solution [24, 31]. A multilevel hypergraph partitioning of a hypergraph constructed from LPnetlib/lp_scfxm2 matrix is presented in Figure 1.

During the coarsening stage, for a hypergraph $H = (V, E)$ a hierarchy of decreasing in size hypergraphs $H^0 = (V^0, E^0), \dots, H^l = (V^l, E^l)$ is constructed. Here l denotes the number of levels in multilevel hierarchy. During the initial partitioning stage, the coarsest hypergraph $H^l = (V^l, E^l)$ is partitioned. Finally, during the refinement stage the solution from coarser levels is projected onto finer ones and refined, typically using a local search heuristic.

2.3 Algebraic distance

Algebraic distance for hypergraphs is a relaxation-based vertex similarity measure [50]. It extends the algebraic distance for graphs [12, 41, 29] by taking into account the non-pairwise nature of the connections between vertices in a hypergraph. Algebraic distance improves on simpler similarity metrics, such as hyperedge weights, by incorporating information about more distant vertex neighborhood, thus better capturing vertex's place in the global structure of the hypergraph. Algebraic distance is inspired by iterative techniques for solving linear systems. An iterative method can be represented in a standard form:

$$x^{(i)} = Hx^{(i-1)} \quad i = 1, 2, 3 \dots \quad (3)$$

where H is the iteration matrix.

Similarly, algebraic distances are computed at each coarsening level using the following stationary iterative relaxation. Let $A \in \mathbb{R}^{|E| \times |V|}$ be hypergraph incidence matrix, i.e., $A_{ij} = 1$ if the hyperedge i contains the vertex j . Let $S^v \in \mathbb{R}^{|V| \times |V|}$ and $S^h \in \mathbb{R}^{|E| \times |E|}$ be diagonal matrices such that

$$S_{jj}^v = w(v_j) \quad \text{and} \quad S_{ii}^h = \frac{w(h_i)}{|h_i|}, \quad (4)$$

where $|h_i|$ denotes the cardinality of the i th hyperedge. Denote

$$W = \begin{bmatrix} 0 & A^T S^h \\ A S^v & 0 \end{bmatrix} \quad (5)$$

and let D be the diagonal matrix with elements $D_{jj} = \sum_i W_{ij}$. Then the iterative step is defined as

$$x^{(i)} = \frac{1}{r-l} \left[\underbrace{\omega D^{-1} W x^{(i-1)} + (1-\omega)x^{(i-1)}}_{x^{*(i-1)}} \right] - \frac{r+l}{2(r-l)} \mathbf{1} \quad (6)$$

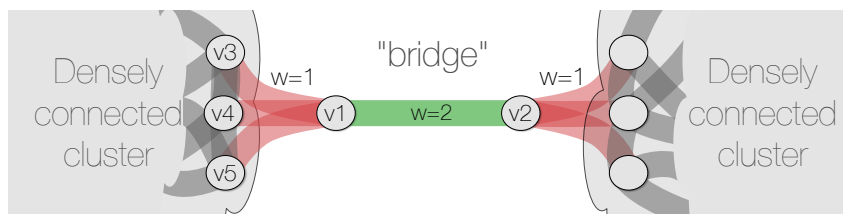
where $\mathbf{1}$ is the vector of all ones, and r and l are the maximum and the minimum of the elements in $x^{*(i-1)}$, respectively. We can simplify the update formula as

$$x^{(i)} = \alpha^{(i-1)} H x^{(i-1)} + \beta^{(i-1)} \mathbf{1}, \quad (7)$$

where

$$H = \omega D^{-1} W + (1-\omega)I, \quad \alpha^{(i-1)} = \frac{1}{r-l}, \quad \text{and} \quad \beta^{(i-1)} = -\frac{r+l}{2(r-l)}. \quad (8)$$

The iterative scheme is performed multiple times for different random initial values $x_0^{(0)}, \dots, x_R^{(0)}$ (called test vectors in algebraic multigrid [34]). Then, the algebraic distance between vertices i and j is set to be the maximum over R random initializations namely, $\text{algdist}_{ij} = \max_R |x_i - x_j|$. For the detailed discussion of algebraic distances on hypergraphs and for convergence analysis of the described iterative scheme the reader is referred to [50].



■ **Figure 2** An example demonstrating the limitations of schemes based on edge weights. Here the “bridge” edge (green) connecting $v1$ and $v2$ has weight two and all other edges (red) have unitary weights. The best cut is achieved by cutting the green “bridge” between $v1$ and $v2$ and therefore matching $v1$ with one of the vertices on the left ($v3, v4$ or $v5$). However, matching schemes based on edge weights can match $v1$ with $v2$ instead and increase the cut.

3 Related work

Practical approaches to solving HGP typically rely on heuristics. Many have been developed over the years, but the most common approach is multilevel. It is implemented in many state-of-the-art hypergraph partitioners including but not limited to Zoltan [15], hMetis [30], KaHyPar [49] and PaToH [10]. In this section we will briefly describe the multilevel approach used by those state-of-the-art partitioners and discuss existing advanced coarsening schemes for hypergraphs. For a more detailed review of hypergraph partitioning, the reader is referred to [2, 3, 39, 52].

3.1 Brief overview of multilevel hypergraph partitioning

The HGP multilevel frameworks consist of three stages: coarsening, initial partitioning and uncoarsening with refinement. During the coarsening, the hypergraph is approximated via a series of decreasing in size hypergraphs. At each coarsening step, the next hypergraph is formed by matching a group of vertices into one, such that a set of vertices at level k becomes one vertex at level $k + 1$. The decision as to which vertices to match is made based on similarity metrics such as inner product (i.e., the total weight of hyperedges connecting two vertices). However, simple metrics often result in a decision that ignores the structure of the hypergraph. Consider the example in Figure 2. It shows two densely connected clusters, separated by a “bridge” between vertices $v1$ and $v2$. Schemes that use naive similarity metrics like hyperedge weights might match $v1$ with $v2$, whereas an algorithm that considers larger neighborhoods to minimize a cut would prefer to match $v1$ with either $v3, v4$ or $v5$ instead. This example demonstrates the challenges of capturing the hypergraph structure by using only local information. In the refinement stage all the aforementioned state-of-the-art partitioners use a combination of Fiduccia-Mattheyses [18] or Kernighan-Lin [32] with the exception of KaHyPar, which uses a novel localized adaptive local search heuristic [1].

3.2 Aggregative coarsening

The standard approach to coarsening used in most state-of-the-art hypergraph partitioners is matching-based. Originally, this meant that at each level pairs of adjacent vertices are selected to become one vertex at the next level. This technique has later been extended to include non-pairwise matchings (i.e., more than two fine vertices can form a coarse vertex). One of the alternative approaches is aggregative coarsening inspired by algebraic multigrid. In aggregative coarsening, the set of vertices V is separated into disjoint sets of seeds and non-seeds, namely, C and F such that $F \cup C = V$. The non-seed vertices aggregate themselves

around the seeds (hence the name aggregative coarsening). The aggregation can be strict (F -vertices are not split) and weighted (F -vertices can be split between multiple seeds with vertex weight conservation). At the refinement stage, the partitioning decision (i.e., partition assignment) is interpolated from each seed to the non-seeds in its aggregate. This separation between seed and non-seeds helps to introduce additional guarantees. For example, on graphs Safro et al. [45] introduce the notion of strong connection and guarantee that each vertex in the graph is strongly connected to at least one seed. The weighted aggregation was initially introduced for several cut problems on graphs [48, 41, 46] including GP [47]. There was an unfinished attempt to extend this approach to hypergraphs. Buluç and Boman [7] describe several challenges in applying aggregative coarsening to hypergraphs, as well as propose two very similar coarsening schemes, strict and weighted. In this paper, we limit our discussion to strict aggregation.

In aggregative coarsening, two main questions have to be addressed: seed selection and aggregation of non-seeds around seeds. In the process of seed selection, Buluç and Boman [7] follow Safro et al. [45] in using the concept of *future volumes*. Future volume is a measure of how many vertices a seed can incorporate into itself (in other words, how large a vertex can grow). They propose computing future volumes on the star expansion of the hypergraph (thus limiting the complexity), then iteratively adding vertices with high future volumes to the set of seeds C until $|C|$ reaches a certain threshold. Aggregation rules are established on the star expansion of the hypergraph. Seeds and non-seeds select a constant number of adjacent hyperedges to “invade” based on the exclusive coarseness (a metric indicating how many seeds an hyperedge contains).

4 Two Aggregation Algorithms

Our algorithm combines the ideas of aggregative coarsening described in [45] and [7] with the algebraic distance [41, 50]. Aggregative coarsening is a two-step process, so we have to address both the seed selection and the rules of aggregation. At each coarsening level, a set of seeds is selected and each seed is assigned a set of non-seeds to form a cluster. The cluster at a given coarsening level becomes one vertex at the next level.

Both introduced schemes utilize algebraic distances by augmenting hyperedge weights with algebraic weights. We define the algebraic weight of hyperedge e as an inverse of the algebraic distance between two farthest apart vertices in e , i.e.,

$$\rho(e) = 1 / \max_{i,j \in e} \text{algdist}_{ij}. \quad (9)$$

4.1 Seed selection

For the seed selection we utilize two core concepts: *future volumes* and *strong connection*. The main goal is to construct a set of seeds C such that every vertex in the graph is *strongly connected* to C . We define *strong connection* as follows: the vertex $i \in F$ is *strongly connected* to C if the sum of algebraic weights of the edges connecting it to C is more than a certain fraction of the total algebraic weight of incident edges:

$$i \text{ is strongly connected to } C \iff \frac{\sum_{j \in C} \rho(e_{ij})}{\sum_j \rho(e_{ij})} > Q, \quad (10)$$

where Q is a parameter (in our experiments $Q = 0.5$). The *future volume* of a vertex is a measure of how large an aggregate seeded by it can grow. Intuitively, we want to add the vertices with very high volume (or the ones that might become centers of the aggregates of

■ **Listing 1** Seed selection.

```

for i in V:
    fv[i] = w[i] +  $\sum_j w[j] (w[e_{ij}] / \sum_k w[e_{jk}])$ 
for i in V:
    if fv[i] > mean(fv) + 2 * stdev(fv):
        C.insert(i)
    else:
        F.remove(i)
for i in F:
    fv[i] = w[i] +  $\sum_{j \in F} w[j] (w[e_{ij}] / \sum_{k \in F} w[e_{jk}])$ 

for i in sort_indices(fv):
    if  $\sum_{j \in C} w[e_{ij}] / \sum_j w[e_{ij}] < Q$ :
        C.insert(i)
        F.remove(i)

```

very high volume) to the set of seeds. *Future volume* of a vertex is defined as follows (note that here we use the hyperedge weights w and not the algebraic weights ρ):

$$fv(i) = w(i) + \sum_j w(j) \frac{w(e_{ij})}{\sum_k w(e_{jk})}. \quad (11)$$

We begin the construction of set C by computing future volumes for all vertices. Then, we initialize C with vertices with large future volumes (if mean future volume is m_{fv} and standard deviation of the distribution of future volumes is σ_{fv} , then $i \in C \iff fv(i) > m_{fv} + 2\sigma_{fv}$) and initialize F with all other vertices, such that $F \cup C = V$. After that the future volumes of vertices in F are recomputed, only taking into account connections with other vertices in F (i.e., in Equation (11) assume $w(e_{ij}) = 0$ if $j \in C$ or $i \in C$). Finally, vertices in F are visited in order of decreasing future volume and added to the set C if they are not strongly connected to C . Note that at the end of this process each vertex in V is strongly connected to the set C and $F \cup C = V$. Pseudocode for this procedure is presented in Listing 1.

4.2 Aggregation

We investigate two approaches to establishing the rules of aggregation. First approach is a scheme similar to inner-product matching used in Zoltan[11] and PaToH[10] but applied in algebraic multigrid setting. Second approach consists of computing a stable assignment [23] between vertices of C and F . Both approaches take advantage of algebraic distances as a similarity measure when establishing aggregation rules.

Inner-product aggregation proceeds by visiting the non-seed vertices in the random order. For each unmatched vertex $v \in F$, a neighboring seed $u \in C$ with the highest inner product is selected and v is added to the cluster C_u seeded by it. The inner product is defined as the total algebraic weight of the edges connecting v with the seed u . Concretely, $ipm(v, u) = \sum_{e|v, u \in e} \rho(e)$. See Listing 2 for pseudocode. We experimented with visiting the non-seeds in order of decreasing future volume and with using connectivity to make decisions when establishing aggregation rules. These approaches are more computationally intensive and do not produce better results (see Appendix B of full version [51] for the comparison of different parameters).

Stable assignment aggregation begins by constructing preference lists. Each seed orders adjacent non-seeds in the order of decreasing total algebraic weight of the hyperedges

■ **Listing 2** Inner-product aggregation.

```
for i in F:
    j = argmaxu∈C ipm(v,u)
    Cj.insert(i)
```

■ **Listing 3** Stable matching aggregation.

```
def propose(i):
    for j in pref_list[i]:
        if waitlist[i].size > threshold:
            return
        if propos[j] == -1: // j holds no proposal
            propos[j] = i
            waitlist[i].push_back(j)
            continue
        if i is preferable to propos[j]:
            rejected = propos[j]
            propos[j] = i
            waitlist[i].push_back(j)
            propose(rejected)

// Step 1: compute preference lists
for i in F:
    for j in seed_neighbors[i]:
        pref[j] = Σρ(eij) // pref is a hashtable
    for j in sort_by_value(pref).keys():
        pref_list[i].push_back(j)
for i in C:
    for j in non_seed_neighbors[i]:
        pref[j] = Σρ(eij)
    for j in sort_by_value(pref).keys():
        pref_list[i].push_back(j)

// Step 2: compute stable assignment
for i in C:
    propose(i)
```

connecting them (and vice versa): $\text{pref}_i(j) = \Sigma\rho(e_{ij})$. Then the stable assignment is computed using an algorithm similar to the classical one described in [19]. Each seed in C proposes to non-seeds in its preference list. If the non-seed does not have a better offer, it tentatively accepts the proposal and is put on the waitlist. If that non-seed later receives a better offer (i.e., an offer from a seed that ranks higher on its preference list), it rejects the current offer and the rejected seed proposes to the next candidate on its preference list. To discourage the creation of very large clusters, we limit the size of waitlist for a seed to the maximal vertex weight on a given coarsening level times three plus ten: $\text{len}(\text{waitlist}) = 3 \times \text{max_vtx_wgt} + 10$. Procedure terminates when each non-seed has been assigned to a waitlist or a seed has been rejected by every non-seed. At this point each seed forms a cluster with all vertices on its waitlist, subject to size constraint (we guarantee that no cluster can be larger than total vertex weight over the number of parts). The fact that we use a classical problem as a subproblem in our heuristic allows us to potentially leverage the previous work in optimizing and parallelizing stable assignment, such as [35],[36] and [22]. The pseudocode is presented in Listing 3.

5 Results

We implemented all algorithms described in this paper within Zoltan [15] package of the Trilinos Project [26]. Zoltan is an open-source toolkit of parallel combinatorial scientific computing algorithms [15]. It includes a hypergraph partitioning algorithm PHG (Parallel HyperGraph partitioner) and interfaces to PaToH and hMetis2. We added our new coarsening schemes and left other phases of the multilevel framework unchanged. Our implementation, data, and full results are available at <http://bit.ly/aggregative2018code>.

The hypergraphs in our benchmark are generated from a selection of matrices using the row-net model. In the row-net model, each column of the matrix represents a vertex, each row represents an edge and a vertex j belongs to the hyperedge i if there is a non-zero element at the intersection of j -th column and i -th row, i.e., $A_{ij} \neq 0$. All matrices (more than 300) were obtained from SuiteSparse Matrix Collection [14] that includes other collections. For each combination of hypergraph/algorithm/set of parameters, we executed 20 experiments.

We compare our algorithm with four state-of-the-art partitioners: hMetis2 [30], PaToH v3.2 [9], Zoltan PHG [15] and Zoltan-AlgD [50]. PaToH is used as a plug-in for Zoltan with default parameters described in Zoltan's User Guide [17]. hMetis2 is used in k -way mode with all parameters set to default: greedy first-choice scheme for coarsening, random k -way refinement, and min-cut objective function. The reason we run hMetis in k -way mode is the way hMetis specifies imbalance constraint. In recursive bisection mode, the imbalance constraint is applied *at each bisection step*, therefore relaxing the constraint as the number of parts increases. We found it almost impossible to compare hMetis in recursive bisection mode fairly (i.e., with the same imbalance) with other partitioners. Both Zoltan and PaToH are used in serial mode.

Optimizing the constants in the running time of the proposed algorithms is beyond the scope of this paper. Currently, for the existing unoptimized implementation the running time of other state-of-the-art hypergraph partitioners is not improved except for those that generate less levels in the hierarchy. In the experiments, the runtime of unoptimized implementation of our algorithms is up to an order of magnitude larger than the runtime of other state-of-the-art partitioners in worst cases. However, we must point out that our algorithm utilizes the building blocks and ideas of algebraic multigrid, which makes it possible to improve the runtime drastically by leveraging a plethora of existing research in optimizing and parallelizing algebraic multigrid solvers (e.g. [25], [40]). Similarly, there exists extensive research into optimizing the performance of stable matching solvers. Manne et al. [36] demonstrate the connection between graph matchings and stable marriage and show the scalability of Gale-Shapely type algorithms. Munera et al. [38] present an adaptive search formulation of stable marriage problem and take advantage of a Cooperative Parallel Local Search framework [37], achieving superlinear speedup. Gelain et al. [21] demonstrate a different efficient local search method for stable marriage problem.

In Figures 3 and 4 the results are presented graphically. In the main body of the paper, we only plot the results for 10% imbalance. For results for other imbalance factors please refer to Appendix A of the full version [51]. In Figure 3, we show the results of inner-product algebraic multigrid aggregation coarsening. In Figure 4, the stable matching aggregation is demonstrated. We use frequency histograms to present the distribution of cut differences between our methods and other state-of-the-art hypergraph partitioners. The value being represented (see horizontal axes) is the ratio

$$\zeta = \frac{\text{cut obtained using another partitioner}}{\text{cut obtained using our method}}. \quad (12)$$

2:10 Aggregative Coarsening for Multilevel Hypergraph Partitioning

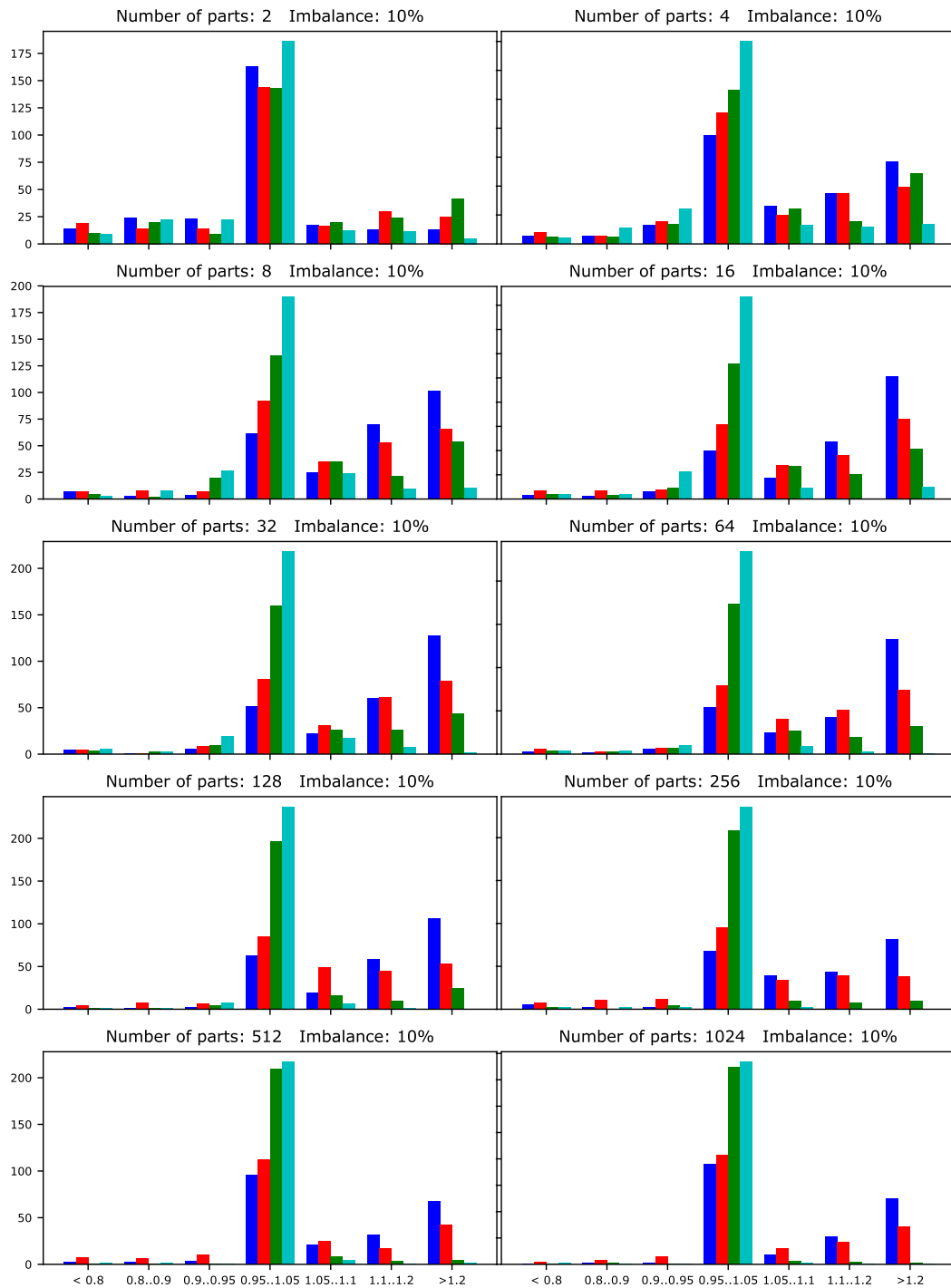
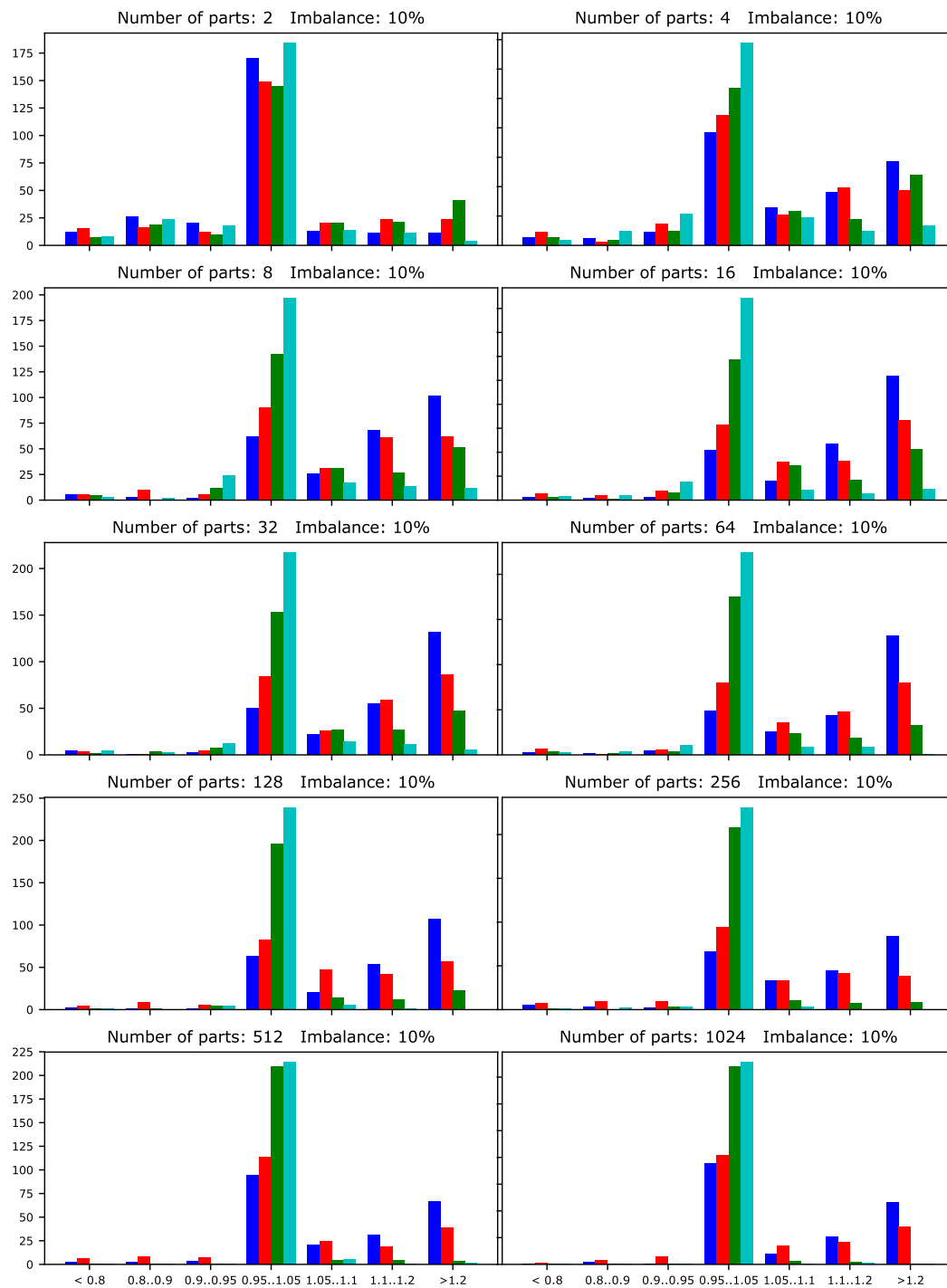


Figure 3 Histogram of ζ for coarsening using algebraic multigrid inner-product aggregation. Blue rectangle corresponds to PaToH, red to hMetis2, green to Zoltan PHG and cyan to Zoltan-AlgD.



■ **Figure 4** Histogram of ζ for coarsening using stable matching aggregation. Blue rectangle corresponds to PaToH, red to hMetis2, green to Zoltan PHG and cyan to Zoltan-AlgD.

Each bin corresponds to a range of the ratios (for example, the middle bin corresponds to the differences of less than $\pm 5\%$ and the rightmost to the improvements of $> 20\%$). Each rectangle corresponds to a partitioner: blue corresponds to PaToH, red corresponds to hMetis2, green corresponds to Zoltan PHG and cyan corresponds to Zoltan-AlgD. For the full results, please refer to <http://bit.ly/aggregative2018results>

The results demonstrate that given the same refinement, the proposed schemes are at least as effective as traditional matching-based schemes, while outperforming them on many instances. Both proposed coarsening schemes almost equally succeed in improving the quality of solvers (see Appendix A of the full version [51] for comparison of the performance of two algorithms). Further investigation of the difference between them is a very interesting future research direction, because, in fact, they represent very different algorithms. Since Zoltan utilizes recursive bisectioning scheme, we can see that improvements decrease as number of parts increases. This can be attributed to refinement becoming more and more important as number of parts increases.

6 Conclusion

We have presented two novel aggregative coarsening schemes for hypergraphs. The introduced schemes incorporate ideas of algebraic multigrid and stable matching into multilevel hypergraph partitioning framework. We have implemented the described algorithms within state-of-the-art hypergraph partitioner Zoltan and compared their performance against a number of other state-of-the-art partitioners on a large benchmark.

The experimental results demonstrate that given the same uncoarsening, the proposed coarsening schemes perform at least as well as traditional matching-based schemes, while outperforming them on many instances. This suggests that algebraic-multigrid-inspired coarsening schemes have great potential when combined with appropriate refinement.

References

- 1 Yaroslav Akhremtsev, Tobias Heuer, Peter Sanders, and Sebastian Schlag. Engineering a direct k -way hypergraph partitioning algorithm. In *19th Workshop on Algorithm Engineering and Experiments, (ALENEX 2017)*, pages 28–42, 2017.
- 2 Charles J Alpert and Andrew B Kahng. Recent directions in netlist partitioning: a survey. *Integration, the VLSI journal*, 19(1-2):1–81, 1995.
- 3 David A Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. Graph partitioning and graph clustering: 10th dimacs implementation challenge, vol. 588. *American Mathematical Society*, 7:210–223, 2013.
- 4 Stephen T Barnard and Horst D Simon. Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Concurrency and computation: Practice and Experience*, 6(2):101–117, 1994.
- 5 V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and R. Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 10:P10008, 2008.
- 6 Achi Brandt and Dorit Ron. Multigrid solvers and multilevel optimization strategies. In *Multilevel optimization in VLSICAD*, pages 1–69. Springer, 2003.
- 7 Aydin Buluç and Erik G Boman. Towards scalable parallel hypergraph partitioning. *CSRI Summer Proceedings 2008*, page 109, 2008.
- 8 Aydin Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in graph partitioning. In *Algorithm Engineering: Selected Results and Surveys*, volume 9220, pages 117–158. Springer, 2016.

- 9 Ümit Çatalyürek and Cevdet Aykanat. Patch (partitioning tool for hypergraphs). In *Encyclopedia of Parallel Computing*, pages 1479–1487. Springer, 2011.
- 10 Umit V Catalyurek and Cevdet Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Trans. Parallel Distrib. Syst.*, 10(7):673–693, 1999.
- 11 Umit V Catalyurek, Erik G Boman, Karen D Devine, Doruk Bozdağ, Robert T Heaphy, and Lee Ann Riesen. A repartitioning hypergraph model for dynamic load balancing. *J. Parallel Distrib. Comput.*, 69(8):711–724, 2009.
- 12 Jie Chen and Ilya Safro. Algebraic distance on graphs. *SIAM J. Sci. Comput.*, 33(6):3468–3490, 2011.
- 13 Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment*, 3(1-2):48–57, 2010.
- 14 Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.
- 15 Karen D Devine, Erik G Boman, Robert T Heaphy, Rob H Bisseling, and Umit V Catalyurek. Parallel hypergraph partitioning for scientific computing. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10–pp. IEEE, 2006.
- 16 Karen D Devine, Erik G Boman, Robert T Heaphy, Bruce A Hendrickson, James D Teresco, Jamal Faik, Joseph E Flaherty, and Luis G Gervasio. New challenges in dynamic load balancing. *Applied Numerical Mathematics*, 52(2-3):133–152, 2005.
- 17 Karen D Devine, Vitus Leung, Erik G Boman, Sivasankaran Rajamanickam, Lee Ann Riesen, and Umit Catalyurek. Zoltan user’s guide, version 3.8.(2014), 2014. URL: http://www.cs.sandia.gov/zoltan/ug_html/ug_alg_patch.html.
- 18 Charles M Fiduccia and Robert M Mattheyses. A linear-time heuristic for improving network partitions. In *Papers on Twenty-five years of electronic design automation*, pages 241–247. ACM, 1988.
- 19 David Gale and Lloyd S Shapley. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962.
- 20 Michael R Garey and David S Johnson. *Computers and intractability*, volume 29. New York, 2002.
- 21 Mirco Gelain, Maria Silvia Pini, Francesca Rossi, K Brent Venable, and Toby Walsh. Local search approaches in stable matching problems. *Algorithms*, 6(4):591–617, 2013.
- 22 Giorgos Georgiadis and Marina Papatriantafyllou. Overlays with preferences: Distributed, adaptive approximation algorithms for matching with preference lists. *Algorithms*, 6(4):824–856, 2013.
- 23 Dan Gusfield and Robert W Irving. *The stable marriage problem: structure and algorithms*. MIT press, 1989.
- 24 Bruce Hendrickson and Robert W Leland. A multi-level algorithm for partitioning graphs. *SC*, 95(28), 1995.
- 25 Van Emden Henson and Ulrike Meier Yang. Boomerang: a parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics*, 41(1):155–177, 2002.
- 26 Michael A Heroux, Roscoe A Bartlett, Vicki E Howle, Robert J Hoekstra, Jonathan J Hu, Tamara G Kolda, Richard B Lehoucq, Kevin R Long, Roger P Pawlowski, Eric T Phipps, et al. An overview of the trilinos project. *ACM Trans. Math. Software*, 31(3):397–423, 2005.
- 27 Y. F. Hu and J. A. Scott. A multilevel algorithm for wavefront reduction. *SIAM J. Sci. Comput.*, 23(4):1352–1375, 2001. doi:10.1137/S1064827500377733.

- 28 Ruoming Jin, Yang Xiang, David Fuhry, and Feodor F. Dragan. Overlapping matrix pattern visualization: A hypergraph approach. *Data Mining, IEEE International Conference on*, 0:313–322, 2008. doi:10.1109/ICDM.2008.102.
- 29 Emmanuel John and Ilya Safro. Single-and multi-level network sparsification by algebraic distance. *Journal of Complex Networks*, 5(3):352–388, 2016.
- 30 George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel hypergraph partitioning: applications in vlsi domain. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 7(1):69–79, 1999.
- 31 George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.
- 32 Brian W Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *Bell Syst. Tech. J.*, 49(2):291–307, 1970.
- 33 K Ashwin Kumar, Abdul Quamar, Amol Deshpande, and Samir Khuller. Sword: workload-aware data placement and replica selection for cloud data management systems. *The VLDB Journal*, 23(6):845–870, 2014.
- 34 Oren E Livne and Achi Brandt. Lean algebraic multigrid (lamg): Fast graph laplacian linear solver. *SIAM Journal on Scientific Computing*, 34(4):B499–B522, 2012.
- 35 Enyue Lu and SQ Zheng. A parallel iterative improvement stable matching algorithm. In *International Conference on High-Performance Computing*, pages 55–65. Springer, 2003.
- 36 Fredrik Manne, Md. Naim, Håkon Lerring, and Mahantesh Halappanavar. *On Stable Marriages and Greedy Matchings*, pages 92–101. SIAM, 2016. doi:10.1137/1.9781611974690.ch10.
- 37 Danny Munera, Daniel Diaz, Salvador Abreu, and Philippe Codognet. A parametric framework for cooperative parallel local search. In *European Conference on Evolutionary Computation in Combinatorial Optimization*, pages 13–24. Springer, 2014.
- 38 Danny Munera, Daniel Diaz, Salvador Abreu, Francesca Rossi, Vijay A Saraswat, and Philippe Codognet. Solving hard stable matching problems via local search and cooperative parallelization. In *AAAI*, pages 1212–1218, 2015.
- 39 David A Papa and Igor L Markov. Hypergraph partitioning and clustering., 2007.
- 40 Jongsoo Park, Mikhail Smelyanskiy, Ulrike Meier Yang, Dheevatsa Mudigere, and Pradeep Dubey. High-performance algebraic multigrid solver optimized for multi-core based distributed parallel systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 54. ACM, 2015.
- 41 Dorit Ron, Ilya Safro, and Achi Brandt. Relaxation-based coarsening and multiscale graph organization. *Multiscale Modeling & Simulation*, 9(1):407–423, 2011.
- 42 Randolph Rotta and Andreas Noack. Multilevel local search algorithms for modularity clustering. *Journal of Experimental Algorithmics (JEA)*, 16:2–3, 2011.
- 43 I. Safro, D. Ron, and A. Brandt. Graph minimum linear arrangement by multilevel weighted edge contractions. *Journal of Algorithms*, 60(1):24–41, 2006.
- 44 I. Safro, D. Ron, and A. Brandt. Multilevel algorithm for the minimum 2-sum problem. *Journal of Graph Algorithms and Applications*, 10(2):237–258, 2006.
- 45 Ilya Safro, Dorit Ron, and Achi Brandt. Graph minimum linear arrangement by multilevel weighted edge contractions. *Journal of Algorithms*, 60(1):24–41, 2006.
- 46 Ilya Safro, Dorit Ron, and Achi Brandt. Multilevel algorithms for linear ordering problems. *ACM Journal of Experimental Algorithmics*, 13, 2008. doi:10.1145/1412228.1412232.
- 47 Ilya Safro, Peter Sanders, and Christian Schulz. Advanced coarsening schemes for graph partitioning. *ACM Journal of Experimental Algorithmics (JEA)*, 19:2–2, 2015.
- 48 Ilya Safro and Boris Temkin. Multiscale approach for the network compression-friendly ordering. *J. Discrete Algorithms*, 9(2):190–202, 2011. doi:10.1016/j.jda.2010.09.007.

- 49 Sebastian Schlag, Vitali Henne, Tobias Heuer, Henning Meyerhenke, Peter Sanders, and Christian Schulz. k -way hypergraph partitioning via n -level recursive bisection. In *18th Workshop on Algorithm Engineering and Experiments, (ALENEX 2016)*, pages 53–67, 2016.
- 50 Ruslan Shaydulin, Jie Chen, and Ilya Safro. Relaxation-based coarsening for multilevel hypergraph partitioning. *arXiv preprint arXiv:1710.06552*, 2017.
- 51 Ruslan Shaydulin and Ilya Safro. Aggregative coarsening for multilevel hypergraph partitioning. *CoRR*, abs/1802.09610, 2018. [arXiv:1802.09610](#).
- 52 Aleksandar Trifunovic. *Parallel algorithms for hypergraph partitioning*. PhD thesis, University of London, 2006.
- 53 C. Walshaw. Multilevel refinement for combinatorial optimisation problems. *Annals Oper. Res.*, 131:325–372, 2004.
- 54 Chris Walshaw. A multilevel approach to the travelling salesman problem. *Operations Research*, 50(5):862–877, 2002.
- 55 Dengyong Zhou, Jiayuan Huang, and Bernhard Schölkopf. Learning with hypergraphs: Clustering, classification, and embedding. In *NIPS*, volume 19, pages 1633–1640, 2006.

Memetic Graph Clustering

Sonja Biedermann

University of Vienna
Vienna, Austria
sonja.biedermann@univie.ac.at

Monika Henzinger

University of Vienna
Vienna, Austria
monika.henzinger@univie.ac.at

Christian Schulz

University of Vienna
Vienna, Austria
christian.schulz@univie.ac.at

Bernhard Schuster

University of Vienna
Vienna, Austria
bernhard.schuster@univie.ac.at

Abstract

It is common knowledge that there is no single best strategy for graph clustering, which justifies a plethora of existing approaches. In this paper, we present a general memetic algorithm, VieClus, to tackle the graph clustering problem. This algorithm can be adapted to optimize different objective functions. A key component of our contribution are natural recombine operators that employ ensemble clusterings as well as multi-level techniques. Lastly, we combine these techniques with a scalable communication protocol, producing a system that is able to compute high-quality solutions in a short amount of time. We instantiate our scheme with local search for modularity and show that our algorithm successfully improves or reproduces all entries of the 10th DIMACS implementation challenge under consideration using a small amount of time.

2012 ACM Subject Classification Information systems → Clustering, Theory of computation → Evolutionary algorithms

Keywords and phrases Graph Clustering, Evolutionary Algorithms

Digital Object Identifier 10.4230/LIPIcs.SEA.2018.3

Related Version A full version of the paper is available at [9], <https://arxiv.org/abs/1802.07034>.

Funding The research leading to these results has received funding from the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013) /ERC grant agreement No. 340506.

Acknowledgements The authors acknowledge support by the state of Baden-Württemberg through bwHPC.



© Sonja Biedermann, Monika Henzinger, Christian Schulz, and Bernhard Schuster; licensed under Creative Commons License CC-BY

17th International Symposium on Experimental Algorithms (SEA 2018).

Editor: Gianlorenzo D'Angelo; Article No. 3; pp. 3:1–3:15

Leibniz International Proceedings in Informatics



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

1 Introduction

Graph clustering is the problem of detecting tightly connected regions of a graph. Depending on the task, knowledge about the structure of the graph can reveal information such as voter behavior, the formation of new trends, existing terrorist groups and recruitment [42] or a natural partitioning of data records onto pages [17]. Further application areas include the study of protein interaction [35], gene expression networks [48], fraud detection [1], program optimization [29, 15] and the spread of epidemics [32] – possible applications are plentiful, as almost all systems containing interacting or coexisting entities can be modeled as a graph.

It is common knowledge that there is no single best strategy 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. [4, 12, 47], allowing only heuristic approaches for optimization. The majority of algorithms for graph clustering are based on the paradigm of intra-cluster density versus inter-cluster sparsity. One successful heuristic to cluster large graphs is the *multi-level* approach [11], e.g. the Louvain method for the optimization of modularity [10]. Here, the graph is recursively *contracted* to obtain smaller graphs which should reflect the same general structure as the input. After applying an *initial clustering* algorithm to the smallest graph, the contraction steps are undone and, at each level, a *local search* method is used to improve the clustering induced by the coarser level w.r.t some objective function measuring the quality of the clustering. The intuition behind this approach is that a good clustering at one level of the hierarchy will also be a good clustering on the next finer level. Hence, depending on the definition of the neighborhood, local search algorithms are able to explore local solution spaces very effectively in this setting. However, these methods are also prone to get trapped in local optima. The multi-level scheme can help to some extent since local search has a more global view on the problem on the coarse levels and a very fine-grained view on the fine levels of the multi-level hierarchy. In addition, as with many other randomized meta-heuristics, several repeated runs can be made in order to improve the final result at the expense of running time.

Still, even a large number of repeated executions can only scratch the surface of the huge search space of possible clusterings. In order to explore the global solution space extensively, we need more sophisticated meta-heuristics. This is where memetic algorithms (MAs), i.e. genetic algorithms combined with local search [25], come into play. Memetic algorithms allow for effective exploration (global search) and exploitation (local search) of the solution space. The general idea behind genetic algorithms is to use mechanisms inspired by biological evolution such as selection, mutation, recombination, and survival of the fittest. A genetic algorithm (GA) starts with a population of individuals (in our case clusterings of the graph) and evolves the population over several generational cycles (rounds). In each round, the GA uses a selection rule to select good individuals and combines them to obtain improved offspring [21]. When an offspring is generated an eviction rule is used to select a member of the population to be replaced by the new offspring. For an evolutionary algorithm it is of major importance to preserve diversity in the population [5], i.e., the individuals should not become too similar in order to avoid a premature convergence of the algorithm. This is usually achieved by using mutation operations and by using eviction rules that take similarity of individuals into account.

In this paper, we present a general memetic algorithm, VieClus (Vienna Graph Clustering), to tackle the graph clustering problem. This algorithm can be adapted to optimize different objective functions simply by using a local search algorithm that optimizes the objective

function desired by the user. A key component of our contribution are natural recombine operators that employ ensemble clusterings as well as multi-level techniques. In machine learning, ensemble methods combine multiple weak classification (or clustering) algorithms to obtain a strong algorithm for classification (or clustering). More precisely, given a number of clusterings, the *overlay/ensemble clustering* is a clustering in which two vertices belong to the same cluster if and only if they belong to the same cluster in each of the input clusterings. Our recombination operators use the overlay of two clusterings from the population to decide whether pairs of vertices should belong to the same cluster [34, 43]. This is combined with a local search algorithm to find further improvements and also embedded into a multi-level algorithm to find even better clusterings. Our general principle is to randomize tie-breaking whenever possible. This diversifies the search and also improves solutions. Lastly, we combine these techniques with a scalable communication protocol, producing a system that is able to compute high-quality solutions in a short amount of time. In our experimental evaluation, we show that our algorithm successfully improves or reproduces all entries of the 10th DIMACS implementation challenge under consideration in a small amount of time. In fact, for most of the small instances, we can improve the old benchmark result *in less than a minute*. Moreover, while the previous best result for different instances has been computed by a variety of solvers, our algorithm can now be used as a single tool to compute the result.

2 Preliminaries

2.1 Basic Concepts

Let $G = (V = \{0, \dots, n-1\}, E)$ be an undirected graph and $N(v) := \{u : \{v, u\} \in E\}$ denote the neighbors of v . The degree of a vertex v is $d(v) := |N(v)|$. The problem that we tackle in this paper is the *graph clustering* problem. A clustering \mathcal{C} is a partition of the set of vertices, i.e. a set of disjoint *blocks/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. The term $\mathcal{C}[v]$ refers to the cluster of a node v . A size-constrained clustering constrains the size of the blocks of a clustering by a given upper bound U . A clustering is *trivial* if there is only one block, or all clusters/blocks contain only one element, i.e., are singletons. We identify a cluster V_i with its node-induced subgraph of G . The set $E(\mathcal{C}) := E \cap (\cup_i V_i \times V_i)$ is the set of *intra-cluster edges*, and $E \setminus E(\mathcal{C})$ is the set of *inter-cluster edges*. We set $|E(\mathcal{C})| =: m(\mathcal{C})$ and $|E \setminus E(\mathcal{C})| =: \bar{m}(\mathcal{C})$. An edge running between two blocks is called *cut edge*. There are different objective functions that are optimized in the literature. We review some of them in Section 2.3. Our main focus in this work is on *modularity*. However, our algorithm can be generalized to optimize other objective functions. The *graph partitioning problem* is also looking for a partition of the vertices. Here, a *balancing constraint* demands that all blocks have weight $|V_i| \leq (1 + \epsilon) \lceil \frac{|V|}{k} \rceil =: L_{\max}$ for some imbalance parameter ϵ . A vertex is a *boundary vertex* if it is incident to a vertex in a different block. The objective is to minimize the total *cut* $|E \cap \bigcup_{i < j} V_i \times V_j|$. Throughout the paper, given a set S , the operator \in_{rnd} selects an element of S uniformly at random.

2.2 Ensemble/Overlay Clusterings

In machine learning, ensemble methods combine multiple weak classification algorithms to obtain a strong classifier. These base clusterings are used to decide whether pairs of vertices should belong to the same cluster [34, 44]. Given two clusterings, the *overlay clustering* is a clustering in which two vertices belong to the same cluster if and only if they belong to the same cluster in each of the input clusterings. More precisely, given two clusterings \mathcal{C}_1

and \mathcal{C}_2 the *overlay clustering* is the clustering where each block corresponds to a connected component of the graph $G_{\mathcal{E}} = (V, E \setminus \mathcal{E})$ where \mathcal{E} is the union of the cut edges of \mathcal{C}_1 and \mathcal{C}_2 , i.e. all edges that run between blocks in either \mathcal{C}_1 or \mathcal{C}_2 . Intuitively, if the input clusters agree that two vertices belong to the same cluster, the two vertices belong together with high confidence. It is easy to see that the number of clusters in the overlay clustering cannot be smaller than the number of clusters in each of the input clusterings.

An overlay clustering can be computed in expected linear-time. More precisely, given two clusterings $\{\mathcal{C}_1, \mathcal{C}_2\}$, we use the following approach to compute the overlay clustering. Initially, the overlay clustering \mathcal{O} is set to the clustering \mathcal{C}_1 . We then iterate through the remaining clusterings and incrementally update the current solution \mathcal{O} . To this end, we use pairs of cluster IDs (i, j) as a key in a hash map \mathcal{H} , where i is a cluster ID of \mathcal{O} and j is a cluster ID of the current clustering \mathcal{C} . We initialize \mathcal{H} to be the empty hashmap and set a counter c to zero. Then we iterate through the nodes. Let v be the current node. If the pair $(\mathcal{O}[v], \mathcal{C}[v])$ is not contained in \mathcal{H} , we set $\mathcal{H}(\mathcal{O}[v], \mathcal{C}[v])$ to c and increment c by one. Afterwards, we update the cluster ID of v in \mathcal{O} to be $\mathcal{H}(\mathcal{O}[v], \mathcal{C}[v])$. At the end of the algorithm, c is equal to the number of clusters contained in the overlay clustering and each vertex is labeled with its cluster ID in \mathcal{O} .

2.3 Objective Functions

There are a variety of measures used to assess the quality of a clustering, such as *coverage* [11], *performance* [46], *inter-cluster conductance* [24], *surprise* [3], *map equation* [38] and *modularity* [33]. The most simple index realizing a traditional measure of clustering quality is coverage. The coverage of a graph clustering \mathcal{C} is defined as the fraction of intra-cluster edges within the complete set of edges $\text{cov}(\mathcal{C}) := \frac{m(\mathcal{C})}{m}$. Intuitively, large values of coverage correspond to a good quality of a clustering. However, one principal drawback of coverage is that the converse is not necessarily true: coverage takes its largest value of 1 in the trivial case where there is only one cluster. *Modularity* fixes this issue by comparing the coverage of a clustering to the same value after rearranging edges randomly keeping node degrees. *Performance* is the fraction of correctly classified vertex pairs, w.r.t the set of edges. *Inter-cluster conductance* returns the worst (i.e. the thickest) bottleneck created by separating a cluster from the rest of the graph. *Surprise* measures the probability that a random graph \mathcal{R} has more intra-cluster edges. *Map equation* is a flow-based and information-theoretic method to assess clustering quality. Our focus in this work is on modularity as it is a widely accepted quality function and has been the main objective function in the 10th DIMACS implementation challenge [7]. For further discussions of these indices we refer the reader to the given references, and simply state the formal definition of modularity here:

$$\mathcal{Q}(\mathcal{C}) := \text{cov}(\mathcal{C}) - \mathbb{E}[\text{cov}(\mathcal{C})] = \frac{m(\mathcal{C})}{m} - \frac{1}{4m^2} \sum_{V_i \in \mathcal{C}} \left(\sum_{v \in V_i} d(v) \right)^2$$

2.4 Related Work

This paper is a summary and extension of the bachelor's thesis by Sonja Biedermann [8]. There has been a *significant* amount of research on graph clustering. We refer the reader to [20, 23] for thorough reviews of the results in this area. Here, we focus on results closely related to our main contributions. It is common knowledge that there is no single best strategy for graph clustering. Moreover, most quality indices for graph clusterings have turned out to be NP-hard to optimize and rather resilient to effective approximations, see

e.g. [4, 12, 47], allowing only heuristic approaches for optimization. Other approaches often rely on specific strategies with high running times, e.g. the iterative removal of central edges [33], or the direct identification of dense subgraphs [16]. Provably good methods with a decent running time include algorithms that have a spectral embedding of the vertices as the basis for a geometric clustering [13], min-cut tree clustering [19], a technique which guarantees certain bounds on bottlenecks and an approach which relies on random walks in graphs staying inside dense regions with high probabilities [46]. The Louvain method is a multi-level clustering algorithm introduced by Blondel et al. [10] that optimizes modularity as an objective function. As we use this method in our algorithm to create the initial population as well as to improve individuals after recombination, we go into more detail for that method in Section 2.6.

Recently, the 10th DIMACS challenge on graph partitioning and graph clustering compared different state-of-the-art graph clustering algorithms w.r.t optimizing modularity [6]. Most of the best results have been obtained by Ovelgönne and Geyer-Schulz [34] as well as Aloise et al. [2]. CGGC [34] used an ensemble learning strategy during the challenge. From several weak clusterings an overlap clustering is computed and a more expensive clustering algorithm is applied. VNS [2] applies the Variable Neighborhood Search heuristic to the graph clustering problem. Later, Džamić et al. [18] outperformed VNS by proposing an ascent-descent VNS. We compare ourselves to the results above in Section 4.

Evolutionary Graph Clustering. Tasgin and Bingol [45] introduce a genetic algorithm using modularity as a quality measure. They chose an integer encoding for representing the population clusterings. Individuals are randomly initialized, with some bias for assigning direct neighbors to the same cluster. Recombination is done one way, i.e. instead of using mutual exchange, clusters are transferred from a source individual to a target individual. More precisely, the operation picks a random vertex in the source individual and clusters all vertices of its cluster together in the destination individual. This is repeated several times in one recombination operation. As for mutation, the authors chose to pick one vertex and move it to a random cluster, which is almost guaranteed to decrease the fitness. This approach does not use local search to improve individuals.

A mutation-less agglomerative hierarchical genetic algorithm is presented by Lipczak and Milios in [27]. The authors represent each cluster as one individual and use one-point and uniform crossovers in that representation. Two synthetic networks have been used for the experimental evaluation. An important advantage of this algorithm is the possibility to distribute its computations.

2.5 Karlsruhe High-Quality Partitioning

Within this work, we use the open source multi-level graph partitioning framework KaHIP [41] (Karlsruhe High-Quality Partitioning). More precisely, we employ partitioning tools contained therein to create high-quality partitions of the graphs. We shortly outline its main components. KaHIP implements many different algorithms, for example, flow-based methods and more-localized local searches within a multi-level framework, as well as several coarse-grained parallel and sequential meta-heuristics. Recently, specialized methods to partition social networks and web graphs have been included in the framework [30].

2.6 Multi-level Louvain Method

The Louvain method is a multi-level clustering algorithm introduced by Blondel et al. [10]. It is an approach to graph clustering that optimizes modularity as an objective function. Since we instantiate our memetic algorithm to optimize for modularity, we give more detail in

order to be self-contained. The algorithm is organized in two phases: a local movement phase and contraction/uncontraction phase. The first phase, *local movement*, works in rounds and is done as follows: In the beginning, each vertex is a singleton cluster. Vertices are then traversed in random order and always moved to the neighboring cluster yielding the highest modularity increase. Computing the best move can be done in time proportional to the degree of a vertex by storing cumulative vertex degrees of clusters. More precisely, the gain in modularity by removing a vertex from a cluster can be computed by

$$\Delta Q(u) = \frac{s}{m} + \frac{d(u)}{4m^2} + \frac{(\sum_{v \in C} d(v))^2}{4m^2} - \frac{(d(u) + \sum_{v \in C} d(v))^2}{4m^2}.$$

A similar formula holds when adding a singleton vertex to a cluster. Once a vertex is moved, the cumulated degrees of the affected clusters are updated. Hence, the best move can be found in time proportional to the degree of a vertex. The local movement algorithm stops when a local maximum of the modularity is attained, that is when no vertex move yields modularity gain. The *second phase* of the algorithm consists in contracting the clustering. Contracting the clustering works as follows: each block of the clustering is contracted into a single node. There is an edge between two vertices u and v in the contracted graph if the two corresponding blocks in the clustering are adjacent to each other in G , i.e. block u and block v are connected by at least one edge. The weight of an edge (A, B) is set to the sum of the weight of the edges that run between block A and block B of the clustering. Moreover, a self-loop is inserted for each vertex in the contracted graph. The weight of this edge is set to be the cumulative weight of the edges in the respective cluster. Note that due to the way the contraction is defined, a clustering of the coarse graph corresponds to a clustering of the finer graph with the same objective. The algorithm then continues with the local movement phase on the contracted graph. In the end, the clustering contractions are undone and at each level local movement improves the current clustering w.r.t. modularity.

3 Memetic Graph Clustering

We now explain the components of our memetic graph clustering algorithm. Our algorithm starts with a population of individuals (in our case one individual is a clustering of the graph) and evolves the population into different populations over several rounds. In each round, the GA uses a selection rule based on the fitness of the individuals (in our case the objective function of the clustering problem under consideration) of the population to select good individuals and recombine them to obtain improved offspring. Our selection process is based on the tournament selection rule [31], i.e. \mathcal{C} is the fittest out of two random individuals R_1, R_2 from the population. When an offspring is generated an elimination rule is used to select a member of the population and replace it with the new offspring. In general, one has to take both into consideration, the fitness of an individual and the distance between individuals in the population [36] in order to avoid premature convergence of the algorithm. We evict the solution that is *most similar* with respect to the edges that run in between clusters with the offspring among those individuals in the population that have a worse or equal objective than the offspring itself. If there is no such individual, then the offspring is rejected and not inserted into the population. The difference between two individuals is defined as the size of the symmetric difference between their sets of cut edges. Our algorithm generates only one offspring per generation.

The core of our algorithm are our novel recombination and mutation operations. We provide two different kinds of operations, flat- and multi-level recombination operations. We also define a mutation operator that splits clusters by employing graph partitioning

techniques. In any case, the offspring is typically improved using a local search algorithm that optimizes for the objective function of the graph clustering problem. We continue this section by explaining the details of our algorithm with a focus on modularity – the results are transferable to other clustering problems by using local search algorithms that optimize the objective function of the problem under consideration.

3.1 Initialization/Creating Individuals

We initialize our population using the following modified Louvain algorithm. We also use the following algorithm to create an individual in recombine and mutation operations. Depending on the objective of the evolutionary algorithm, the choice of algorithms to initialize the population may be different. We describe our scheme for modularity.

We mainly use the Louvain method to create clusterings. Due to the non-deterministic nature of this algorithm – the order of nodes visited is randomized – we obtain different initial graph clusterings and later on individuals. In order to introduce more diversification, we modify the approach by using a different coarsening strategy based on the size-constrained label propagation algorithm [30]. Label propagation works similar to the local movement phase of the Louvain method. However, the objective of the algorithm is different. Without size-constraints, it was proposed by Raghavan et al. [37]. Initially, each vertex is in its own cluster/block, i.e. the initial block ID of a vertex is set to its vertex ID. The algorithm then works in rounds. In each round, the vertices of the graph are traversed in a random order. When a vertex v is visited, it is *moved* to the block that has the strongest connection to v , i.e. it is moved to the cluster V_i that maximizes $\omega(\{(v, u) \mid u \in N(v) \cap V_i\})$ whilst not overloading the target cluster w.r.t to the size-constraint bound U . Ties are broken randomly. The process is repeated until the process has converged. Here, we perform at most ℓ rounds of the algorithm, where ℓ is a tuning parameter, and stop the algorithm if less than five percent of the vertices changed its cluster during one round. One LPA round can be implemented to run in $\mathcal{O}(n + m)$ time.

We modify the coarsening stage of the Louvain method using size-constrained label propagation as follows: For the first $\lambda \in_{\text{rnd}} [0, 4]$ levels of the multi-level hierarchy we use size-constrained label propagation to compute the clustering to be contracted instead of the local movement phase to compute a clustering. We choose $U \in_{\text{rnd}} [n/10, n]$ in the beginning of the multi-level algorithm. Afterwards, we switch to the local movement phase of the Louvain method to compute a clustering to be contracted. Note that for $\lambda = 0$ the method is the Louvain method. In any case, in the uncoarsening phase local movement improves the current clustering by optimizing modularity.

3.2 Recombination

We define flat- and multi-level recombination operations which we are going to explain now. *Both* operations use the notion of overlay clustering to pass on good parts of the solutions to the offspring. Some of our recombination operators ensure that the offspring has non-decreasing fitness. Moreover, our recombine operations can recombine a solution from the population with an arbitrary clustering of the graph. Due to the fact that our local search and multi-level algorithms are randomized, a recombine operation performed twice using the same parents can yield different offsprings.

Flat Recombination. The *basic flat recombination operation* starts by taking two clusterings $\mathcal{C}_1, \mathcal{C}_2$ from the population as input and computes its overlay. The overlay is then contracted such that vertices represent blocks in the overlay clustering and edges represent edges between

vertices of two blocks. Edge weights are equal to the summed weight of the edges that run between the respective blocks, and self-loops are inserted with the edge weight being equal to the summed internal edge weight of that corresponding block. Note that due to the way contraction is defined, a clustering of the contracted graph can be transformed into a clustering of the input network having the same modularity score. Hence, we use the Louvain method to cluster the contracted graph, which then constitutes the offspring. Also, note that the contraction of the overlay ensures that vertices that are clustered together in both inputs will belong to the same cluster in the offspring. Only vertices which are split by one of the input clusterings will be affected by the Louvain method. What follows are multiple variations of the basic flat recombination method that ensure non-decreased fitness of the input individual, i.e. the offspring has fitness at least as good as the better input individual.

Apply Input Clustering. This operator uses the better of the two parent clusterings as a starting point for the Louvain method on the contracted graph. Due to the way contraction is defined the objective function of the starting point on the contracted graph is the same as of the corresponding clustering on the input network. The resulting offspring can be expected to be less diverse than when using the plain recombination operator, but may significantly more improve upon the parent.

Cluster-/Partition Recombination. This operator differs from the previous two operators insofar that only one of the parents is selected from the population. The other parent is manufactured on the spot by running either the size-constrained label propagation (SCLP) or a graph partitioning algorithm on the input graph. The resulting clustering can be very different from the other parent. This introduces more diversification which helps local search to explore a larger search space. When using SCLP clustering for the recombination, we use the parameters as described in Section 3.1. When using a partition, we use the KaHIP with the fastsocial preconfiguration using $k \in_{\text{rnd}} [2, 64]$ and $\epsilon \in_{\text{rnd}} [0.03, 0.5]$.

Multi-level Recombination. We now explain our multi-level recombine operator. This recombine operator also ensures that the solution quality of the offspring is *at least as good as the best of both parents*. For our recombine operator, let \mathcal{C}_1 and \mathcal{C}_2 again be two individuals from the population. Both individuals are used as input for the multi-level Louvain method in the following sense. Let \mathcal{E} be the set of edges that are cut edges, i.e. edges that run between two blocks, in either \mathcal{C}_1 or \mathcal{C}_2 . All edges in \mathcal{E} are blocked during the coarsening phase, i.e. they are *not* contracted during the coarsening phase. In other words, these edges cannot be contracted during the multi-level scheme. We ensure this by modifying the local movement phase of the Louvain method, i.e. clusters can only grow inside connected components of the overlay $(V, E \setminus \mathcal{E})$. We stop contracting clusterings when no contractable edge is left.

When coarsening is stopped, we then apply the better out of both input individuals w.r.t. the objective to the coarsest graph and use this as initial clustering instead of running the Louvain method on the coarsest graph. However, during uncoarsening local search still optimizes modularity on each level of the hierarchy. Note that this is possible since we did not contract any cut edge of both inputs. Also, note that this way we obtain a clustering of the coarse graph having a modularity score being equal to the better of both input individuals. Local movement guarantees no worsening of the clustering. Hence, the offspring is at least as good as the best input individual. Note that the coarsest graph is the same as the graph obtained in a flat recombination. However, the operation now is able to pass on good parts of the solution on multi-levels of the hierarchy during uncoarsening and hence has a more fine-grained view on both individuals.

3.3 Mutation

Our recombination operators can only decrease the number of clusters present in the solution. To counteract this behavior, we define a mutation operator that selects a subset of clusters and splits each of them in half. Splitting is done using the KaHIP graph partitioning framework. The number of clusters to be split is set to $p_s|\mathcal{C}|$ where p_s is the splitting probability and $|\mathcal{C}|$ is the number of clusters that the clustering selected from the population has. That means that we split a cluster into two balanced blocks such that there are only a small amount of edges running between them. We do this operation with two individuals from the population that are found by tournament selection. This results in two output individuals which are used as input to a multi-level recombination operation. The computed offspring of that operation is then inserted into the population as described above.

3.4 Parallelization

We now explain the island-based parallelization that we use. We use a parallelization scheme that has been successfully used in graph partitioning [40]. Each processing element (PE) basically performs the same operations using different random seeds. First, we estimate the population size \mathcal{S} : each PE creates an individual and measures the time \bar{t} spent. We then choose \mathcal{S} such that the time for creating \mathcal{S} clusterings is approximately t_{total}/c where the fraction c is a tuning parameter and t_{total} is the total running time that the algorithm is given to produce a clustering of the graph. The minimum amount of individuals in the population is set to 3, the maximum amount of the individuals in the population is set to 100. The lower bound on the population size is chosen to ensure a certain minimum of diversity, while the upper bound is used to ensure convergence. Each PE then builds its own population. Afterwards, the algorithm proceeds in rounds as long as time is left. Either a mutation or recombination operation is performed. Over time, the best individuals are exchanged between the PEs. Our communication protocol is similar to *randomized rumor spreading* which has shown to be scalable in previous work [40]. We refer to the full version of the paper [9] for a description of the communication protocol.

4 Experimental Evaluation

System and Methodology. We implemented the memetic algorithm described in the previous section within the KaHIP (Karlsruhe High Quality Partitioning) framework. The code is written in C++ and MPI. It has been compiled using g++-5.2 with flags `-O3` and OpenMPI 1.6.5. We refer to the algorithm presented in this paper as VieClus. The code is available at <http://vieclus.taa.univie.ac.at/>. Throughout this section, our main objective is modularity. All experiments comparing VieClus with competing algorithms are performed on a cluster with 512 nodes, where each node has two Intel Xeon E5-2670 Octa-Core (Sandy Bridge) processors clocked at 2.6 GHz, 64 GB main memory, 20 MB L3- and 8x256 KB L2-Cache and runs RHEL 7.4. We use the *arithmetic mean* when averaging over solutions of the same instance and the *geometric mean* when averaging over different instances in order to give every instance a comparable influence on the final result. It is well known that the algorithms that scored most of the points during the 10th DIMACS challenge compute better results than the Louvain method. Hence, we refrain from doing additional experiments with the Louvain method.

■ **Table 1** Results of our algorithm on the benchmark test set. Columns from left to right: average modularity achieved by our algorithm (Avg. Q), \bar{t} average time in minutes needed to beat the old challenge result, the best score computed of *our* algorithm (Max. Q), the best scores achieved by challenge participants, running time in minutes needed to create the previous best entry according to [26] and reference to the solver that achieved the result during the 10th DIMACS challenge.

Graph	Avg. Q	\bar{t} [m]	Max. Q	Q [7]	t_{sol} [m]	Solver
Small Instances						
as-22july06	0,679 391	<1	0,679 396	0,678 267	6,6	CGGC[34]
astro-ph	0,746 285	<1	0,746 292	0,744 621	11,9	VNS[2]
celegans_metabol	<i>0,453 248</i>	<1	<i>0,453 248</i>	0,453 248	<1	VNS[2]
cond-mat-2005	0,750 065	<1	0,750 171	0,746 254	40,9	CGGC[34]
email	<i>0,582 829</i>	<1	<i>0,582 829</i>	0,582 829	<1	VNS[2]
PGPgiantcompo	0,886 853	<1	0,886 853	0,886 564	1,9	CGGC[34]
polblogs	<i>0,427 105</i>	<1	<i>0,427 105</i>	0,427 105	<1	VNS[2]
power	0,940 975	<1	0,940 977	0,940 851	<1	VNS[2]
smallworld	0,793 186	<1	0,793 187	0,793 042	16,8	VNS[2]
memplus	0,701 242	2,6	0,701 275	0,700 473	3,2	CGGC[34]
G_n_pin_pout	0,500 457	3,9	0,500 466	0,500 098	64,8	CGGC[34]
caidaRouterLevel	0,872 804	5,0	0,872 828	0,872 042	81,0	CGGC[34]
rgg_n17	0,978 448	5,0	0,978 454	0,978 324	37,5	VNS[2]
luxembourg.osm	0,989 665	7,3	0,989 672	0,989 621	40,9	VNS[2]
Large Instances						
coAuthorsCiteseer	0,906 804	3,9	0,906 830	0,905 297	91,3	CGGC[34]
citationCiteseer	0,825 518	12,9	0,825 545	0,823 930	77,6	CGGC[34]
coPapersDBLP	0,868 019	20,5	0,868 058	0,866 794	603,3	CGGC[34]
belgium.osm	0,995 062	29,5	0,995 064	0,994 940	102,9	CGGC[34]
ldoor	0,970 521	35,1	0,970 555	0,969 370	485,6	ParMod[14]
eu-2005	0,941 575	65,8	0,941 575	0,941 554	341,5	CGGC[34]
in-2004	0,980 684	237,4	0,980 690	0,980 622	244,0	CGGC[34]
333SP	0,989 316	297,1	0,989 356	0,989 095	976,9	ParMod[14]
prefAttachment	0,315 843	*	0,316 089	0,315 994	1 353,1	VNS[2]

Parameters. Our algorithm is not very sensitive to the precise choice parameters. We *did not* perform a tuning of the parameters of the algorithm, rather we chose the parameters described above and below to be reasonable and to introduce a large amount of diversification or we chose parameters that were a good choice in previous evolutionary algorithms [40]. We expect the parameters to work well with a variety of instances. As our main design goal is to introduce as much diversification as possible, we use all recombination and mutation operations in our algorithm. The ratio of mutation to recombine operations has been set to 1:9 as this has been a good choice in previous evolutionary algorithms [40]. When we perform a recombine operation, we pick the recombine operation uniformly at random and diversify the parameters as described above. When performing a mutation operation, we use a splitting probability p_s uniformly at random in $[0.01, 0.1]$. We invest 1/10 of the total time to create the initial population.

Instances. We use the graphs that have been used for the 10th DIMACS implementation challenge on graph clustering and graph partitioning [7]. A list of the instances as well as the modularity scores that have been obtained during the challenge can be found in Table 1. We exclude the instances `cage15`, `audikw1`, `er-fact1.5-scale25`, `kron_*`, `uk-2002`, `uk-2007-05` from the challenge testbed in our evaluation, because they are either too large to be feasible for an evolutionary algorithm or they do not contain a significant cluster structure as indicated by the reported modularity score [6].

4.1 Evolutionary Graph Clustering

We now run our algorithm on all the DIMACS instances under consideration using the rules used there, i.e. *running time is not an issue* but we want to achieve modularity values as large as possible for each instance. On the small instances, we give our algorithm 2 hours of time to compute a solution and on the large instances, we set the time limit to 16 hours. In any case, we use 16 cores of our machine, i.e. one node of the machine. We perform the test five times with different random seeds. Table 1 summarizes the results of our experiment, and the technical report [9] contains convergence plots for a selected subset of the instances.

First of all, on *every* instance under consideration, our algorithm is able to compute a result that is better than the currently reported modularity value in literature. More precisely, in 98 out of 115 runs of our algorithm, the previous benchmark result of the 10th DIMACS implementation challenge is outperformed. In further 15 out of 115 runs, we reproduce the results. This is the case for each of the runs for the graphs `celegans_metabolic`, `email` and `polblogs`. The two cases in which our algorithm does not beat the previous best solver, are 2 runs on the graph `prefAttachment`. The other 3 runs on that graph outperform the previous result. The time needed to compute a clustering having a similar score on that graph is roughly 95% of the total running time. Moreover, as convergence plots in [9] show this may be fixable by giving the algorithm a larger amount of time to compute the solution.

Overall, the time needed to outperform the previous benchmark results ranges from less than a minute to a couple of minutes on the small instances. On the large instances our algorithm needs more time, but on most instances, the previous benchmark result can be computed in roughly an hour. Table 1 also reports the running time of the solver that obtained the result during the DIMACS challenge. Our algorithm is faster in every case compared to the previous solver (eventually by more than an order of magnitude), however, the machines used for the experiment are different and our algorithm is a parallel algorithm whereas previous solvers are sequential. Note that the final improvements over the old result are fairly small (<0.1% on average). This is not surprising, as previous solvers already invested a large amount of time to compute the results. However, note that the previous result has been computed by different solvers and our evolutionary algorithm can be seen as a single tool to compute the result.

We now compare the results with recently published results [26, 22, 28, 39]. LaSalle [26] reports results on all graphs from the DIMACS challenge subset. However, each of the best computed result is worse than the result computed by the respective best algorithm during the DIMACS challenge (and hence worse compared to our algorithm). Moreover, LaSalle [26] reports that his Nerstrand algorithm is on average equal or slightly better than the Louvain method on the instances used here. Lu et al. [28] present a result for `coPapersDBLP` ($Q = 0,858\,088$) and Ryu and Kim [39] report modularity for `email` ($Q = 0,568$) which are worse compared to the results that we report here. Hamann et al. [22] report the result for `in-2004` ($Q = 0,980$) which is comparable to the result that we report here. Džamić [18] build upon VNS proposing an ascent-decent VNS. Results are reported for `celegans_metabolic` ($Q = 0.453248$), `email` ($Q = 0,582\,829$), `polblogs` ($Q = 0,427\,105$) for which their algorithm computes the same results as all other tools reported in Table 1. Moreover, the following best results are reported `as-22july06` ($Q = 0,678\,381$), `astro-ph` ($Q = 0,745\,246$), `cond-mat-2005` ($Q = 0,747\,181$), `PGPgiantcompo` ($Q = 0.886647$), as well as `power` ($Q = 0.940974$) which are indeed better than the previous best result obtained during the 10th DIMACS challenge, but still worse than average result of our algorithm in every case. Hence, overall we consider our algorithm as a new state-of-the-art heuristic for solving the modularity clustering problem.

■ **Table 2** Columns from left to right: the approximate bound computed with the heuristic algorithm to balance cluster volumes and the ratio of our best value and the bound.

Graph	B	Q/B
Small Instances		
as-22july06	0,973 684	0,697 758
astro-ph	0,999 070	0,746 987
celegans_metabol	0,888 889	0,509 904
cond-mat-2005	0,999 468	0,750 570
email	0,900 000	0,647 588
PGPgiantcompo	0,989 796	0,895 996
polblogs	0,996 168	0,428 748
power	0,975 610	0,964 501
smallworld	0,995 652	0,796 651
memplus	0,984 127	0,712 586
G_n_pin_pout	0,993 865	0,503 555
caidaRouterLevel	0,997 758	0,874 789
rgg_n17	0,992 537	0,985 811
luxembourg.osm	0,996 283	0,993 364
Large Instances		
coAuthorsCiteseer	0,995 575	0,910 861
citationCiteseer	0,993 289	0,831 123
coPapersDBLP	0,995 283	0,872 172
belgium.osm	0,998 358	0,996 701
ldoor	0,989 796	0,980 561
eu-2005	0,996 564	0,944 821
in-2004	0,999 136	0,981 538
333SP	0,995 726	0,993 603
prefAttachment	0,875 000	0,361 245

4.2 Approximate Bound for Modularity

The modularity clustering problems seeks to maximize $\text{cov}(\mathcal{C}) - \mathbb{E}[\text{cov}(\mathcal{C})]$. However, the problem is NP-complete and our algorithm heuristically finds solutions. Moreover, modularity is trivially bounded by 1. The true optimum clustering, however, has typically a value below that. Thus we try to find a bound or a value to normalize the score such that one gets a clearer picture on how far away from the optimum score the achieved modularity value is.

To do so, let us define $\sum_{v \in V_i} d(v)$ to be the volume $\text{vol}(V_i)$ of the cluster V_i . Note that the formula $\frac{1}{4m^2} \sum_{V_i \in \mathcal{C}} \text{vol}(V_i)^2$ is minimized if all clusters have the same volume. But again, finding the optimum value of this equation is hard. As $\sum_{V_i \in \mathcal{C}} \text{vol}(V_i) = 2m$, the equation is minimized when $\text{vol}(V_i) = 2m/k \forall i = 1 \dots k$, where k is the number of clusters. Thus it follows that $\frac{1}{4m^2} \sum_{V_i \in \mathcal{C}} \text{vol}(V_i)^2 \geq \frac{1}{4m^2} \cdot k \cdot \left(\frac{2m}{k}\right)^2 = \frac{1}{k}$. This gives an upper of $1 - \frac{1}{k}$ for $Q(\mathcal{C})$ if the number of clusters is an input to the clustering problem. This upper bound is, however, not very far from the trivial upper bound of large k .

Thus for each graph we also clustered the nodes into cluster V_i using the following heuristic to try to balance the volumes of the resulting clusters as much as possible: We sort the nodes in decreasing order of their degree and then assign them step by step to the cluster having the smallest volume. We break ties by using the cluster with the smallest ID among those having the smallest volume. For this heuristic, we use the number of clusters that our algorithm has computed as value for k . For the resulting clustering \mathcal{C}^* we compute the value $\mathbb{E}[\text{cov}(\mathcal{C}^*)]$ and use $B := 1 - \mathbb{E}[\text{cov}(\mathcal{C}^*)]$ as a value to normalize $Q(\mathcal{C})$. Table 2 shows the resulting bound and compares them against our results.

5 Conclusion

We presented a parallel memetic algorithm, VieClus, that tackles the graph clustering problem. A key component of our contribution are natural recombine operators that employ ensemble clusterings as well as multi-level techniques. We combine these techniques with a scalable communication protocol, producing a system that is able to reproduce or improve previous all entries of the 10th DIMACS implementation challenge under consideration as well as results recently reported in the literature in a short amount of time. Moreover, while the previous best result for different instances has been computed by a variety of solvers, our algorithm can now be used as a single tool to compute the result. Hence, overall we consider our algorithm as a new state-of-the-art heuristic for solving the modularity clustering problem. In the future, it may be interesting to instantiate our scheme for different objective functions depending on the application domain or to use a more diverse set of initial solvers to create the population. We also want to look at distributed memory parallel multi-level algorithms for the problem that can use the depicted algorithm as an initial clustering scheme on the coarsest level of the hierarchy.

References

- 1 L. Akoglu, H. Tong, and D. Koutra. Graph Based Anomaly Detection and Description: A Survey. *Data Min. Knowl. Discov.*, 29(3):626–688, may 2015. doi:10.1007/s10618-014-0365-y.
- 2 D. Aloise, G. Caporossi, S. Perron, P. Hansen, L. Liberti, and M. Ruiz. Modularity Maximization in Networks by Variable Neighborhood Search. In *10th DIMACS Impl. Challenge Workshop*. Georgia Inst. of Technology, Atlanta, GA, 2012.
- 3 V. Arnau, S. Mars, and I. Marín. Iterative Cluster Analysis of Protein Interaction Data. *Bioinformatics*, 21(3):364–378, 2004.
- 4 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.
- 5 T. Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. PhD thesis, Informatik Centrum Dortmund, Germany, 1996.
- 6 D. Bader, A. Kappes, H. Meyerhenke, P. Sanders, C. Schulz, and D. Wagner. Benchmarking for Graph Clustering and Partitioning. In *Encyclopedia of Social Network Analysis and Mining*. Springer, 2014.
- 7 D. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, editors. *Proc. of the 10th DIMACS Impl. Challenge*, Cont. Mathematics. AMS, 2012.
- 8 S. Biedermann. Evolutionary Graph Clustering. Bachelor’s Thesis, Universität Wien, 2017.
- 9 S. Biedermann, M. Henzinger, C. Schulz, and B. Schuster. Memetic Graph Clustering (see ArXiv preprint arXiv:1802.07034). *Technical Report*. arXiv:1802.07034, 2018.
- 10 V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast Unfolding of Communities in Large Networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, 2008. URL: <http://stacks.iop.org/1742-5468/2008/i=10/a=P10008>.
- 11 U. Brandes. *Network Analysis: Methodological Foundations*, volume 3418. Springer Science & Business Media, 2005.
- 12 U. Brandes, D. Delling, M. Gaertler, R. Gorke, M. Hofer, Z. Nikoloski, and D. Wagner. On Modularity Clustering. *IEEE Transactions on Knowledge and Data Engineering*, 20(2):172–188, 2008.

- 13 U. Brandes, M. Gaertler, and D. Wagner. Engineering Graph Clustering: Models and Experimental Evaluation. *ACM Journal of Experimental Algorithmics*, 12(1.1):1–26, 2007.
- 14 Ü. V. Çatalyürek, K. Kaya, J. Langguth, and B. Uçar. A Divisive Clustering Technique for Maximizing the Modularity. In *10th DIMACS Impl. Challenge Workshop*. Georgia Inst. of Technology, Atlanta, GA, 2012.
- 15 J. Demme and S. Sethumadhavan. Approximate Graph Clustering for Program Characterization. *ACM Trans. Archit. Code Optim.*, 8(4):21:1–21:21, 2012. doi:10.1145/2086696.2086700.
- 16 I. Derényi, G. Palla, and T. Vicsek. Clique Percolation in Random Networks. *Physical review letters*, 94(16):160202, 2005.
- 17 A. A. Diwan, S. Rane, S. Seshadri, and S. Sudarshan. Clustering Techniques for Minimizing External Path Length. In *Proceedings of the 22th International Conference on Very Large Data Bases, VLDB '96*, pages 342–353, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc. URL: <http://dl.acm.org/citation.cfm?id=645922.673636>.
- 18 D. Džamić, D. Aloise, and N. Mladenović. Ascent–descent Variable Neighborhood Decomposition Search for Community Detection by Modularity Maximization. *Annals of Operations Research*, Jun 2017. doi:10.1007/s10479-017-2553-9.
- 19 G. W. Flake, R. E. Tarjan, and K. Tsioutsouliklis. Graph Clustering and Minimum Cut Trees. *Internet Mathematics*, 1(4):385–408, 2004.
- 20 S. Fortunato. Community Detection in Graphs. *Physics reports*, 486(3):75–174, 2010.
- 21 D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.
- 22 M. Hamann, B. Strasser, D. Wagner, and T. Zeitz. Simple Distributed Graph Clustering using Modularity and Map Equation. *arXiv preprint arXiv:1710.09605*, 2017.
- 23 T. Hartmann, A. Kappes, and D. Wagner. Clustering Evolving Networks. In *Algorithm Engineering*, pages 280–329. Springer, 2016.
- 24 R. Kannan, S. Vempala, and A. Vetta. On clusterings: Good, bad and spectral. *Journal of the ACM (JACM)*, 51(3):497–515, 2004.
- 25 J. Kim, I. Hwang, Y. H. Kim, and B. R. Moon. Genetic Approaches for Graph Partitioning: A Survey. In *Proceedings of the 13th Annual Genetic and Evolutionary Computation Conference (GECCO'11)*, pages 473–480. ACM, 2011. doi:10.1145/2001576.2001642.
- 26 D. LaSalle. Graph Partitioning, Ordering, and Clustering for Multicore Architectures, 2015.
- 27 M. Lipczak and E. E. Milios. Agglomerative Genetic Algorithm for Clustering in Social Networks. In Franz Rothlauf, editor, *GECCO*, pages 1243–1250. ACM, 2009. URL: <http://dblp.uni-trier.de/db/conf/gecco/gecco2009.html#LipczakM09>.
- 28 H. Lu, M. Halappanavar, and A. Kalyanaraman. Parallel heuristics for scalable community detection. *Parallel Computing*, 47:19–37, 2015.
- 29 S. McFarling. Program Optimization for Instruction Caches. *SIGARCH Comput. Archit. News*, 17(2):183–191, 1989. doi:10.1145/68182.68200.
- 30 H. Meyerhenke, P. Sanders, and C. Schulz. Partitioning Complex Networks via Size-Constrained Clustering. In *SEA*, volume 8504 of *Lecture Notes in Computer Science*, pages 351–363. Springer, 2014.
- 31 B. L. Miller and D. E. Goldberg. Genetic Algorithms, Tournament Selection, and the Effects of Noise. *Evolutionary Computation*, 4(2):113–131, 1996.
- 32 M. E. J. Newman. Properties of Highly Clustered Networks. *Physical Review E*, 68(2):026121, 2003.
- 33 M. E. J. Newman and M. Girvan. Finding and Evaluating Community Structure in Networks. *Physical review E*, 69(2):026113, 2004.

- 34 M. Ovelgönne and A. Geyer-Schulz. An Ensemble Learning Strategy for Graph Clustering. In *Graph Partitioning and Graph Clustering*, number 588 in Contemporary Mathematics, 2013.
- 35 J. B. Pereira-Leal, A. J. Enright, and C. A. Ouzounis. Detection of Functional Modules from Protein Interaction Networks. *Proteins: Structure, Function, and Bioinformatics*, 54(1):49–57, 2004. doi:10.1002/prot.10505.
- 36 D. C. Porumbel, J.-K. Hao, and P. Kuntz. Spacing Memetic Algorithms. In *13th Annual Genetic and Evolutionary Computation Conference, GECCO 2011, Proceedings, Dublin, Ireland, July 12-16, 2011*, pages 1061–1068, 2011.
- 37 U. N. Raghavan, R. Albert, and S. Kumara. Near Linear Time Algorithm to Detect Community Structures in Large-Scale Networks. *Physical Review E*, 76(3), 2007.
- 38 M. Rosvall, D. Axelsson, and C. T. Bergstrom. The Map Equation. *The European Physical Journal-Special Topics*, 178(1):13–23, 2009.
- 39 S. Ryu and D. Kim. Quick Community Detection of Big Graph Data Using Modified Louvain Algorithm. In *High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2016 IEEE 18th International Conference on*, pages 1442–1445. IEEE, 2016.
- 40 P. Sanders and C. Schulz. Distributed Evolutionary Graph Partitioning. In *Proc. of the 12th Workshop on Algorithm Engineering and Experimentation (ALENEX'12)*, pages 16–29, 2012.
- 41 P. Sanders and C. Schulz. Think Locally, Act Globally: Highly Balanced Graph Partitioning. In *12th International Symposium on Experimental Algorithms (SEA'13)*. Springer, 2013.
- 42 S. E. Schaeffer. Survey: Graph Clustering. *Comput. Sci. Rev.*, 1(1):27–64, 2007. doi:10.1016/j.cosrev.2007.05.001.
- 43 C. L. Staudt and H. Meyerhenke. Engineering High-Performance Community Detection Heuristics for Massive Graphs. In *Proceedings 42nd Conference on Parallel Processing (ICPP'13)*, 2013.
- 44 C. L. Staudt and H. Meyerhenke. Engineering Parallel Algorithms for Community Detection in Massive Networks. *IEEE Trans. on Parallel and Distributed Systems*, 27(1):171–184, 2016. doi:10.1109/TPDS.2015.2390633.
- 45 M. Tasgin and H. Bingol. Community Detection in Complex Networks using Genetic Algorithm. In *ECCS '06: Proc. of the European Conference on Complex Systems*, 2006. arXiv:cond-mat/0604419.
- 46 S. M. Van Dongen. *Graph Clustering by Flow Simulation*. PhD thesis, Utrecht University, 2001.
- 47 D. Wagner and F. Wagner. Between Min Cut and Graph Bisection. In *Proceedings of the 18th International Symposium on Mathematical Foundations of Computer Science*, pages 744–750. Springer, 1993.
- 48 Y. Xu, V. Olman, and D. Xu. Clustering Gene Expression Data using a Graph-Theoretic Approach: an Application of Minimum Spanning Trees. *Bioinformatics*, 18(4):536–545, 2002.

ILP-based Local Search for Graph Partitioning

Alexandra Henzinger

Stanford University, Stanford, CA, USA
ahenz@stanford.edu

Alexander Noe

University of Vienna, Vienna, Austria
alexander.noe@univie.ac.at

Christian Schulz

University of Vienna, Vienna, Austria
christian.schulz@univie.ac.at

Abstract

Computing high-quality graph partitions is a challenging problem with numerous applications. In this paper, we present a novel meta-heuristic for the balanced graph partitioning problem. Our approach is based on integer linear programs that solve the partitioning problem to optimality. However, since those programs typically do not scale to large inputs, we adapt them to heuristically improve a given partition. We do so by defining a much smaller model that allows us to use symmetry breaking and other techniques that make the approach scalable. For example, in Walshaw’s well-known benchmark tables we are able to improve roughly half of all entries when the number of blocks is high.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms, Theory of computation → Integer programming

Keywords and phrases Graph Partitioning, Integer Linear Programming

Digital Object Identifier 10.4230/LIPIcs.SEA.2018.4

Related Version A full version of the paper is available at <https://arxiv.org/abs/1802.07144>.

Funding The research leading to these results has received funding from the European Research Council under the European Community’s Seventh Framework Programme (FP7/2007-2013)/ERC grant agreement No. 340506. Moreover, the authors gratefully acknowledge the Gauss Centre for Supercomputing e.V. (<http://www.gauss-centre.eu>) for funding this project by providing computing time on the GCS Supercomputer SuperMUC at Leibniz Supercomputing Centre (<https://www.lrz.de>).

1 Introduction

Balanced graph partitioning is an important problem in computer science and engineering with an abundant amount of application domains, such as VLSI circuit design, data mining and distributed systems [38]. It is well known that this problem is NP-complete [8] and that no approximation algorithm with a constant ratio factor exists for general graphs unless $P=NP$ [8]. Still, there is a large amount of literature on methods (with worst-case exponential time) that solve the graph partitioning problem to optimality. This includes methods dedicated to the bipartitioning case [3, 4, 12, 13, 14, 15, 24, 21, 30, 39] and some methods that solve the general graph partitioning problem [16, 40]. Most of these methods rely on the branch-and-bound framework [28]. However, these methods can typically solve



© Alexandra Henzinger, Alexander Noe, and Christian Schulz;
licensed under Creative Commons License CC-BY

17th International Symposium on Experimental Algorithms (SEA 2018).

Editor: Gianlorenzo D’Angelo; Article No. 4; pp. 4:1–4:15

Leibniz International Proceedings in Informatics



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

only very small problems as their running time grows exponentially, or if they can solve large bipartitioning instances using a moderate amount of time [12, 13], the running time highly depends on the bisection width of the graph. Methods that solve the general graph partitioning problem [16, 40] have huge running times for graphs with up to a few hundred vertices. Thus in practice mostly heuristic algorithms are used.

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. Here, we focus on the case when the bounds on the size are very strict, including the case of *perfect balance* when the maximal block size has to equal the average block size.

Our focus in this paper is on solution quality, i.e. minimize the number of edges that run between blocks. During the past two decades there have been numerous researchers trying to improve the best graph partitions in Walshaw’s well-known partitioning benchmark [41, 42]. Overall there have been more than forty different approaches that participated in this benchmark. Indeed, high solution quality is of major importance in applications such as VLSI Design [1, 2] where even minor improvements in the objective can have a large impact on the production costs and quality of a chip. High-quality solutions are also favorable in applications where the graph needs to be partitioned only once and then the partition is used over and over again, implying that the running time of the graph partitioning algorithms is of a minor concern [11, 18, 27, 29, 32, 31]. Thirdly, high-quality solutions are even important in areas in which the running time overhead is paramount [41], such as finite element computations [37] or the direct solution of sparse linear systems [20]. Here, high-quality graph partitions can be useful for benchmarking purposes, i.e. measuring how much more running time can be saved by higher quality solutions.

In order to compute high-quality solutions, state-of-the-art local search algorithms exchange vertices between blocks of the partition trying to decrease the cut size while also maintaining balance. This highly restricts the set of possible improvements. Recently, we introduced new techniques that relax the balance constraint for vertex movements but globally maintain balance by combining multiple local searches [36]. This was done by reducing this combination problem to finding negative cycles in a graph. In this paper, we extend the neighborhood of the combination problem by employing integer linear programming. This enables us to find even more complex combinations and hence to further improve solutions. More precisely, our approach is based on integer linear programs that solve the partitioning problem to optimality. However, out of the box those programs typically do not scale to large inputs, in particular because the graph partitioning problem has a very large amount of symmetry – given a partition of the graph, each permutation of the block IDs gives a solution having the same objective and balance. Hence, we adapt the integer linear program to improve a given input partition. We do so by defining a much smaller graph, called *model*, and solve the graph partitioning problem on the model to optimality by the integer linear program. More specifically, we select vertices close to the cut of the given input partition for potential movement and contract all remaining vertices of a block into a single vertex. A feasible partition of this model corresponds to a partition of the input graph having the same balance and objective. Moreover, this model enables us to use symmetry breaking, which allows us to scale to much larger inputs. To make the approach even faster, we combine it with initial bounds on the objective provided by the input partition, as well as providing the input partition to the integer linear program solver. Overall, we arrive at a system that is able to improve more than half of all entries in Walshaw’s benchmark when the number of blocks is high.

The rest of the paper is organized as follows. We begin in Section 2 by introducing basic concepts. After presenting some related work in Section 3 we outline the integer linear

program as well as our novel local search algorithm in Section 4. Here, we start by explaining the very basic idea that allows us to find combinations of simple vertex movements. We then explain our strategies to improve the running time of the solver and strategies to select vertices for movement. A summary of extensive experiments done to evaluate the performance of our algorithms is presented in Section 5. Finally, we conclude in Section 6.

2 Preliminaries

2.1 Basic concepts

Let $G = (V = \{0, \dots, n-1\}, E)$ be an undirected graph. We consider positive, real-valued edge and vertex weight functions ω resp. c and extend them to sets, i.e., $\omega(E') := \sum_{x \in E'} \omega(x)$ and $c(V') := \sum_{x \in V'} c(x)$. Let $N(v) := \{u : \{v, u\} \in E\}$ denote the neighbors of v . The degree of a vertex v is $d(v) := |N(v)|$. A vertex is a *boundary vertex* if it is incident to at least one vertex in a different block. We are looking for disjoint *blocks* of vertices V_1, \dots, V_k that partition V ; i.e., $V_1 \cup \dots \cup V_k = V$. The *balancing constraint* demands that each block has weight $c(V_i) \leq (1 + \epsilon) \lceil \frac{c(V)}{k} \rceil =: L_{\max}$ for some imbalance parameter ϵ . We call a block V_i *overloaded* if its weight exceeds L_{\max} . The objective of the problem is to minimize the total cut $\omega(E \cap \bigcup_{i < j} V_i \times V_j)$ subject to the balancing constraints. We define the *gain* of a vertex as the maximum decrease in the cut value when moving it to a different block.

3 Related Work

There has been a *huge* amount of research on graph partitioning and we refer the reader to the surveys given in [6, 9, 37, 43] for most of the material. Here, we focus on issues closely related to our main contributions. All general-purpose methods that are able to obtain good partitions for large real-world graphs are based on the multi-level principle. Well-known software packages based on this approach include Jostle [43], KaHIP [34], Metis [25] and Scotch [33].

Chris Walshaw's well-known benchmark archive has been established in 2001 [41, 42]. Overall it contains 816 instances (34 graphs, 4 values of imbalance, and 6 values of k). In this benchmark, the running time of the participating algorithms is not measured or reported. Submitted partitions will be validated and added to the archive if they improve on a particular result. This can either be an improvement in the number of cut edges or, if they match the current best cut size, an improvement in the weight of the largest block. Most entries in the benchmark have as of Feb. 2018 been obtained by Galinier et al. [19] (more precisely an implementation of that approach by Frank Schneider), Hein and Seitzer [22] and the Karlsruhe High-Quality Graph Partitioning (KaHIP) framework [36]. More precisely, Galinier et al. [19] use a memetic algorithm that is combined with tabu search to compute solutions and Hein and Seitzer [22] solve the graph partitioning problem by providing tight relaxations of a semi-definite program into a continuous problem.

The Karlsruhe High-Quality Graph Partitioning (*KaHIP*) framework implements many different algorithms, for example flow-based methods and more-localized local searches, as well as several coarse-grained parallel and sequential meta-heuristics. KaBaPE [36] is a coarse-grained parallel evolutionary algorithm, i.e. each processor has its own population (set of partitions) and a copy of the graph. After initially creating the local population, each processor performs multi-level combine and mutation operations on the local population. This is combined with a meta-heuristic that combines local searches that individually violate the balance constraint into a more global feasible improvement. For more details, we refer the reader to [36].

4 Local Search based on Integer Linear Programming

We now explain our algorithm that combines integer linear programming and local search. We start by explaining the integer linear program that can solve the graph partitioning problem to optimality. However, out-of-the-box this program does not scale to large inputs, in particular because the graph partitioning problem has a very large amount of symmetry. Thus, we reduce the size of the graph by first computing a partition using an existing heuristic and based on it collapsing parts of the graph. Roughly speaking, we compute a small graph, called *model*, in which we only keep a small number of selected vertices for potential movement and perform graph contractions on the remaining ones. A partition of the model corresponds to a partition of the input network having the same objective and balance. The computed model is then solved to optimality using the integer linear program. As we will see this process enables us to use symmetry breaking in the linear program, which in turn drastically speeds up computation times.

4.1 Integer Linear Program for the Graph Partitioning Problem

We now introduce a generalization of an integer linear program formulation for balanced bipartitioning [7] to the general graph partitioning problem. First, we introduce binary decision variables for all edges and vertices of the graph. More precisely, for each edge $e = \{u, v\} \in E$, we introduce the variable $e_{uv} \in \{0, 1\}$ which is one if e is a cut edge and zero otherwise. Moreover, for each $v \in V$ and block k , we introduce the variable $x_{v,k} \in \{0, 1\}$ which is one if v is in block k and zero otherwise. Hence, we have a total of $|E| + k|V|$ variables. We use the following constraints to ensure that the result is a valid k -partition:

$$\forall \{u, v\} \in E, \forall k : e_{uv} \geq x_{u,k} - x_{v,k} \quad (1)$$

$$\forall \{u, v\} \in E, \forall k : e_{uv} \geq x_{v,k} - x_{u,k} \quad (2)$$

$$\forall k : \sum_{v \in V} x_{v,k} c(v) \leq L_{\max} \quad (3)$$

$$\forall v \in V : \sum_k x_{v,k} = 1 \quad (4)$$

The first two constraints ensure that e_{uv} is set to one if the vertices u and v are in different blocks. For an edge $\{u, v\} \in E$ and a block k , the right-hand side in this equation is one if one of the vertices u and v is in block k and the other one is not. If both vertices are in the same block then the right-hand side is zero for all values of k . Hence, the variable can either be zero or one in this case. However, since the variable participates in the objective function and the problem is a minimization problem, it will be zero in an optimum solution.

The third constraint ensures that the balance constraint is satisfied for each partition. And finally, the last constraint ensures that each vertex is assigned to exactly one block. To sum up, our program has $2k|E| + k + |V|$ constraints and $k \cdot (6|E| + 2|V|)$ non-zeros. Since we want to minimize the weight of cut edges, the objective function of our program is written as:

$$\min \sum_{\{u,v\} \in E} e_{uv} \cdot \omega(\{u, v\}) \quad (5)$$

4.2 Local Search

The graph partitioning problem has a large amount of symmetry – each permutation of the block IDs gives a solution with equal objective and balance. Hence, the integer linear program described above will scan many branches that contain essentially the same solutions so that the program does not scale to large instances. Moreover, it is not immediately clear how to improve the scalability of the program by using symmetry breaking or other techniques.

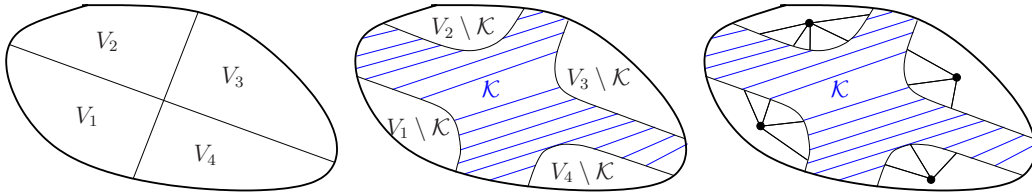
Our goal in this section is to develop a local search algorithm using the integer linear program above. Given a partition as input to be improved, our *main idea* is to contract vertices “that are far away” from the cut of the partition. In other words, we want to keep vertices close to the cut and contract all remaining vertices into one vertex for each block of the input partition. This ensures that a partition of the contracted graph yields a partition of the input graph with the same objective and balance. Hence, we apply the integer linear program to the model and solve the partitioning problem on it to optimality. Note, however, that due to the performed contractions this does not imply an optimal solution on the input graph.

We now outline the details of the algorithm. Our local algorithm has two inputs, a graph G and a partition V_1, \dots, V_k of its vertices. For now assume that we have a set of vertices $\mathcal{K} \subset V$ which we want to keep in the coarse model, i.e. a set of vertices which we do not want to contract. We outline in Section 4.4 which strategies we have to select the vertices \mathcal{K} . For the purpose of contraction we define k sets $\mathcal{V}_i := V_i \setminus \mathcal{K}$. We obtain our coarse model by contracting each of these vertex sets. The contraction of a vertex set \mathcal{V}_i works as follows: the set of vertices is contracted into a single vertex μ_i . The weight of μ_i is set to the sum of the weight of all vertices in the set that is contracted. There is an edge between two vertices μ_i and v in the contracted graph if there is an edge between a vertex of the set and v in the original graph G . The weight of an edge (μ_i, v) is set to the sum of the weight of edges that run between the vertices of the set and v . After all contractions have been performed the coarse model contains $k + |\mathcal{K}|$ vertices, and potentially much less edges than the input graph. Figure 1 gives an abstract example of our model.

There are two things that are important to see: first, due to the way we perform contraction, the given partition of the input network yields a partition of our coarse model that has the same objective and balance simply by putting μ_i into block i and keeping the block of the input for the vertices in \mathcal{K} . Moreover, if we compute a new partition of our coarse model, we can build a partition in the original graph with the same properties by putting the vertices \mathcal{V}_i into the block of their coarse representative μ_i together with the vertices of \mathcal{K} that are in this block. Hence, we can solve the integer linear program on the coarse model to compute a partition for the input graph. After the solver terminates, i.e. found an optimum solution of our mode or has reached a predefined time limit \mathcal{T} , we transfer the best solution to the original graph. Note that the latter is possible since an integer linear program solver typically computes intermediate solutions that may not be optimal.

4.3 Optimizations

Independent of the vertices \mathcal{K} that are selected to be kept in the coarse model, the approach above allows us to define optimizations to solve our integer linear program faster. We apply four strategies: (i) symmetry breaking, (ii) providing a start solution to the solver, (iii) add the objective of the input as a constraint as well as (iv) using the parallel solving facilities of the underlying solver. We outline the first three strategies in greater detail:



■ **Figure 1** From left to right: a graph that is partitioned into four blocks, the set \mathcal{K} close to the boundary that will stay in the model, and lastly the model in which the sets $V_i \setminus \mathcal{K}$ have been contracted.

Symmetry Breaking. If the set \mathcal{K} is small, then the solver will find a solution much faster. Typically, our algorithms selects the vertices \mathcal{K} such that $c(\mu_i) + c(\mu_j) > L_{\max}$. In other words, no two contracted vertices can be clustered in one block. We can use this to break symmetry in our integer linear programming by adding constraints that fix the block of μ_i to block i , i.e. we set $x_{\mu_i, i} = 1$ and $x_{\mu_i, j} = 0$ for $i \neq j$. Moreover, for those vertices we can remove the constraint which ensures that the vertex is assigned to a single unique block – since we assigned those vertices to a block using the new additional constraints.

Providing a Start Solution to the Solver. The integer linear program performs a significant amount of work in branches which correspond to solutions that are worse than the input partitioning. Only very few - if any - solutions are better than the given partition. However, we already know a fairly good partition (the given partition from the input) and give this partition to the solver by setting according initial values for all variables. This ensures that the integer linear program solver can omit many branches and hence speeds up the time needed to solve the integer linear program.

Solution Quality as a Constraint. Since we are only interested in improved partitions, we can add an additional constraint that disallows solutions which have a worse objective than the input partition. Indeed, the objective function of the linear program is linear, and hence the additional constraint is also linear. Depending on the objective value, this reduces the number of branches that the linear program solver needs to look at. However, note that this comes at the cost of an additional constraint that needs to be evaluated.

4.4 Vertex Selection Strategies

The algorithm above works for different vertex sets \mathcal{K} that should be kept in the coarse model. There is an obvious trade-off: on the one hand, the set \mathcal{K} should not be too large, otherwise the coarse model would be large and hence the linear programming solver needs a large amount of time to find a solution. On the other hand, the set should also not be too small, since this restricts the amount of possible vertex movements, and hence the approach is unlikely to find an improved solution. We now explain different strategies to select the vertex set \mathcal{K} . In any case, while we add vertices to the set \mathcal{K} , we compute the number of non-zeros in the corresponding ILP. We stop to add vertices when the number of non-zeros in the corresponding ILP is larger than a parameter \mathcal{N} .

Vertices Close to Input Cut. The intuition of the first strategy, **Boundary**, is that changes or improvements of the partition will occur reasonable close to the input partition. In this simple strategy our algorithm tries to use all *boundary vertices* as the set \mathcal{K} . In order to adhere to the constraint on the number of non-zeros in the ILP, we add the vertices

of the boundary uniformly at random and stop if the number of non-zeros \mathcal{N} is reached. If the algorithm managed to add all boundary vertices whilst not exceeding the specified number of non-zeros, we do the following extension: we perform a breadth-first search that is initialized with a random permutation of the boundary vertices. All additional vertices that are reached by the BFS are added to \mathcal{K} . As soon as the number of non-zeros \mathcal{N} is reached, the algorithm stops.

Start at Promising Vertices. Especially for high values of k the boundary contains many vertices. The **Boundary** strategy quickly adds a lot of random vertices while ignoring vertices that have high gain. However, note that even in good partitions it is possible that vertices with positive gain exist but cannot be moved due to the balance constraint.

Hence, our second strategy, **Gain $_{\rho}$** , tries to fix this issue by starting a breadth-first search initialized with only high gain vertices. More precisely, we initialize the BFS with each vertex having gain $\geq \rho$ where ρ is a tuning parameter. Our last strategy, **TopVertices $_{\delta}$** , starts by sorting the boundary vertices by their gain. We break ties uniformly at random. Vertices are then traversed in decreasing order (highest gain vertices first) and for each start vertex v our algorithm adds all vertices with distance $\leq \delta$ to the model. The algorithm stops as soon as the number of non-zeros exceeds \mathcal{N} .

Early gain-based local search heuristics for the ϵ -balanced graph partitioning problem searched for pairwise swaps with positive gain [17, 26]. More recent algorithms generalized this idea to also search for cycles or paths with positive total gain [36]. An important advantage of our new approach is that we solve the combination problem to optimality, i.e. our algorithm finds the best combination of vertex movements of the vertices in \mathcal{K} w.r.t to the input partition of the original graph. Therefore we can also find more complex optimizations that cannot be reduced to positive gain cycles and paths.

5 Experiments

5.1 Experimental Setup and Methodology

We implemented the algorithms using C++-17 and compiled all codes using g++-7.2.0 with full optimization (-O3). We use Gurobi 7.5.2 as an ILP solver and always use its parallel version. All of our experiments were conducted on a machine with two Haswell Xeon E5-2697 v3 processors. The machine has 28 cores at 2.6GHz as well as 64GB of main memory and runs the SUSE Linux Enterprise Server (SLES) operating system. Unless otherwise mentioned, our approach uses the shared-memory parallel variant of Gurobi using all 28 cores. In general, we perform five repetitions per instance and report the average running time as well as cut. Unless otherwise mentioned, we use a time limit for the integer linear program. When the time limit is passed, the integer linear program solver outputs the best solution that has currently been discovered. This solution does not have to be optimal. Note that we do not perform experiments with Metis [25] and Scotch [33] in here, since previous papers, e.g. [34, 35], have already shown that solution quality obtained is much worse than results achieved in the Walshaw benchmark. When averaging over multiple instances, we use the geometric mean in order to give every instance the same influence on the *final score*.

Performance Plots. These plots relate the fastest running time to the running time of each other ILP-based local search algorithm on a per-instance basis. For each algorithm, these ratios are sorted in increasing order. The plots show the ratio $t_{\text{best}}/t_{\text{algorithm}}$ on the y-axis to highlight the instances in which each algorithm performs badly. For plots in which we measure solution quality, the y-axis shows the ratio $\text{cut}_{\text{best}}/\text{cut}_{\text{algorithm}}$. A point close to

■ **Table 1** Basic properties of the benchmark instances.

Graph	n	m	Graph	n	m
Walshaw Graphs (Set B)			Walshaw Graphs (Set B)		
add20	2 395	7 462	wing	62 032	≈ 121K
data	2 851	15 093	brack2	62 631	≈ 366K
3elt	4 720	13 722	finan512	74 752	≈ 261K
uk	4 824	6 837	fe_tooth	78 136	≈ 452K
add32	4 960	9 462	fe_rotor	99 617	≈ 662K
bcsstk33	8 738	≈ 291K	598a	110 971	≈ 741K
whitaker3	9 800	28 989	fe_ocean	143 437	≈ 409K
crack	10 240	30 380	144	144 649	≈ 1.1M
wing_nodal	10 937	75 488	wave	156 317	≈ 1.1M
fe_4elt2	11 143	32 818	m14b	214 765	≈ 1.7M
vibrobox	12 328	≈ 165K	auto	448 695	≈ 3.3M
bcsstk29	13 992	≈ 302K	Parameter Tuning (Set A)		
4elt	15 606	45 878	delanay_n15	32 768	98 274
fe_sphere	16 386	49 152	rgg_15	32 768	≈ 160K
cti	16 840	48 232	2cubes_sphere	101 492	≈ 772K
memplus	17 758	54 196	cf2	123 440	≈ 1.5M
cs4	22 499	43 858	boneS01	127 224	≈ 3.3M
bcsstk30	28 924	≈ 1.0M	Dubcova3	146 689	≈ 1.7M
bcsstk31	35 588	≈ 572K	G2_circuit	150 102	≈ 288K
fe_pwt	36 519	≈ 144K	thermal2	1 227 087	≈ 3.7M
bcsstk32	44 609	≈ 985K	as365	3 799 275	≈ 11.4M
fe_body	45 087	≈ 163K	adaptive	6 815 744	≈ 13.6M
t60k	60 005	89 440			

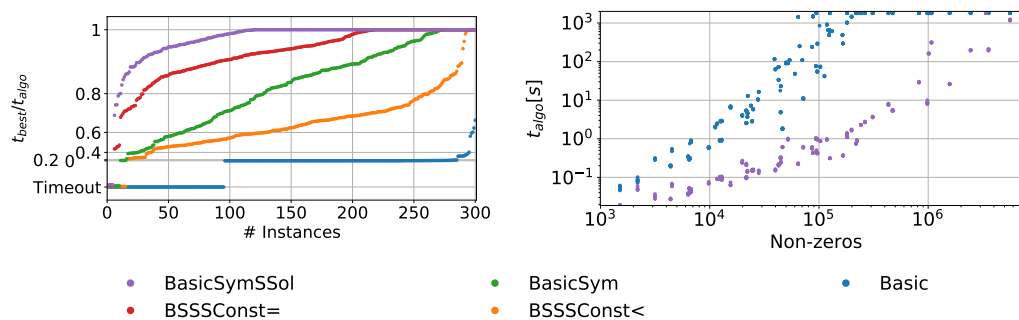
zero indicates that the running time/quality of the algorithm was considerably worse than the fastest/best algorithm on the same instance. A value of one therefore indicates that the corresponding algorithm was one of the fastest/best algorithms to compute the solution. Thus an algorithm is considered to outperform another algorithm if its corresponding ratio values are above those of the other algorithm. In order to include instances that hit the time limit, we set the corresponding values to a negative value for ratio computations.

Instances. We perform experiments on two sets of instances. Set A is used to determine the performance of the integer linear programming optimizations and to tune the algorithm. We obtained these instances from the Florida Sparse Matrix collection [10] and the 10th DIMACS Implementation Challenge [5] to test our algorithm. Set B are all graphs from Chris Walshaw’s graph partitioning benchmark archive [41, 42]. This archive is a collection of instances from finite-element applications, VLSI design and is one of the default benchmarking sets for graph partitioning.

Table 1 gives basic properties of the graphs from both benchmark sets. We ran the unoptimized integer linear program that solves the graph partitioning problem to optimality from Section 4.1 on the five smallest instances from the Walshaw benchmark set. With a time limit of 30 minutes, the solver has only been able to compute a solution for the graphs uk and add32 with $k = 2$. For higher values of k the solver was unable to find any solution in the time limit. Even giving a starting solution does not increase the number of ILPs solved. Hence, we omit further experiments in which we run an ILP solver on the full graph.

5.2 Impact of Optimizations

We now evaluate the impact of the optimization strategies for the ILP that we presented in Section 4.3. In this section, we use the variant of our local search algorithm in which \mathcal{K} is



■ **Figure 2** *Left*: performance plot for five variants of our algorithm: **Basic** does not contain any optimizations; **BasicSym** enables symmetry breaking; **BasicSymSSol** additionally gives the input partitioning to the ILP solver. The two variants **BSSSSConst=** and **BSSSSConst<** are the same as **BasicSymSSol** with additional constraints: **BSSSSConst=** has the additional constraint that the objective has to be smaller or equal to the start solution, **BSSSSConst<** has the constraint that the solution must be better than the start solution. *Right*: performance of the slowest (**Basic**) and fastest ILPs (**BasicSymSSol**) depending on the number of non-zeros in the ILP.

obtained by starting depth-one breadth-first search at the 25 highest gain vertices, and set the limit on the non-zeros in the ILP to $\mathcal{N} = \infty$. However, due to preliminary experiments we expect the results in terms of speedup to be similar for different vertex selection strategies. To evaluate the ILP performance, we run KaFFPa using the strong preconfiguration on each of the graphs from set A using $\epsilon = 0$ and $k \in \{2, 4, 8, 16, 32, 64\}$ and then use the computed partition as input to each ILP (with the different optimizations). As the optimizations do not change the objective value achieved in the ILP, we only report running times of our different approaches. We set the time limit of the ILP solver to 30 minutes.

We use five variants of our algorithm: **Basic** does not contain any optimizations; **BasicSym** enables symmetry breaking; **BasicSymSSol** additionally gives the input partitioning to the ILP solver. The two variants **BSSSSConst=** and **BSSSSConst<** are the same as **BasicSymSSol** with additional constraints: **BSSSSConst=** has the additional constraint that the objective has to be smaller or equal to the start solution, **BSSSSConst<** has the constraint that the objective value of a solution must be better than the objective value of the start solution. Figure 2 summarises the results.

In our experiments, the basic configuration reaches the time limit in 95 out of the 300 runs. Overall, enabling symmetry breaking drastically speeds up computations. On all of the instances which the **Basic** configuration could solve within the time limit, each other configuration is faster than the **Basic** configuration. Symmetry breaking speeds up computations by a factor of 41 in the geometric mean on those instances. The largest obtained speedup on those instances was a factor of 5663 on the graph adaptive for $k = 32$. The configuration solves all but the two instances (boneS01, $k = 32$) and (Dubcova3, $k = 16$) within the time limit. Additionally providing the start solution (**BasicSymSSol**) gives an addition speedup of 22% on average. Over the **Basic** configuration, the average speedup is 50 with the largest speedup being 6495 and the smallest speedup being 47%. This configuration can solve all instances within the time limit except the instance boneS01 for $k = 32$. Providing the objective function as a constraint (or strictly smaller constraint) does not further reduce the running time of the solver. Instead, the additional constraints even increase the running time. We attribute this to the fact that the solver has to do additional work to evaluate the constraint. We conclude that **BasicSymSSol** is the fastest configuration of the ILP. Hence, we use this configuration in all the following experiments. Moreover, from Figure 2 we can see that this configuration can solve most of the instance within the time

■ **Table 2** From top to bottom: Number of improvements found by different vertex selection rules relative to the total number of instances, average running time of the strategy on the subset of instances (graph, k) in which all strategies finished within the time limit, and the relative number of instances in which the strategy computed the lowest cut. Best values are highlighted in bold.

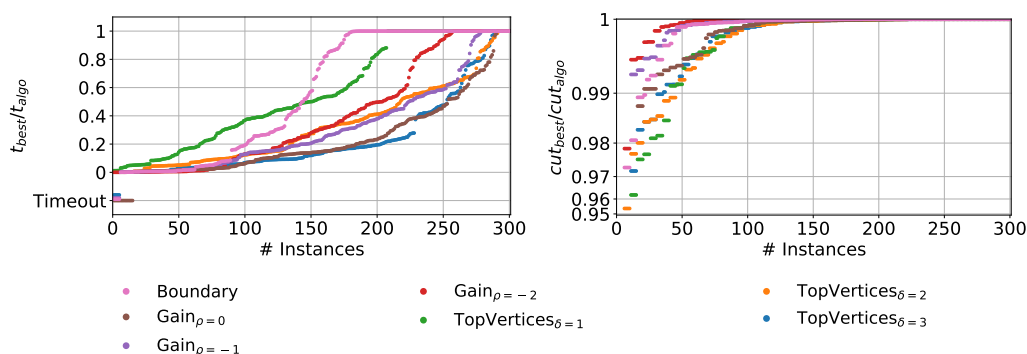
k	Gain			TopVertices			Boundary
	$\rho = 0$	$\rho = -1$	$\rho = -2$	$\delta = 1$	$\delta = 2$	$\delta = 3$	
	Relative Number of Improvements						
2	70%	70%	70%	50%	70%	70%	70%
4	50%	60%	80%	70%	70%	70%	80%
8	50%	60%	78%	60%	60%	60%	48%
16	30%	50%	70%	40%	30%	30%	40%
32	60%	60%	46%	50%	50%	20%	20%
64	70%	70%	50%	30%	20%	20%	0%
	Average Running Time						
2	189.943s	292.573s	357.145s	34.045s	61.152s	92.452s	684.198s
4	996.934s	628.950s	428.353s	87.357s	255.223s	558.578s	1467.595s
8	552.183s	244.470s	244.046s	105.737s	167.164s	340.900s	96.763s
16	118.532s	52.547s	90.363s	53.385s	141.814s	243.957s	34.790s
32	40.300s	24.607s	94.146s	27.156s	80.252s	116.023s	7.596s
64	15.866s	21.908s	24.253s	14.627s	30.558s	44.813s	4.187s
	Relative Number Best Algorithm						
2	20%	60%	50%	10%	10%	0%	60%
4	10%	0%	50%	10%	0%	0%	30%
8	0%	20%	30%	10%	10%	10%	26%
16	0%	10%	54%	10%	0%	10%	20%
32	0%	8%	38%	0%	0%	0%	4%
64	0%	16%	36%	0%	0%	0%	0%

limit if the number of non-zeros in the ILP is below 10^6 . Hence, we set the parameter \mathcal{N} to 10^6 in the following section.

5.3 Vertex Selection Rules

We now evaluate the vertex selection strategies to find the set of vertices \mathcal{K} that model the ILP. We look at all strategies described in Section 4.4, i.e. **Boundary**, **Gain** $_{\rho}$ with the parameter $\rho \in \{-2, -1, 0\}$ as well as **TopVertices** $_{\delta}$ for $\delta \in \{1, 2, 3\}$. To evaluate the different selection strategies, we use the best of five runs of KaFFPa strong on each of the graphs from set A using $\epsilon = 0$ and $k \in \{2, 4, 8, 16, 32, 64\}$ and then use the computed partition as input to the ILP (with different sets \mathcal{K}). Table 2 summarizes the results of the experiment, i.e. the number of cases in which our algorithm was able to improve the result, the average running time in seconds for these selection strategies as well as the number of cases in which the strategy computed the best result (the partition having the lowest cut). We set the time limit to 2 days to be able to finish almost all runs without running into timeout. For the average running time we exclude all graphs in which at least one algorithm did not finish in 2 days (rgg_15 $k = 16$, delaunay_n15 $k = 4$, G2_circuit $k = 4, 8$). If multiple runs share the best result, they are all counted. However, when no algorithm improves the input partition on a graph, we do not count them.

Looking at the number of improvements, the **Boundary** strategy is able to improve the input for small values of k , but with increasing number of blocks k improvements decrease to no improvement in all runs with $k = 64$. Because of the limit on the number of non-zeros, the ILP contains only random boundary vertices for large values of k in this case. Hence,



■ **Figure 3** *Left*: performance plot for all vertex selection strategies *Right*: cut value of vertex selection strategies in comparison to the best result given by any strategy.

there are not sufficiently many high gain vertices in the model and fewer improvements for large values of k are expected. For small values of $k \in \{2, 4\}$, the **Boundary** strategy can improve as many as the **Gain $_{\rho=-2}$** strategy but the average running times are higher.

For $k = \{2, 4, 8, 16\}$, the strategy **Gain $_{\rho=-2}$** has the highest number of improvements, for $k = \{32, 64\}$ it is surpassed by the strategy **Gain $_{\rho=-1}$** . However, the strategy **Gain $_{\rho=-2}$** finds the best cuts in most cases among all tested strategies. Due to the way these strategies are designed, they are able to put a lot of high gain vertices into the model as well as vertices that can be used to balance vertex movements. The **TopVertices** strategies are overall also able to find a large number of improvements. However, the found improvements are typically smaller than the **Gain** strategies. This is due to the fact that the **TopVertices** strategies grow BFS balls with a predefined depth around high gain vertices first, and later on are not able to include vertices that could be used to balance their movement. Hence, there are less potential vertex movements that could yield an improvement.

For almost all strategies, we can see that the average running time decreases as the number of blocks k increases. This happens because we limit the number of non-zeros \mathcal{N} in our ILP. As the number of non-zeros grows linearly with the underlying model size, the models are far smaller for higher values of k . Using symmetry breaking, we already fixed the block of the k vertices μ_i which represent the vertices not part of \mathcal{K} . Thus the ILP solver can quickly prune branches which would place vertices connected heavily to one of these vertices in a different block. Additionally, our data indicate that a large number of small areas in our model results faster in solve times than when the model contains few large areas. The performance plot in Figure 3 shows that the strategies **Boundary**, **TopVertices $_{\delta=1}$** and **Gain $_{\rho=-2}$** have lower running times than other strategies. These strategies all select a large number of vertices to initialize the breadth-first search. Therefore they output a vertex set \mathcal{K} that is the union of many small areas around these vertices. Variants that initialize the breadth-first search with fewer vertices have fewer areas, however each of the areas is larger.

5.4 Walshaw Benchmark

In this section, we present the results when running our best configuration on all graphs from Walshaw’s benchmark archive. Note that the rules of the benchmark imply that running time is not an issue, but algorithms should achieve the smallest possible cut value while satisfying the balance constraint. We run our algorithm in the following setting: We take existing partitions from the archive and use those as input to our algorithm. As indicated by the experiments in Section 5.3, the vertex selection strategies **Gain $_{\rho \in \{-1, -2\}}$** perform best

■ **Table 3** Relative number of improved instances in the Walshaw Benchmark starting from current entries reported in the Walshaw benchmark.

$k \setminus \epsilon$	0%	1%	3%	5%
2	6%	12%	6%	6%
4	18%	9%	6%	18%
8	26%	24%	12%	15%
16	50%	26%	29%	29%
32	62%	47%	47%	53%
64	68%	59%	71%	76%
overall	38%	29%	28%	33%

for different values of k . Thus we use the variant $\text{Gain}_{\rho=-2}$ for $k \leq 16$ and both $\text{Gain}_{\rho=-2}$ and $\text{Gain}_{\rho=-1}$ otherwise in this section. We repeat the experiment once for each instance (graph, k) and run our algorithm for $k = \{2, 4, 8, 16, 32, 64\}$ and $\epsilon \in \{0, 1\%, 3\%, 5\%\}$. For larger values of $k \in \{32, 64\}$, we strengthen our strategy and use $\mathcal{N} = 5 \cdot 10^6$ as a bound for the number of non-zeros. Table 3 summarizes the results. Detailed per-instance results are given in the full version of this paper [23].

When running our algorithm using the currently best partitions provided in the benchmark, we are able to improve 38% of the currently reported perfectly balanced results. We are able to improve a larger number of results for larger values of k , more specifically, out of the partitions with $k \geq 16$, we can improve 60% of all perfectly balanced partitions. This is due to the fact that the graph partitioning problem becomes more difficult for larger values of k . There is a wide range of improvements with the smallest improvement being 0.0008% for graph auto with $k = 32$ and $\epsilon = 3\%$ and with the largest improvement that we found being 1.72% for fe_body for $k = 32$ and $\epsilon = 0\%$. The largest absolute improvement we found is 117 for bcsstk32 with $k = 64$ and $\epsilon = 0\%$. In general, the total number of improvements is lower if some imbalance is allowed. This is also expected since traditional local search methods have a larger amount of freedom to move vertices. However, the number of improvements still shows that the method is also able to improve a many partitions even if some imbalance is allowed.

6 Conclusions and Future Work

We presented a novel meta-heuristic for the balanced graph partitioning problem. Our approach is based on an integer linear program that solves a model to combine unconstrained vertex movements into a global feasible improvement. Through a given input partition, we were able to use symmetry breaking and other techniques that make the approach scale to large inputs. In Walshaw’s benchmark, we were able to improve a large number of partitions.

We plan to further improve our implementation and integrate it into the KaHIP framework. We would like to look at other objective functions as long as they can be modelled linearly. Moreover, we want to investigate whether this kind of contractions can be useful for other ILPs. It may be interesting to find cores for contraction by using the information provided an evolutionary algorithm like KaFFPaE [35], i.e. if many of the individuals of the population of the evolutionary algorithm agree that two vertices should be put together in a block then those should be contracted in our model. Lastly, besides using other exact techniques like branch-and-bound to solve the model, it may also be worthwhile to use a heuristic algorithm instead.

References

- 1 C. J. Alpert and A. B. Kahng. Recent Directions in Netlist Partitioning: A Survey. *Integration, the VLSI Journal*, 19(1-2):1–81, 1995.
- 2 C. J. Alpert, A. B. Kahng, and S. Z. Yao. Spectral Partitioning with Multiple Eigenvectors. *Discrete Applied Mathematics*, 90(1):3–26, 1999.
- 3 M. Armbruster. *Branch-and-Cut for a Semidefinite Relaxation of Large-Scale Minimum Bisection Problems*. PhD thesis, 2007.
- 4 M. Armbruster, M. Fügenschuh, C. Helmberg, and A. Martin. A Comparative Study of Linear and Semidefinite Branch-and-Cut Methods for Solving the Minimum Graph Bisection Problem. In *Proc. of the 13th International Conference on Integer Programming and Combinatorial Optimization*, volume 5035 of *LNCS*, pages 112–124. Springer, 2008.
- 5 D. A. Bader, H. Meyerhenke, P. Sanders, C. Schulz, A. Kappes, and D. Wagner. Benchmarking for Graph Clustering and Partitioning. In *Encyclopedia of Social Network Analysis and Mining*, pages 73–82. Springer, 2014.
- 6 C. Bichot and P. Siarry, editors. *Graph Partitioning*. Wiley, 2011.
- 7 R. Brillout. A Multi-Level Framework for Bisection Heuristics. 2009.
- 8 T. N. Bui and C. Jones. Finding Good Approximate Vertex and Edge Partitions is NP-Hard. *Information Processing Letters*, 42(3):153–159, 1992.
- 9 A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. Recent Advances in Graph Partitioning. In *Algorithm Engineering*, pages 117–158. Springer, 2016.
- 10 T. Davis. The University of Florida Sparse Matrix Collection.
- 11 D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Customizable Route Planning. In *Proc. of the 10th International Symposium on Experimental Algorithms*, volume 6630 of *LCNS*, pages 376–387. Springer, 2011.
- 12 D. Delling, A. V. Goldberg, I. Razenshteyn, and R. F. Werneck. Exact Combinatorial Branch-and-Bound for Graph Bisection. In *Proc. of the 12th Workshop on Algorithm Engineering and Experimentation (ALENEX'12)*, pages 30–44, 2012.
- 13 D. Delling and R. F. Werneck. Better Bounds for Graph Bisection. In *Proc. of the 20th European Symposium on Algorithms*, volume 7501 of *LNCS*, pages 407–418, 2012.
- 14 A. Feldmann and P. Widmayer. An $O(n^4)$ Time Algorithm to Compute the Bisection Width of Solid Grid Graphs. In *Proc. of the 19th European Conference on Algorithms*, volume 6942 of *LNCS*, pages 143–154. Springer, 2011.
- 15 A. Felner. Finding Optimal Solutions to the Graph Partitioning Problem with Heuristic Search. *Annals of Mathematics and Artificial Intelligence*, 45:293–322, 2005.
- 16 C. E. Ferreira, A. Martin, C. C. De Souza, R. Weismantel, and L. A. Wolsey. The Node Capacitated Graph Partitioning Problem: A Computational Study. *Mathematical Programming*, 81(2):229–256, 1998.
- 17 C. M. Fiduccia and R. M. Mattheyses. A Linear-Time Heuristic for Improving Network Partitions. In *Proc. of the 19th Conference on Design Automation*, pages 175–181, 1982.
- 18 J. Fietz, M. Krause, C. Schulz, P. Sanders, and V. Heuveline. Optimized Hybrid Parallel Lattice Boltzmann Fluid Flow Simulations on Complex Geometries. In *Proc. of Euro-Par 2012 Parallel Processing*, volume 7484 of *LNCS*, pages 818–829. Springer, 2012.
- 19 P. Galinier, Z. Boujbel, and M. C. Fernandes. An Efficient Memetic Algorithm for the Graph Partitioning Problem. *Annals of Operations Research*, 191(1):1–22, 2011.
- 20 A. George. Nested Dissection of a Regular Finite Element Mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, 1973.
- 21 W. W. Hager, D. T. Phan, and H. Zhang. An Exact Algorithm for Graph Partitioning. *Mathematical Programming*, 137(1-2):531–556, 2013. doi:10.1007/s10107-011-0503-x.
- 22 M. Hein and S. Setzer. Beyond Spectral Clustering - Tight Relaxations of Balanced Graph Cuts. In *Advances in Neural Information Processing Systems*, pages 2366–2374, 2011.

- 23 A. Henzinger, A. Noe, and C. Schulz. ILP-based Local Search for Graph Partitioning. *arXiv preprint arXiv:1802.07144*, 2018.
- 24 S. E. Karisch, F. Rendl, and J. Clausen. Solving Graph Bisection Problems with Semidefinite Programming. *INFORMS Journal on Computing*, 12(3):177–191, 2000.
- 25 G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.
- 26 B. W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell System Technical Journal*, 49(1):291–307, 1970.
- 27 T. Kieritz, D. Luxen, P. Sanders, and C. Vetter. Distributed Time-Dependent Contraction Hierarchies. In *Proc. of the 9th International Symposium on Experimental Algorithms*, volume 6049 of *LNCS*, pages 83–93. Springer, 2010.
- 28 A. H. Land and A. G. Doig. An Automatic Method of Solving Discrete Programming Problems. *Econometrica*, 28(3):497–520, 1960.
- 29 U. Lauther. An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background, 2004.
- 30 A. Lisser and F. Rendl. Graph Partitioning using Linear and Semidefinite Programming. *Mathematical Programming*, 95(1):91–101, 2003. doi:10.1007/s10107-002-0342-x.
- 31 D. Luxen and D. Schieferdecker. Candidate Sets for Alternative Routes in Road Networks. In *Proc. of the 11th International Symposium on Experimental Algorithms (SEA'12)*, volume 7276 of *LNCS*, pages 260–270. Springer, 2012.
- 32 R. H. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm. Partitioning Graphs to Speedup Dijkstra's Algorithm. *Journal of Experimental Algorithmics (JEA)*, 11(2006), 2007.
- 33 F. Pellegrini. Scotch Home Page. <http://www.labri.fr/pelegrin/scotch>.
- 34 P. Sanders and C. Schulz. Engineering Multilevel Graph Partitioning Algorithms. In *Proc. of the 19th European Symp. on Algorithms*, volume 6942 of *LNCS*, pages 469–480. Springer, 2011.
- 35 P. Sanders and C. Schulz. Distributed Evolutionary Graph Partitioning. In *Proc. of the 12th Workshop on Algorithm Engineering and Experimentation (ALENEX'12)*, pages 16–29, 2012.
- 36 P. Sanders and C. Schulz. Think Locally, Act Globally: Highly Balanced Graph Partitioning. In *Proc. of the 12th Int. Symp. on Experimental Algorithms (SEA'13)*, LNCS. Springer, 2013.
- 37 K. Schloegel, G. Karypis, and V. Kumar. Graph Partitioning for High Performance Scientific Simulations. In *The Sourcebook of Parallel Computing*, pages 491–541, 2003.
- 38 C. Schulz and D. Strash. Graph Partitioning – Formulations and Applications to Big Data. In *Encyclopedia on Big Data Technologies*, 2018, to appear.
- 39 M. Sellmann, N. Sensen, and L. Timajev. Multicommodity Flow Approximation used for Exact Graph Partitioning. In *Proc. of the 11th European Symposium on Algorithms*, volume 2832 of *LNCS*, pages 752–764. Springer, 2003.
- 40 N. Sensen. Lower Bounds and Exact Algorithms for the Graph Partitioning Problem Using Multicommodity Flows. In *Proc. of the 9th European Symposium on Algorithms*, volume 2161 of *LNCS*, pages 391–403. Springer, 2001.
- 41 A. J. Soper, C. Walshaw, and M. Cross. A Combined Evolutionary Search and Multilevel Optimisation Approach to Graph-Partitioning. *Journal of Global Optimization*, 29(2):225–241, 2004.
- 42 C. Walshaw. Walshaw Partitioning Benchmark. <http://staffweb.cms.gre.ac.uk/~wc06/partition/>.


- 43 C. Walshaw and M. Cross. JOSTLE: Parallel Multilevel Graph-Partitioning Software – An Overview. In *Mesh Partitioning Techniques and Domain Decomposition Techniques*, pages 27–58. 2007.

Decision Diagrams for Solving a Job Scheduling Problem Under Precedence Constraints

Kosuke Matsumoto

Kyushu University
matumoabyss@gmail.com

Kohei Hatano

Kyushu University / RIKEN AIP
hatano@inf.kyushu-u.ac.jp
 <https://orcid.org/0000-0002-1536-1269>

Eiji Takimoto

Kyushu University
eiji@inf.kyushu-u.ac.jp

Abstract

We consider a job scheduling problem under precedence constraints, a classical problem for a single processor and multiple jobs to be done. The goal is, given processing time of n fixed jobs and precedence constraints over jobs, to find a permutation of n jobs that minimizes the total flow time, i.e., the sum of total wait time and processing times of all jobs, while satisfying the precedence constraints. The problem is an integer program and is NP-hard in general. We propose a decision diagram π -MDD, for solving the scheduling problem exactly. Our diagram is suitable for solving linear optimization over permutations with precedence constraints. We show the effectiveness of our approach on the experiments on large scale artificial scheduling problems.

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

Keywords and phrases decision diagram, permutation, scheduling, NP-hardness, precedence constraints, MDD

Digital Object Identifier 10.4230/LIPIcs.SEA.2018.5

Supplement Material Source codes are available at https://bitbucket.org/kohei_hatano/pimdd/.

Funding This work is supported in part by JSPS KAKENHI Grant Number JP16K00305 and JSPS KAKENHI Grant Number JP15H02667, respectively.

Acknowledgements We thank Fumito Miyake for insightful discussions and anonymous reviewers for helpful comments.

1 Introduction

Scheduling problems are typical problems which are known to be NP-hard in general. Hence, practical fast approximate solvers for scheduling are quite useful in practice. Among them, the job scheduling problem of a single machine with precedence constraints is a classical one, where, given n fixed jobs and their processing times, as well as precedence constraints over jobs (i.e., job i must be done prior to job j), the task is to find a permutation of n jobs (a schedule) which minimizes the sum of processing times and wait times (called flow time) of all jobs among those permutations satisfying precedence constraints. This



© Kosuke Matsumoto, Kohei Hatano, and Eiji Takimoto;
licensed under Creative Commons License CC-BY
17th International Symposium on Experimental Algorithms (SEA 2018).

Editor: Gianlorenzo D'Angelo; Article No. 5; pp. 5:1–5:12



Leibniz International Proceedings in Informatics

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

problem is NP-hard [15, 16] as well and 2-approximation polynomial time algorithms are known [24, 11, 5, 17, 4]. For further details, see, e.g., [8, 2]. On the other hand, practical exact algorithms are quite non-trivial to obtain; Naive algorithms using integer programming solvers still take prohibitive time.

BDDs (Binary Decision Diagrams) [1, 3] and ZDDs (Zero Suppressed BDDs) [20, 21, 14] are data structures which represent sets of binary vectors (or sets of fixed objects). BDDs/ZDDs can compress sets succinctly and various functions over sets (such as union and intersection) are efficiently computed using the data structures. In particular, ZDDs are suitable for representing sparse sets and often advantageous in practice (see, e.g., [22, 12]). A variant of ZDDs called π DDs are specially designed for representing permutations [23]. The structure is suitable for counting or enumeration, but not designed for optimization. In addition, we are not aware of other non-trivial applications of BDDs/ZDDs for the scheduling problem with precedence constraints. MDDs (Multiple-Valued Decision Diagrams)[19] are of variants of BDDs which can treat multiple values naturally and applications of MDDs to scheduling problems are known [6, 7]. The scheduling problems considered are different from ours and thus are not applicable.

In this paper, we propose a data structure π -MDD, which directly represents a set of permutations over $\{1, \dots, n\}$ ¹. A π -MDD is a DAG and each path in the DAG represents a permutation. Using the data structure, we show an exact optimization scheme for the job scheduling problem under precedence constraints. More specifically, our scheme consists of the following two parts.

- (i) We propose an algorithm which, when given a set of precedence constraints represented by a DAG as input, constructs a π -MDD representing permutations satisfying the precedence constraints in output-linear time. We show that the size of the π -MDD made by the algorithm is $O(h(G)(n/h(G) + 1)^{h(G)})$, where G is the DAG representing precedence constraints and $h(G)$ is the width of the graph.
- (ii) Given a π -MDD which represents a set of permutations, and processing times of jobs, we show a method for finding a permutation π optimizing the flow time among the set which is a linear optimization over the permutations in the set. Like BDDs/ZDDs, linear optimization of the set of interest can be reduced to the shortest path problem over the corresponding MDD which represents the set. Thus the computation time for the optimization is linear in the size of the π -MDD.

A potential advantage of our method (and other BDDs/ZDDs/MDDs based approaches) over naive integer-programming based methods is that once we construct a π -MDD representing permutations satisfying precedence constraints, we can re-use it for different cost criteria without reconstructing π -MDDs. This advantage is crucial for (i) the case where several different cost criteria are considered and (ii) a repeated game version of the job scheduling under fixed precedence constraints under uncertainty (see, e.g., [9]).

In our preliminary experiments over large artificial data sets of job scheduling under precedence constraints, our method outperforms naive methods based on the integer programming, especially when there are more precedence constraints.

1.1 Related Work

Note that the data structure π -MDD is a special case of MDDs and not new itself. Furthermore, the structure of π -MDD is quite similar to ones used in the previous work of Hadzic et al. [10] and Ciré and van Hoesve [6, 7]. However, their approach to construct the data

¹ More precisely, π -MDDs can deal with permutations over n fixed different numbers.

structure is totally different from ours. Their approach is to construct a relaxed MDD which represents a super set of the feasible solutions first and to solve the problem by refining the MDD as well as filtering infeasible solutions. On the other hand, our approach directly constructs the exact set of feasible solutions.

The technical contribution of the paper is not to derive a new data structure, but to derive an efficient construction method of π -MDDs satisfying precedence constraints as well as an efficient exact optimization of the job scheduling problem using the structure.

2 Preliminaries

2.1 Notations

Let $[n] = \{1, 2, \dots, n\}$ be the set of integers $1, \dots, n$. A permutation π over $[n]$ is a bijection from $[n]$ to $[n]$. Each permutation π can be represented as the corresponding vector $\boldsymbol{\pi} = (\pi(1), \dots, \pi(n))$. For convenience, for each $i \in [n]$, let π_i be the i -th element of $\boldsymbol{\pi}$, and π_i^{-1} be the position of element i , respectively. Let $S_{[n]}$ be the set of permutations over $[n]$. For example, $S_{[3]} = \{(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)\}$.

A directed graph (DAG) $G = (V, E)$ is a pair, where V is the set of nodes and $E \subseteq V \times V$ is the set of directed edges in which there is no directed cycle, i.e., there is no sequence $(v_1, v_2), (v_2, v_3) \dots, (v_k, v_{k+1}) \in E^k$ with $v_1 = v_{k+1}$.

Let d_v^+ denote the out-degree of node v in V , that is $d_v^+ = |\{v' \in V \mid (v, v') \in E\}|$. We say that a node $v \in V$ is reachable from $v' \in V$ if there is a directed path starting from v' ending at v , i.e., a sequence of directed edges $(v', v_1), (v_1, v_2), \dots, (v_{k-1}, v_k), (v_k, v) \in E$. For a DAG $G = (V, E)$, the width $h(G)$ denote the maximum size of the set $V' \subseteq V$ where each pair of nodes are not reachable from each other. A partially ordered set (poset) (P, R) is a pair, where P is a set and $R \subset P \times P$ is a binary relation satisfying reflexivity ($\forall a \in P, aRa$), transitivity ($\forall a, b, c \in P, aRb$ and bRc implies aRc), and antisymmetry ($\forall a, b \in P, aRb$ and bRa implies $a = b$). A poset (P, R) can be viewed as a DAG $G = (V, E)$ with $V = P$ and $R = E$ and known as a Hasse diagram.

Let $S_V(G)$ denote the set of permutations over $V \subseteq [n]$ satisfying the precedence constraints corresponding to the DAG $G = (V, E)$ and is defined as

$$S_V(G) = \{\boldsymbol{\pi} \in S_V \mid \forall (v, v') \in E \ \pi_v < \pi_{v'}\}.$$

Similarly, let $S_V^{-1}(G)$ the set of inverses of permutations in $S_V(G)$, i.e.,

$$S_V^{-1}(G) = \{\boldsymbol{\pi}^{-1} \in S_V \mid \boldsymbol{\pi} \in S_V(G)\}$$

Note that, by definition of the inverse, $S_V^{-1}(G) = \{\boldsymbol{\pi} \in S_V \mid \forall (v, v') \in E \ \pi_v^{-1} < \pi_{v'}^{-1}\}$. We will make use of this property extensively in later discussions.

2.2 The job scheduling problem under precedence constraints

We consider the job scheduling problem of n jobs with a single machine under the precedence constraints given as a DAG $G = ([n], E)$. Given processing times of jobs represented as a vector $\boldsymbol{w} \in \mathbb{R}^n$ and the precedence constraints G , the task is to find a permutation over the set $[n]$ of jobs minimizing the sum of flow times (the sum of processing time and wait time) of all jobs. For example, when $n = 4$, jobs 3, 2, 4, 1 are done successively, flow times of these jobs are $w_3, w_3 + w_2, w_3 + w_2 + w_4, w_3 + w_2 + w_4 + w_1$, respectively and the sum of flow times is $4w_3 + 3w_2 + 2w_4 + w_1$. If we represent the schedule as a permutation $\boldsymbol{\pi} = (1, 3, 4, 2)$ (each component i can be viewed as its priority), the sum of flow times of the schedule is exactly

the inner product $\boldsymbol{\pi} \cdot \boldsymbol{w}$. This relationship holds in general. That is, a permutation $\boldsymbol{\pi} \in S_{[n]}$ represents a schedule where the priority of job i is π_i (i.e., the job is the $(n + 1 - \pi_i)$ -th to be done) and the sum of flow times is $\boldsymbol{\pi} \cdot \boldsymbol{w}$.

Now we define the job scheduling problem under the precedence constraints represented as a DAG $G = ([n], E)$ and $\boldsymbol{w} \in \mathbb{R}^n$ as the following linear optimization problem:

$$\begin{aligned} \text{Input} &: \text{ DAG } G = ([n], E), \boldsymbol{w} \in \mathbb{R}^n \\ \text{Output} &: \boldsymbol{\pi}^* = \arg \min_{\boldsymbol{\pi} \in S_{[n]}(G)} \boldsymbol{\pi} \cdot \boldsymbol{w} \end{aligned} \quad (1)$$

The problem can be formulated as an integer program where variables takes values in $[n]$ and it is NP-hard [15, 16]. We will solve this problem exactly using a new data structure later.

This problem can be reduced to the 0-1 integer programs in two ways. The first reduction represents a permutation as a permutation matrix as follows: Given a permutation $\boldsymbol{\pi} \in S_{[n]}$, the corresponding permutation matrix $X \in \{0, 1\}^{n \times n}$ is defined as $X_{i,j} = 1$ if $\pi_j = i$ and otherwise $X_j = 0$ for each $j \in [n]$. For example, for $\boldsymbol{\pi} = (2, 3, 1)$, the permutation matrix X is

$$X = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}.$$

When we represent permutations as permutation matrices and we are given $\boldsymbol{w} \in \mathbb{R}^n$, let

$$W = \begin{bmatrix} w_1 & w_2 & \cdots & w_n \\ 2w_1 & 2w_2 & \cdots & 2w_n \\ \vdots & \vdots & \ddots & \vdots \\ nw_1 & nw_2 & \cdots & nw_n \end{bmatrix}.$$

Then the problem can be given as the following integer program.

$$\begin{aligned} \text{minimize}_{X \in \{0,1\}^{n \times n}} & \sum_{i=1}^n \sum_{j=1}^n X_{i,j} W_{i,j} \\ \text{subject to} & \end{aligned} \quad (2a)$$

$$\forall i \in [n] \quad \sum_{j=1}^n X_{i,j} = 1 \quad (2b)$$

$$\forall j \in [n] \quad \sum_{i=1}^n X_{i,j} = 1 \quad (2c)$$

$$\forall (v, v') \in E \quad \forall i \in [n] \quad \sum_{j=i}^n X_{j,v} \leq \sum_{j=i}^n X_{j,v'} \quad (2d)$$

This formulation has n^2 variables and $n(|E| + 2)$ linear constraints.

The second reduction to an 0-1 integer program uses a comparison matrix as a representation of a permutation. A comparison matrix $Y \in \{0, 1\}^n$ corresponding to a permutation $\boldsymbol{\pi}$ is defined as

$$Y_{i,j} = \begin{cases} 0 & (\pi_i > \pi_j) \\ 1 & (\pi_i \leq \pi_j) \end{cases} \quad (3)$$

By adopting the comparison matrix representation, the scheduling problem is given as the following integer program with the input

$$W = \begin{bmatrix} w_1 & w_2 & \cdots & w_n \\ w_1 & w_2 & \cdots & w_n \\ \vdots & \vdots & \ddots & \vdots \\ w_1 & w_2 & \cdots & w_n \end{bmatrix}.$$

$$\text{minimize}_{Y \in \{0,1\}^{n \times n}} \sum_{i=1}^n \sum_{j=1}^n Y_{i,j} W_{i,j}$$

subject to

$$\forall i, j, k \in [n] \quad 1 \geq Y_{i,j} - Y_{i,k} + Y_{j,k} \geq 0 \quad (4a)$$

$$\forall i, j \in [n] \quad \begin{cases} Y_{i,j} + Y_{j,i} = 1 & (i \neq j) \\ Y_{i,j} = 1 & (i = j) \end{cases} \quad (4b)$$

$$\forall (v, v') \in E \quad Y_{v,v'} = 1 \quad (4c)$$

The optimum of the scheduling problem is the permutation represented by the solution Y . This problem has n^2 variables and $2n^3 + n(n+1)/2 + |E|$ constraints. This formulation is well-known in the scheduling literature. For details, see, e.g., the result of Chudak and Hochbaum [5].

3 π -MDD

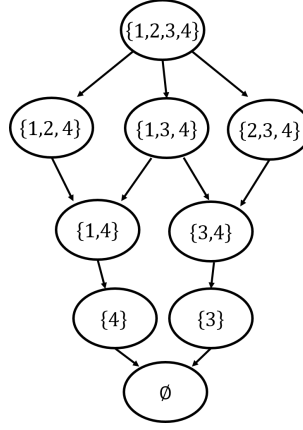
We propose π -MDD, a variant of MDD (Multiple-Valued Decision Diagram[18][19]). An MDD is a data structure representing a set of vectors, while a π -MDD represents a set of permutation vectors.

A π -MDD $D = (V_D, E_D)$ over $N \subset \mathbb{N}$, $|N| = n$ is a DAG with the root node \mathbf{r} whose in-degree is 0 and the terminal node \mathbf{t} whose out-degree is 0, where $V_D \subseteq 2^N$ consists of sets of nodes. Each node of V_D corresponds to a subset of N . In particular, $\mathbf{r} = N$ and $\mathbf{t} = \emptyset$. The structure of a π -MDD has several layers. At the first layer, only the root $\mathbf{r} = N$ exists and i -th layer consists of nodes u with size $|u| = n - i + 1$ ($i = 1, \dots, n$). Edges of a π -MDD appear only between consecutive layers. More precisely, $(u, u') \in E_D$ if and only if $u' \subset u$ and u and u' differ in exactly one element. Each path from the root \mathbf{r} to the terminal \mathbf{t} has length exactly n and each node u in V_D with distance i from \mathbf{t} corresponds to a set of size i , that is $|u| = i$. Let P_D be the set of all paths from the root $\mathbf{r} = N$ to the terminal $\mathbf{t} = \emptyset$. For convenience, we sometimes regard a path in P_D as a sequence of vertices along the directed path. That is, $P_D \subset (2^N)^{n+1}$. Figure 1 illustrates a π -MDD. Given a π -MDD D , a path $\mathbf{p} = (p_1, \dots, p_{n+1}) \in P_D$ defines a permutation $\pi_{\mathbf{p}}$ as follows:

$$\pi_{\mathbf{p}} = (\pi_{\mathbf{p},1}, \dots, \pi_{\mathbf{p},n}) \text{ s.t. } \pi_{\mathbf{p},i} \in p_{n-i+1} \setminus p_{n-i+2} \quad (i = 1, \dots, n).$$

Note that the set $p_{n-i+1} \setminus p_{n-i+2}$ is a singleton and thus the definition is well-defined. For example, in Figure 1, for the path $\mathbf{p} = (\{1, 2, 3, 4\}, \{2, 3, 4\}, \{3, 4\}, \{3\}, \emptyset)$, the corresponding permutation $\pi_{\mathbf{p}}$ is $(3, 4, 2, 1)$. Similarly, two sets of permutations associated with a π -MDD D over $N \subset \mathbb{N}$ are defined as follows:

$$\Pi(D) = \{\pi_{\mathbf{p}} \mid \mathbf{p} \in P_D\}, \text{ and } \Pi^{-1}(D) = \{\pi^{-1} \mid \pi \in \Pi(D)\}.$$



■ **Figure 1** An illustration of a π -MDD D .

For the π -MDD in Figure 1, $\Pi(D) = \{(4, 1, 2, 3), (4, 1, 3, 2), (3, 4, 1, 2), (3, 4, 2, 1)\}$, and $\Pi^{-1}(D) = \{(2, 3, 4, 1), (2, 4, 3, 1), (3, 4, 1, 2), (4, 3, 1, 2)\}$. $S_{\pi}^{-1}(D) S_{\pi}(D)$, respectively.

Now we propose a method to solve the scheduling problem using π -MDDs. The method consists of two parts.

1. Given a DAG $G = ([n], E)$ which represents precedence constraints, construct a π -MDD D such that $\Pi(D) = S_{[n]}^{-1}(G)$. That is, the π -MDD D represents inverses of permutations satisfying the constraints.
2. Given the π -MDD D over $[n]$ and a weight vector $\mathbf{w} \in \mathbb{R}^n$, solve

$$\boldsymbol{\pi} = \arg \min_{\boldsymbol{\pi} \in S_{[n]}(G)} \boldsymbol{\pi} \cdot \mathbf{w}.$$

3.1 Construction of a π -MDD

In this subsection, we consider the following problem:

Input : DAG $G = ([n], E)$

Output : π -MDD D s.t. $\Pi(D) = S_{[n]}^{-1}(G)$.

Let $G(V')$ denote the subgraph of G induced by the vertex subset V' , that is, $G(V') = (V', E')$, where $E' = \{(v, v') \in V' \mid (v, v') \in E\}$. Also, let $E(V') = \{(v, v') \in V' \mid (v, v') \in E\}$. First, we describe the algorithm Make π -MDD in Algorithm 1.

The algorithm Make π -MDD recursively constructs a π -MDD from the root node $\mathbf{r} = [n]$ to the terminal node $\mathbf{t} = \emptyset$. For any $\boldsymbol{\pi} \in S_V$ and an integer q , we denote $\boldsymbol{\pi}q$ as $\boldsymbol{\pi}q = (\pi_1, \dots, \pi_{|V|}, q)$. For any set $Y \subseteq \mathbb{R}^n$ and any real number $y \in \mathbb{R}$, let $Y \times y = \{(\mathbf{y}, y) \in \mathbb{R}^{n+1} \mid \mathbf{y} \in Y\}$. Then we prove an important property of $S_V^{-1}(G)$.

► **Lemma 1.** For a DAG $G = (V, E)$ and $Q = \{q \in V \mid d_q^+ = 0\}$,

$$S_V^{-1}(G) = \bigcup_{q \in Q} S_{V \setminus \{q\}}^{-1}(G(V \setminus \{q\})) \times q. \quad (5)$$

Algorithm 1 Make π -MDD.

Require: DAG $G = (V, E)$, where $V \subseteq [n]$

- 1: **if** $V = \emptyset$ **then**
- 2: **return** node \emptyset
- 3: **else if** have never memorized the π -MDD D_G for G **then**
- 4: π -MDD $D \leftarrow (V_D, E_D)$ with $V_D = \{V\}$ and $E_D = \emptyset$.
- 5: **for** each $v \in V$ whose out-degree is 0 in $G = (V, E)$ **do**
- 6: $V' \leftarrow V \setminus \{v\}$
- 7: π -MDD $D' \leftarrow \text{Make}\pi\text{-MDD}(G(V'))$ // root node is V'
- 8: $D \leftarrow D$ with D' and edge (V, V')
- 9: **end for**
- 10: memorize D as D_G
- 11: **end if**
- 12: **return** D_G

Proof. For any fixed $q \in Q$,

$$\begin{aligned}
& \pi q \in S_{V \setminus \{q\}}^{-1}(G(V \setminus \{q\})) \times q \\
\Leftrightarrow & \pi q \in S_V, \forall (v, v') \in E(V \setminus \{q\}), \pi_v^{-1} \leq \pi_{v'}^{-1} \\
& \quad \text{(by definition of } S_V^{-1}(G) \text{ and its property)} \\
\Leftrightarrow & \pi' = \pi q \in S_V, \forall (v, v') \in E(V \setminus \{q\}) \cup \{(i, q) \in V^2 \mid i \in V \setminus \{q\}\}, \pi_v'^{-1} \leq \pi_v^{-1} \\
& \quad \text{(since } \pi_{|V|} = q \Leftrightarrow \pi_q'^{-1} = |V| \text{)} \\
\Rightarrow & \pi' = \pi q \in S_V, \forall (v, v') \in V, \pi_v'^{-1} \leq \pi_v^{-1} = S_V^{-1}(G) \\
& \quad \text{(since } E \subseteq E(V \setminus \{q\}) \cup \{(i, q) \in V^2 \mid i \in V \setminus \{q\}\}\text{),}
\end{aligned}$$

which implies that $\cup_{q \in Q} S_{V \setminus \{q\}}^{-1}(G(V \setminus \{q\})) \times q \subseteq S_V^{-1}(G)$.

For the opposite direction, let π be any member of $S_V^{-1}(G)$. Then, by definition, for any $(v, v') \in E$, it holds that $\pi_v^{-1} \leq \pi_{v'}^{-1}$. Let $q = \pi_{|V|}$. Then, we have $\pi_q^{-1} = |V| > \pi_v^{-1}$ for any $v \in V \setminus \{q\}$. Therefore, there is no out-going edge from q (if exists, it implies a contradiction) and thus $q \in Q$. Together with the fact that $\pi \in S_{V \setminus \{q\}}^{-1}(G(V \setminus \{q\})) \times q$, the opposite direction also holds. \blacktriangleleft

The following lemma holds for Make π -MDD. Now we describe an important relationship between the output of Make π -MDD and the input DAG $G = (V, E)$. Roughly speaking, the π -MDD made by the algorithm represents a set of inverses of permutations satisfying the precedence constraints.

► **Lemma 2.** *Make π -MDD constructs a π -MDD D such that $S_\pi(D) = S_V^{-1}(G)$.*

Proof. The proof is done by induction on the size $k = |V|$ in the input DAG $G = (V, E)$. (i) For $|V|=0$, Make π -MDD outputs the π -MDD $D = (V_D, E_D)$ with $V_D = \{\emptyset\}$ and $E = \emptyset$. Thus the statement is true. (ii) For $|V| = k$, assume that the statement is true. Let $D_{G(V')}$ be the π -MDD made by Make π -MDD($G(V')$). Then,

$$\begin{aligned}
\Pi(D) &= \bigcup_{q \in Q} (\Pi(D_{G(V \setminus \{q\})}) \times q) \\
&= \bigcup_{q \in Q} (S_{V \setminus \{q\}}^{-1}(G(V \setminus \{q\})) \times q) \text{ (by the inductive assumption)} \\
&= \bigcup_{q \in Q} (\{\pi \in S_{V \setminus \{q\}} \mid \forall (v, v') \in E(V \setminus \{q\}) \ \pi_v^{-1} < \pi_{v'}^{-1}\} \times q) \\
&= S_V^{-1}(G) \text{ (by Lemma 1),} \tag{6}
\end{aligned}$$

which completes the inductive proof. \blacktriangleleft

► **Corollary 3.** For the output D of $\text{Make}\pi\text{-MDD}(G)$ with $G = ([n], E)$, $\Pi^{-1}(D) = S_{[n]}(G)$.

The size of the π -MDD can be bounded based on the result of Inoue and Minato [13].

► **Lemma 4.** For a DAG $G = (V, E)$, let $h(G)$ be the width of G . Then, the size $|D|$ of π -MDD D obtained from $\text{Make}\pi\text{-MDD}(G = (V, E))$ is

$$|D| = O(h(G)(n/h(G) + 1)^{h(G)}).$$

Proof. Let $IS(G)$ be the set of DAGs which can be constructed by updating $G = G(V \setminus \{v \in V \mid d^+(v) = 0\})$ recursively. The total number of recursions in $\text{Make}\pi\text{-MDD}$ is at most $|IS(G)|$. It is known that the size $|IS(G)|$ is at most $(n/h(G) + 1)^{h(G)}$ [13]. Since any pair of elements in the set $Q = \{v \in V' \mid d_v^+ = 0\}$ are not reachable to each other, $|Q| \leq h(G)$. Therefore, at each recursion, at most $h(G)$ edges are added, and thus the total edges in the π -MDD made by the algorithm is $O(h(G)(n/h(G) + 1)^{h(G)})$. \blacktriangleleft

3.2 Optimization over a π -MDD

We describe how to find an optimal solution of the scheduling problem (1) using a π -MDD. More precisely, we deal with the following optimization problem.

$$\begin{aligned}
&\text{Input : } \pi\text{-MDD } D = (V_D, E_D) \text{ s.t. } \Pi(D) = S_{[n]}^{-1}(G) \text{ for some DAG } G, \text{ and } \mathbf{w} \in \mathbb{R}^n \\
&\text{Output : } \boldsymbol{\pi} = \arg \min_{\boldsymbol{\pi} \in S_{[n]}(G)} \boldsymbol{\pi} \cdot \mathbf{w}
\end{aligned}$$

We solve this problem by reducing it to the shortest path problem over the π -MDD D . The reduction is as follows: For each edge $(u, u') \in E_D$, we set the cost $L_{(u, u')}$ as

$$L_{(u, u')} = |u|w_{u \setminus u'}.$$

The cost $L_{\mathbf{p}}$ of each path $\mathbf{p} \in P_D$ is defined as $L_{\mathbf{p}} = \sum_{i \in [n]} L_{(p_i, p_{i+1})}$. Then we consider the shortest path problem over D from the root $\mathbf{r} = [n]$ to the terminal $\mathbf{t} = \emptyset$ with cost $L_{(v, v')}$ for each edge $(v, v') \in E_D$. This problem is can be solved in time $O(|D|)$. We prove the following relationship between the cost of each path $\mathbf{p} \in P_D$ and its corresponding permutation.

► **Lemma 5.** For any $\mathbf{p} \in P(D)$,

$$L_{\mathbf{p}} = \boldsymbol{\pi}_{\mathbf{p}}^{-1} \cdot \mathbf{w}.$$

Proof.

$$\begin{aligned}
L_{\mathbf{p}} &= \sum_{i \in [n]} L_{(p_i, p_{i+1})} \\
&= |p_1|w_{p_1 \setminus p_2} + \cdots + |p_n|w_{p_n \setminus p_{n+1}} \\
&= nw_{p_1 \setminus p_2} + \cdots + w_{p_n \setminus p_{n+1}} \\
&= nw_{\pi_{\mathbf{p}, n}} + \cdots + w_{\pi_{\mathbf{p}, 1}} \\
&= \sum_{i \in [n]} iw_{\pi_{\mathbf{p}, i}} \\
&= \sum_{i \in [n]} i \sum_{j \in [n]} \mathbb{1}[j = \pi_{\mathbf{p}, i}]w_j \\
&= \sum_{i \in [n]} i \sum_{j \in [n]} \mathbb{1}[\pi_{\mathbf{p}, j}^{-1} = i]w_j \\
&= \sum_{j \in [n]} \sum_{i \in [n]} \mathbb{1}[\pi_{\mathbf{p}, j}^{-1} = i]iw_j \\
&= \sum_{j \in [n]} \pi_{\mathbf{p}, j}^{-1} w_j \\
&= \boldsymbol{\pi}_{\mathbf{p}}^{-1} \cdot \mathbf{w}
\end{aligned}$$

By combining Lemma 2, 4, 5, we obtain the main result.

► **Theorem 6.** *There is an algorithm that, given a DAG $G = ([n], E)$ as precedence constraints, computes a solution of problem (1) in time $O(h(G)(n/h(G) + 1)^{h(G)})$.*

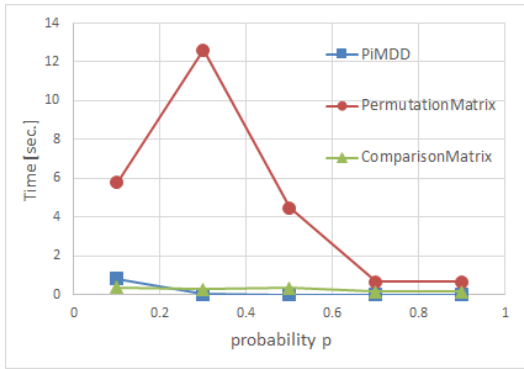
Proof. By Lemma 2, given a DAG $G = ([n], E)$, Make π -MDD constructs a π -DD D such that $\Pi(D) = S_{[n]}^{-1}(G)$. By Lemma 5, we can compute the linear optimization problem over $\Pi^{-1}(D) = S_{[n]}(G)$. By Lemma 4, both constructing a π -MDD and solving the shortest path problem take time $O(h(G)(n/h(G) + 1)^{h(G)})$. ◀

4 Experimental Results

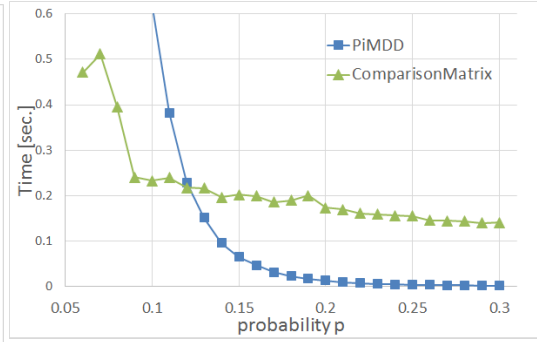
In this section, we show experimental results on artificial data. We construct the artificial data sets of the scheduling problem with precedence constraints by generating DAGs and weight vectors randomly. More precisely, given $V = [n]$, for each $(v_i, v_j) \in V \times V (v_i < v_j)$, we assign the edge $(v_i, v_j) \in E$ with probability p ($0 < p < 1$). Note that, because of the constraint that $v_i < v_j$, the resulting random graph is a DAG. We generate a random weight vector by generating each w_i according to the uniform distribution over $[0, 1]$ for $i \in [n]$. We use the parameters $n = 25$, and $p \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$. We compare the proposed methods with π -MDD, and integer programming (IP) with permutation matrices and comparison matrices, respectively. We implemented these methods in C++ and used the Gurobi optimizer 6.5.0 to solve integer programs. We run them in a machine with Intel(R) Xeon(R) CPU X5560 2.80GHz and 198GB memory.

Figure 2 shows the computation times of each method for different choices of p . The shown results are obtained by averaging over 500 random instances for each fixed choice of p .

The proposed method is fastest among others when $p > 0.15$, i.e., precedence constraints are not sparse. (Figure3 shows the detailed results). In particular, for $p \geq 0.2$, the speed up by the proposed method is 10 times w.r.t. the IP with comparison matrices and more than 20 times w.r.t. the IP with permutation matrices. Also, as can be seen in Figure 2, computation



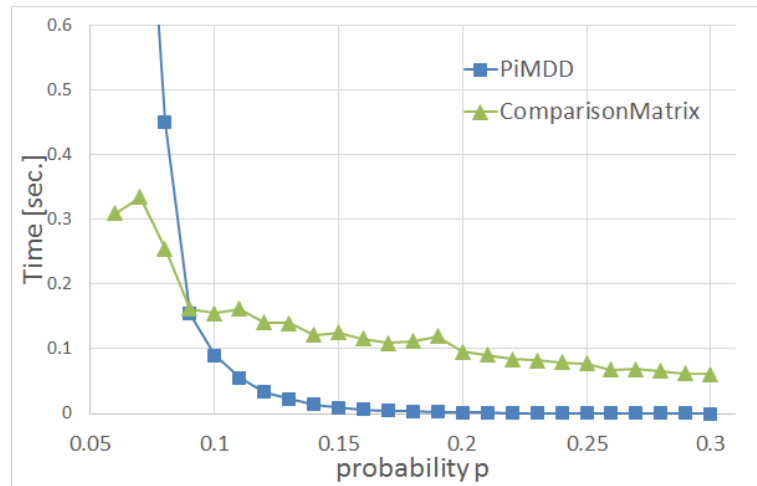
■ **Figure 2** Average running time for $n = 25$.



■ **Figure 3** Average running time for $n = 25$ in details.

■ **Table 1** Average number of constraints for $n = 25$.

	$k = 0.1$	$k = 0.3$	$k = 0.5$	$k = 0.7$	$k = 0.9$
permutation matrix	799.3	2304.8	3801.9	5294.0	6803.4
comparison matrix	31580.0	31640.2	31700.1	31759.8	31820.1



■ **Figure 4** Average computation time for optimization for $n = 25$.

time of the IP with permutation matrices is much larger than that of the IP with comparison matrices by 10 times for $p \leq 0.5$. In contrast, as in Table 1, the number of constraints used in IP with comparison matrices is much larger. So, the number of constraints does not seem to affect the computation time.

In general, if the DAG $G = (V, E)$ is sparse, the width of G tends to be larger. In fact, for $p = 0.1$, the average width of the random graph is 11.808, while for $p = 0.5$, the average width is 3.596. Note that the worst case time complexity bound $O(h(n/h + 1)^h)$ for constructing a π -MDD depends on the width h .

Next, for sparse precedence constraints, we compare performances of our π -MDD based method and the IP with comparison matrices. For $n = 25$, and $p \in \{0.06, 0.07, \dots, 0.30\}$, we generate 500 random DAGs for precedence constraints, and run these algorithms for each random instance. Figure3 shows averaged results. The average running time of the proposed method becomes smaller than the IP-based method for $p \geq 0.13$.

Now we compare computation times of these methods for optimization only. Here we separate computation time into preprocessing time and time for optimization. For our π -MDD based method, time for constructing a π -MDD is for preprocessing and solving the shortest path problem over the π -MDD corresponds to the optimization part. For IP with comparison matrices, we regard time for constructing a problem instance (e.g., adding constraints) as preprocessing time.

Figure 4 shows computation times of the proposed method and the IP with comparison matrices for optimization only. For optimization, the proposed method is faster than the IP for $p \geq 0.09$. This result indicates that the proposed method is better suited when we solve scheduling problems with the same precedence constraints and different weight vectors.

References

- 1 Sheldon B. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, 1978.
- 2 Christoph Ambühl, Monaldo Mastrolilli, Nikolaus Mutsanas, and Ola Svensson. On the Approximability of Single-Machine Scheduling with Precedence Constraints Christoph Ambühl. *Mathematics of Operations Research*, 36(4):653–669, 2011.
- 3 Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, aug 1986.
- 4 Chandra Chekuri and Rajeev Motwani. Precedence constrained scheduling to minimize sum of weighted completion times on a single machine. *Discrete Applied Mathematics*, 98(1-2):29–38, 1999.
- 5 Fabian A. Chudak and Dorit. S. Hochbaum. A half-integral linear programming relaxation for scheduling precedence-constrained jobs on a single machine. *Operations Research Letters*, 25:199–204, 1999.
- 6 André A Ciré and Willem Jan van Hoeve. MDD Propagation for Disjunctive Scheduling. In *Proceedings of the 22nd International Conference on Automated Planning and Scheduling (ICAPS'12)*, 2012.
- 7 André A Ciré and Willem-Jan van Hoeve. Multivalued Decision Diagrams for Sequencing Problems. *Operations Research*, 61(6):1411–1428, 2013.
- 8 José R. Correa and Andreas S. Schulz. Single-Machine Scheduling with Precedence Constraints. *Mathematics of Operations Research*, 30(4):1005–1021, 2005.
- 9 Takahiro Fujita, Kohei Hatano, Shuji Kijima, and Eiji Takimoto. Online Linear Optimization for Job Scheduling under Precedence Constraints. In *Proceedings of 26th International Conference on Algorithmic Learning Theory(ALT 2015)*, volume 6331 of *LNCS*, pages 345–359, 2015.
- 10 Tarik Hadzic, John N Hooker, Barry O’Sullivan, and Peter Tiedemann. Approximate Compilation of Constraints into Multivalued Decision Diagrams. In *Proceedings of the 14th International Conference on Principles and Practice of Constraint Programming (CP 2008)*, *LNCS* 5202, pages 448–462, 2008.
- 11 Leslie A. Hall, Andreas S. Schulz, David B. Shmoys, and Joel Wein. Scheduling to Minimize Average Completion Time: Off-Line and On-Line Approximation Algorithms. *Mathematics of Operations Research*, 22(3):513–544, 1997.
- 12 Takeru Inoue, Keiji Takano, Takayuki Watanabe, Jun Kawahara, Ryo Yoshinaka, Akihiro Kishimoto, Koji Tsuda, Shin-ichi Minato, and Yasuhiro Hayashi. Distribution Loss Minimization With Guaranteed Error Bound. *IEEE Transactions on Smart Grid*, 5(1):102–111, 2014.

- 13 Yuma Inoue and Shin-ichi Minato. An Efficient Method for Indexing All Topological Orders of a Directed Graph. In *Proceedings of the 25th international Symposium on Algorithms and Computation (ISAAC'14)*, pages 103–114, 2014.
- 14 Donald E. Knuth. *The Art of Computer Programming, Volume 4A, Combinatorial Algorithms, Part 1*. Addison-Wesley Professional, 2011.
- 15 Eugene L. Lawler. On Sequencing jobs to minimize weighted completion time subject to precedence constraints. *Annals of Discrete Mathematics* 2, 2:75–90, 1978.
- 16 Jan K. Lenstra and Alexander H. G. Rinnooy Kan. Complexity of scheduling under precedence constraints. *Operations Research*, 26:22–35, 1978.
- 17 François Margot, Maurice Queyranne, and Yaoguang Wang. Decompositions, Network Flows, and a Precedence Constrained Single-Machine Scheduling Problem. *Operations Research*, 51(6):981–992, 2003.
- 18 Michael D. Miller. Multiple-Valued Logic Design Tools. In *Proceedings of the 23rd IEEE International Symposium on Multiple-Valued Logic (ISMVL'93)*, pages 2–11, 1993.
- 19 Michael D. Miller and Rolf Drechsler. Implementing a Multiple-Valued Decision Diagram Package. In *Proceedings of the 28th IEEE International Symposium on Multiple-Valued Logic (ISMVL'98)*, pages 52–57, 1998.
- 20 Shin-ichi Minato. Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. In *Proceedings of the 30th international Design Automation Conference (DAC'93)*, pages 272–277, 1993.
- 21 Shin-ichi Minato. Zero-suppressed BDDs and their applications. *International Journal on Software Tools for Technology Transfer*, 3(2):156–170, 2001.
- 22 Shin-ichi Minato. Efficient Database Analysis Using VSOP Calculator Based on Zero-Suppressed BDDs. In *New Frontiers in Artificial Intelligence, Joint JSAI 2005 Workshop Post-Proceedings*, pages 169–181, 2005.
- 23 Shin-ichi Minato. π DD: A New Decision Diagram for Efficient Problem Solving in Permutation Space. In *Proceedings of the 14th International Conference on Theory and Applications of Satisfiability Testing (SAT'11)*, pages 90–104, 2011.
- 24 Andreas S. Schulz. Scheduling to Minimize Total Weighted Completion Time: Performance Guarantees of LP-Based Heuristics and Lower Bounds. In *Proceedings of the 5th Conference on Integer Programming and Combinatorial Optimization (IPCO1996)*, pages 301–315, 1996.

Speeding up Dualization in the Fredman-Khachiyan Algorithm B

Nafiseh Sedaghat

School of Computing Science, Simon Fraser University
nf_sedaghat@sfu.ca

Tamon Stephen¹

Department of Mathematics, Simon Fraser University
tamon@sfu.ca

Leonid Chindelevitch²

School of Computing Science, Simon Fraser University
leonid@sfu.ca

Abstract

The problem of computing the dual of a monotone Boolean function f is a fundamental problem in theoretical computer science with numerous applications. The related problem of duality testing (given two monotone Boolean functions f and g , declare that they are dual or provide a certificate that shows they are not) has a complexity that is not yet known. However, two quasi-polynomial time algorithms for it, often referred to as FK -A and FK -B, were proposed by Fredman and Khachiyan in 1996, with the latter having a better complexity guarantee. These can be naturally used as a subroutine in computing the dual of f .

In this paper, we investigate this use of the FK -B algorithm for the computation of the dual of a monotone Boolean function, and present practical improvements to its performance. First, we show how FK -B can be modified to produce multiple certificates (Boolean vectors on which the functions defined by the original f and the current dual g do not provide outputs consistent with duality). Second, we show how the number of redundancy tests - one of the more costly and time-consuming steps of FK -B - can be substantially reduced in this context. Lastly, we describe a simple memoization technique that avoids the solution of multiple identical subproblems.

We test our approach on a number of inputs coming from computational biology as well as combinatorics. These modifications provide a substantial speed-up, as much as an order of magnitude, for FK -B dualization relative to a naive implementation. Although other methods may end up being faster in practice, our work paves the way for a principled optimization process for the generation of monotone Boolean functions and their duals from an oracle.

2012 ACM Subject Classification Computing methodologies → Boolean algebra algorithms

Keywords and phrases Monotone boolean functions, dualization, Fredman-Khachiyan algorithm, algorithm engineering, metabolic networks

Digital Object Identifier 10.4230/LIPIcs.SEA.2018.6

1 Introduction

Boolean functions are a powerful modeling tool. In many applications, such as those described in [9], the relevant Boolean functions have a natural monotone structure, and can thus be understood and manipulated in terms of their minimal true and maximal false settings.

¹ Partially supported by an NSERC Discovery grant.

² Partially supported by an NSERC Discovery grant and a Sloan Foundation fellowship.



Obtaining and translating between these representations is a challenging and deep theoretical question. There are numerous applications to do this in areas such as graph theory (generating the transversal of a hypergraph), combinatorics (finding minimal hitting sets), and machine learning (model-based fault diagnosis). It is also of interest in more applied fields from security to networking and from distributed systems to computational biology. Our interest in the problem stems begins from computational biology, notably in computing elementary flux modes (EFMs) and minimal cut sets (MCSs) following [13].

Given the list of minimal true settings, the problem of generating the list of maximal false settings is the classical problem of hypergraph dualization, which has arisen in several contexts, see for example [9] for a survey, but is not well understood theoretically. The problem of jointly generating both lists when the function is presented as an oracle is an attractive problem. Notably Fredman and Khachiyan [7] found two novel algorithms for dualization, that, unlike other known algorithms for the problem, extend to oracle-based generation [10]. These algorithms, which we refer to as *FK-A* and *FK-B*, verify duality or generate new clauses in incremental quasi-polynomial time $N^{o(\log N^2)}$ and $N^{o(\log N)}$ respectively, where N is the current joint size of the input and output lists. These algorithms are poorly understood in theory and in practice. The only available open-source code for oracle-based generation is `cl-jointgen` [12] for *FK-A*, though some experiments on *FK-A* and *FK-B* are described in [11], and an *FK-A* based dualization algorithm by Elbassioni is also now available [5]. Highly parallel *FK*-type algorithms were proposed by Elbassioni [6] and Boros and Makino [1].

In this paper we address some of the computational challenges of using the *FK-B* algorithm for dualizing a monotone Boolean function, although our techniques can also be directly applied to the setting of jointly generating the minimal true and maximal false settings of a monotone Boolean function given as an oracle. Our main contribution is an extension of the basic *FK-B* algorithm to produce multiple conflicting assignments (abbreviated as CA) in a single iteration. Our second contribution is a substantial reduction in the number of redundancy tests (namely, a test to make sure that no clause is a strict superset of another one, required to maintain correctness) during the execution of *FK-B*. Lastly, we find that a number of the subproblems arising during the different calls to *FK-B* are identical, and this leads to our final contribution - the use of a memoization technique to speed up dualization. Each improvement alone produces a substantial speed-up, and in combination, they result in an order of magnitude speed gain relative to a naive (unoptimized) implementation.

2 Definitions

Let n be fixed. We write \mathcal{B} to denote the set $\{0, 1\}$. A Boolean function $f : \mathcal{B}^n \rightarrow \mathcal{B}$ is *monotone* if $f(s) \leq f(t)$ for any two vectors $s \leq t \in \mathcal{B}^n$, where the inequality is interpreted component-wise. In other words, replacing a 0 with a 1 in the input cannot decrease f 's value. Monotone functions are precisely those that can be constructed using the OR and AND operations, without any NOT gates.

The dual of a Boolean function f is the function f^d defined by:

$$f^d(x) = \overline{f(\bar{x})} \tag{1}$$

for all $x = (x_1, x_2, \dots, x_n) \in \{0, 1\}^n$, where $\bar{x} = (\bar{x}_1, \bar{x}_2, \dots, \bar{x}_n)$.

A monotone Boolean function f is said to be in Disjunctive Normal Form (DNF) if it is represented as an OR of ANDs, i.e. as

$$f = \bigvee_{j=1}^m M_j, \text{ where } M_j = \bigwedge_{i \in T_j} x_i.$$

Here, the monomials M_j are called *implicants* of f . If the underlying sets T_j satisfy the *Sperner property*, i.e. $T_j \not\subseteq T_k$ whenever $j \neq k$, then each M_j is also called a *prime implicant* of f and m is called the *size* of f . In this case, the point x defined by

$$x_i = \begin{cases} 1 & \text{if } i \in T_j \\ 0 & \text{otherwise} \end{cases}$$

is a *minimal true point* of f ; indeed, for this x we have $f(x) = 1$ and $f(y) = 0$ for any $y < x$, where $y < x$ if and only if $y \leq x$ and $y \neq x$.

Similarly, a monotone Boolean function f is said to be in *Conjunctive Normal Form (CNF)* if it is represented as an AND of ORs, i.e. as

$$f = \bigwedge_{j=1}^m C_j, \text{ where } C_j = \bigvee_{i \in S_j} x_i.$$

Here, the clauses C_j are called *implicates* of f . Once again, if the underlying sets S_j satisfy the Sperner property, then each C_j is also called a *prime implicate* of f and m is called the *size* of f . In this case, the point x defined by

$$x_i = \begin{cases} 0 & \text{if } i \in S_j \\ 1 & \text{otherwise} \end{cases}$$

is a *maximal false point* of f ; indeed, for this x we have $f(x) = 0$ and $f(y) = 1$ for any $y > x$.

Lastly, we define the *support* of x , denoted $\text{supp}(x)$, as the set $\{i \in \{1, 2, \dots, n\} | x_i = 1\}$.

In the context of a metabolic network model M , which we use as one of the test cases here, the monotone Boolean function f is defined on subsets of reactions via $f(x) = 1$ if and only if the support of x enables biomass production. In this setting the minimal true points of f are called *elementary flux modes (EFMs)* and the maximal false points of f are called *minimal cut sets (MCSs)*, see for example [20, 14].

We can define two related problems for monotone Boolean functions:

- testing the equivalence of two monotone Boolean functions defined by a DNF and a CNF;
- the dualization problem: computing the equivalent CNF when a monotone DNF is given.

These two problems can be easily transformed into one another. The dualization problem is equivalent to *Transversal Hypergraph Generation*, also called the *Minimal Hitting Set Enumeration* problem. For background on these problems and applications, we refer the reader to [4, 11, 3, 9] and references therein.

The study of monotone Boolean functions is a vibrant area of ongoing research. The algorithm with the best known worst-case performance guarantee for the dualization of a monotone Boolean function f has incremental quasi-polynomial running time. More precisely, starting from a description of f in DNF, each iteration obtains an additional clause of the equivalent CNF, in $N^{O(\log N)}$ time, where N is the total size of the DNF and the current, possibly incomplete, CNF [7].

2.1 The *FK*-Dualization Algorithm

Algorithms 1 and 2 show the *FK* dualization and *FK-B* duality checking procedure respectively, following the presentations of [3] and [15].

Let ϕ be a DNF or CNF, and let x be a splitting variable. Then ϕ_0^x denotes the formula that consists of the terms of ϕ from which x is removed: $\phi_0^x = \{t - \{x\} : t \in \phi\}$. Analogously, ϕ_1^x denotes the formula that consists of all terms of ϕ that do not contain x : $\phi_1^x = \{t : t \in \phi \text{ and } x \notin t\}$.

Algorithm 1 Fredman-Khachiyan Dualization.

Input: A positive Boolean function f on B^n expressed by its complete DNF.

Output: The complete DNF of f^d .

```

1: function  $FK\_DUALIZATION(f)$ 
2:    $g = 0$ ;
3:   Call  $FK_B$  on the pair  $(f, g)$ ;
4:   if the returned value is "Yes" then
5:     halt;
6:   else
7:     let  $X^* \in B^n$  be the point returned by  $FK_B$ ;
8:     compute a maximal false point of  $f$ , say  $Y^*$ , such that  $X^* \leq Y^*$ ;
9:      $g = g \vee \bigwedge_{j \in \text{supp}(\overline{Y^*})} x_j$ ;
10:    return to Step 3.

```

A variable x is called at most μ -frequent in D if its frequency in D is at most $1/\mu(|D| \cdot |C|)$, i.e. $|\{m \in D : x \in m\}|/|D| \leq 1/\mu(|D| \cdot |C|)$ where $\mu(n) \sim \log n / \log \log n$. A similar definition applies to C .

The original FK -B algorithm returns the first conflicting assignment (CA) that it finds between the given CNF and DNF. Hence computing the dual of a given DNF requires $N_{CNF} + 1$ iterations, where N_{CNF} is the size of the CNF that is dual to the given DNF.

3 Methods

3.1 Pre-processing and Post-processing Steps

When analyzing metabolic networks to obtain the MCSs from the EFMs, it is beneficial to pre-process the given EFMs before starting the dualization procedure. The preprocessing involves three steps:

- removing any reactions that are not part of any EFMs (also known as blocked reactions [2, 8]), which correspond to unused variables;
- removing any reactions involved in all the EFMs (also referred to as essential reactions [2, 8]), adding them as singleton MCSs in post-processing;
- collapsing any set of k reactions whose presence/absence patterns in clauses are identical (a special case of this is referred to as enzyme subsets [2, 8]) into a single reaction, expanding each of the final MCSs involving this reaction into k copies in post-processing.

The pre-processing and post-processing steps are not necessary and can be ignored. However, they reduce the original problem and make the dualization procedure faster, so we routinely perform these steps.

3.2 Finding Multiple Conflicting Assignments

Given that we can use any conflicting assignment between the current CNF and DNF to compute a new clause in CNF, we can find Multiple Conflicting Assignments (MCAs) at the same time to generate more than one clause per iteration of the dualization procedure, and reduce the running time of the algorithm by reducing the total number of required iterations.

Algorithm 2 The Fredman-Khachiyan Algorithm B (*FK-B*).

Input: irredundant, monotone DNF D and CNF C .**Output:** \emptyset in case of equivalence; otherwise, assignment \mathcal{A} with $\mathcal{A}(D) \neq \mathcal{A}(C)$.

```

1: function FK-B( $C, D$ )
2:   make  $D$  and  $C$  irredundant;
3:   if a necessary condition is violated then return conflicting assignment;
4:   if  $\min\{|D|, |C|\} \leq 2$  then return conflicting assignment found by a trivial check;
5:   else
6:     choose a splitting variable  $x$  from the formulae
7:     if  $x$  is at most  $\mu$ -frequent in  $D$  then
8:        $\mathcal{A} \leftarrow$  FK-B( $D_1^x, C_0^x \wedge C_1^x$ ) // recursive call for  $x$  set to false
9:       if  $\mathcal{A} \neq \emptyset$  then return  $\mathcal{A}$ 
10:      for all clauses  $c \in C_0^x$  do do
11:         $\mathcal{A} \leftarrow$  FK-B( $D_0^{c,x}, C_1^{c,x}$ ) // see ⟨1⟩
12:        if  $\mathcal{A} \neq \emptyset$  then return  $\mathcal{A} \cup \{x\}$ 
13:      else if  $x$  is at most  $\mu$ -frequent in  $C$  then
14:         $\mathcal{A} \leftarrow$  FK-B( $D_0^x \vee D_1^x, C_1^x$ ) // recursive call for  $x$  set to true
15:        if  $\mathcal{A} \neq \emptyset$  then return  $\mathcal{A} \cup \{x\}$ 
16:        for all monomials  $m \in D_0^x$  do do
17:           $\mathcal{A} \leftarrow$  FK-B( $D_1^{m,x}, C_0^{m,x}$ ) // see ⟨2⟩
18:          if  $\mathcal{A} \neq \emptyset$  then return  $\mathcal{A} \cup \{m\}$ 
19:        else
20:           $\mathcal{A} \leftarrow$  FK-B( $D_1^x, C_0^x \wedge C_1^x$ ) // recursive call for  $x$  set to false
21:          if  $\mathcal{A} = \emptyset$  then
22:             $\mathcal{A} \leftarrow$  FK-B( $D_0^x \vee D_1^x, C_1^x$ ) // recursive call for  $x$  set to true
23:            if  $\mathcal{A} \neq \emptyset$  then return  $\mathcal{A} \cup \{x\}$ 
24:          return  $\mathcal{A}$ 

```

⟨1⟩: $D_1^x \equiv C_0^x \wedge C_1^x$: recursive call for all maximal non-satisfying assignments of C_0^x for x set to *true*. $D_0^{c,x}$ and $C_1^{c,x}$ denote the formulae we obtain if we set all variables in c to *false*.

⟨2⟩: $D_0^x \vee D_1^x \equiv C_1^x$: recursive call for all minimal satisfying assignments of D_0^x for x set to *false*. $D_1^{m,x}$ and $C_0^{m,x}$ denote the formula we obtain if we set all variables in m to *true*.

To this end, MCAs can be computed in the situations below without significant increase of computational effort. The first two situations arise during the assessment of the first two conditions necessary for equivalence in *FK-B*.

The first condition that we assess in the *FK-B* algorithm is the existence of a non-empty intersection between every clause in CNF and every monomial in DNF. If there is no intersection between monomial $m \in DNF$ and clause $c \in CNF$, then m makes the DNF *true* and the CNF *false*, so it is a CA. During the dualization procedure, especially early on, many of the monomials and clauses have no intersection. We thus consider intersections between every clause in the CNF and every monomial in the DNF at once and can return more than one CA.

The second condition that we assess in the *FK-B* algorithm is the presence of exactly the same variables in the CNF and the DNF. If this condition is not met, a CA is determined

from the extra variable(s) in the CNF or the DNF. If multiple variables are present in exactly one of the CNF and the DNF, we consider all possible conflicting assignments instead of returning only one.

The third situation in which we compute MCAs is in the case where $\min(|C|, |D|) \leq 2$. In such cases, the conflicting assignment is directly derived from Boolean algebra. We only consider the case $|C| \leq 2$ here; the case $|D| \leq 2$ is symmetric and is processed analogously.

The following cases may happen during this step:

- $|C| = 1$
Here, we look for a variable x in the unique CNF clause, denoted $C[1]$, such that D does not contain the singleton monomial x , in which case $\{x\}$ is a conflicting assignment.
- $|C| = 2$
In this case, we denote the two clauses by $C[1]$ and $C[2]$. There are three sub-cases:
 - Let $A_0 := C[1] \cap C[2]$. If $x \in A_0$ is a variable such that D does not contain the singleton monomial x , then $\{x\}$ is a conflicting assignment.
 - Let $A_1 := C[1] - C[2]$ and $A_2 := C[2] - C[1]$. Note that $A_1, A_2 \neq \emptyset$. If some monomial m in D is a subset of one of the A_i 's, then $\{x|x \in m\}$ is a conflicting assignment.
 - Let $(x, y) \in A_1 \times A_2$ (defined in the previous case). If no monomial in D is a subset of $\{x, y\}$ then $\{x, y\}$ is a conflicting assignment.

In all the aforementioned cases, whenever there is more than one conflicting assignment we return all of them. An issue regarding MCAs is that sometimes more than one CA can be mapped to a single clause in the CNF.

In the following sections, we refer to this version of *FK-B* as *FK_M*.

3.3 Reducing the Number of Redundancy Tests in the *FK-B* Algorithm

In the original *FK-B* algorithm, Algorithm 2, the first two instructions (line 2) remove redundancy in both the CNF and the DNF. During redundancy removal one performs an all-pairs comparison of the clauses (the monomials) in the CNF (the DNF) and remove any supersets found. In logic, removing redundancy is equivalent to applying the absorption rule to simplify the Boolean function.

In the *FK-B* algorithm, this procedure is a bottleneck due to the large number of pairwise comparisons that must be performed in each recursive call, and this is compounded by the fact that when we perform dualization, the *FK-B* algorithm is iterated many times to find the clauses of the CNF.

We reduce the number of redundancy tests performed in *FK-B*, and consequently in *FK-dualization*, by noting that when we set a variable to *true* (*false*) in the CNF (DNF), there is no need to check the redundancy of the CNF (DNF) in the next recursive call because such a setting results in one or more clauses (monomials) in the CNF (DNF) being removed, which cannot generate redundancy.

This can simply be implemented using two binary flags, which are respectively set if and only if the redundancy in the CNF (the DNF) should be checked, and cleared otherwise. For simplicity, we call this algorithm *FK_R*. It differs from the baseline, algorithm 2, in the following ways. First, the redundancy of the CNF (the DNF) is only checked if the corresponding flag is set. Second, in lines 8 and 20, where a variable x is set to *false*, the flag for the CNF is set and the flag for the DNF is cleared, since we only need to check the redundancy in the CNF, not the DNF. Conversely, in lines 14 and 22, the variable x is set to *true*, so the flag for the DNF is set and the flag for the CNF is cleared. Note that in lines

11 and 17, it is assumed that variable x is respectively set to *true* and *false*, and then the variables in c and m are respectively set to *false* and *true*. For this reason, redundancy can be produced in those lines, so the next call to FK_R needs to check the redundancy in both the CNF and the DNF.

3.4 Dealing with Repeated subproblems

Given that the FK -B algorithm is a recursive algorithm which is called multiple times during dualization, we encounter many subproblems solved in the previous recursive calls or past iterations. In this case, memoizing (storing for future retrieval) these subproblems and their solutions (in the form of CAs) is beneficial, as it can reduce the running time of both the FK -B algorithm as well as FK -dualization as a whole.

To this end, we use a hash table whose keys are combination of the CNF and the DNF and whose values are the CAs between them. To implement this idea, we compute the key for a given CNF and DNF prior to calling the FK algorithm. If it is already in the hash table, we retrieve the value, i.e. the corresponding CAs, bypassing a recursive call to FK . Otherwise, we call FK and store the computed CAs as a new record in the hash table.

As we experimented with different settings in implementation of the memoization idea, we realized that solving small subproblems, with $|C| < 3$ and $|D| < 3$, from scratch was faster than storing them in the hash table and retrieving the CAs. Thus, in our implementation we do not use the hashing technique for small subproblems. The method that uses hashing in the FK -B algorithm is referred to as FK_H .

4 Experimental Results

To assess the proposed algorithms, we run them on 12 problems including seven biological (metabolic network) models downloaded from the BioModels database³ as well as 5 synthetic models. The code and examples that we used are available on GitHub [16].

We first parsed the biological models using the SBML parser in *MATLAB* to obtain a stoichiometric matrix containing the reactions and the metabolites, then applied EFMTTool [18]/FluxModeCalculator [19] to extract the EFMs into a matrix. We converted this matrix into a binary one by setting all non-zero values to one, and used this as the input DNF, as is standard when looking for MCSs. We have chosen models with small to medium sizes.

The synthetic models included in our experiments are the 3×3 magic squares, called ‘*ms-33*’, 3×3 semi-magic squares, called ‘*sms-33*’, and three problems ‘ac-200k’, ‘SDFP16’, and ‘SDFP23’ taken from the Hypergraph Dualization Repository⁴. ‘ac-200k’ is the complement of the set of maximal frequent itemsets with support threshold 200,000 from the “accident” dataset. ‘SDFP16’ and ‘SDFP23’ are the Self-Dual Fano Plane hypergraphs with respectively $n = 16$ and $n = 23$ vertices, and $(k_n - 2)^2/4 + k_n/2 + 1$ hyperedges, where $k_n = (n - 2)/7$.

4.1 Evaluating the Speed of the Algorithms

To measure the running time of the proposed algorithms we used the `timeit` function in *MATLAB*, which measures the typical running time of functions in seconds. `timeit` automatically repeats the input function in a timing loop a number of runs determined by a built-in heuristic method, and returns the median value across the runs.

³ <http://www.ebi.ac.uk/biomodels-main/publmodels>

⁴ <http://research.nii.ac.jp/~uno/dualization.html>

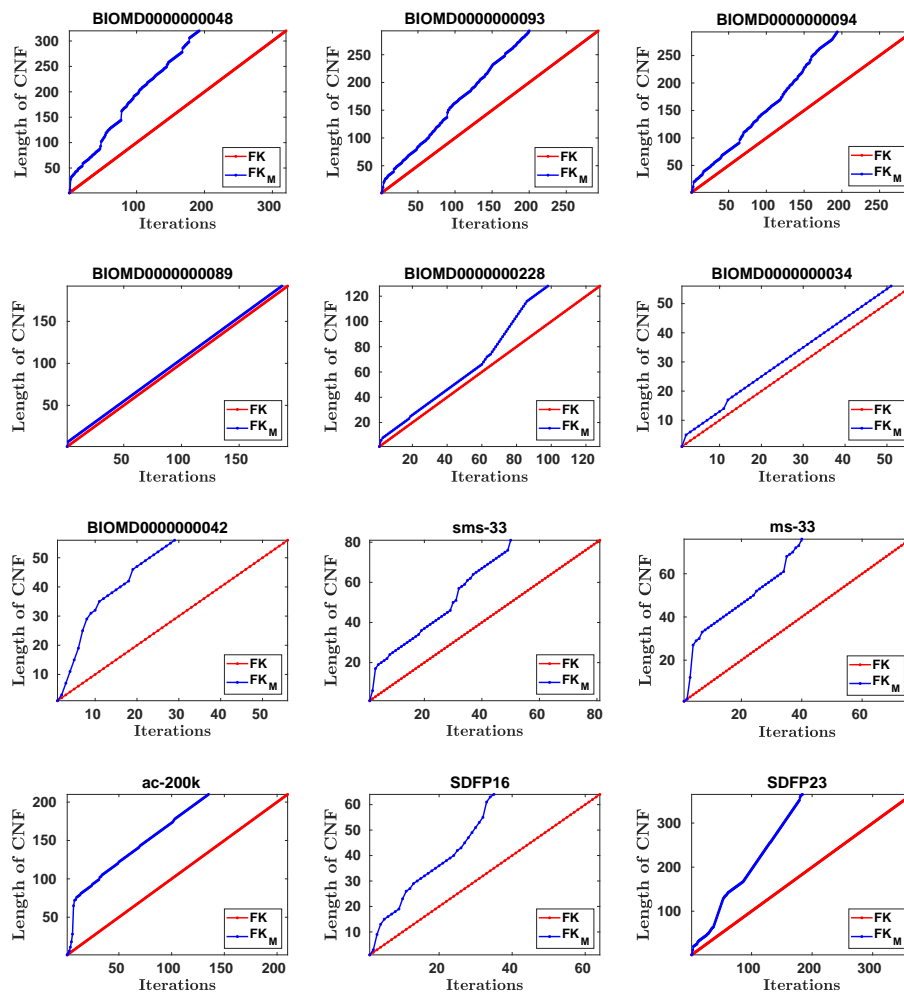
■ **Table 1** Characteristics of models; n_{metab} : number of metabolites, n_{EFM} : number of elementary flux modes or monomials in DNF, $n_r^{<pre}$: Number of reactions/variables before preprocessing steps, $n_r^{>pre}$: Number of reactions/variables after preprocessing, $n_{MCS}^{<post}$: Number of minimal cut sets or clauses in CNF before postprocessing, $n_{MCS}^{>post}$: Number of minimal cut sets or clauses in CNF after post-processing.

Model	n_{metab}	n_{EFM}	$n_r^{<pre}$	$n_r^{>pre}$	$n_{MCS}^{<post}$	$n_{MCS}^{>post}$
BIOMD0000000048	23	63	25	14	320	12960
BIOMD0000000093	34	24	46	24	293	2001
BIOMD0000000094	34	23	45	23	293	667
BIOMD0000000089	16	20	36	28	192	15552
BIOMD0000000228	9	13	22	20	128	512
BIOMD0000000034	9	13	22	22	56	56
BIOMD0000000042	15	35	25	20	56	188
sms_33	-	48	9	9	81	81
ms_33	-	40	9	9	76	76
ac_200k	-	81	64	21	210	253
SDFP16	-	64	16	16	64	64
SDFP23	-	365	23	23	365	365

■ **Table 2** Running time in seconds; FK_M : FK -dualization algorithm that returns multiple conflicting assignments, FK_R : Variant of FK with a reduction in the number of redundancy tests, FK_{MHR} : Variant of FK_M with a reduction in the number of redundancy tests that stores solved subproblems in a hash table, FK_{MHCR} : Variant of FK_{MHR} that stores solved subproblems in a hash table and uses the canonical form for small ones. In the last four columns, τ is the threshold such that if $|C| < \tau$ and $|D| < \tau$, the hash table is not used (when $\tau = 0$ it is always used).

	FK	FK_M	FK_R	FK_{MHR} ($\tau = 0$)	FK_{MHR} ($\tau = 3$)	FK_{MHCR} ($\tau = 0$)	FK_{MHCR} ($\tau = 3$)
BIOMD0000000048	119.06	62.71	86.13	12.82	12.67	17.29	12.80
BIOMD0000000093	136.68	88.69	117.54	14.44	15.84	25.24	15.92
BIOMD0000000094	140.37	78.81	122.28	19.31	21.69	37.71	21.81
BIOMD0000000089	31.27	26.6	25.96	9.59	8.52	15.62	8.57
BIOMD0000000228	9.30	6.41	7.82	2.42	2.81	5.57	3.06
BIOMD0000000034	1.56	1.34	1.36	0.45	0.42	1.05	0.43
BIOMD0000000042	7.74	3.72	7.24	0.19	0.19	0.20	0.22
sms_33	3.95	1.72	3.27	0.46	0.55	3.06	2.45
ms_33	2.90	1.26	2.28	0.46	0.64	2.79	2.17
ac_200k	311.57	35.22	305.69	7.13	7.85	13.44	11.65
SDFP16	6.67	2.80	5.86	0.40	0.58	1.90	0.56
SDFP23	648.59	251.05	504.30	103.66	108.98	130.81	108.67

Table 2 shows the running time of finding the dual of a given DNF. The time measurements also show the time required for dualization, exclusive of pre- and post-processing.

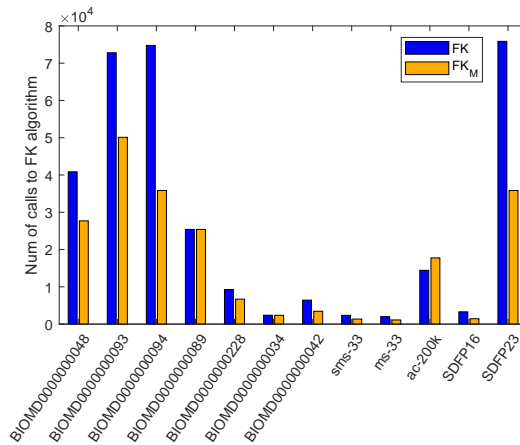


■ **Figure 1** Size of the CNF versus number of iterations in FK -dualization and FK_M -dualization.

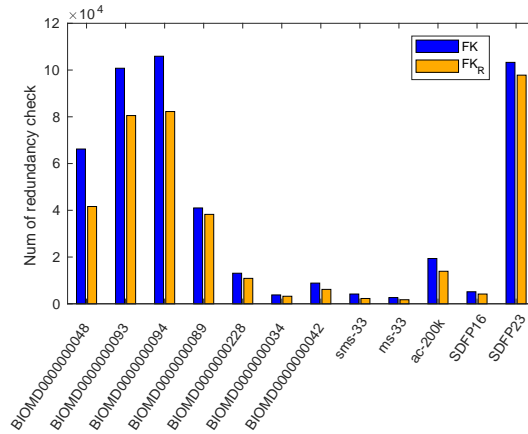
4.2 Reducing the Number of Iterations in FK -dualization Using FK_M

To show how effective FK_M -dualization is in comparison to FK -dualization we run both methods on our 12 problems and stored the size of the CNF in each iteration of the algorithms. Figure 1 shows the results. As can be seen in this figure, FK_M -dualization constructs the final CNF faster and requires as few as half the iterations in comparison to FK -dualization.

We have also counted the number of recursive calls to FK and FK_M in their corresponding dualization procedures. It turns out that by returning multiple conflicting assignments, FK_M requires fewer recursive calls than FK for every problem except ‘ac-200k’, as illustrated in Figure 2. Although finding multiple conflicting assignments always reduces the total number of iterations (except for one of the metabolic models, where it stays the same, as shown in Figure 1), the number of recursive calls to FK_M can occasionally exceed the number of recursive calls to FK , presumably due to changes in the traversal of the assignment tree.



■ **Figure 2** Comparing the number of recursive calls to FK and FK_M during dualization.



■ **Figure 3** Comparing the number of redundancy tests in FK and FK_R dualization.

4.3 Reducing the Number of Redundancy Tests in the FK -B Algorithm

In this experiment, we show how using the two flags in the FK_R algorithm in comparison with FK reduces the number of redundancy tests in both the CNF and the DNF. To this end we count the number of calls to the redundancy testing function.

Figure 3 shows the results. As expected, using the two redundancy flags reduces the number of redundancy tests in the FK_R algorithm relative to the baseline FK algorithm.

In small problems, the reduction is not significant, however, in large problems like $BIOMD00000000048$, the number of redundancy tests in FK_R is about $\frac{2}{3}$ of what it is in FK . Given that the procedure of redundancy test is a bottleneck of the FK algorithm, fewer calls to this function reduces the time required to find the dual of a given monotone Boolean function.

4.4 Dealing with Repeated subproblems Using a Hash Table

In this experiment we show the efficiency of using a hash table for dualization. We count the number of successful key and value retrievals from the hash table. Figure 4 shows the characteristics of the subproblems, e.g. the frequency and the size of the subproblem, corresponding to the first five most popular keys in FK_{MH} dualization. For each model there are two figures, corresponding to $\tau = 0$ and $\tau = 3$, meaning that we use the hash table only for subproblems with $|C| > \tau$ and $|D| > \tau$.

Comparing the figures in columns with $\tau = 0$ to the figures in columns with $\tau = 3$ demonstrates that when we use hash table for every subproblem, i.e. $\tau = 0$, the size of frequent subproblems is very small while in the other case, i.e. $\tau = 3$, the size of the frequent subproblems is larger.

Since there is the possibility that some of the frequently occurring Boolean functions differ only by a permutation of the variables, we have implemented a simple scheme for reducing functions of up to seven variables to a canonical form as is done in Stephen and Yusun [17]. We found that the most popular keys in the implementation of the hash table with and without the canonical form are the same, thus, we only present the results without the canonical form here, as FK_{MH} .

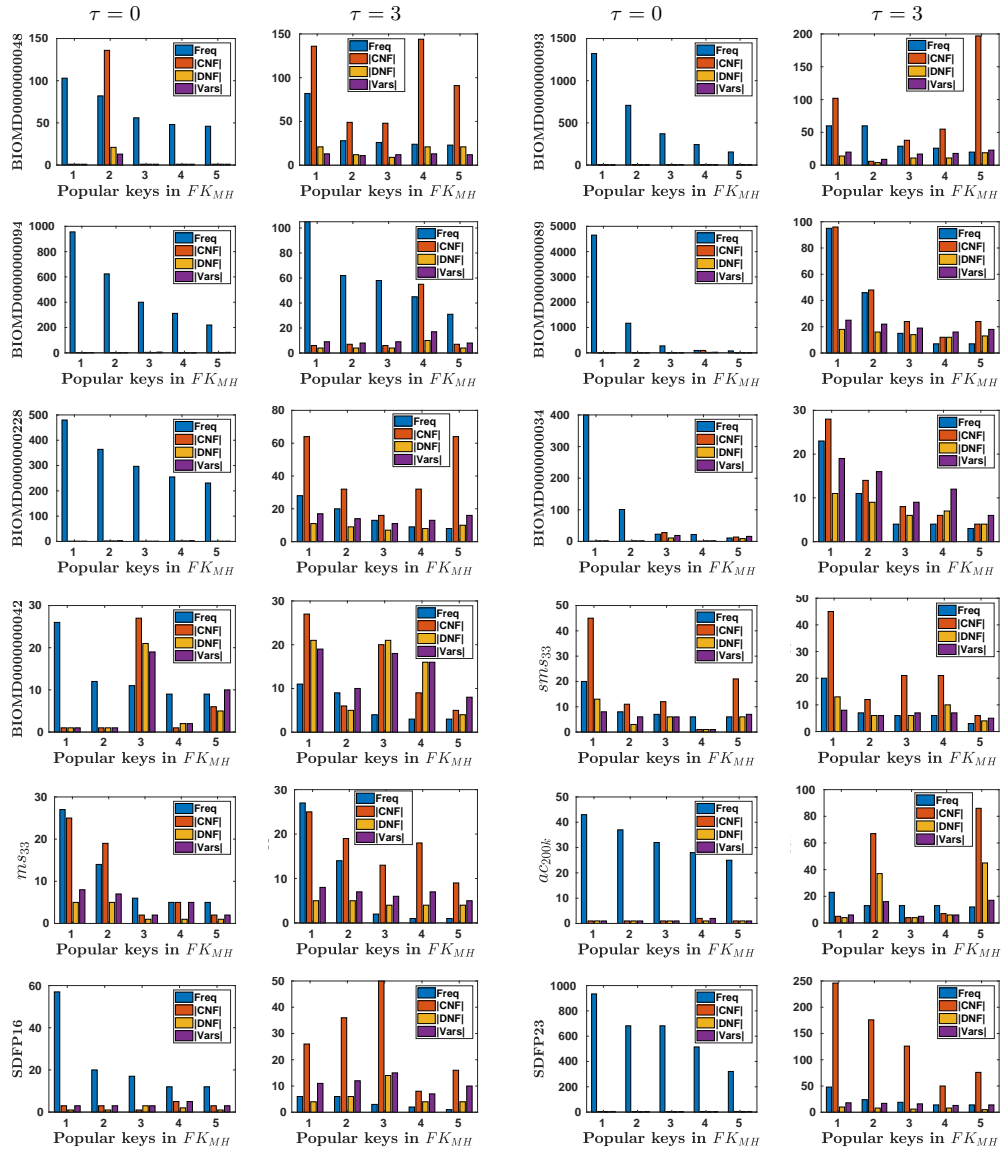
5 Discussion

The FK -B algorithm was a theoretical breakthrough at the time of its discovery in 1996, showing for the first time that the problem of testing the duality of two monotone Boolean functions could be solved in quasi-polynomial time. Because of its ability to produce a certificate of non-duality, this also means that, given a Boolean function in explicit form, its dual can be generated in incremental quasi-polynomial time. In addition, the FK -B algorithm also applies to joint generation, namely, an explicit description of the function f and of its dual in the case where f is given implicitly by an oracle (an algorithm that, given an input x , returns the value $f(x)$) - with the same complexity guarantees provided that the oracle runs in polynomial time.

However, despite these exciting developments the FK -B algorithm is not easily usable for generating the dual of a large monotone Boolean function in practice. In this paper we provided several improvements that reduce the overall running time by an order of magnitude on most of the examples we considered. Although they do not change the worst-case algorithmic complexity of dualization via the FK -B algorithm, they take it closer to being usable in practice on medium-to-large-scale problems.

All our techniques also apply directly to the problem of joint generation. The generation of multiple conflicting assignments per iteration, the reduction in the number of redundancy tests, and the use of memoization could all turn out to be beneficial in this scenario as well. One of our future directions is to explore how helpful these techniques are in the context of joint generation (which, for metabolic network models, would amount to simultaneously generating the elementary modes and the minimal cut sets of the network).

In conclusion, our work paves the way for a systematic exploration of algorithmic improvements to Monotone boolean function dualization and joint generation algorithms that use a duality testing algorithm such as FK -B as a subroutine. We expect that, with additional algorithmic improvements, this approach will ultimately result in a practical method for solving this important problem.



■ **Figure 4** Characteristics of most frequent subproblems in the hash table in FK_{MH} dualization. In this set of experiments, hash table has been used for subproblems with $|C| > \tau$ and $|D| > \tau$.

References

- 1 Endre Boros and Kazuhisa Makino. A fast and simple parallel algorithm for the monotone duality problem. In *Automata, languages and programming. Part I*, volume 5555 of *Lecture Notes in Comput. Sci.*, pages 183–194. Springer, Berlin, 2009.
- 2 Leonid Chindelevitch, Jason Trigg, Aviv Regev, and Bonnie Berger. An exact arithmetic toolbox for a consistent and reproducible structural analysis of metabolic network models. *Nature Communications*, 5(4893):165–182, 2014.
- 3 Yves Crama and Peter L Hammer. *Boolean functions: Theory, algorithms, and applications*. Cambridge University Press, 2011.

- 4 Thomas Eiter, Kazuhisa Makino, and Georg Gottlob. Computational aspects of monotone dualization: a brief survey. *Discrete Appl. Math.*, 156(11):2035–2049, 2008.
- 5 Khaled Elbassioni. C implementation of Fredman and Khachiyan’s algorithm A. Available from <https://github.com/VeraLiconoResearchGroup/MHSGenerationAlgorithms/blob/master/containers/fka-begk>.
- 6 Khaled M. Elbassioni. On the complexity of monotone dualization and generating minimal hypergraph transversals. *Discrete Appl. Math.*, 156(11):2109–2123, 2008.
- 7 Michael L Fredman and Leonid Khachiyan. On the complexity of dualization of monotone disjunctive normal forms. *Journal of Algorithms*, 21(3):618–628, 1996.
- 8 Julien Gagneur and Steffen Klamt. Computation of elementary modes: a unifying framework and the new binary approach. *BMC Bioinformatics*, 5(175), 2004.
- 9 Andrew Gainer-Dewar and Paola Vera-Licona. The minimal hitting set generation problem: algorithms and computation. *SIAM J. Discrete Math.*, 31(1):63–100, 2017.
- 10 V. Gurvich and L. Khachiyan. On generating the irredundant conjunctive and disjunctive normal forms of monotone Boolean functions. *Discrete Appl. Math.*, 96/97:363–373, 1999. The satisfiability problem (Certosa di Pontignano, 1996); Boolean functions.
- 11 Matthias Hagen, Peter Horatschek, and Martin Mundhenk. Experimental comparison of the two Fredman-Khachiyan-algorithms. In *Proceedings of the Eleventh Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 154–161. SIAM, 2009.
- 12 Utz-Uwe Haus. `cl-jointgen`, a common lisp implementation of the joint-generation method. Available from <https://sourceforge.net/projects/cl-jointgen/> and in C at <https://sourceforge.net/projects/jointgen-c.cl-jointgen.p/>.
- 13 Utz-Uwe Haus, Steffen Klamt, and Tamon Stephen. Computing knock-out strategies in metabolic networks. *J. Comput. Biol.*, 15(3):259–268, 2008.
- 14 Steffen Klamt and Ernst Dieter Gilles. Minimal cut sets in biochemical reaction networks. *Bioinformatics*, 20(2):226–234, 2004.
- 15 Martin Mundhenk and Robert Zeranski. How to apply SAT-solving for the equivalence test of monotone normal forms. In Karem A. Sakallah and Laurent Simon, editors, *Theory and Applications of Satisfiability Testing - SAT 2011*, pages 105–119, Berlin, 2011. Springer.
- 16 Nafiseh Sedaghat. Matlab implementation of Fredman and Khachiyan’s algorithm B. Available from <https://github.com/WGS-TB/FK/tree/master/Modified%20FK-B%20Algorithm>.
- 17 Tamon Stephen and Timothy Yusun. Counting inequivalent monotone Boolean functions. *Discrete Appl. Math.*, 167:15–24, 2014.
- 18 Marco Terzer and Jörg Stelling. Large-scale computation of elementary flux modes with bit pattern trees. *Bioinformatics*, 24(19):2229–2235, 2008.
- 19 Jan Bert Van Klinken and Ko Willems van Dijk. FluxModeCalculator: an efficient tool for large-scale flux mode computation. *Bioinformatics*, 32(8):1265–1266, 2015.
- 20 Jürgen Zanghellini, David E. Ruckerbauer, Michael Hanscho, and Christian Jungreuthmayer. Elementary flux modes in a nutshell: Properties, calculation and applications. *Biotechnology Journal*, 8(9):1009–1016, 2013. doi:10.1002/biot.201200269.


An Ambiguous Coding Scheme for Selective Encryption of High Entropy Volumes

M. Oğuzhan Külekci¹

Informatics Institute, Istanbul Technical University

Istanbul, Turkey

kulekci@itu.edu.tr

 <https://orcid.org/0000-0002-4583-6261>

Abstract

This study concentrates on the security of high-entropy volumes, where entropy-encoded multimedia files or compressed text sequences are the most typical sources. We consider a system in which the cost of encryption is hefty in terms of some metric (e.g., time, memory, energy, or bandwidth), and thus, creates a bottleneck. With the aim of reducing the encryption cost on such a system, we propose a data coding scheme to achieve the data security by encrypting significantly less data than the original size without sacrifice in secrecy. The main idea of the proposed technique is to represent the input sequence by *not* uniquely-decodable codewords. The proposed coding scheme splits a given input into two partitions as the *payload*, which consists of the ambiguous codeword sequence, and the *disambiguation information*, which is the necessary knowledge to properly decode the payload. Under the assumed condition that the input data is the output of an entropy-encoder, and thus, on ideal case independently and identically distributed, the payload occupies $\approx \frac{(d-2)}{d}$, and the disambiguation information takes $\approx \frac{2}{d}$ of the encoded stream, where $d > 2$ denotes a chosen parameter typically between 6 to 20. We propose to encrypt the payload and keep the disambiguation information in plain to reduce the amount of data to be encrypted, where recursive representation of the payload with the proposed coding can decrease the to-be-encrypted volume further. When $2 \cdot 2^d \leq n \leq \tau \cdot d \cdot 2^d$, for $\tau = \frac{d-1.44}{2}$, we show that the contraction of the possible message space 2^n due to the public disambiguation information is accommodated by keeping the codeword set secret. We discuss possible applications of the proposed scheme in practice.

2012 ACM Subject Classification Information systems → Data encryption, Information systems → Multimedia databases, Mathematics of computing → Combinatorics, Mathematics of computing → Coding theory, Security and privacy → Management and querying of encrypted data

Keywords and phrases Non-prefix-free codes, selective encryption, massive data security, multimedia data security, high-entropy data security, source coding, security in resource-limited environments

Digital Object Identifier 10.4230/LIPIcs.SEA.2018.7

1 Introduction

Achieving security of massive data volumes with less encryption makes sense on platforms where the cost of encryption defines a bottleneck as being heavy according to some metrics, e.g., time, memory, energy, or bandwidth. For example, let us assume a system at which the items in a queue are waiting for the encryption/decryption unit to get processed, and consequently delays occur. One simple solution might be to increase the number of the

¹ This work has been supported by TUBITAK-ARDEB-1001 program grant number 117E865.



© Muhammed Oğuzhan Külekci;

licensed under Creative Commons License CC-BY

17th International Symposium on Experimental Algorithms (SEA 2018).

Editor: Gianlorenzo D'Angelo; Article No. 7; pp. 7:1–7:13

Leibniz International Proceedings in Informatics



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

serving units, but on the other hand usually the congestion only appears at certain times, and the expense of the additional unit may not be feasible. While waiting on the queue, the items can be encoded such that the amount of data to-be-encrypted is reduced, and the items are processed more quickly in the system. Such a reduction can simply improve the throughput of a security pipeline without a need to upgrade the infrastructure.

Similarly, in battery-constrained environments such as mobile devices [9], sensor networks [3], or unmanned aerial vehicles [18], performing less encryption may also help to increase the battery life. It had been shown that symmetric security algorithms roughly doubles the energy consumption of normal operation in those environments, and asymmetric security algorithms increase the energy usage per bit in order of magnitudes (around 5 fold) [14].

Previously, selective encryption schemes [12] have been proposed to reduce the encryption load, particularly on transmission of video/image files [19, 11, 8]. In selective encryption, segments of the data, which are assumed to include important information, e.g., the I-frames in a video stream, are encrypted, while rest of the data is kept plain. We introduce an alternative approach to reduce the amount of encryption required to secure a source data. As opposed to the partial security provided by the selective encryption schemes, we aim to provide the security of the whole data by benefiting from the intrinsic ambiguity of non-prefix-free (NPF) coding.

In NPF coding a codeword may be a prefix of some others, and thus, the nice self-delimiting property of the prefix-free schemes [2] does not apply. Therefore, the codeword boundaries on the encoded stream should be explicitly specified for correct decoding. In other words, the disambiguation information required to decode an NPF codewords stream is the identification of the codeword boundaries on that sequence.

The NPF coding has not been addressed much in the literature except a few studies [4, 10, 1] due to that unique decodability problem, which limits, if not totally removes, its possible usage in practical applications, particularly in data compression. However, we consider that this lack of unique decodability in NPF coding may provide us an interesting opportunity in terms of security. It is noteworthy that the hardness of decoding an encoded data without the knowledge of the used codeword set had been addressed as early as in 1979 [15], and later by others [6, 7]. More recently, non-prefix-free codes have also been mentioned [13] in that sense.

The main idea of the proposed technique here is to represent the input sequence by *not* uniquely-decodable codewords, which can be summarized as follows. We process the n bit long input bit sequence in blocks of d bits according to a predetermined d parameter such that $d \cdot 2^d \leq n$. Due to some limitations that will be described in the paper, typically d is expected to be between 6 and 20. We create 2^d non-prefix codewords by using a *secret* permutation of the numbers $[1 \dots 2^d]$, and then replace every d -bits long symbol in the input with its corresponding NPF codeword of varying bit-length in between 1 to d . We call the resulting bit stream the payload since it includes the actual information of the source. This sequence is not decodable without the codeword boundaries. Therefore, we need to maintain an efficient representation of the codeword boundaries on the payload. This second stream is referred as the *disambiguation information* throughout the study. The total space consumed by the payload and the disambiguation information introduces an overhead of $2(d-1)/d \cdot 2^d \approx \frac{1}{2^{d-1}}$ bits per each original bit, which becomes negligible as d increases, for example, it is less than 7 bits per a thousand bit when $d = 8$. Thus, proposed scheme actually splits the input into two partitions, which occupy almost the same space.

We prove that the *payload* occupies $\approx \frac{(d-2)}{d}$, and the *disambiguation information* takes $\approx \frac{2}{d}$ of the final volume. When the payload, which is the main source information, is

encrypted and disambiguation information is stored in plain, the amount of to-be-encrypted volume decreases by $2/d$ of the original size, e.g., for $d = 8$, 25% of the data is not required to be encrypted. The payload can be subject to the same process recursively, which gives us the opportunity to tune the size of the encrypted volume with processing power. For instance, in case $d = 8$, a second level of encoding of the first level's payload increases the gain in encryption from 25% to 43%.

In this scenario, it is important to analyze the information leakage by the plain disambiguation information. Since the payload is encrypted, it should not be easier for an attacker to guess the payload from the disambiguation information rather than breaking the ciphered payload. When $2 \cdot 2^d \leq n \leq \tau \cdot d \cdot 2^d$, for $\tau = \frac{d-1.44}{2}$, we show that the contraction of the possible message space 2^n due to the public disambiguation information is accommodated by keeping the codeword set secret.

It might have captured the attention of the reader that the analysis assumes the input bit stream to be uniformly i.i.d., which seems a bit restrictive at a first glance. However, the target data types of the introduced method are mainly the sources that have been previously entropy encoded² such as the video files in mpeg4 format, sound files in mp3, and similar others. The output of the compression tools squeezing data down to its entropy actually is actually quite nice input for our proposal. We support this observation by the experiments performed on various compressed file types showed that the results on real data is very close to the theoretical bounds computed by the uniformly i.i.d. assumption.

The outline of the paper is as follows. In Section 2 we introduce the proposed ambiguous encoding method based on the non-prefix-free codes, and analyze its basic properties mostly focusing on the space consumption of the partitions. We also provide verification of the theoretical claims based on uniformly i.i.d. assumption on some files that are already entropy-encoded. Section 3 focuses on using the ambiguous coding to reduce the number of encryption operations, and investigates the information leakage by the disambiguation information, which is proposed to be stored in plain format without encryption. We finalize our study by summarizing the results and discussing further related research avenues.

2 Ambiguous Data Coding

Let $\mathcal{A} = a_1 a_2 \dots a_n$ denotes a uniformly independently and identically distributed bit sequence, and $d > 1$ is a predetermined block length. Without loss of generality we assume n is divisible by d . Otherwise, it is padded with random bits. \mathcal{A} can be represented as $\mathcal{B} = b_1 b_2 \dots b_r$, for $r = \frac{n}{d}$ such that each d -bits long b_i in \mathcal{B} is from the alphabet $\Sigma = \{0, 1, 2, \dots, 2^d - 1\}$.

We will first define the *minimum binary representation* of an integer, and then use this definition to state our encoding scheme.

► **Definition 1.** The **minimum binary representation (MBR)** of an integer $i \geq 2$ is its binary representation without the leftmost 1 bit.

As an example, $MBR(21) = 0101$ by omitting the leftmost set bit in its binary representation as $21 = (\mathbf{1}0101)_2$.

² Any lossless data compression scheme, where each symbol is represented by minimum number of bits close to the entropy of the symbol according to Shannon's theorem [17].

7:4 Ambiguous Coding For Selective Encryption

$\Sigma =$	0	1	2	3	4	5	6	7
$\Sigma' =$	6	0	5	1	7	2	4	3
	ϵ_1	ϵ_2	ϵ_3	ϵ_4	ϵ_5	ϵ_6	ϵ_7	ϵ_8
	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8
$W =$	{000,011,100,111}	0	11	1	{001,010,101,110}	00	10	01
$\mathcal{A} =$	001	110	101	011	010	111	100	000
$\mathcal{B} =$	1	6	5	3	2	7	4	0
$NPF(\mathcal{A}) =$	w_2	w_7	w_6	w_4	w_3	w_8	w_5	w_1
$NPF(\mathcal{A}) =$	0	10	00	1	11	01	101	000
$DisInfo(\mathcal{A}) =$	01	1	1	01	1	1	00	00

■ **Figure 1** A simple sketch of the non-prefix-free coding of an input bit sequence \mathcal{A} , where \mathcal{B} is the representation of \mathcal{A} with the block length $d = 3$. Σ' is a random permutation of the corresponding alphabet Σ , and W is the non-prefix-free codeword set generated for Σ' according to Definition 2. The disambiguation information $DisInfo(\mathcal{A})$ is computed according to Lemma 5.

► **Definition 2.** Let $\Sigma' = \{\epsilon_1, \epsilon_2, \dots, \epsilon_{2^d}\}$ be a permutation of the given alphabet $\Sigma = \{0, 1, 2, \dots, 2^d - 1\}$, and $W = \{w_1, w_2, \dots, w_{2^d}\}$ is a codeword set such that

$$w_i = \begin{cases} MBR(2 + \epsilon_i) & \text{,if } \epsilon_i < 2^d - 2 \\ \{MBR(2^d + \zeta) : \forall \zeta \in \{0, 1, \dots, 2^d - 1\}, \text{where } \zeta = \{0, 3\} \pmod{4}\} & \text{,if } \epsilon_i = 2^d - 2 \\ \{MBR(2^d + \zeta) : \forall \zeta \in \{0, 1, \dots, 2^d - 1\}, \text{where } \zeta = \{1, 2\} \pmod{4}\} & \text{,if } \epsilon_i = 2^d - 1 \end{cases}$$

The representation of the input $\mathcal{A} = \mathcal{B} = b_1 b_2 \dots b_r$ with the *non-prefix-free* codeword set W is shown by $NPF(\mathcal{A}) = c_1 c_2 \dots c_r$ such that $c_i = w_{1+b_i}$. When a codeword c_i has multiple options, a randomly selected one among the possibilities is used.

The NPF coding of a sample sequence according to the Definitions 1 and 2 with the parameter $d = 3$ is shown in Figure 1. The codewords w_1 and w_5 are *sets* as their corresponding $\epsilon_1 = 6$ and $\epsilon_5 = 7$ values are greater than or equal to $6 = 2^3 - 2$. Thus, when $c_i = w_1$ or $c_i = w_5$, a randomly selected codeword respectively from sets w_1 or w_5 , is inserted.

► **Proposition 3.** In a codeword set W that is generated for a block length $d > 1$ according to Definition 2, the lengths of the codewords in bits range from 1 to d , where the number of ℓ -bits long codewords for each $\ell \in \{1, 2, \dots, d - 1\}$ is 2^ℓ , and for $\ell = d$ there exist 2 sets of codewords each of which includes 2^{d-1} elements.

Proof. According to Definition 2, the entities in W are minimum binary representations of numbers $\{2, 3, \dots, 2^{d+1} - 1\}$. Since the MBR bit-lengths of those numbers range from 1 to d , there are d distinct codeword lengths in W .

Each codeword length $\ell \in \{1, 2, \dots, d - 1\}$ defines 2^ℓ distinct codewords, and thus, total number of codewords defined by all possible $\ell < d$ values becomes $\sum_{i=1}^{d-1} 2^i = 2^d - 2$. The remaining 2 codewords out of the $|W| = 2^d$ items require d -bits long bit sequences.

For example, when $d = 3$, the W includes $2(= 2^1)$ codewords of 1-bit long, $4(= 2^2)$ codewords of length 2, and $2(= 2^3 - 6)$ codeword *sets* of length 3-bits as shown in Figure 1. ◀

► **Lemma 4.** The $NPF(\mathcal{A})$ is expected to occupy $n \cdot (1 - \frac{2}{d} + \frac{2(d+1)}{d \cdot 2^d})$ bits space for a uniformly i.i.d. input \mathcal{A} of length $n = r \cdot d$ bits.

Proof. The total bit length of the NPF codewords is simply $\sum_{\ell=1}^d C_\ell \cdot \ell$, where C_ℓ denotes the number of occurrences of the b_i values represented by ℓ -bits long codewords in \mathcal{B} . Assuming the uniform distribution of \mathcal{B} , each $b_i \in \{0, 1, 2, \dots, 2^d - 1\}$ appears $\frac{r}{2^d}$ times. The number

Codelength	# of occurrences	represented by	space consumption
$d - 1$	$\frac{r}{2} = \frac{r}{2^d} \cdot 2^{d-1}$	1	$1 \cdot \frac{r}{2}$
$d - 2$	$\frac{r}{4} = \frac{r}{2^d} \cdot 2^{d-2}$	01	$2 \cdot \frac{r}{4}$
$d - 3$	$\frac{r}{8} = \frac{r}{2^d} \cdot 2^{d-3}$	001	$3 \cdot \frac{r}{8}$
...
1	$\frac{r}{2^{d-1}} = \frac{r}{2^d} \cdot 2$	00...1	$(d - 1) \cdot \frac{r}{2^{d-1}}$
d	$\frac{r}{2^{d-1}} = \frac{r}{2^d} \cdot 2$	00...0	$(d - 1) \cdot \frac{r}{2^{d-1}}$
Total space occupied:			$r(2 - \frac{1}{2^{d-2}})$

■ **Figure 2** The representation of the codeword lengths to specify the codeword boundaries on the NPF stream.

of distinct b_i values represented by a codeword of length ℓ is 2^ℓ for $1 \leq \ell < d$, and two of the b_i values require $\ell = d$ bit long codewords as stated in Proposition 3. Thus, $C_\ell = \frac{r}{2^d} \cdot 2^\ell$ for $1 \leq \ell < d$, and $C_d = \frac{r}{2^d} \cdot 2$. The length of the $NPF(\mathcal{B})$ bit-stream can then be computed by

$$|NPF(\mathcal{A})| = \frac{r}{2^d} \cdot (1 \cdot 2 + 2 \cdot 2^2 + \dots + (d - 1) \cdot 2^{d-1} + d \cdot 2) \quad (1)$$

$$= \frac{r}{2^d} \cdot (2d + \sum_{i=1}^{d-1} i \cdot 2^i) = \frac{r}{2^d} \cdot (2d + 2^d \cdot (d - 2) + 2) \quad (2)$$

$$= \frac{r}{2^d} \cdot (2^d(d - 2) + 2(d + 1)) \quad (3)$$

$$= r \cdot d - r \cdot (2 - \frac{d + 1}{2^{d-1}}) \quad (4)$$

$$= r \cdot (d - 2 + \frac{d + 1}{2^{d-1}}) \quad (5)$$

$$= \frac{n}{d} \cdot (d - 2 + \frac{d + 1}{2^{d-1}}) \quad (6)$$

$$= n \cdot (1 - \frac{2}{d} + \frac{2(d + 1)}{d \cdot 2^d}) \quad (7)$$

While computing the summation term in equation (2), we use the formula from basic algebra that $\sum_{i=1}^p i \cdot 2^i = 2^{p+1}(p - 1) + 2$, and substitute $p = d - 1$. ◀

The input sequence \mathcal{A} is originally n bit long, and the NPF coding reduces that space by $n \cdot (\frac{2}{d} - \frac{2(d+1)}{2^d})$ bits. However, since non-prefix-free codes are not uniquely decodable, $NPF(\mathcal{A})$ cannot be decoded back correctly in absence of the codeword boundaries. Therefore, we need to represent these boundary positions on $NPF(\mathcal{A})$. Lemma 5 states an efficient method to achieve this task.

► **Lemma 5.** *The expected number of bits to specify the codeword boundaries in the $NPF(\mathcal{A})$ is $n \cdot (\frac{2}{d} - \frac{4}{d \cdot 2^d})$, where $|\mathcal{A}| = n = r \cdot d$.*

Proof. Due to Proposition 3 there are 2^ℓ distinct codewords with length ℓ for $\ell \in \{1, 2, \dots, d - 1\}$ and 2 codewords (sets) are generated for $\ell = d$. Since each d -bits block has equal probability of appearance on \mathcal{A} , the number of occurrences of codewords having length $\ell \in \{1, 2, \dots, d - 1\}$ is $\frac{r}{2^d} \cdot 2^\ell$. The most frequent codeword length is $(d - 1)$, which appears at half of the r codewords as $\frac{r}{2^d} \cdot 2^{d-1} = \frac{r}{2}$. It is followed by the codeword length $(d - 2)$ that is observed

7:6 Ambiguous Coding For Selective Encryption

■ **Table 1** The payload, disambiguation information, and overhead bits per each original bit introduced by the proposed ambiguous coding for some selected d values.

$d =$	4	6	8	10	12	14	16	20
Overhead per bit $\frac{d-1}{d \cdot 2^{d-1}} \approx$	0,094	0,026	0,007	0,002	$1.1 \cdot 10^{-4}$	$4.4 \cdot 10^{-4}$	$2,8 \cdot 10^{-5}$	$1.8 \cdot 10^{-6}$
Payload per bit $1 - \frac{2}{d} + \frac{2(d+1)}{d \cdot 2^d} \approx$	0.656	0.703	0.759	0.802	0.834	0.857	0.875	0.900
Dis.Info. per bit $\frac{2}{d} - \frac{4}{d \cdot 2^d} \approx$	0,438	0.323	0.248	0.200	0.167	0.143	0.125	0.100

$\frac{r}{4}$ times. When we examine the number of codewords with length $\ell \in \{1, 2, \dots, d-1\}$, we see that this distribution is geometric, as depicted in Figure 2. The optimal prefix-free codes for the codeword lengths are then $\{1, 01, 001, \dots, 0^{d-2}1, 0^{d-1}\}$, which correspond to codeword lengths $\{d-1, d-2, d-3, \dots, 1, d\}$ respectively. Thus, codeword length $\ell = (d-i) \in \{1, 2, \dots, d-1\}$, which appears $\frac{r}{2^d} \cdot 2^{d-i}$ times on \mathcal{A} , can be shown by i bits. We use $(d-1)$ consecutive zeros to represent the codeword length $\ell = 2$ as the number of occurrences of d -bits long codewords is equal to the number of 1 bit long codewords on \mathcal{A} . Notice that the representation of the codeword lengths are prefix-free that can be uniquely decoded.

Total number of bits required to represent the individual lengths of the codewords can be computed by

$$\frac{r}{2^d} (2(d-1) + \sum_{i=1}^{d-1} i \cdot 2^{d-i}) = r \cdot \left(\frac{2(d-1)}{2^d} + \sum_{i=1}^{d-1} i \cdot 2^{-i} \right) \quad (8)$$

$$= r \left(\frac{d-1}{2^{d-1}} + \frac{2^d - d - 1}{2^{d-1}} \right) \quad (9)$$

$$= r \left(2 - \frac{1}{2^{d-2}} \right) \quad (10)$$

$$= \frac{n}{d} \cdot \left(2 - \frac{1}{2^{d-2}} \right) \quad (11)$$

$$= n \left(\frac{2}{d} - \frac{4}{d \cdot 2^d} \right) \quad (12)$$

◀

► **Theorem 6.** *The ambiguous encoding of n bit long uniformly i.i.d. input \mathcal{A} sequence is achieved with $\frac{2(d-1)}{d \cdot 2^d}$ bits overhead per each original bit.*

Proof. Total overhead can be computed by subtracting the original length n from the sum of the space consumption described in Lemmas 4 and 5. Dividing this value by the n returns the overhead per bit as shown below.

$$\frac{1}{n} \cdot \left[n \cdot \left(1 - \frac{2}{d} + \frac{2(d+1)}{d \cdot 2^d} \right) + n \cdot \left(\frac{2}{d} - \frac{4}{d \cdot 2^d} \right) - n \right] = \frac{d-1}{d \cdot 2^{d-1}} = \frac{2(d-1)}{d \cdot 2^d} \quad (13)$$

◀

Table 1 summarizes the amount of extra bits introduced by the proposed encoding per each original bit in \mathcal{A} . A large overhead, which seems significant for small d , e.g., $d < 8$, may inhibit the usage of the method. However, thanks to the to the exponentially increasing denominator (2^d) in the overhead amount that the extra space consumption quickly becomes

very small, and even negligible. For instance, when $d = 8$, the method produces only 6.8 extra bits per a thousand bit. Similarly, the overhead becomes less than 3 bits per 100K bits, and less than 2 bits per a million bits for the values of $d = 16$ and $d = 20$, respectively. Thus, for $d \geq 8$, an input uniformly i.i.d. bit sequence can be represented with a negligible space overhead by the proposed ambiguous encoding scheme.

2.1 Experimental Verification

During the calculations of the payload and disambiguation information sizes as well as the overhead, the input data has been assumed to be independently and identically distributed. In practice, the input to the proposed method is supposed to be the output of an entropy coder, where the distribution of d -bits long items in such a file may deviate from the perfect assumptions. We would like to evaluate whether such entropy-encoded files still provide enough good uniformity close to the theoretical claims based on uniformly i.i.d. assumption. Therefore, we have conducted experiments on different compressed files to observe how much these theoretical values are caught in practice.

We have selected 16 publicly available files³, where the first ten are *gzip* compressed data from different sources and the remaining six are multimedia files of *mp3*, *mp4*, *jpg*, *webm*, *ogv*, and *flv* formats. The first $d \cdot 2^d$ bits of each file is inspected for distinct values of $d = \{8, 12, 16, 20\}$, and the corresponding observed payload and disambiguation information sizes are computed as well as the overhead bits in each case.

Table 2 includes the comparisons of the observed and theoretical values on each analyzed dimension. The payload size, which is the total length of the concatenated NPF codewords, and the disambiguation information size, which is the total length of the prefix-free encoded codeword lengths, are both observed to be compatible with the theoretical claims. This is also reflected on the overhead bits as a consequence. Thus, in terms of space consumption, the experimental results on compressed data support the theoretical findings based on perfect uniformly i.i.d. input data assumption.

3 Data Security with Reduced Encryption Operations

Given a high-entropy input bit-stream \mathcal{M} , two secret keys \mathcal{K}_1 and \mathcal{K}_2 , and a properly chosen d parameter, the data security scheme $\mathcal{S}(\mathcal{K}_1, \mathcal{K}_2, \mathcal{M}, d)$ aiming reduced encryption operations starts with generating the permutation $\Sigma' = \{\epsilon_1, \epsilon_2, \dots, \epsilon_{2^d}\}$ via a cryptographically secure pseudo-random number generator seeded with the secret key \mathcal{K}_1 . The input data \mathcal{M} is then encoded with the ambiguous coding described in previous section. This encoding generates the *payload*, which is the concatenated NPF codewords, and the *disambiguation information*, which simply specifies the lengths of individual codewords via an optimal prefix-free code. The payload is encrypted with some selected encryption algorithm by using the key \mathcal{K}_2 , where the disambiguation information is kept in plain. Hence, we need to analyze how much information is revealed by the public disambiguation information, and show that the leakage by the disambiguation information section does not provide an advantage for an attacker to break the cipher on the payload.

³ First ten files are available from <http://corpus.canterbury.ac.nz>. and <http://people.unipmn.it/~manzini/lightweight/corpus/>. The multimedia files are from <https://github.com/johndyer/mediaelement-files>.

7:8 Ambiguous Coding For Selective Encryption

■ **Table 2** Verification of the theoretical claims on selected files for $d = \{8, 12, 16, 20\}$.

File name	Payload Size		Disambiguation Information Size		Overhead Bits	
	Thr.	Obs.	Thr.	Obs.	Thr.	Obs.
chr22		1555		500		7
etext99		1515		533		0
gcc		1491		578		21
howtobwt		1510		538		0
howto		1551		511		14
jdk		1522		540		14
rctail		1535		527		14
rfc	1554	1522	508	526	14	0
sprot34		1538		524		14
w3c2		1529		519		0
mp3		1470		585		7
jpg		1426		636		14
mp4		1415		654		21
webm		1496		566		14
ogv		1456		592		0
flv		1571		484		7
<hr/>						
chr22		40961		8213		22
etext99		41103		8060		11
gcc		40764		8410		22
howtobwt		41058		8127		33
howto		41079		8095		22
jdk		41074		8122		44
rctail		41075		8088		11
rfc	40986	40769	8188	8394	22	11
sprot34		41049		8125		22
w3c2		41016		8158		22
mp3		41021		8131		0
jpg		40819		8344		11
mp4		41373		7779		0
webm		40835		8317		0
ogv		40985		8189		22
flv		40796		8367		11
<hr/>						
chr22		917579		131027		30
etext99		917289		131302		15
gcc		917397		131209		30
howtobwt		917518		131088		30
howto		917812		130794		30
jdk		917412		131179		15
rctail		917139		131437		0
rfc	917538	917346	131068	131275	30	45
sprot34		918158		130433		15
w3c2		917707		130899		30
mp3		914926		133695		45
jpg		915821		132770		15
mp4		905887		142689		0
webm		917075		131561		60
ogv		916558		132108		90
flv		915335		133286		45
<hr/>						
chr22		18873040		2098575		95
etext99		18872686		2098853		19
gcc		18879975		2091602		57
howtobwt		18875332		2096207		19
howto		18875502		2096037		19
jdk		18873190		2098406		76
rctail		18876497		2095042		19
rfc	18874410	18873175	2097148	2098364	38	19
sprot34		18878705		2092891		76
w3c2		18876613		2094945		38
mp3		18863837		2107721		38
jpg		18878914		2092625		19
mp4		18898789		2072826		95
webm		18873348		2098210		38
ogv		18875407		2096208		95
flv		18909861		2061735		76

► **Lemma 7.** *The number of distinct messages that can be generated from a given disambiguation information is $2^{n - \frac{2n}{d} + \frac{4n}{d^2}}$ by assuming the codeword set W , parameter d , and message length n are known.*

Proof. A codeword length $\ell = d - i$ appears $\frac{r}{2^i}$ times in the disambiguation information for $i = 1$ to $d - 1$, and represents 2^ℓ distinct symbols. The d bit long codewords appear $\frac{r}{2^{d-1}}$ times, and represent two distinct symbols. Thus, the total number of distinct sequences that can be generated from a known disambiguation information can be counted by

$$2^{\frac{r(d-1)}{2}} \cdot 2^{\frac{r(d-2)}{4}} \cdot \dots \cdot 2^{\frac{r}{2^{d-1}}} \cdot 2^{\frac{r}{2^{d-1}}} = 2^{rd \sum_{i=1}^{d-1} 2^{-i}} \cdot 2^r \sum_{i=1}^{d-1} i 2^{-i} \cdot 2^{\frac{r}{2^{d-1}}} \quad (14)$$

$$= 2^{\frac{rd(2^{d-1}-1) - r(2^d - d - 1) + 1}{2^{d-1}}} \quad (15)$$

$$= 2^{r(d-2 + \frac{4}{2^d})} \quad (16)$$

$$= 2^{n - \frac{2n}{d} + \frac{4n}{d^2}} \quad (17)$$

◀

The result of Lemma 7 is consistent with previous Lemma 5 such that the disambiguation information is not squeezing the possible message space by more than its size. In other words, when the codeword set W is known, plain disambiguation information reduces the possible 2^n message space to $2^{n-\epsilon}$, where $\epsilon = n(\frac{2}{d} - \frac{4}{d \cdot 2^d})$.

However, in the proposed scheme, W is private, and we need to investigate whether that secrecy of W accommodates the loss of information by the public disambiguation data. Lemma 8 shows that for an attacker using the knowledge revealed by the disambiguation information does not provide an advantage over breaking the encryption on the payload as long as the codeword set W is kept secret.

► **Lemma 8.** *The shrinkage in the possible message space due to public disambiguation information can be accommodated by keeping the codeword set W secret in the ambiguous coding of $n \leq \tau \cdot d \cdot 2^d$ bit long data for $\tau = \frac{d-1.44}{2}$.*

Proof. W is a secret permutation of the set $\{0, 1, 2, \dots, 2^d - 1\}$ containing 2^d numbers. Thus, there are $2^d!$ distinct possibilities, which corresponds to $\log 2^d!$ bits of information. On the other hand, the amount of revealed knowledge about the n bit long input by the disambiguation information is $n(\frac{2}{d} - \frac{4}{d \cdot 2^d})$ bits. The advantage gained by keeping W secret should accommodate the loss by making disambiguation information public. This simply yields the following equation.

$$\log(2^d!) \geq n \cdot \left(\frac{2}{d} - \frac{4}{d \cdot 2^d} \right) \quad (18)$$

$$\frac{\ln(2^d!)}{\ln 2} \approx \frac{2^d \cdot \ln 2^d - 2^d}{\ln 2} \geq n \cdot \frac{2}{d} \quad (19)$$

$$\frac{2^d \cdot d \cdot \ln 2 - 2^d}{\ln 2} \geq n \cdot \frac{2}{d} \quad (20)$$

$$2^d(d - 1.44) \geq n \cdot \frac{2}{d} \quad (21)$$

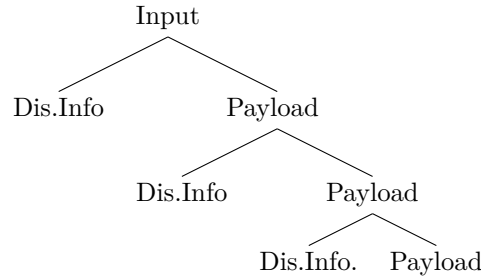
$$d \cdot 2^d \cdot \left(\frac{d - 1.44}{2} \right) \geq n \quad (22)$$

◀

7:10 Ambiguous Coding For Selective Encryption

■ **Table 3** The minimum ($d \cdot 2^d$) and maximum ($d \cdot 2^d \cdot (\frac{d-1.44}{2})$) block sizes that are appropriate according to the proposed ambiguous coding scheme for selected d values.

d	Block Size in bits		d	Block Size in bits	
	min	max		min	max
6	384	875	14	230K	144K
8	2K	6.7K	16	1M	7M
10	10K	43K	18	4.7M	39M
12	49K	256K	20	21M	194M



■ **Figure 3** Sketch of three rounds recursive application of the proposed scheme for further reduction in encryption amount.

Yet another point not to neglect in practice is to choose d such that $2^{d!}$ is no smaller than $2^{\mathcal{K}_1}$. This is to make sure that it should not be easier to try all possible permutations than breaking the secret key (\mathcal{K}_1) of the pseudo-random number generator used in creating the permutation. Assuming the keys used in symmetric encryption schemes are at least 256 bits, $d \geq 6$ seems a lower bound for a security level provided by a 256 bit symmetric encryption since $2^{6!} > 2^{295} > 2^{256}$.

Due to Lemma 8, the choice of d creates an upper bound on the size of the input data that will be subject to the proposed ambiguous coding scheme. On the other hand, it would be appropriate to select d such that the input size is at least $d \cdot 2^d$ bits to confirm with the computations in the size arguments of the payload and the disambiguation information, which assumed all possible 2^d symbols are uniformly i.i.d. on the input. The minimum and maximum block sizes defined by the d parameter are listed in Table 3 considering these facts. Therefore, given an input bit string \mathcal{A} , the ambiguous encoding to be achieved in blocks of the any preferred size in between these values seems appropriate in practice.

The value of d plays a crucial role both in the security and in the to-be-encrypted data size. It is good to choose large d for better security with less (even negligible when $d > 8$) overhead. On the other hand, the payload size is inversely proportional with d , and thus, the reduction in the data volume to be encrypted decreases when d increases.

Thus, to achieve better reductions in the encryption amount, the ambiguous coding can be recursively applied on the payload generated after the first round. Figure 3 sketches this by considering three rounds and Table 4 lists the percentage of gain for each level. For instance, when $d = 8$, the gain in to-be-encrypted volume is around 25 percent. If the payload, which is roughly 75 percent of the original data at the end of this round, is again encoded with the proposed scheme, then gain is improved to more than 40 percent. Even one more round reaches near 60 percent less encryption requirement. Notice that in case of multiple application of the ambiguous coding, the latest payload is encrypted and the remaining disambiguation information are kept plain.

■ **Table 4** Percentages of the disambiguation and payload data with 3 rounds of recursion for various d values calculated according to the lemmas 4 and 5. The bold values represents the percentage of the data to be encrypted.

$d :$		6	8	10	12	14	16
1st Round	Dis. Info.	32.29	24.80	19.96	16.66	14.28	12.50
	Payload	70.31	75.88	80.21	83.39	85.73	87.50
	Overhead	2.60	0.68	0.18	0.04	0.01	0.00
2nd Round	Dis. Info.	22.71	18.82	16.01	13.89	12.25	10.94
	Payload	49.44	57.58	64.34	69.53	73.49	76.57
	Overhead	4.44	1.20	0.32	0.08	0.02	0.01
3rd Round	Dis. Info.	15.96	14.28	12.84	11.58	10.50	9.57
	Payload	34.76	43.69	51.61	57.98	63.00	67.00
	Overhead	5.72	1.60	0.43	0.11	0.03	0.01

4 Conclusions

We have presented an ambiguous coding scheme based on variable-length non-prefix-free codes that splits an input bit-stream into two as the payload and the disambiguation information. We have proved that the overhead at the end of this coding becomes negligible, particularly when $d \geq 8$. The encryption of the payload is supposed to be performed by standard ways, and the disambiguation information is kept plain. Thus, there appears a gain in the amount to be encrypted, which is equal to the disambiguation information size. Proposed ambiguous coding can be applied recursively on the payload generated as depicted in Figure 3 to increase that gain in encryption amount.

We assumed that the input to the ambiguous encoder is uniformly i.i.d. in ideal case, and empirically verified that the compressed volumes ensures the mentioned results. Actually, applying entropy coding before the encryption is a common daily practice, which makes the proposed method to be directly integrated. The *mpeg4* video streams, *jpg* images, compressed text sequences, or *mp3* songs are all typical data sources of high-entropy. Related previous work [5, 16] had stated that although the perfect security of an input data requires a key length equal to its size (one-time pad), high-entropy data can be perfectly secured with much shorter keys. This study addresses another dimension and investigates achieving security of such volumes by encrypting less than their original sizes by using the introduced ambiguous coding scheme.

Reducing the amount of data to-be-encrypted can make sense in scenarios where the encryption process defines a bottleneck in terms of some metrics. Ambiguous coding becomes particularly efficient on securing large collections over power-limited devices, where the cost of encryption becomes heavy in terms of energy. This reduction also helps to increase the throughput of a security pipeline without a need to expand the relatively expensive security hardware. For instance, let's assume a case where the data is waiting to be processed by a hardware security unit. When the amount of data exceeds the capacity of this unit, a bottleneck appears, which can be resolved by increasing the number of such security units. However, adding and managing more security units is costly, particularly when the bottleneck is not so frequent, but only appearing at some time. An alternative solution is to use the proposed ambiguous coding, where instead of expanding the security units, data can be processed appropriately while waiting in the queue, and the amount to be encrypted can be reduced up to desired level by applying the scheme recursively if needed. Notice that as

opposed to previous selective encryption schemes, ambiguous coding supports the security of the whole file instead of securing only the selected partitions. Besides massive multimedia files, small public key files around a few kilobytes that are used in asymmetric encryption schemes are also very suitable inputs for the ambiguous coding. The exchange of public keys via symmetric ciphers can also benefit from the reduction introduced.

The non-prefix-free codes have not received much attention in the literature due to their intrinsic decodability problem. However, such a disadvantage may turn to be an advantage in terms of security systems as investigated in this study. Further security applications based on such ambiguous codes have the potential to be out-of-the box solutions particularly in privacy preserving information retrieval and secure text processing applications.

References

- 1 Boran Adaş, Ersin Bayraktar, and M Oğuzhan Külekci. Huffman codes versus augmented non-prefix-free codes. In *Experimental Algorithms*, pages 315–326. Springer, 2015.
- 2 Norman L Biggs. Prefix free codes. In *Codes: An Introduction to Information Communication and Cryptography*, pages 1–14. Springer, 2008.
- 3 R. Chandramouli, S. Bapatla, K. P. Subbalakshmi, and R. N. Uma. Battery power-aware encryption. *ACM Trans. Inf. Syst. Secur.*, 9(2):162–180, 2006. doi:10.1145/1151414.1151417.
- 4 Marco Dalai and Riccardo Leonardi. Non prefix-free codes for constrained sequences. In *Information Theory, 2005. ISIT 2005. Proceedings. International Symposium on*, pages 1534–1538. IEEE, 2005.
- 5 Yevgeniy Dodis and Adam Smith. Entropic security and the encryption of high entropy messages. In *Theory of Cryptography Conference*, pages 556–577. Springer, 2005.
- 6 Aviezri S. Fraenkel and Shmuel T. Klein. Complexity aspects of guessing prefix codes. *Algorithmica*, 12(4-5):409–419, 1994.
- 7 David W Gillman, Mojdeh Mohtashemi, and Ronald L Rivest. On breaking a huffman code. *IEEE Transactions on Information theory*, 42(3):972–976, 1996.
- 8 Marco Grangetto, Enrico Magli, and Gabriella Olmo. Multimedia selective encryption by means of randomized arithmetic coding. *IEEE Transactions on Multimedia*, 8(5):905–917, 2006.
- 9 Fadi Hamad, Leonid Smalov, and Anne James. Energy-aware security in m-commerce and the internet of things. *IETE Technical review*, 26(5):357–362, 2009.
- 10 Muhammed Oğuzhan Külekci. Uniquely decodable and directly accessible non-prefix-free codes via wavelet trees. In *Information Theory Proceedings (ISIT), 2013 IEEE International Symposium on*, pages 1969–1973. IEEE, 2013.
- 11 Shiguo Lian, Zhongxuan Liu, Zhen Ren, and Haila Wang. Secure advanced video coding based on selective encryption algorithms. *IEEE Transactions on Consumer Electronics*, 52(2):621–629, 2006.
- 12 Ayoub Massoudi, Frédéric Lefebvre, Christophe De Vleeschouwer, Benoit Macq, and J-J Quisquater. Overview on selective encryption of image and video: challenges and perspectives. *EURASIP Journal on Information Security*, 2008(1):1, 2008.
- 13 Rashmi Bangalore Muralidhar. Substitution cipher with nonprefix codes. Master’s thesis, San Jose State University, 2011.
- 14 Nachiketh R Potlapally, Srivaths Ravi, Anand Raghunathan, and Niraj K Jha. A study of the energy consumption characteristics of cryptographic algorithms and security protocols. *IEEE Transactions on Mobile Computing*, 5(2):128–143, 2006.
- 15 Frank Rubin. Cryptographic aspects of data compression codes. *Cryptologia*, 3(4):202–205, 1979.

- 16 Alexander Russell and Hong Wang. How to fool an unbounded adversary with a short key. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 133–148. Springer, 2002.
- 17 Claude Elwood Shannon. A mathematical theory of communication. *Bell system technical journal*, 27(3):379–423, 1948.
- 18 Balemir Uragun. Energy efficiency for unmanned aerial vehicles. In *Machine Learning and Applications and Workshops (ICMLA), 2011 10th International Conference on*, volume 2, pages 316–320. IEEE, 2011.
- 19 Marc Van Droogenbroeck and Raphaël Benedett. Techniques for a selective encryption of uncompressed and compressed images. *ACIVS Advanced Concepts for Intelligent Vision Systems, Proceedings*, pages 90–97, 2002.

A $\frac{3}{2}$ -Approximation Algorithm for the Student-Project Allocation Problem

Frances Cooper¹

School of Computing Science, University of Glasgow
Glasgow, Scotland, UK
f.cooper.1@research.gla.ac.uk
 <https://orcid.org/0000-0001-6363-9002>

David Manlove²

School of Computing Science, University of Glasgow
Glasgow, Scotland, UK
david.manlove@glasgow.ac.uk
 <https://orcid.org/0000-0001-6754-7308>

Abstract

The *Student-Project Allocation problem with lecturer preferences over Students* (SPA-S) comprises three sets of agents, namely students, projects and lecturers, where students have preferences over projects and lecturers have preferences over students. In this scenario we seek a *stable matching*, that is, an assignment of students to projects such that there is no student and lecturer who have an incentive to deviate from their assignee/s. We study SPA-ST, the extension of SPA-S in which the preference lists of students and lecturers need not be strictly ordered, and may contain ties. In this scenario, stable matchings may be of different sizes, and it is known that MAX SPA-ST, the problem of finding a maximum stable matching in SPA-ST, is NP-hard. We present a linear-time $\frac{3}{2}$ -approximation algorithm for MAX SPA-ST and an Integer Programming (IP) model to solve MAX SPA-ST optimally. We compare the approximation algorithm with the IP model experimentally using randomly-generated data. We find that the performance of the approximation algorithm easily surpassed the $\frac{3}{2}$ bound, constructing a stable matching within 92% of optimal in all cases, with the percentage being far higher for many instances.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases Matching problems, Approximation, Algorithms, Stability

Digital Object Identifier 10.4230/LIPIcs.SEA.2018.8

1 Introduction

Background and motivation. In universities all over the world, students need to be assigned to projects as part of their degree programmes. Lecturers typically offer a range of projects, and students may rank a subset of the available projects in preference order. Lecturers may have preferences over students, or over the projects they offer, or they may not have explicit preferences at all. There may also be capacity constraints on the maximum numbers of students that can be allocated to each project and lecturer. The problem of allocating students to projects subject to these preference and capacity constraints is called the *Student-Project Allocation problem* (SPA) [7, Section 5.5][2, 3]. Variants of this problem can be defined for the cases that lecturers have preferences over the students that rank their projects [1],

¹ Supported by an Engineering and Physical Sciences Research Council Doctoral Training Account

² Supported by Engineering and Physical Sciences Research Council grant EP/P028306/01



or over the projects they offer [9], or not at all [6]. In this paper we focus on the first of these cases, where lecturers have preferences over students – the so-called *Student-Project Allocation problem with lecturer preferences over Students* (SPA-S).

Finding an optimal allocation of students to projects manually is time-consuming and error-prone. Consequently many universities automate the allocation process using a centralised algorithm. Given the typical sizes of problem instances (e.g., 130 students at the University of Glasgow, School of Computing Science), the efficiency of the matching algorithm is of paramount importance. In the case of SPA-S, the desired matching must be *stable* with respect to the given preference lists, meaning that no student and lecturer have an incentive to deviate from the given allocation and form an assignment with one another [10].

Abraham et al. [1] described a linear-time algorithm to find a stable matching in an instance I of SPA-S when all preference lists in I are strictly ordered. They also showed that, under this condition, all stable matchings in I are of the same size. In this paper we focus on the variant of SPA-S in which preference lists of students and lecturers can contain ties, which we refer to as the *Student-Project Allocation problem with lecturer preferences over Students including Ties* (SPA-ST). Ties allow both students and lecturers to express indifference in their preference lists (in practice, for example, lecturers may be unable to distinguish between certain groups of students). A stable matching in an instance of SPA-ST can be found in linear time by breaking the ties arbitrarily and using the algorithm of Abraham et al. [1].

The *Stable Marriage problem with Ties and Incomplete lists* (SMTI) is a special case of SPA-ST in which each project and lecturer has capacity 1, and each lecturer offers one project. Given an instance of SMTI, it is known that stable matchings can have different sizes [8], and thus the same is true for SPA-ST. Yet in practical applications it is desirable to match as many students to projects as possible. This motivates MAX SPA-ST, the problem of finding a maximum (cardinality) stable matching in an instance of SPA-ST. This problem is NP-hard, since the corresponding optimisation problem restricted to SMTI, which we refer to as MAX SMTI, is NP-hard [8]. Király [5] described a $\frac{3}{2}$ -approximation algorithm for MAX SMTI. He also showed how to extend this algorithm to the case of the *Hospitals-Residents problem with Ties* (HRT), where HRT is the special case of SPA-ST in which each lecturer l offers one project p , and the capacities of l and p are equal. Yanagisawa [11] showed that MAX SMTI is not approximable within a factor of $\frac{33}{29}$ unless P=NP; the same bound applies to MAX SPA-ST.

Our contribution. In this paper we describe a linear-time $\frac{3}{2}$ -approximation algorithm for MAX SPA-ST. This algorithm is a non-trivial extension of Király’s approximation algorithm for HRT as mentioned above. We also describe an Integer Programming (IP) model to solve MAX SPA-ST optimally. Through a series of experiments on randomly-generated data, we then compare the sizes of stable matchings output by our approximation algorithm with the sizes of optimal solutions obtained from our IP model. Our main finding is that the performance of the approximation algorithm easily surpassed the $\frac{3}{2}$ bound on the generated instances, constructing a stable matching within 92% of optimal in all cases, with the percentage being far higher for many instances.

Note that a natural “cloning” technique, involving transforming an instance I of SPA-ST into an instance I' of SMTI, and then using Király’s $\frac{3}{2}$ -approximation algorithm for SMTI [5] in order to obtain a similar approximation in SPA-ST, does not work in general, as shown in [4, Appendix A]. This motivates the need for a bespoke algorithm for the SPA-ST case.

Structure of this paper. Section 2 gives a formal definition of SPA-ST. Section 3 describes the $\frac{3}{2}$ -approximation algorithm, and the IP model for MAX SPA-ST is given in Section 4. The experimental evaluation is described in Section 5, and Section 6 discusses future work.

2 Formal definition of SPA-ST

An instance I of SPA-ST comprises a set $S = \{s_1, s_2, \dots, s_{n_1}\}$ of *students*, a set $P = \{p_1, p_2, \dots, p_{n_2}\}$ of *projects*, and a set $L = \{l_1, l_2, \dots, l_{n_3}\}$ of *lecturers*. Each project is *offered* by one lecturer, and each lecturer l_k *offers* a set of projects $P_k \subseteq P$, where P_1, \dots, P_k partitions P . Each project $p_j \in P$ has a *capacity* $c_j \in \mathbb{Z}_0^+$, and similarly each lecturer $l_k \in L$ has a *capacity* $d_k \in \mathbb{Z}_0^+$. Each student $s_i \in S$ has a set $A_i \subseteq P$ of *acceptable* projects that they rank in order of preference. *Ties* are allowed in preference lists, where a tie t in a student s_i 's list indicates that s_i is indifferent between all projects in t . Each lecturer $l_k \in L$ has a preference list over the students s_i for which $A_i \cap P_k \neq \emptyset$. Ties may also exist in lecturer preference lists. The *rank* of project p_j on student s_i 's list, denoted $\text{rank}(s_i, p_j)$, is defined as 1 plus the number of projects that s_i strictly prefers to p_j . An analogous definition exists for the rank of a student on a lecturer's list, denoted $\text{rank}(l_k, s_i)$.

An *assignment* M in I is a subset of $S \times P$ such that, for each pair $(s_i, p_j) \in M$, $p_j \in A_i$, that is, s_i finds p_j acceptable. Let $M(s_i)$ denote the set of projects assigned to a student $s_i \in S$, let $M(p_j)$ denote the set of students assigned to a project $p_j \in P$, and let $M(l_k)$ denote the set of students assigned to projects in P_k for a given lecturer $l_k \in L$. A *matching* M is an assignment such that $|M(s_i)| \leq 1$ for all $s_i \in S$, $|M(p_j)| \leq c_j$ for all $p_j \in P$ and $|M(l_k)| \leq d_k$ for all $l_k \in L$. If $s_i \in S$ is assigned in a matching M , we let $M(s_i)$ denote s_i 's assigned project, otherwise $M(s_i)$ is empty.

Given a matching M in I , let $(s_i, p_j) \in (S \times P) \setminus M$ be a student-project pair, where p_j is offered by lecturer l_k . Then (s_i, p_j) is a *blocking pair* of M [1] if 1, 2 and 3 hold as follows:

1. s_i finds p_j acceptable;
2. s_i either prefers p_j to $M(s_i)$ or is unassigned in M ;
3. Either a, b or c holds as follows:
 - a. p_j is *undersubscribed* (i.e., $|M(p_j)| < c_j$) and l_k is *undersubscribed* (i.e., $|M(l_k)| < d_k$);
 - b. p_j is undersubscribed, l_k is full and either $s_i \in M(l_k)$ or l_k prefers s_i to the worst student in $M(l_k)$;
 - c. p_j is full and l_k prefers s_i to the worst student in $M(p_j)$.

Let (s_i, p_j) be a blocking pair of M . Then we say that (s_i, p_j) is of *type* $(3x)$ if 1, 2 and $3x$ are true in the above definition, where $x \in \{a, b, c\}$. In order to more easily describe certain stages of the approximation algorithm, blocking pairs of type $(3b)$ are split into two subtypes as follows. $(3bi)$ defines a blocking pair of type $(3b)$ where s_i is already assigned to another project of l_k 's. $(3bii)$ defines a blocking pair of type $(3b)$ where this is not the case.

A matching M in an instance I of SPA-ST is *stable* if it admits no blocking pair. Define MAX SPA-ST to be the problem of finding a maximum stable matching in SPA-ST and let M_{opt} denote a maximum stable matching for a given instance. Similarly, let MIN SPA-ST be the problem of finding a minimum stable matching in SPA-ST.

3 Approximation algorithm

3.1 Introduction and preliminary definitions

We begin by defining key terminology before describing the approximation algorithm itself in Section 3.2, which is a non-trivial extension of Király's HRT algorithm [5].

A student $s_i \in S$ is either in *phase* 1, 2 or 3. In *phase* 1 there are still projects on s_i 's list that they have not applied to. In *phase* 2, s_i has iterated once through their list and are doing so again whilst a priority is given to s_i on each lecturer's preference list, compared to

other students who tie with s_i . In *phase 3*, s_i is considered unassigned and carries out no more applications. A project p_j is *fully available* if p_j and l_k are both undersubscribed, where lecturer l_k offers p_j . A student s_i *meta-prefers* project p_{j_1} to p_{j_2} if either (i) $\text{rank}(s_i, p_{j_1}) < \text{rank}(s_i, p_{j_2})$, or (ii) $\text{rank}(s_i, p_{j_1}) = \text{rank}(s_i, p_{j_2})$ and p_{j_1} is fully available, whereas p_{j_2} is not. In phase 1 or 2, s_i may be either *available*, *provisionally assigned* or *confirmed*. Student s_i is *available* if they are not assigned to a project. Student s_i is *provisionally assigned* if s_i has been assigned in phase 1 and there is a project still on s_i 's list that meta-prefers to p_j . Otherwise, s_i is *confirmed*.

If a student s_i is a provisionally assigned to project p_j , then (s_i, p_j) is said to be *precarious*. A project p_j is *precarious* if it is assigned a student s_i such that (s_i, p_j) is precarious. A lecturer is *precarious* if they offer a project p_j that is precarious. Lecturer l_k *meta-prefers* s_{i_1} to s_{i_2} if either (i) $\text{rank}(l_k, s_{i_1}) < \text{rank}(l_k, s_{i_2})$, or (ii) $\text{rank}(l_k, s_{i_1}) = \text{rank}(l_k, s_{i_2})$ and s_{i_1} is in phase 2, whereas s_{i_2} is not. The *favourite* projects F_i of a student s_i are defined as the set of projects on s_i 's preference list for which there is no other project on s_i 's list meta-preferred to any project in F_i . A *worst assignee* of lecturer l_k is defined to be a student in $M(l_k)$ of worst rank, with priority given to phase 1 students over phase 2 students. Similarly, a *worst assignee of lecturer l_k in $M(p_j)$* is defined to be a student in $M(p_j)$ of worst rank, prioritising phase 1 over phase 2 students, where l_k offers p_j .

We remark that some of the above terms such as *favourite* and *precarious* have been defined for the SPA-ST setting by extending the definitions of the corresponding terms as given by Király in the HRT context [5].

3.2 Description of the algorithm

Algorithm 1 begins with an empty matching M which will be built up over the course of the algorithm's execution. All students are initially set to be available and in phase 1. The algorithm proceeds as follows. While there are still available students in phase 1 or 2, choose some such student s_i . Student s_i applies to a favourite project p_j at the head of their list, that is, there is no project on s_i 's list that s_i meta-prefers to p_j . Let l_k be the lecturer who offers p_j . We consider the following cases.

- If p_j and l_k are both undersubscribed then (s_i, p_j) is added to M . Clearly if (s_i, p_j) were not added to M , it would potentially be a blocking pair of type (3a).
- If p_j is undersubscribed, l_k is full and l_k is precarious where precarious pair $(s_{i'}, p_{j'}) \in M$ for some project $p_{j'}$ offered by l_k , then we remove $(s_{i'}, p_{j'})$ from M and add pair (s_i, p_j) . This notion of precariousness allows us to find a stable matching of sufficient size even when there are ties in student preference lists (there may also be ties in lecturer preference lists). Allowing a pair $(s_{i'}, p_{j'}) \in M$ to be precarious means that we are noting that $s_{i'}$ has other fully available project options in their preference list at equal rank to $p_{j'}$. Hence, if another student applies to $p_{j'}$ when $p_{j'}$ is full, or to a project offered by l_k where l_k is full, we allow this assignment to happen removing $(s_{i'}, p_{j'})$ from M , since there is a chance that the size of the resultant matching could be increased.
- If on the other hand p_j is undersubscribed, l_k is full and l_k meta-prefers s_i to a worst assignee $s_{i'}$, where $(s_{i'}, p_{j'}) \in M$ for some project $p_{j'}$ offered by l_k , then we remove $(s_{i'}, p_{j'})$ from M and add pair (s_i, p_j) . It makes intuitive sense that if l_k is full and gets an offer to an undersubscribed project from a student s_i that they prefer to a worst assigned student $s_{i'}$, then l_k would want to remove $s_{i'}$ from $p_{j'}$ and take on s_i for $p_{j'}$. Student $s_{i'}$ will subsequently remove $p_{j'}$ from their preference list as l_k will not want to assign to them on re-application. This is done via the Remove-pref method (Algorithm 2).

Algorithm 1 3/2-approximation algorithm for SPA-ST.

Require: An instance I of SPA-ST**Ensure:** Return a stable matching M where $|M| \geq \frac{2}{3}|M_{opt}|$

```

1:  $M \leftarrow \emptyset$ 
2: all students are initially set to be available and in phase 1
3: while there exists an available student  $s_i \in S$  who is in phase 1 or 2 do
4:   let  $l_k$  be the lecturer who offers  $p_j$ 
5:    $s_i$  applies to a favourite project  $p_j \in A(s_i)$ 
6:   if  $p_j$  is fully available then
7:      $M \leftarrow M \cup \{(s_i, p_j)\}$ 
8:   else if  $p_j$  is undersubscribed,  $l_k$  is full and ( $l_k$  is precarious or  $l_k$  meta-prefers  $s_i$  to
a worst assignee) then  $\triangleright$  according to the worst assignee definition in Section 3.1
9:     if  $l_k$  is precarious then
10:       let  $p_{j'}$  be a project in  $P_k$  such that there exists  $(s_{i'}, p_{j'}) \in M$  that is precarious
11:     else  $\triangleright l_k$  is not precarious
12:       let  $s_{i'}$  be a worst assignee of  $l_k$  such that  $l_k$  meta-prefers  $s_i$  to  $s_{i'}$  and let
 $p_{j'} = M(s_{i'})$ 
13:       Remove-Pref( $s_{i'}, p_{j'}$ )
14:     end if
15:      $M \leftarrow M \setminus \{(s_{i'}, p_{j'})\}$ 
16:      $M \leftarrow M \cup \{(s_i, p_j)\}$ 
17:   else if  $p_j$  is full and ( $p_j$  is precarious or  $l_k$  meta-prefers  $s_i$  to a worst assignee in
 $M(p_j)$ ) then
18:     if  $p_j$  is precarious then
19:       identify a student  $s_{i'} \in M(p_j)$  such that  $(s_{i'}, p_j)$  is precarious
20:     else  $\triangleright p_j$  is not precarious
21:       let  $s_{i'}$  be a worst assignee of  $l_k$  in  $M(p_j)$  such that  $l_k$  meta-prefers  $s_i$  to  $s_{i'}$ 
22:       Remove-Pref( $s_{i'}, p_j$ )
23:     end if
24:      $M \leftarrow M \setminus \{(s_{i'}, p_j)\}$ 
25:      $M \leftarrow M \cup \{(s_i, p_j)\}$ 
26:   else
27:     Remove-Pref( $s_i, p_j$ )
28:   end if
29: end while
30: Promote-students( $M$ )
31: return  $M$ ;

```

- If p_j is full and precarious then pair (s_i, p_j) is added to M while precarious pair $(s_{i'}, p_j)$ is removed. As before, this allows $s_{i'}$ to potentially assign to other fully available projects at the same rank as p_j on their list. Since $s_{i'}$ does not remove p_j from their preference list, $s_{i'}$ will get another chance to assign to p_j if these other applications to fully available projects at the same rank are not successful.
- If p_j is full and l_k meta-prefers s_i to a worst assignee $s_{i'}$ in $M(p_j)$, then pair (s_i, p_j) is added to M while $(s_{i'}, p_j)$ is removed. As this lecturer's project is full (and not precarious) the only time they will want to add a student s_i to this project (meaning the removal of another student) is if s_i is preferred to a worst student $s_{i'}$ assigned to that project. Similar to before, $s_{i'}$ will not subsequently be able to assign to this project and so removes it from their preference list via the Remove-pref method (Algorithm 2).

Algorithm 2 Remove-Pref(s_i, p_j) – remove a project from a student’s preference list.

Require: An instance I of SPA-ST and a student s_i and project p_j

Ensure: Return an instance I where p_j is removed from s_i ’s preference list

```

1: remove  $p_j$  from  $s_i$ ’s preference list
2: if  $s_i$ ’s preference list is empty then
3:   reinstate  $s_i$ ’s preference list
4:   if  $s_i$  is in phase 1 then
5:     move  $s_i$  to phase 2
6:   else if  $s_i$  is in phase 2 then
7:     move  $s_i$  to phase 3
8:   end if
9: end if
10: return instance  $I$ 

```

Algorithm 3 Promote-students(M) – remove all blocking pairs of type (3bi).

Require: SPA-ST instance I and matching M that does not contain blocking pairs of type (3a), (3bii) or (3c).

Ensure: Return a stable matching M .

```

1: while there are still blocking pairs of type (3bi) do
2:   Let  $(s_i, p_{j'})$  be a blocking pair of type (3bi)
3:    $M \leftarrow M \setminus \{(s_i, M(s_i))\}$ 
4:    $M \leftarrow M \cup \{(s_i, p_{j'})\}$ 
5: end while
6: return  $M$ 

```

When removing a project from a student s_i ’s preference list (the Remove-pref operation of Algorithm 2), if s_i has removed all projects from their preference list and is in phase 1 then their preference list is reinstated and they are set to be in phase 2. If on the other hand they were already in phase 2, then they are set to be in phase 3 and are hence inactive. The proof that Algorithm 1 produces a stable matching (see [4, Appendix B]) relies only on the fact that a student iterates once through their preference list. Allowing students to iterate through their preference lists a second time when in phase 2 allows us to find a stable matching of sufficient size when there are ties in lecturer preference lists (there may also be ties in student preference lists). This is due to the meta-prefers definition where a lecturer favours one student s_i over another $s_{i'}$ if they are the same rank and s_i is in phase 2 whereas $s_{i'}$ is not. Similar to above, this then allows s_i to steal a position from $s_{i'}$ with the chance that $s_{i'}$ may find another assignment and increase the size of the resultant matching.

After the main while loop has terminated, the final part of the algorithm begins where all blocking pairs of type (3bi) are removed using the Promote-students method (Algorithm 3).

3.3 Proof of correctness

► **Theorem 1.** *Let M be a matching found by Algorithm 1 for an instance I of SPA-ST. Then M is stable and $|M| \geq \frac{2}{3}|M_{opt}|$, where M_{opt} is a maximum stable matching in I .*

Proof. Theorems 18, 22 and Theorem 30, proved in [4, Appendix B], show that M is stable, and that Algorithm 1 runs in polynomial time and has performance guarantee $\frac{3}{2}$. The proofs required for this algorithm are naturally longer and more complex than given by Király [5]

for SMTI, as SPA-ST generalises SMTI to the case that lecturers can offer multiple projects, and projects and lecturers may have capacities greater than 1. These extensions add extra components to the definition of a blocking pair (given in Section 2) which in turn adds complexity to the algorithm and its proof of correctness. ◀

Appendix B.5 in [4] gives a simple example instance where a matching found by Algorithm 1 is *exactly* $\frac{2}{3}$ times the optimal size, hence the analysis of the performance guarantee is tight.

4 IP model

In this section we present an IP model for MAX SPA-ST. For the stability constraints in the model, it is advantageous to use an equivalent condition for stability, as given by the following lemma, whose proof can be found in [4, Appendix C].

► **Lemma 2.** *Let I be an instance of SPA-ST and let M be a matching in I . Then M is stable if and only if the following condition, referred to as condition $(*)$ holds: For each student $s_i \in S$ and project $p_j \in P$, if s_i is unassigned in M and finds p_j acceptable, or s_i prefers p_j to $M(s_i)$, then either:*

- l_k is full, $s_i \notin M(l_k)$ and l_k prefers the worst student in $M(l_k)$ to s_i or is indifferent between them, or;
- p_j is full and l_k prefers the worst student in $M(p_j)$ to s_i or is indifferent between them, where l_k is the lecturer offering p_j .

The key variables in the model are binary-valued variables x_{ij} , defined for each $s_i \in S$ and $p_j \in P$, where $x_{ij} = 1$ if and only if student s_i is assigned to project p_j . Additionally, we have binary-valued variables α_{ij} and β_{ij} for each $s_i \in S$ and $p_j \in P$. These variables allow us to more easily describe the stability constraints below. For each $s_i \in S$ and $l_k \in L$, let

$$T_{ik} = \{s_u \in S : \text{rank}(l_k, s_u) \leq \text{rank}(l_k, s_i) \wedge s_u \neq s_i\}.$$

That is, T_{ik} is the set of students ranked at least as highly as student s_i in lecturer l_k 's preference list not including s_i . Also, for each $p_j \in P$, let

$$T_{ijk} = \{s_u \in S : \text{rank}(l_k, s_u) \leq \text{rank}(l_k, s_i) \wedge s_u \neq s_i \wedge p_j \in A(s_u)\}.$$

That is, T_{ijk} is the set of students s_u ranked at least as highly as student s_i in lecturer l_k 's preference list, such that project p_j is acceptable to s_u , not including s_i . Finally, let $S_{ij} = \{p_r \in P : \text{rank}(s_i, p_r) \leq \text{rank}(s_i, p_j)\}$, that is, S_{ij} is the set of projects ranked at least as highly as project p_j in student s_i 's preference list, including p_j . Figure 1 shows the IP model for MAX SPA-ST.

Equation (1) enforces $x_{ij} = 0$ if s_i finds p_j unacceptable. Inequality (2) ensures that a student may be assigned to a maximum of one project. Inequalities (3) and (4) ensure that project and lecturer capacities are enforced. In the left hand side of Inequality (5), if $1 - \sum_{p_r \in S_{ij}} x_{ir} = 1$, then either s_i is unmatched or s_i prefers p_j to $M(s_i)$. This also ensures that either $\alpha_{ij} = 1$ or $\beta_{ij} = 1$, described in Inequalities (6) and (7). Inequality (6) ensures that, if $\alpha_{ij} = 1$, the number of students ranked at least as highly as student s_i by l_k (not including s_i) and assigned to l_k must be at least l_k 's capacity d_k . Inequality (7) ensures that, if $\beta_{ij} = 1$, the number of students ranked at least as highly as student s_i in lecturer l_k 's preference list (not including s_i) and assigned to p_j must be at least p_j 's capacity c_j .

Finally, for our optimisation we maximise the sum of all x_{ij} variables in order to maximise the number of students assigned. The following result, proved in [4, Appendix C], establishes the correctness of the IP model.

maximise: $\sum_{s_i \in S} \sum_{p_j \in P} x_{ij}$	
subject to:	
1. $x_{ij} = 0$	$\forall s_i \in S \quad \forall p_j \in P, p_j \notin A(s_i)$
2. $\sum_{p_j \in P} x_{ij} \leq 1$	$\forall s_i \in S$
3. $\sum_{s_i \in S} x_{ij} \leq c_j$	$\forall p_j \in P$
4. $\sum_{s_i \in S} \sum_{p_j \in P_k} x_{ij} \leq d_k$	$\forall l_k \in L$
5. $1 - \sum_{p_r \in S_{ij}} x_{ir} \leq \alpha_{ij} + \beta_{ij}$	$\forall s_i \in S \quad \forall p_j \in P$
6. $\sum_{s_u \in T_{ik}} \sum_{p_r \in P_k} x_{ur} \geq d_k \alpha_{ij}$	$\forall s_i \in S \quad \forall p_j \in P$
7. $\sum_{s_u \in T_{ijk}} x_{uj} \geq c_j \beta_{ij}$	$\forall s_i \in S \quad \forall p_j \in P$
$x_{ij} \in \{0, 1\}, \quad \alpha_{ij} \in \{0, 1\}, \quad \beta_{ij} \in \{0, 1\}$	$\forall s_i \in S \quad \forall p_j \in P$

■ **Figure 1** IP model for MAX SPA-ST.

► **Theorem 3.** *Given an instance I of SPA-ST, let J be the IP model as defined in Figure 1. A maximum stable matching in I corresponds to an optimal solution in J and vice versa.*

5

 Experimental evaluation

5.1 Methodology

Experiments were conducted on the approximation algorithm and the IP model using randomly-generated data in order to measure the effects on matching statistics when changing parameter values relating to (1) instance size, (2) probability of ties in preference lists, and (3) preference list lengths. Two further experiments (referred to as (4) and (5) below) explored scalability properties for both techniques. Instances were generated using both existing and new software. The existing software is known as the *Matching Algorithm Toolkit* and is a collaborative project developed by students and staff at the University of Glasgow.

For a given SPA-ST instance, let the total project and lecturer capacities be denoted by c_P and d_L , respectively. Note that these capacities were distributed randomly, subject to there being a maximum difference of 1 between the capacities of any two projects or any two lecturers (to ensure uniformity). The minimum and maximum size of student preference lists is given by l_{min} and l_{max} , and t_s represents the probability that a project on a student's preference list is tied with the next project. Lecturer preference lists were generated initially from the student preference lists, where a lecturer l_k must rank a student if a student ranks a project offered by l_k . These lists were randomly shuffled and t_l denotes the ties probability for lecturer preference lists. A linear distribution was used to make some projects more popular than others and in all experiments the most popular project is around 5 times more

popular than the least. This distribution influenced the likelihood of a student finding a given project acceptable. Parameter details for each experiment are given below.

- (1) **Increasing instance size:** 10 sets of 10,000 instances were created (labelled SIZE1, ..., SIZE10). The number of students n_1 increased from 100 to 1000 in steps of 100, with $n_2 = 0.6n_1$, $n_3 = 0.4n_1$, $c_P = 1.4n_1$, $d_L = 1.2n_1$. The probabilities of ties in preference lists were $t_s = t_l = 0.2$ throughout all instance sets. Lengths of preference lists $l_{min} = 3$ and $l_{max} = 5$ also remained the same and were kept low to ensure a wide variability in stable matching size per instance.
- (2) **Increasing probability of ties:** 11 sets of 10,000 instances were created (labelled TIES1, ..., TIES11). Throughout all instance sets $n_1 = 300$, $n_2 = 250$, $n_3 = 120$, $c_P = 420$, $d_L = 360$, $l_{min} = 3$ and $l_{max} = 5$. The probabilities of ties in student and lecturer preference lists increased from $t_s = t_l = 0.0$ to $t_s = t_l = 0.5$ in steps of 0.05.
- (3) **Increasing preference list lengths:** 10 sets of 10,000 instances were generated (labelled PREF1, ..., PREF10). Similar to the TIES cases, throughout all instance sets $n_1 = 300$, $n_2 = 250$, $n_3 = 120$, $c_P = 420$ and $d_L = 360$. Additionally, $t_s = t_l = 0.2$. Preference list lengths increased from $l_{min} = l_{max} = 1$ to $l_{min} = l_{max} = 10$ in steps of 1.
- (4) **Instance size scalability:** 5 sets of 10 instances were generated (labelled SCALS1, ..., SCALS5). All instance sets in this experiment used the same parameter values as the SIZE experiment, except the number of students n_1 increased from 10,000 to 50,000 in steps of 10,000.
- (5) **Preference list scalability:** Finally, 6 sets of 10 instances were created (labelled SCALP1, ..., SCALP6). Throughout all instance sets $n_1 = 500$ with the same values for other parameters as the SIZE experiment. However in this case ties were fixed at $t_s = t_l = 0.4$, and $l_{min} = l_{max}$ increasing from 25 to 150 in steps of 25.

For each generated instance, we ran the $\frac{3}{2}$ -approximation algorithm and then used the IP model to find a maximum stable matching. We also computed a minimum stable matching using a simple adaptation of our IP model for MAX SPA-ST, in order to measure the spread in the sizes of stable matchings. A timeout of 1800 seconds (30 minutes) was imposed on all instance runs. All experiments were conducted using a machine with 32 cores, 8×64GB RAM and Dual Intel® Xeon® CPU E5-2697A v4 processors. The operating system was Ubuntu version 17.04 with all code compiled in Java version 1.8, where the IP models were solved using Gurobi version 7.5.2. Each approximation algorithm instance was run on a single thread while each IP instance was run on two threads. No attempt was made to parallelise Java garbage collection. Repositories for the code and data can be found at <https://doi.org/10.5281/zenodo.1183221> and <https://doi.org/10.5281/zenodo.1186823> respectively.

Correctness testing was conducted over all generated instances. This consisted of (1) ensuring that each matching produced by the approximation algorithm was at least $\frac{2}{3}$ the size of maximum stable matching, as found by the IP, and, (2) testing that a given allocation was stable and adhered to all project and lecturer capacities. This was run over all output from both the approximation algorithm and the IP-based algorithm.

5.2 Experimental results

Experimental results can be seen in Tables 1, 2, 3 and 4. Tables 1, 2 and 3 show the results from Experiments 1, 2 and 3 respectively (in which the instance size, probability of ties and preference list lengths were increased, respectively). From this point onwards an *optimal* matching refers to a maximum stable matching. In these tables, column ‘minimum A/Max’ gives the minimum ratio of approximation algorithm matching size to optimal

matching size that occurred, ‘% A=Max’ displays the percentage of times the approximation algorithm achieved an optimal result, and ‘% A \geq 0.98Max’ shows the percentage of times the approximation algorithm achieved a result at least 98% of optimal. The ‘average size’ columns are somewhat self explanatory, with sub-columns ‘A/Max’ and ‘Min/Max’ showing the average approximation algorithm matching size and minimum stable matching size as a fraction of optimal. Finally, ‘average total time’ indicates the time taken for model creation, solving and outputting results *per instance*. The main findings are summarised below.

- *The approximation algorithm consistently far exceeds its $\frac{3}{2}$ bound.* Considering the column labelled ‘minimum A/Max’ in Tables 1, 2 and 3, we see that the smallest value was within the SIZE1 instance set with a ratio of 0.9286. This is well above the required bound of $\frac{2}{3}$.
- *On average the approximation algorithm provides results that are closer in size to the average maximum stable matching than the minimum stable matching.* The columns ‘A/Max’ and ‘Min/Max’ show that, on average, for each instance set, the approximation algorithm produces a solution that is within 98% of maximum and far closer to the maximum size than to the minimum size.

Table 4 shows the scalability results for increasing instance sizes (Experiment 4) and increasing preference list lengths (Experiment 5). The ‘instances completed’ column indicates the number of instances completed before timeout occurred. In addition to showing the average total time taken (where ‘total’ includes model creation time and solution time), the column ‘average solve time’ displays the time taken to either execute the approximation algorithm, or solve the IP model (in both cases, model creation time is excluded).

For Experiment 4, the number of instances solved within the 30-minute timeout reduced from 10 to 0 for the IP-based algorithm finding the maximum stable matching. However, even for the largest instance set sizes the approximation algorithm was able to solve all instances on average within a total of 21 seconds (0.8 seconds of which was used to actually execute the algorithm).

For Experiment 5, with a higher probability of ties and increasing preference list lengths, the IP-based algorithm was only able to solve all the instances of one instance set (SCALP2) within 30 minutes each, however the approximation algorithm took less than 0.3 seconds on average to return a solution for each instance. This shows that the approximation algorithm is useful for either larger or more complex instances than the IP-based algorithm can handle, motivating its use for real world scenarios.

6 Future work

This paper has described a $\frac{3}{2}$ -approximation algorithm for MAX SPA-ST. It remains open to describe an approximation algorithm that has a better performance guarantee, and/or to prove a stronger lower bound on the inapproximability of the problem than the current best bound of $\frac{33}{29}$ [11]. Further experiments could also measure the extent to which the order that students apply to projects in Algorithm 1 affects the size of the stable matching generated.

The work in this paper has mainly focused on the size of stable matchings. However, it is possible for a stable matching to admit a *blocking coalition*, where a permutation of student assignments could improve the allocations of the students and lecturers involved without harming anyone else. Since permutations of this kind cannot change the size of the matching they are not studied further here, but would be of interest for future work.

Table 1 Increasing instance size experimental results.

Case	minimum		% A=Max	% A ≥ 0.98Max	A	Min	average size		average total time (ms)		
	A/Max	%					Max	A/Max	Min/Max	A	Min
SIZE1	0.9286	17.8	0.0	62.7	96.4	92.0	97.8	0.986	0.941	147.6	137.8
SIZE2	0.9585	1.6	0.0	62.6	192.6	183.4	195.7	0.984	0.937	230.6	210.6
SIZE3	0.9556	0.1	0.0	63.7	288.7	274.9	293.7	0.983	0.936	346.4	313.4
SIZE4	0.9644	0.0	0.0	65.6	384.9	366.4	391.7	0.983	0.935	488.7	429.3
SIZE5	0.9654	0.0	0.0	66.5	481.0	457.7	489.6	0.982	0.935	660.3	555.6
SIZE6	0.9641	0.0	0.0	66.8	577.2	549.3	587.7	0.982	0.935	862.3	713.0
SIZE7	0.9679	0.0	0.0	65.4	673.3	640.5	685.7	0.982	0.934	1127.8	900.6
SIZE8	0.9684	0.0	0.0	67.4	769.5	732.0	783.8	0.982	0.934	1437.3	1098.2
SIZE9	0.9653	0.0	0.0	68.6	865.6	823.4	881.7	0.982	0.934	1784.3	1343.9
SIZE10	0.9701	0.0	0.0	68.0	961.7	914.7	979.7	0.982	0.934	2281.2	1651.0

Table 2 Increasing probability of ties experimental results.

Case	minimum		% A=Max	% A ≥ 0.98Max	A	Min	average size		average total time (ms)		
	A/Max	%					Max	A/Max	Min/Max	A	Min
TIES1	1.0000	100.0	0.0	100.0	284.0	284.0	284.0	1.000	1.000	184.0	186.9
TIES2	0.9792	38.0	0.0	100.0	284.9	282.0	285.8	0.997	0.987	192.4	194.7
TIES3	0.9722	12.1	0.0	99.3	285.9	279.9	287.9	0.993	0.972	201.0	203.1
TIES4	0.9655	3.4	0.0	95.2	287.0	277.6	289.9	0.990	0.958	213.3	214.5
TIES5	0.9626	1.0	0.0	82.5	288.0	275.1	291.9	0.986	0.942	234.3	231.0
TIES6	0.9558	0.4	0.0	66.7	289.2	272.4	294.0	0.984	0.927	274.2	260.6
TIES7	0.9486	0.2	0.0	52.9	290.3	269.4	295.7	0.982	0.911	358.3	311.3
TIES8	0.9527	0.2	0.0	46.4	291.4	266.2	297.2	0.980	0.896	577.3	380.7
TIES9	0.9467	0.2	0.0	50.4	292.5	262.7	298.3	0.980	0.880	1234.1	427.5
TIES10	0.9529	0.5	0.0	61.9	293.7	258.9	299.1	0.982	0.866	2903.4	409.1
TIES11	0.9467	1.0	0.0	74.2	294.8	254.8	299.5	0.984	0.851	5756.9	377.4

■ **Table 3** Increasing preference list length experimental results.

Case	minimum		% A=Max	% A≥0.98Max	A	Min	average size		Min/Max	A	average total time (ms)	
	A/Max	100.0					Max	A/Max			Min	Max
PREF1	1.0000	100.0	100.0	100.0	215.0	215.0	215.0	1.000	1.000	74.3	107.5	105.1
PREF2	0.9699	12.3	99.0	99.0	249.1	249.1	264.1	0.993	0.943	67.5	133.8	128.7
PREF3	0.9617	1.2	84.0	84.0	266.4	266.4	284.7	0.987	0.936	68.1	181.4	174.0
PREF4	0.9623	1.0	82.8	82.8	277.0	277.0	293.9	0.987	0.943	69.1	249.7	242.6
PREF5	0.9661	4.2	95.1	95.1	283.9	283.9	297.7	0.990	0.954	68.3	346.7	340.3
PREF6	0.9732	15.7	99.5	99.5	288.7	288.7	299.1	0.994	0.965	66.1	472.4	440.6
PREF7	0.9767	36.2	100.0	100.0	292.1	292.1	299.7	0.997	0.975	64.5	638.3	550.9
PREF8	0.9833	58.2	100.0	100.0	294.4	294.4	299.9	0.998	0.982	64.1	811.9	660.3
PREF9	0.9866	75.5	100.0	100.0	296.1	296.1	299.9	0.999	0.987	63.4	1032.2	789.1
PREF10	0.9900	87.3	100.0	100.0	297.4	297.4	300.0	1.000	0.991	104.3	1239.4	931.0

■ **Table 4** Scalability experimental results.

Case	instances completed			average solve time (ms)			average total time (ms)		
	A	Min	Max	A	Min	Max	A	Min	Max
SCALS1	10	10	10	136.5	126162.8	225917.9	1393.8	127980.3	227764.3
SCALS2	10	10	9	242.4	348849.4	1091424.2	5356.7	353272.3	1096045.6
SCALS3	10	10	0	491.7	777267.7	N/A	13095.3	785421.2	N/A
SCALS4	10	7	0	718.8	1049122.0	N/A	18883.5	1062076.4	N/A
SCALS5	10	7	0	803.5	1288961.1	N/A	20993.0	1307728.7	N/A
SCALP1	10	0	9	25.1	N/A	93086.0	193.3	N/A	94242.9
SCALP2	10	1	10	23.3	1425177.0	626774.9	189.4	1428844.0	631225.2
SCALP3	10	0	3	31.7	N/A	867107.7	196.6	N/A	882251.0
SCALP4	10	0	1	37.8	N/A	1551376.0	248.5	N/A	1594201.0
SCALP5	10	0	0	59.0	N/A	N/A	283.7	N/A	N/A
SCALP6	10	0	0	45.7	N/A	N/A	288.4	N/A	N/A

References

- 1 D.J. Abraham, R.W. Irving, and D.F. Manlove. Two algorithms for the student-project allocation problem. *Journal of Discrete Algorithms*, 5:73–90, 2007.
- 2 R. Calvo-Serrano, G. Guillén-Gosálbez, S. Kohn, and A. Masters. Mathematical programming approach for optimally allocating students' projects to academics in large cohorts. *Education for Chemical Engineers*, 20:11–21, 2017.
- 3 M. Chiarandini, R. Fagerberg, and S. Gualandi. Handling preferences in student-project allocation. *Annals of Operations Research*, to appear, 2018.
- 4 F. Cooper and D. Manlove. A $\frac{3}{2}$ -approximation algorithm for the Student-Project Allocation problem. Technical Report 1804.02731, Computing Research Repository, Cornell University Library, 2018. Available from <http://arxiv.org/abs/1804.02731>.
- 5 Z. Király. Linear time local approximation for maximum stable marriage. *Algorithms*, 6:471–484, 2013.
- 6 A. Kwanashie, R.W. Irving, D.F. Manlove, and C.T.S. Sng. Profile-based optimal matchings in the Student–Project Allocation problem. In *Proceedings of IWOCA '14: the 25th International Workshop on Combinatorial Algorithms*, volume 8986 of *Lecture Notes in Computer Science*, pages 213–225. Springer, 2015.
- 7 D.F. Manlove. *Algorithmics of Matching Under Preferences*. World Scientific, 2013.
- 8 D.F. Manlove, R.W. Irving, K. Iwama, S. Miyazaki, and Y. Morita. Hard variants of stable marriage. *Theoretical Computer Science*, 276(1-2):261–279, 2002.
- 9 D.F. Manlove and G. O'Malley. Student-project allocation with preferences over projects. *Journal of Discrete Algorithms*, 6:553–560, 2008.
- 10 A.E. Roth. The evolution of the labor market for medical interns and residents: a case study in game theory. *Journal of Political Economy*, 92(6):991–1016, 1984.
- 11 H. Yanagisawa. *Approximation Algorithms for Stable Marriage Problems*. PhD thesis, Kyoto University, School of Informatics, 2007.

How Good Are Popular Matchings?

Krishnapriya A M¹

Citrix Research and Development India
krishnapriya.am@citrix.com

Meghana Nasre

Indian Institute of Technology Madras India
meghana@cse.iitm.ac.in

Prajakta Nimbhorkar

Chennai Mathematical Institute India and UMI ReLaX
prajakta@cmi.ac.in

Amit Rawat²

University of Massachusetts Amherst USA
amitrawat@umass.edu

Abstract

In this paper, we consider the Hospital Residents problem (HR) and the Hospital Residents problem with Lower Quotas (HRLQ). In this model with two sided preferences, stability is a well accepted notion of optimality. However, in the presence of lower quotas, a stable and feasible matching need not exist. For the HRLQ problem, our goal therefore is to output a *good* feasible matching assuming that a feasible matching exists. Computing matchings with minimum number of blocking pairs (Min-BP) and minimum number of blocking residents (Min-BR) are known to be NP-Complete. The only approximation algorithms for these problems work under severe restrictions on the preference lists. We present an algorithm which circumvents this restriction and computes a *popular* matching in the HRLQ instance. We show that on data-sets generated using various generators, our algorithm performs very well in terms of blocking pairs and blocking residents. Yokoi [20] recently studied *envy-free* matchings for the HRLQ problem. We propose a simple modification to Yokoi's algorithm to output a *maximal envy-free* matching. We observe that popular matchings outperform envy-free matchings on several parameters of practical importance, like size, number of blocking pairs, number of blocking residents.

In the absence of lower quotas, that is, in the Hospital Residents (HR) problem, stable matchings are guaranteed to exist. Even in this case, we show that popularity is a practical alternative to stability. For instance, on synthetic data-sets generated using a particular model, as well as on real world data-sets, a popular matching is on an average 8-10% larger in size, matches more number of residents to their top-choice, and more residents prefer the popular matching as compared to a stable matching. Our comprehensive study reveals the practical appeal of popular matchings for the HR and HRLQ problems. To the best of our knowledge, this is the first study on the empirical evaluation of popular matchings in this setting.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis

Keywords and phrases bipartite graphs, hospital residents, lower-quotas, popular matchings

Digital Object Identifier 10.4230/LIPIcs.SEA.2018.9

¹ Part of this work was done when the author was an M.Tech. student at IIT Madras.

² Part of this work was done when the author was an MS student at IIT Madras.



$r_1 :$	h_1	h_2	$[0, 2]$	$h_1 :$	r_1	r_2	r_3
$r_2 :$	h_1	h_2	$[1, 1]$	$h_2 :$	r_2	r_1	
$r_3 :$	h_1						

■ **Figure 1** A hospital residents instance G with $\mathcal{R} = \{r_1, r_2, r_3\}$ and $\mathcal{H} = \{h_1, h_2\}$. The quotas of the hospitals are $q^-(h_1) = 0, q^+(h_1) = 2, q^-(h_2) = q^+(h_2) = 1$. The preferences can be read from the tables as follows: r_1 prefers h_1 over h_2 and so on. The three matchings M_1, M_2 and M_3 are shown below. M_2 and M_3 are feasible but unstable since (r_2, h_1) blocks M_2 and (r_1, h_1) blocks M_3 .
 $M_1 = \{(r_1, h_1), (r_2, h_1)\}$ $M_2 = \{(r_1, h_1), (r_2, h_2), (r_3, h_1)\}$ $M_3 = \{(r_1, h_2), (r_2, h_1), (r_3, h_1)\}$.

Related Version A full version of the paper is available at
<http://www.cse.iitm.ac.in/~meghana/papers/SEA-full-version.pdf>.

Supplement Material Data [1], <http://www.cse.iitm.ac.in/~meghana/projects/datasets/popular.zip>, Code [5], <https://github.com/rawatamit/GraphMatching>

1 Introduction

In this paper, we study two problems – the Hospital Residents (HR) problem and the Hospital Residents problem with Lower Quotas (HRLQ). The input to the HR problem is a bipartite graph $G = (\mathcal{R} \cup \mathcal{H}, E)$ where \mathcal{R} denotes a set of residents, \mathcal{H} denotes a set of hospitals, and an edge $(r, h) \in E$ denotes that r and h are acceptable to each other. Each vertex has a *preference list* which is a strict ordering on its neighbors. Further, each hospital has a positive upper-quota $q^+(h)$. In the HRLQ problem, additionally, a hospital has a non-negative lower-quota $q^-(h)$. A *matching* M is a subset of E such that every resident is assigned at most one hospital and every hospital is assigned at most upper-quota many residents. Let $M(r)$ denote the hospital to which resident r is matched in M . Analogously, let $M(h)$ denote the set of residents that are matched to h in M . A matching M in an HRLQ instance is *feasible* if for every hospital h , $q^-(h) \leq |M(h)| \leq q^+(h)$. The goal is to compute a feasible matching that is *optimal* with respect to the preferences of the residents and the hospitals. When both sides of the bipartition express preferences, *stability* is a well-accepted notion of optimality. A stable matching is defined by the absence of a *blocking pair*.

► **Definition 1.** A pair $(r, h) \in E \setminus M$ blocks M if either r is unmatched in M or r prefers h over $M(r)$ and either $|M(h)| < q^+(h)$ or h prefers r over some $r' \in M(h)$. A matching M is *stable* if there does not exist any blocking pair w.r.t. M , else M is unstable.

From the seminal result of Gale and Shapley [12], it is known that every instance of the HR problem admits a stable matching and such a matching can be computed in linear time in the size of the instance. In contrast, there exist simple instances of the HRLQ problem which do not admit any matching that is both *feasible and stable*.

Figure 1 shows an HRLQ instance where h_2 has a lower-quota of 1. The instance admits a unique stable matching M_1 which is not feasible. The instance admits two maximum cardinality matchings M_2 and M_3 , both of which are feasible but unstable. This raises the question what is an optimal feasible matching in the HRLQ setting?

Our goal in this paper is to propose *popularity* as a viable option in the HRLQ setting, and compare and contrast it by an extensive experimental evaluation with other approaches proposed for this problem. Before giving a formal definition of popularity, we describe some approaches from the literature to the HRLQ problem.

Hamada et al. [14] proposed the optimality notions (i) minimum number of blocking pairs (Min-BP) or (ii) minimum number of residents that participate in any blocking pair (Min-BR). Unfortunately, computing a matching which is optimal according to any of the two notions is NP-Hard as proved by Hamada et al. [14]. On the positive side, they give approximation algorithms for a special case of HRLQ, complemented by matching inapproximability results. Their approximation algorithms require that all the hospitals with non-zero lower-quota have *complete* preference lists (*CL-restriction*).

There are two recent works [18, 20] which circumvent the CL-restriction and work with the natural assumption that the HRLQ instance admits some feasible matching. Nasre and Nimbhorkar [18] consider the notion of *popularity* and show how to compute a feasible matching which is a maximum cardinality popular matching in the set of feasible matchings. Yokoi [20] considers the notion of *envy-freeness* in the HRLQ setting. We define both popularity and envy-freeness below.

Popular matchings. Popular matchings are defined based on comparison of two matchings with respect to votes of the participants. To define popular matchings, first we describe how residents and hospitals vote between two matchings M and M' . We use the definition from [18]. For a resident r unmatched in M , we set $M(r) = \perp$ and assume that r prefers to be matched to any hospital in his preference list (set of acceptable hospitals) over \perp . Similarly, for a hospital h with capacity $q^+(h)$, we set the positions $q^+(h) - |M(h)|$ in $M(h)$ to \perp , so that $|M(h)|$ is always equal to $q^+(h)$. For a vertex $u \in \mathcal{R} \cup \mathcal{H}$, $vote_u(x, y) = 1$ if u prefers x over y , $vote_u(x, y) = -1$ if u prefers y over x and $vote_u(x, y) = 0$ if $x = y$. Thus, for a resident r , $vote_r(M, M') = vote_r(M(r), M'(r))$.

Voting for a hospital. A hospital h is assigned $q^+(h)$ -many votes to compare two matchings M and M' ; this can be viewed as one vote per position of the hospital. A hospital is indifferent between M and M' as far as its $|M(h) \cap M'(h)|$ positions are concerned. For the remaining positions of hospital h , the hospital defines a function \mathbf{corr}_h . This function allows h to decide any pairing of residents in $M(h) \setminus M'(h)$ to residents in $M'(h) \setminus M(h)$. Under this pairing, for a resident $r \in M(h) \setminus M'(h)$, $\mathbf{corr}_h(r, M, M')$ is the resident in $M'(h) \setminus M(h)$ corresponding to r . Then $vote_h(M, M') = \sum_{r \in M(h) \setminus M'(h)} vote_h(r, \mathbf{corr}_h(r, M, M'))$. As an example, consider $q^+(h) = 3$ and $M(h) = \{r_1, r_2, r_3\}$ and $M'(h) = \{r_3, r_4, r_5\}$. To decide its votes, h compares between $\{r_1, r_2\}$ and $\{r_4, r_5\}$ and one possible \mathbf{corr}_h is to pair r_1 with r_4 and r_2 with r_5 . Another \mathbf{corr}_h function is to pair r_1 with r_5 and r_2 with r_4 . The choice of the pairing of residents using \mathbf{corr}_h is determined by the hospital h . With the voting scheme as above, popularity can be defined as follows:

► **Definition 2.** A matching M is more popular than M' if $\sum_{u \in \mathcal{R} \cup \mathcal{H}} vote_u(M, M') > \sum_{u \in \mathcal{R} \cup \mathcal{H}} vote_u(M', M)$. A matching M is popular if there is no matching M' more popular than M .

It is interesting to note that the algorithms for computing popular matchings do not need the \mathbf{corr} function as an input. The matching produced by the algorithms presented in this paper is popular for *any* \mathbf{corr} function that is chosen.

► **Definition 3.** Given a feasible matching M in an HRLQ instance, a resident r has *justified envy* towards r' with $M(r') = h$ if h prefers r over r' and r is either unmatched or prefers h over $M(r)$. A matching is envy-free if there is no resident that has a justified envy towards another resident.

Thus envy-freeness is a relaxation of stability. A matching that is popular amongst feasible matchings always exists [18], but there exist simple instances of the HRLQ problem that admit a feasible matching but do not admit a feasible envy-free matching. Yokoi [20] gave a characterization of HRLQ instances that admit an envy-free matching and an efficient algorithm to compute some envy-free matching.

An envy-free matching need not even be *maximal*. In the example in Figure 1 the matching $M = \{(r_2, h_2)\}$ is envy-free. Note that M is not maximal, and not even a *maximal envy-free matching*. The matching $M' = \{(r_1, h_1), (r_2, h_2)\}$ is a maximal envy-free matching, since addition of any edge to M' violates the envy-free property. The matchings M_2 and M_3 shown in Figure 1 are not envy-free, since r_2 has justified envy towards r_3 in M_2 whereas r_1 has a justified envy towards r_3 in M_3 . The algorithm in [20] outputs the matching M which need not even be maximal; Algorithm 2 in Section 2.2 outputs M' which is guaranteed to be maximal envy-free. Finally, the Algorithm 1 in Section 2.1 outputs M_2 (see Figure 1) which is both maximum cardinality as well as popular.

Popularity is an interesting alternative to stability even in the absence of lower-quotas, that is, in the HR problem. The HR problem is motivated by large scale real-world applications like the National Residency Matching Program (NRMP) in US [3] and the Scottish Foundation Allocation Scheme (SFAS) in Europe [4]. In applications like NRMP and SFAS it is desirable from the social perspective to match as many residents as possible so as to reduce the number of unemployed residents and to provide better staffing to hospitals. Birò et al. [7] report that for the SFAS data-set (2006–2007) the administrators were interested in knowing if relaxing stability would lead to gains in the size of the matching output. It is known that the size of a stable matching could be half that of the maximum cardinality matching M^* , whereas the size of a maximum cardinality popular matching is at least $\frac{2}{3}|M^*|$ (see e.g. [16]). Thus, theoretically, popular matchings give an attractive alternative to stable matchings.

Popular matchings have gained lot of attention and there have been several interesting results in the recent past [6, 8, 10, 9, 16, 18, 19]. The algorithms used and developed in this paper, are inspired by a series of papers [9, 11, 16, 18, 19]. One of the goals in this paper is to complement these theoretical results with an extensive experimental evaluation of the quality of popular matchings. We now summarize our contributions below:

1.1 Our contribution

Results for HRLQ Algorithm for popular matchings. We propose a variant of the algorithm in [18]. Whenever the input HRLQ instance admits a feasible matching, our algorithm outputs a (feasible) popular matching amongst all feasible matchings. We report the following experimental findings:

- *Min-BP and Min-BR objectives:* In all the data-sets, the matching output by our algorithm is at most 3 times the optimal for the Min-BP problem and very close to optimal for the Min-BR problem. In the HRLQ setting, no previous approximation algorithm is known for incomplete preference lists. The only known algorithms which output matchings with theoretical guarantees for the Min-BP and Min-BR problems work under the CL-restricted model [14]. We remark that the algorithm of [18] can not provide any bounded approximation ratio for Min-BP and Min-BR problems as it may output a feasible matching with non-zero blocking pairs and non-zero blocking residents even when the instance admits a stable feasible matching.
- *Comparison with envy-free matchings:* Based on the algorithm by Yokoi [20], we give an algorithm to compute a maximal envy-free matching, if it exists, and compare its *quality* with a popular matching. Besides the fact that popular matchings exist whenever feasible matchings exist, which is not the case with envy-free matchings, our experiments

illustrate that, compared to an envy-free matching, a popular matching is about 32%–43% larger, matches about 15%–38% more residents to their top choice and has an order of magnitude fewer blocking pairs and blocking residents.

Empirical Evaluation of known algorithms for HR. For the HR problem, we implement and extensively evaluate known algorithms for maximum cardinality popular matching and popular amongst maximum cardinality matching on synthetic data-sets as well as limited number of real-world data-sets.

- Our experiments show that a maximum cardinality popular matching, as well as popular amongst maximum cardinality matchings are 8–10% larger in size than a stable matching. We also observe that a maximum cardinality popular matching almost always fares better than a stable matching with respect to the number of residents matched to their rank-1 hospitals, and the number of residents favoring a maximum cardinality popular matching over a stable matching.

We note that these properties cannot be proven theoretically as there are instances where they do not hold. Despite these counter-examples (omitted in the interest of space), our empirical results show that the desirable properties hold on synthetic as well as real-world instances. Hence we believe that popular matchings are a very good practical alternative.

Organization of the paper. In Section 2.1, we present our algorithm to compute a matching that is popular amongst feasible matchings for the HRLQ problem. Our algorithm for computing a maximal envy-free matching is described in Section 2.2. In Section 3 we give details of our experimental setup and the data-generation models developed by us. Section 4 gives our empirical results for the HRLQ and HR problems. We refer the reader to [9, 19] for known algorithms for computing popular matchings in the HR problem.

2 Algorithms for HRLQ problem

In Section 2.1, we present our algorithm to compute a popular matching in the HRLQ problem. As mentioned earlier, our algorithm is a variant of the algorithms in [18]. The algorithm to find a maximal envy-free matching is given in Section 2.2, and its output is a superset of the envy-free matching computed by Yokoi’s algorithm [20].

2.1 Algorithm for Popular matching in HRLQ

In this section, we present an algorithm (Algorithm 1) that outputs a feasible matching M in a given HRLQ instance G , such that M is popular amongst all the feasible matchings in G . We assume that G admits a feasible matching, which can be checked in polynomial time [14].

Algorithm 1 is a modification of the standard hospital-proposing Gale and Shapley algorithm [12] and can be viewed as a two-phase algorithm. The first phase simply computes a stable matching M_s in the input instance ignoring lower quotas. If M_s satisfies the lower quotas of all the hospitals, it just outputs M_s . Otherwise hospitals that are *deficient* in M_s propose with *increased priority*. A hospital h is deficient w.r.t. a matching M if $|M(h)| < q^-(h)$. We implement the increased priority by assigning a *level* to each hospital, so that higher level corresponds to higher priority. A resident always prefers a higher level hospital in its preference list to a lower level hospital, irrespective of their relative positions in its preference list. We show that at most $|\mathcal{R}|$ levels suffice to output a feasible matching in the instance, if one exists. We give a detailed description below.

Algorithm 1 Popular matching in HRLQ.

```

1: Input :  $G = (\mathcal{R} \cup \mathcal{H}, E)$ 
2: set  $M = \emptyset$ ;  $Q = \emptyset$ ;
3: for each  $h \in \mathcal{H}$  do set  $\text{level}(h) = 0$ ; add  $h$  to  $Q$ ;
4: for each  $r \in \mathcal{R}$  do set  $\text{level}(r) = -1$ ;
5: while  $Q$  is not empty do
6:   let  $h = \text{head}(Q)$ ;
7:   let  $\mathcal{S}_h = \text{residents to whom } h \text{ has not yet proposed with level}(h)$ ;
8:   if  $\mathcal{S}_h \neq \emptyset$  then
9:     let  $r$  be the most preferred resident in  $\mathcal{S}_h$ 
10:    if  $r$  is matched in  $M$  (to say  $h'$ ) then
11:      if  $\text{pref}(r, h, h') == 0$  then
12:        add  $h$  to  $Q$ ; goto Step 5;
13:      else
14:         $M = M \setminus \{(r, h')\}$ ;
15:        add  $h'$  to  $Q$  if  $h'$  is not present in  $Q$ ;
16:         $M = M \cup \{(r, h)\}$ ;  $\text{level}(r) = \text{level}(h)$ ;
17:      if  $\text{level}(h) == 0$  and  $|M(h)| < q^+(h)$  then
18:        add  $h$  to  $Q$ ;
19:      else if  $h \in \mathcal{H}_{lq}$  and  $|M(h)| < q^-(h)$  then
20:        add  $h$  to  $Q$ ;
21:      else if  $h \in \mathcal{H}_{lq}$  and  $\text{level}(h) < |\mathcal{R}|$  then
22:         $\text{level}(h) = \text{level}(h) + 1$ ; add  $h$  to  $Q$ ;
23:        //  $h$  starts proposing from the beginning of the preference list.

```

Let $G = (\mathcal{R} \cup \mathcal{H}, E)$ be the given HRLQ instance. Let \mathcal{H}_{lq} denote the set of hospitals which have non-zero lower quota – we call $h \in \mathcal{H}_{lq}$ a lower-quota hospital; we call $h \notin \mathcal{H}_{lq}$ a non-lower quota hospital.

Algorithm 1 begins by initializing the matching M and a queue Q of hospitals to be empty (Step 2). As described above, level of a hospital denotes its current priority, and initially all the hospitals have level 0. All hospitals are added to Q (Step 3). We also assign a level to each resident r , which stores the level of the hospital $M(r)$ at the time when r gets matched to $M(r)$. Initially, all residents are assigned level -1 (Step 4). The main while loop of the algorithm (Step 5) executes as long as Q is non-empty. Steps 6–16 are similar to the execution of the hospital-proposing Gale-Shapley algorithm [13]. When a matched resident r gets a proposal from a hospital h , r compares its current match h' with h using $\text{pref}(r, h, h')$. We define $\text{pref}(r, h, h') = 1$ if (i) $\text{level}(r) < \text{level}(h)$, which means h proposes to r with a higher priority than h' did, or (ii) $\text{level}(r) = \text{level}(h)$ and h has a higher position than h' in the preference list of r . We define $\text{pref}(r, h, h') = 0$ otherwise. Thus matched resident r accepts the proposal of h and rejects h' if and only if $\text{pref}(r, h, h') = 1$. In case the edge (r, h) is added to M (Step 16), the algorithm also sets the level of r equal to the $\text{level}(h)$.

When a hospital h finishes proposing all the residents on its preference list with $\text{level}(h) = i$ and still $|M(h)| < q^-(h)$, $\text{level}(h)$ is incremented by 1 and h is added back to Q . This is done in Step 21. Now h restarts proposing all the residents in its preference list with the new value of $\text{level}(h)$.

Running time and correctness. To see that the algorithm terminates we observe that every non-lower quota hospital proposes to residents in its preference list at most once. Every lower-quota hospital proposes to residents on its preference list at most $|\mathcal{R}|$ times. Thus the running time of our algorithm is $O((|\mathcal{R}| + |\mathcal{H}| + |E|) \cdot |\mathcal{R}|)$. To see the correctness note that when G admits a feasible stable matching our algorithm degenerates to the standard Gale-Shapley hospital proposing algorithm. As proved in [19], a stable matching is popular amongst the set of feasible matchings. In case G admits a feasible matching but no feasible

Algorithm 2 Maximal envy-free matching in HRLQ.

-
- 1: Input : $G = (\mathcal{R} \cup \mathcal{H}, E)$
 - 2: Compute a matching M_1 by Yokoi's algorithm.
 - 3: **if** Yokoi's algorithm declares "no envy-free matching" **then**
 - 4: Return $M = \emptyset$.
 - 5: Let \mathcal{R}' be the set of residents unmatched in M_1 .
 - 6: Let \mathcal{H}' be the set of hospitals such that $|M_1(h)| < q^+(h)$ in G .
 - 7: Let $G' = (\mathcal{R}' \cup \mathcal{H}', E')$ be an induced subgraph of G , where $E' = \{(r, h) \mid r \in \mathcal{R}', h \in \mathcal{H}', h \text{ prefers } r \text{ over its threshold resident } r_h\}$. Set $q^+(h)$ in G' as $q^+(h) - q^-(h)$ in G .
 - 8: Each h has the same relative ordering on its neighbors in G' as in G .
 - 9: $M_2 =$ stable matching in $G' = (\mathcal{R}' \cup \mathcal{H}', E')$.
 - 10: Return $M = M_1 \cup M_2$.
-

stable matching, techniques as in [18] can be employed to show that the output is popular amongst the set of feasible matchings.

► **Theorem 4.** *In an HRLQ instance G , Algorithm 1 outputs a matching that is feasible and popular amongst the set of feasible matchings. If G admits a feasible and stable matching, then Algorithm 1 outputs a stable matching.*

2.2 Algorithm for Maximal Envy-free Matching

Yokoi [20] has given an algorithm to compute an envy-free matching in an HRLQ instance $G = (\mathcal{R} \cup \mathcal{H}, E)$. Yokoi's algorithm works in the following steps:

1. Set the upper quota of each hospital to its lower quota.
2. Set the lower quota of each hospital to 0, call this modified instance G_1 .
3. Find a stable matching M_1 in the modified instance.
4. Return M_1 , if every hospital gets matched to as many residents as its upper quota in G_1 , otherwise declare that there is no envy-free matching in G .

It can be seen that Yokoi's algorithm, as stated above, does not return a *maximal envy-free* matching. In particular, any hospital with lower-quota 0 does not get matched to any resident in Yokoi's algorithm. Hence we propose a simple extension of Yokoi's algorithm to compute a maximal envy-free matching, which contains the matching output by Yokoi's algorithm. We call a matching M in G maximal envy-free if, for any edge $e = (r, h) \in E \setminus M$, $M \cup \{e\}$ is not envy-free. The following definition with respect to Yokoi's output matching M_1 is useful for our algorithm. Our algorithm is described as Algorithm 2.

► **Definition 5.** Let h be a hospital that is under-subscribed in G with respect to M_1 , that is $|M_1(h)| < q^+(h)$. A *threshold resident* r_h for h , if one exists, is the most preferred resident in the preference list of h such that (i) r_h is matched in M_1 , to say h' , and (ii) r_h prefers h over h' . If no such resident exists, we assume a unique dummy resident r_h at the end of h 's preference list to be the threshold resident for h .

Below we prove that the output of Algorithm 2 is a maximal envy-free matching.

► **Theorem 6.** *If G admits an envy-free matching, then Algorithm 2 outputs M which is maximal envy-free in G .*

Proof. Since G admits an envy-free matching, M_1 output by Yokoi's algorithm is non-empty. We prove that M is an envy-free feasible matching, and for each edge $e \in E \setminus M$, $M \cup \{e\}$ is not an envy-free matching.

M is envy-free: Assume, for the sake of contradiction, that a resident r' has a justified envy towards a resident r with respect to M . Thus r' prefers $h = M(r)$ over $h' = M(r')$, and h prefers r' over r . The edge $(r, h) \in M$ and hence either $(r, h) \in M_1$ or $(r, h) \in M_2$. Suppose $(r, h) \in M_1$. Recall that M_1 is stable in the instance G_1 used in Yokoi's algorithm. In this case, the edge (r', h) blocks M_1 in G_1 a contradiction to the stability of M_1 in G_1 . Thus, if possible, let $(r, h) \in M_2$, and hence $(r, h) \in E'$. If r' is unmatched in M_1 , then $(r', h) \in E'$ and (r', h) blocks M_2 in G' , contradicting the stability of M_2 in G' . If r' is matched in M_1 then $r' \notin \mathcal{R}$ and hence $(r', h) \notin E'$. In this case, the threshold resident r_h of h is either same as r' or is a resident whom h prefers over r' . Since h prefers r' over r , h prefers r_h over r . Therefore (r, h) can not be in E' by construction, and hence $(r, h) \notin M_2$. This proves that M is envy-free.

M is maximal envy-free: We now prove that, for any $e = (r, h) \notin M$, $M \cup \{e\}$ is not envy-free. Let $e = (r, h)$, $h \in \mathcal{H}$, $r \in \mathcal{R}$. Clearly, h must be *under-subscribed* in M i.e. $|M(h)| < q^+(h)$, and r must be unmatched in M , otherwise $M \cup \{e\}$ is not a valid matching in G . Let r_h be the threshold resident of h with respect to M_1 . Since r is unmatched in M and hence in M_1 , $r \in \mathcal{R}'$. (i) If h prefers r over r_h , then $(r, h) \in E'$ blocks M_2 in G' , which contradicts the stability of M_2 in G' . (ii) If h prefers r_h over r , then adding the edge (r, h) to M makes r_h have a justified envy towards r . Thus, $M \cup \{(r, h)\}$ is not envy-free. ◀

3 Experimental Setup

The experiments were performed on a machine with a single Intel Core i7-4770 CPU running at 3.40GHz, with 32GB of DDR3 RAM at 1600MHz. The OS was Linux (Kubuntu 16.04.2, 64 bit) running kernel 4.4.0-59. The code [5] is developed in C++ and was compiled using the clang-3.8 compiler with -O3 optimization level. To evaluate the performance of our algorithms, we developed data generators which model preferences of the participants in real-world instances. We use a limited number of publicly available data-sets as well as real-world data-sets from elective allocation at IIT-Madras. All the data-sets generated and used by us are available at [1].

3.1 Data generation models and available data-sets

There are a variety of parameters and all of them could be varied to generate synthetic data-sets. We focus on four prominent parameters – (i) the number of residents $|\mathcal{R}|$, (ii) the number of hospitals $|\mathcal{H}|$, (iii) the length of the preference list of each resident k , (iv) capacity of every hospital cap (by default $cap = |\mathcal{R}|/|\mathcal{H}|$). In the synthetic data-sets, all hospitals have uniform capacity and all residents have the same length of preference list. We use the following three models of data generation, and data-sets publicly available from [2].

- 1. Master:** Here, we model the real-world scenario that there are some hospitals which are in high demand among residents and hence have a larger chance of appearing in the preference list of a resident. Hospitals on the other hand, rely on some global criteria to rank residents. We set up a geometric probability distribution with $p = 0.10$ over the hospitals which denotes the probability with which a hospital is chosen for being included in the preference list of a resident. Each resident samples k hospitals according to the distribution and orders them arbitrarily. We also assume that there exists a master list over the set of residents. For all the neighbours of a hospital, the hospital ranks them according to the master list of residents.

2. **Shuffle:** This model is similar to the first model except that we do not assume a master list on the residents. The residents draw their preference lists as above. Every hospital orders its neighbours uniformly at random. This models the scenario that hospitals may have custom defined ranking criteria. Our **Shuffle** model is closely inspired by the one described by Mahdian and Immorlica [15].
3. **Publicly available HR with couples data-set:** Other than generating data-sets using the three models described above, we used a freely available data-set [2] by Manlove et al. [17] for the HR problem with couples (HRC problem). The instances were only modified with respect to the preference list of residents that participate in a couple, all other aspects of the instance remain the same. Residents participating in a couple can have different copies of a same hospitals on their preference list which are not necessarily contiguous. The preference list is created by only keeping the first unique copy of a hospital in the preference list of a resident.
4. **Elective allocation data from IIT-M:** We use a limited number of real-world data-sets available from the IIT Madras elective allocation. The data-sets are obtained from the SEAT (Student Elective Allocation Tool) which allocates humanities electives and outside department electives for under-graduate students across the institute every semester. The input consists of a set of students and a set of courses with capacities (upper-quotas). Every student submits a preference ordering over a subset of courses and every course ranks students based on custom ranking criteria. This is exactly the HR problem.

HRLQ instances. The above three data-generation models (Master, Shuffle, and Random) are common for generating preferences in HRLQ and HR data-sets. For HRLQ data-sets we additionally need lower quotas. In all our data-sets, around 90% of the hospitals had lower quota at least 1. This is to ensure that the instances are HRLQ instances rather than *nearly* HR instances. Also, the sum of the lower quota of all the hospitals was kept around 50% of the total number of positions available. This ensures that at least half of the residents must be matched to a lower quota hospital. Lastly, we consider only those HRLQ instances which admit a feasible matching but *no stable feasible matching*. This is done by simply discarding instances that admit a feasible stable matching. We discard such instances because if an instance with feasible stable matching, the stable matching is optimal with respect to popularity, envy-freeness and min-BP and min-BR objectives.

Methodology. For reporting our results, we fix a model of generation and the parameters $|\mathcal{R}|, |\mathcal{H}|, k, cap$. For the chosen model and the parameters, we generate 10 data-sets and report arithmetic average for all output parameters on these data-sets.

4 Empirical Evaluation

Here, we present our empirical observations on HRLQ and HR instances. In each case, we define set of parameters on the basis of which we evaluate the quality of our matchings.

4.1 HRLQ instances

In this section we show the performance of Algorithm 1 and Algorithm 2 on data-sets generated using models described earlier. Let G be an instance of the HRLQ problem, and M_s be the stable matching in the instance ignoring lower-quotas. For all our instances M_s is infeasible. Let M_p denote a popular matching output Algorithm 1 in G . Let M_e denote a

9:10 How Good Are Popular Matchings?

■ **Table 1** Data generated using the **Master** model. All values are absolute. $|\mathcal{R}| = 1000, k = 5$.

$ \mathcal{H} $	$Def(M_s, G)$	$S(M) \uparrow$		$BPC(M) \downarrow$		$BR(M) \downarrow$		$\mathcal{R}_1(M) \uparrow$	
		M_p	M_e	M_p	M_e	M_p	M_e	M_p	M_e
100	30.80	885.40	559.00	78.50	2747.00	34.80	822.00	554.10	174.00
20	23.60	897.90	510.66	67.70	3067.33	27.50	803.66	570.40	195.33
10	27.50	912.80	535.50	85.10	2945.00	31.10	770.87	600.40	226.87

■ **Table 2** Data generated using the **Shuffle** model. All values are absolute. $|\mathcal{R}| = 1000, k = 5$.

$ \mathcal{H} $	$Def(M_s, G)$	$S(M) \uparrow$		$BPC(M) \downarrow$		$BR(M) \downarrow$		$\mathcal{R}_1(M) \uparrow$	
		M_p	M_e	M_p	M_e	M_p	M_e	M_p	M_e
100	17.00	892.70	–	27.20	–	19.40	–	350.30	–
20	20.60	915.30	547.00	34.40	2838.20	23.60	808.00	343.90	185.80
10	35.40	930.00	490.33	57.80	3388.00	35.40	853.00	309.30	147.00

maximal envy-free matching output by Algorithm 2. Both M_p and M_e are feasible for G and hence unstable.

Parameters of interest. We now define the parameters of the matching that are of interest in the HRLQ problem. For $M \in \{M_p, M_e\}$ we compute the following.

- $S(M)$: size of the matching M in G .
- $BPC(M)$: number of blocking pairs w.r.t. M in G . Since M is not stable in G , this parameter is expected to be positive; however we would like this parameter to be small.
- $BR(M)$: number of residents that participate in at least one blocking pair w.r.t. M .
- $\mathcal{R}_1(M)$: number of residents matched to their rank-1 hospitals in M .

We additionally compute the deficiency of the stable matching M_s . Hamada et al. [14] showed that $Def(M_s, G)$ is a lower bound on the number of blocking pairs and the number of blocking residents in an HRLQ instance. To analyze the goodness of M_p and M_e w.r.t. the Min-BP and Min-BR objectives, we compare the number of blocking pairs and blocking residents with the lower bound of $Def(M_s, G)$.

- $Def(M_s, G)$: This parameter denotes the deficiency of the stable (but not feasible) matching M_s in G . For every hospital h , let $def(M_s, h) = \max\{0, q^-(h) - |M_s(h)|\}$. The deficiency of the instance G is the sum of the deficiencies of all hospitals.

We now describe our observations for the data-sets generated using various models. For a particular model, we vary the parameters $|\mathcal{R}|, |\mathcal{H}|, k, cap$ to generate HRLQ data-sets using the different models. In all our tables a column with the legend \uparrow implies that larger values are better. Analogously, a column with the legend \downarrow implies smaller values are better.

4.1.1 Popular Matchings versus Maximal Envy-free Matchings

Here we report the quality of popular matchings and envy-free matchings on the parameters of interest listed above on different models.

Table 1 shows the results for popular matchings (M_p) and maximal envy-free matchings (M_e) on data-sets generated using the **Master** model.

Table 2 shows the results for popular matchings (M_p) and maximal envy-free matchings (M_e) on data-sets generated using the **Shuffle** model.

We observe the following from the above two tables.

- **Guaranteed existence:** As noted earlier, envy-free matchings are not guaranteed to exist in contrast to popular matchings which always exist in HRLQ instances. For instance, in the **Shuffle** model, for $|\mathcal{R}| = 1000, |\mathcal{H}| = 100, k = 5$ (Table 2, Row 1) none of the instances admit an envy-free matching. Thus, for the columns M_e we take an average over the instances that admit an envy-free matching.
- **Size:** It is evident from the tables that in terms of size popular matchings are about 32%–43% larger as compared to envy-free matchings (when they exist). See Column $S(M)$ in Table 1 and Table 2.
- **BPC and BR:** In terms of the blocking pairs and blocking residents, popular matchings beat envy-free matchings by over an order of magnitude. We remark that to ensure envy-freeness, several hospitals may to be left under-subscribed. This explains the unusually large blocking pairs and blocking residents in envy-free matchings. Furthermore, note that $Def(M_s, G)$ is a lower-bound on both the number of blocking pairs and blocking residents. On all instances, for popular matchings the number of blocking pairs (BPC) is at most 3 times the optimal whereas the number of blocking residents (BR) is close to the optimal value.
- **Number of Envy-pairs:** Although we do not report it explicitly, the number of envy pairs in an envy-free matching is trivially zero and for any matching it is upper bounded by the number of blocking pairs. Since the number of blocking pairs is significantly small for popular matchings, we conclude that the number of envy-pairs is also small.
- **Number of residents matched to rank-1 hospitals:** For any matching, a desirable criteria is to match as many participants to their top-choice. Again on this count, we see that popular matchings match about 15%–38% more residents to their top choice hospital (See Column $\mathcal{R}_1(M)$ in the above tables).

4.2 Results on HR instances

To analyze the quality of popular matchings on various data-sets, we generate an HR instance $G = (\mathcal{H} \cup \mathcal{R}, E)$, compute a resident optimal stable matching M_s , a maximum cardinality popular matching M_p and a popular matching among maximum cardinality matchings M_m . The matchings M_p and M_m are generated by creating reduced instances G_2 and $G_{|\mathcal{R}|}$ respectively, and executing the resident proposing Gale-Shapley algorithm in the respective instance. Theoretically, we need to construct $G_{|\mathcal{R}|}$ for computing M_m , practically we observe that almost always a constant number suffices say constructing G_{10} suffices. We now describe our output parameters.

Parameters of interest. For any matching M , let $\mathcal{R}_1(M)$ denote the number of residents matched to rank-1 hospitals in M . For two matchings M and M' , let $\mathcal{V}_{\mathcal{R}}(M, M')$ denote the number of residents that prefer M over M' . For each instance we compute the following:

- $S(M_s)$: size of the stable matching M_s in G .
- For M to be one of M_p or M_m define:
 - Δ : $\frac{|M| - |M_s|}{|M_s|} \times 100$. This denotes the percentage increase in size of M_s when compared to M . When comparing M_s with either M_p or M_m , Δ is guaranteed to be non-negative. The larger this value, the better the matching in terms of size as compared to M_s .
 - Δ_1 : $\frac{\mathcal{R}_1(M) - \mathcal{R}_1(M_s)}{\mathcal{R}_1(M_s)} \times 100$. This denotes the percentage increase in number of rank-1 residents of M_s when compared to M . As discussed in the Introduction, there is no guarantee that this value is non-negative. However, we prefer that the value is as large as possible.

9:12 How Good Are Popular Matchings?

■ **Table 3** Data generated using the **Master** model. All values except $S(M_s)$ are percentages.

$ \mathcal{R} = 1000, k = 5$		M_p vs M_s				M_m vs M_s			
$ \mathcal{H} $	$S(M_s)$	$\Delta \uparrow$	$BP(M_p) \downarrow$	$\Delta_1 \uparrow$	$\Delta_{\mathcal{R}} \uparrow$	$\Delta \uparrow$	$BP(M_m) \downarrow$	$\Delta_1 \uparrow$	$\Delta_{\mathcal{R}} \uparrow$
1000	757.90	11.81	4.66	-3.49	5.25	12.79	5.34	-4.02	6.41
100	823.50	12.93	8.57	-1.64	7.41	13.99	9.96	-2.80	8.58
20	870.70	11.65	12.22	0.24	7.32	12.25	14.47	-0.36	7.47
10	890.00	10.68	16.32	0.76	2.37	10.80	16.57	0.73	2.64

- $\Delta_{\mathcal{R}} : \frac{\mathcal{V}_{\mathcal{R}}(M, M_s) - \mathcal{V}_{\mathcal{R}}(M_s, M)}{|\mathcal{R}|} \times 100$. This denotes the percentage increase in number of resident votes of M when compared to M_s . Similar to Δ_1 , there is no apriori guarantee that more residents prefer M over M_s . A positive value for this parameter indicates that M is *more resident popular* as compared to M_s . That is, in an election where only residents vote, a majority of the residents would like to move from M_s to M .
- $BP(M) : \frac{\text{number of blocking pairs in } M}{|E| - |M|} \times 100$. The minimum value for $BP(M)$ can be 0 (for a stable matching) and the maximum value can be 100, due to the choice of the denominator ($|E| - |M|$). Since matchings M_m and M_p are not stable, this parameter is expected to be positive and we consider it as the *price* we pay to get positive values for Δ , Δ_1 and $\Delta_{\mathcal{R}}$.

We now present our results on data-sets generated using different models. In each case we start with a stable marriage instance (with $|\mathcal{R}| = |\mathcal{H}|$) and gradually increase the capacity. As before, a column with the legend \uparrow implies that larger values are better for that column. Analogously, a column with the legend \downarrow implies smaller values are better.

Master. Table 3 shows our results on data-sets generated using the **Master** model. Here, we see that for all data-sets we get at least 10.5% increase in the size when comparing M_s vs M_p (column 3) and M_s vs M_m (column 7). The negative value of Δ_1 (columns 5 and 9) indicates that we reduce the number of residents matched to their rank-1 hospitals by at most 4.02% in our experiments and we also marginally gain for smaller values of $|\mathcal{H}|$. The parameter $\Delta_{\mathcal{R}}$ is observed to be positive which shows that a majority of residents prefer M_p over M_s (column 6) and also prefer M_m over M_s (column 10). Finally, the value of BP (columns 4 and 8) goes on increasing as we reduce the value of $|\mathcal{H}|$.

Shuffle. The results obtained on data-sets generated using the **Shuffle** model are presented in Table 4. The size gains are at least 6% when comparing M_s with M_p (column 3) and M_m (column 7). Looking at the values of Δ_1 from column 5 and column 9 we see that the number of residents matched to their rank-1 hospitals are almost always more, with up to 18% getting their rank-1 choices. We also observe that $\Delta_{\mathcal{R}}$ is always positive, which implies that majority of the residents prefer M_p and M_m over M_s .

Processed HR couples data-set. Table 5 shows our results on publicly available HR with couples data-set [2]. As seen from the columns with different Δ values, popular matchings perform favourably on all desired parameters on these data-sets. This is similar to our observations on data generated using the **Shuffle** model. We also investigated the relation between data generated using **Shuffle** model and the data used in this experiment and found that the data-sets are similar in their characteristics. This confirms that **Shuffle** is a reasonable model. Our results on these data-sets confirm that popular matchings perform favourably on variants of the **Shuffle** model.

■ **Table 4** Data generated using the **Shuffle** model. All values except $S(M_s)$ are percentages.

$ \mathcal{R} = 1000, k = 5$		M_p vs M_s				M_m vs M_s			
$ \mathcal{H} $	$S(M_s)$	$\Delta \uparrow$	$BP(M_p) \downarrow$	$\Delta_1 \uparrow$	$\Delta_{\mathcal{R}} \uparrow$	$\Delta \uparrow$	$BP(M_m) \downarrow$	$\Delta_1 \uparrow$	$\Delta_{\mathcal{R}} \uparrow$
1000	776.80	9.39	2.33	0.52	4.27	10.20	2.80	-0.14	5.39
100	856.00	8.56	3.55	8.72	7.80	9.23	4.13	9.79	9.38
20	900.80	7.10	5.50	13.87	9.86	7.52	6.01	15.55	11.37
10	935.40	6.03	16.57	17.35	5.77	6.15	16.76	18.02	6.32

■ **Table 5** Processed data-sets from [2]. All values except $S(M_s)$ are percentages.

$ \mathcal{R} = 100, k = 3 \dots 5$		M_p vs M_s				M_m vs M_s			
$ \mathcal{H} $	$S(M_s)$	$\Delta \uparrow$	$BP(M_p) \downarrow$	$\Delta_1 \uparrow$	$\Delta_{\mathcal{R}} \uparrow$	$\Delta \uparrow$	$BP(M_m) \downarrow$	$\Delta_1 \uparrow$	$\Delta_{\mathcal{R}} \uparrow$
90	84.67	10.10	6.26	2.79	4.62	11.81	8.55	1.20	7.16
50	87.19	9.61	8.25	4.53	4.99	11.01	10.77	3.12	6.47
20	91.35	7.92	13.57	9.30	4.28	8.41	14.93	8.22	3.86
10	93.53	6.61	19.94	5.34	-1.86	6.73	20.43	4.99	-2.00

Real world data-sets from IIT-M. Table 6 shows our results on data-sets obtained from the IIT-M elective allocation (the three rows in the table correspond to the Aug–Nov 2016, Jan–May 2017 and Aug–Nov 2017 humanities elective allocation data respectively).

For each data-set we list the number of students ($|\mathcal{R}|$), the number of courses ($|\mathcal{H}|$), the sum of total preferences (m) and the average preference list over the set of students (apl). On an average, each course has a capacity of 50. For each course a custom ranking criteria is used to rank students who have expressed preference in the course. As seen in Table 6, for the Jan–May 2017 (row 2) and Aug–Nov 2017 (row 3) data-sets, popular matchings perform very favourably as compared to stable matching.

References

- 1 Data sets. URL: <http://www.cse.iitm.ac.in/~meghana/projects/datasets/popular.zip>.
- 2 HR with couples data-set. URL: <http://researchdata.gla.ac.uk/303/>.
- 3 National Residency Matching Program. URL: <http://www.nrmp.org>.
- 4 Scottish Foundation Association Scheme. URL: <http://www.matching-in-practice.eu/the-scottish-foundation-allocation-scheme-sfas>.
- 5 Source code repository. URL: <https://github.com/rawatamit/GraphMatching>.
- 6 David J. Abraham, Robert W. Irving, Telikepalli Kavitha, and Kurt Mehlhorn. Popular Matchings. *SIAM Journal on Computing*, 37(4):1030–1045, 2007.
- 7 Péter Biró, David F. Manlove, and Shubham Mittal. Size versus stability in the marriage problem. *Theoretical Computer Science*, 411(16):1828–1841, 2010.
- 8 Péter Biró, Robert W. Irving, and David Manlove. Popular matchings in the marriage and roommates problems. In *7th International Conference on Algorithms and Complexity CIAC*, pages 97–108, 2010.
- 9 Florian Brandl and Telikepalli Kavitha. Popular matchings with multiple partners. In *Proceedings of 37th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 19:1–19:15, 2017.
- 10 Ágnes Cseh. Popular matchings. In U. Endriss, editor, *Trends in Computational Social Choice*, pages 105–122. COST (European Cooperation in Science and Technology), 2017.

9:14 How Good Are Popular Matchings?

■ **Table 6** Real data-sets from IIT-M elective allocation. All values except $S(M_s)$ are percentages.

$ \mathcal{R} , \mathcal{H} , m, \text{ avg. pref. length}$					$M_p \text{ vs } M_s$				$M_m \text{ vs } M_s$			
$ \mathcal{R} $	$ \mathcal{H} $	$ E $	apl	$S(M_s)$	$\Delta \uparrow$	$BP(M_p) \downarrow$	$\Delta_1 \uparrow$	$\Delta_{\mathcal{R}} \uparrow$	$\Delta \uparrow$	$BP(M_m) \downarrow$	$\Delta_1 \uparrow$	$\Delta_{\mathcal{R}} \uparrow$
483	18	5313	11.00	481	0.41	1.32	0	-0.20	0.41	1.32	0	-0.20
729	16	4534	6.21	675	8.00	31.98	7.39	-5.48	8.00	31.98	7.39	-5.48
655	14	2689	4.10	487	18.27	16.42	14.97	7.17	23.81	29.91	24.06	5.49


- 11 Ágnes Cseh and Telikepalli Kavitha. Popular Edges and Dominant Matchings. In *Proceedings of the Eighteenth Conference on Integer Programming and Combinatorial Optimization*, pages 138–151, 2016.
- 12 David Gale and Lloyd Shapley. College Admissions and the Stability of Marriage. *American Mathematical Monthly*, 69:9–14, 1962.
- 13 Dan Gusfield and Robert W. Irving. *The Stable Marriage Problem: Structure and Algorithms*. MIT Press, 1989.
- 14 Koki Hamada, Kazuo Iwama, and Shuichi Miyazaki. The Hospitals/Residents Problem with Lower Quotas. *Algorithmica*, 74(1):440–465, 2016.
- 15 Nicole Immorlica and Mohammad Mahdian. Marriage, honesty, and stability. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 53–62, 2005.
- 16 Telikepalli Kavitha. A Size-Popularity Tradeoff in the Stable Marriage Problem. *SIAM Journal on Computing*, 43(1):52–71, 2014.
- 17 David F. Manlove, Iain McBride, and James Trimble. “Almost-stable” matchings in the Hospitals / Residents problem with Couples. *Constraints*, 22(1):50–72, 2017.
- 18 Meghana Nasre and Prajakta Nimbhorkar. Popular matchings with lower quotas. In *Proceedings of 37th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 44:1–44:15, 2017.
- 19 Meghana Nasre and Amit Rawat. Popularity in the generalized hospital residents setting. In *Proceedings of the 12th International Computer Science Symposium in Russia*, pages 245–259, 2017.
- 20 Yu Yokoi. Envy-free matchings with lower quotas. In *Proceedings of the 28th International Symposium on Algorithms and Computation, ISAAC*, pages 67:1–67:12, 2017.

Evaluating and Tuning n -fold Integer Programming


Kateřina Altmanová¹

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

Duřan Knop²

Department of Informatics, University of Bergen, Bergen, Norway and
Department of Applied Mathematics, Charles University, Prague, Czech Republic
dusan.knop@uib.no
 <https://orcid.org/0000-0003-2588-5709>

Martin Koutecký³

Faculty of Industrial Engineering and Management, Technion – Israel Institute of Technology
Haifa, Israel
koutecky@technion.ac.il
 <https://orcid.org/0000-0002-7846-0053>

Abstract

In recent years, algorithmic breakthroughs in stringology, computational social choice, scheduling, etc., were achieved by applying the theory of so-called n -fold integer programming. An n -fold integer program (IP) has a highly uniform block structured constraint matrix. Hemmecke, Onn, and Romanchuk [Math. Programming, 2013] showed an algorithm with runtime $a^{O(rst+r^2s)}n^3$, where a is the largest coefficient, r, s , and t are dimensions of blocks of the constraint matrix and n is the total dimension of the IP; thus, an algorithm efficient if the blocks are of small size and with small coefficients. The algorithm works by iteratively improving a feasible solution with augmenting steps, and n -fold IPs have the special property that augmenting steps are guaranteed to exist in a not-too-large neighborhood. However, this algorithm has never been implemented and evaluated.

We have implemented the algorithm and learned the following along the way. The original algorithm is practically unusable, but we discover a series of improvements which make its evaluation possible. Crucially, we observe that a certain constant in the algorithm can be treated as a tuning parameter, which yields an efficient heuristic (essentially searching in a smaller-than-guaranteed neighborhood). Furthermore, the algorithm uses an overly expensive strategy to find a “best” step, while finding only an “approximately best” step is much cheaper, yet sufficient for quick convergence. Using this insight, we improve the asymptotic dependence on n from n^3 to $n^2 \log n$ which yields the currently asymptotically fastest algorithm for n -fold IP.

Finally, we tested the behavior of the algorithm with various values of the tuning parameter and different strategies of finding improving steps. First, we show that decreasing the tuning parameter initially leads to an increased number of iterations needed for convergence and eventually to getting stuck in local optima, as expected. However, surprisingly small values of the parameter already exhibit good behavior. Second, our new strategy for finding “approximately best” steps wildly outperforms the original construction.

2012 ACM Subject Classification Software and its engineering → Software design engineering, Theory of computation → Parameterized complexity and exact algorithms

¹ Author was supported the project 17-09142S of GA ĀR.

² Author supported supported by the project NFR MULTIVAL and the project P202/12/G061 of GA ĀR.

³ Author supported by Technion postdoctoral fellowship.



© Kateřina Altmanová, Duřan Knop, and Martin Koutecký;
licensed under Creative Commons License CC-BY

17th International Symposium on Experimental Algorithms (SEA 2018).

Editor: Gianlorenzo D’Angelo; Article No. 10; pp. 10:1–10:14

Leibniz International Proceedings in Informatics



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

Keywords and phrases n -fold integer programming, integer programming, analysis of algorithms, primal heuristic, local search

Digital Object Identifier 10.4230/LIPIcs.SEA.2018.10

Supplement Material <https://github.com/katealtmanova/nfoldexperiment>

1 Introduction

In this article we consider the general integer linear programming (ILP) problem in standard form,

$$\min \{ \mathbf{w}\mathbf{x} \mid A\mathbf{x} = \mathbf{b}, \mathbf{l} \leq \mathbf{x} \leq \mathbf{u}, \mathbf{x} \in \mathbb{Z}^n \}. \quad (\text{ILP})$$

with A an integer $m \times n$ matrix, $\mathbf{b} \in \mathbb{Z}^m$, $\mathbf{w} \in \mathbb{Z}^n$, $\mathbf{l}, \mathbf{u} \in (\mathbb{Z} \cup \{\pm\infty\})^n$. It is well known to be strongly NP-hard, but models many important problems in combinatorial optimization such as planning [28], scheduling [13] and transportation [4] and thus powerful generic solvers have been developed for it [25]. Still, theory is motivated to search for tractable special cases. One such special case is when the constraint matrix A has a so-called N -fold structure:

$$A = E^{(N)} = \begin{pmatrix} E_1 & E_1 & \cdots & E_1 \\ E_2 & 0 & \cdots & 0 \\ 0 & E_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & E_2 \end{pmatrix}.$$

Here, $r, s, t, N \in \mathbb{N}$, $\mathbf{u}, \mathbf{l}, \mathbf{w} \in \mathbb{Z}^{Nt}$, $\mathbf{b} \in \mathbb{Z}^{r+Ns}$, $E^{(N)}$ is an $(r + Ns) \times Nt$ -matrix, $E_1 \in \mathbb{Z}^{r \times t}$ is an $r \times t$ -matrix and $E_2 \in \mathbb{Z}^{s \times t}$ is an $s \times t$ -matrix. We call $E^{(N)}$ the N -fold product of $E = \begin{pmatrix} E_1 \\ E_2 \end{pmatrix}$. Problem (ILP) with $A = E^{(N)}$ is known as N -fold integer programming (N -fold IP). Hemmecke, Onn, and Romanchuk [16] prove the following.

► **Proposition 1** ([16, Theorem 6.2]). *There is an algorithm that solves⁴ (ILP) with $A = E^{(N)}$ encoded with L bits in time $\Delta^{O(trs+t^2s)} \cdot n^3 L$, where $\Delta = 1 + \max\{\|E_1\|_\infty, \|E_2\|_\infty\}$.*

Recently, algorithmic breakthroughs in stringology [20], computational social choice [21], scheduling [5, 18, 22], etc., were achieved by applying this algorithm and its subsequent non-trivial improvements.

The algorithm belongs to the larger family of augmentation (primal) algorithms. It starts with an initial feasible solution $\mathbf{x}_0 \in \mathbb{Z}^{Nt}$ and produces a sequence of increasingly better solutions $\mathbf{x}_1, \dots, \mathbf{x}_s$ (better means $\mathbf{w}\mathbf{x}_s < \mathbf{w}\mathbf{x}_{s-1} < \dots < \mathbf{w}\mathbf{x}_0$). It is guaranteed that the algorithm terminates, that \mathbf{x}_s is an optimal solution, and that the algorithm converges quickly, i.e., s is polynomial in the length of the input. A key property of N -fold IPs is that, if an augmenting step exists, then it can be decomposed into a bounded number of elements of the so-called *Graver basis* of A , which in turn makes it possible to compute it using dynamic programming [16, Lemma 3.1]. In a sense, this property makes the algorithm a local search algorithm which is always guaranteed to find an improvement in a not-too-large neighborhood. The bound on the number of elements or the size of the neighborhood which

⁴ Given an IP, we say that to *solve* it is to either (i) declare it infeasible or unbounded or (ii) find a minimizer of it.

needs to be searched is called the *Graver complexity* of E . This, in turn, implies that, if an augmenting step exists, then there is always one with small ℓ_1 -norm; for a matrix A , we denote this bound $g_1(A) = \max_{\mathbf{g} \in \mathcal{G}(A)} \|\mathbf{g}\|_1$ [24, Theorem 4]. However, the algorithm has never been implemented and evaluated.

1.1 Our Contributions

We have implemented the algorithm and tested it on two problems for which n -fold formulations were known: makespan minimization on uniformly related machines ($Q||C_{\max}$) and CLOSEST STRING; we have used randomly generated instances.

In the course of implementing the algorithm we learn the following. The algorithm in its initial form is practically unusable due to an *a priori* construction of the Graver basis $\mathcal{G}(E_2)$ of size exponential in r, s, t and a related (even larger) set $Z(E)$. However, we discover a series of improvements (some building on recent insights [24]) which avoid the construction of these two sets. Moreover, we adjust the algorithm to treat $g_1(A)$ as a tuning parameter \mathbf{g}_1 , which turns it into a heuristic.

We also study the *augmentation strategy*, which is the way the algorithm chooses an augmenting step among all the possible options. The original algorithm uses an overly expensive strategy to find a “best” step, which means that a large number of possible steps is evaluated in each iteration. We show that finding only an “approximately best” step is sufficient to obtain asymptotically equivalent convergence rate, and the work per iteration decreases exponentially. Using this insight, we improve the asymptotic dependence on N from N^3 to $N^2 \log N$. Together with recent improvements, this yields the currently asymptotically fastest algorithm for N -fold IP:

► **Theorem 2.** *Problem (ILP) with $A = E^{(N)}$ can be solved in time $\Delta^{r^2+s+rs^2} (Nt)^2 \log(Nt)M$, where $M = \log(\mathbf{w}\mathbf{x}^* - \mathbf{w}\mathbf{x}_0)$ for some minimizer \mathbf{x}^* of $\mathbf{w}\mathbf{x}$.*

Finally, we evaluate the behavior of the algorithm. We ask how is the performance of the algorithm (in terms of number of dynamic programming calls and quality of the returned solution) influenced by

1. the choice of the tuning parameter $1 < \mathbf{g}_1 \leq g_1(A)$?
2. the choice of the augmentation strategy between “best step”, “approximate best step” and “any step”?

As expected, as \mathbf{g}_1 moves from $g_1(A)$ to 1, we first see an increase in the number of iterations needed for convergence and eventually the algorithm gets stuck in local optima. However, surprisingly small values (e.g. $\mathbf{g}_1 = 5$ when $g_1(A) > 10^5$) of the parameter already exhibit close to optimal behavior. Second, our new strategy for finding “approximately best” steps outperforms the original construction by orders of magnitude, while the naive “any step” strategy behaves erratically.

We note that at this stage we are *not* (yet) interested in showing supremacy over existing algorithms; we simply want to understand the practical behavior of an algorithm whose theoretical importance was recently highlighted. For this reasons our experimental focus is on the two aforementioned questions rather than simply measuring the time. Similarly, due to the rigid format of $E^{(N)}$ we are limited to few problems for which N -fold formulations are known. For CLOSEST STRING we use the same instances as Chimani et al. [6]; for MAKESPAN MINIMIZATION we generate our own data because standard benchmarks are not limited to short jobs or few types of jobs.

1.2 Related Work

Our work mainly relates to *primal heuristics* [3] for MIPs which are used to help reach optimality faster and provide good feasible solutions early in the termination process. Specifically, our algorithm is a *neighborhood* (or *local*) *search algorithm*. The standard paradigm is *Large Neighborhood Search* (LNS) [27] with specializations such as for example *Relaxation Induced Neighborhood Search* (RINS) [7] and *Feasibility Pump* [2]. Using this paradigm, our proposed algorithm searches in the neighborhood induced by the ℓ_1 -distance from the current feasible solution and the search procedure is formulated as an ILP subproblem with the additional constraint $\|\mathbf{x}\|_1 \leq \mathbf{g}_1$. In this sense the closest technique to ours is *local branching* [11] which also searches in the ℓ_1 -neighborhood; however, we treat a discovered step as a *direction* and apply it exhaustively, so, unlike in local branching, we make long steps. Moreover, local branching was mainly applied to binary programs without any additional structure of the constraint matrix.

On the theoretical side, very recently Koutecký et al. [24] have studied parameterized strongly polynomial algorithms for various block-structured ILPs, not just N -fold IP. Eisenbrand et al. [9] independently (and using slightly different techniques) arrive at the same complexity of N -fold IP as our Theorem 2.

2 Preliminaries

For positive integers m, n we set $[m, n] = \{m, \dots, n\}$ and $[n] = [1, n]$. We write vectors in boldface (e.g., \mathbf{x}, \mathbf{y}) and their entries in normal font (e.g., the i -th entry of \mathbf{x} is x_i). Given the problem (ILP), we say that \mathbf{x} is *feasible* for (ILP) if $A\mathbf{x} = \mathbf{b}$ and $\mathbf{l} \leq \mathbf{x} \leq \mathbf{u}$.

Graver bases and augmentation. Let us now introduce Graver bases and discuss how they can be used for optimization. We also recall N -fold IPs; for background, we refer to the books of Onn [26] and De Loera et al. [8].

Let \mathbf{x}, \mathbf{y} be n -dimensional integer vectors. We call \mathbf{x}, \mathbf{y} *sign-compatible* if they lie in the same orthant, that is, for each $i \in [n]$ the sign of x_i and y_i is the same. We call $\sum_i \mathbf{g}^i$ a *sign-compatible sum* if all \mathbf{g}^i are pair-wise sign-compatible. Moreover, we write $\mathbf{y} \sqsubseteq \mathbf{x}$ if \mathbf{x} and \mathbf{y} are sign-compatible and $|y_i| \leq |x_i|$ for each $i \in [n]$. Clearly, \sqsubseteq imposes a partial order called “conformal order” on n -dimensional vectors. For an integer matrix $A \in \mathbb{Z}^{m \times n}$, its *Graver basis* $\mathcal{G}(A)$ is the set of \sqsubseteq -minimal non-zero elements of the *lattice* of A , $\ker_{\mathbb{Z}}(A) = \{\mathbf{z} \in \mathbb{Z}^n \mid A\mathbf{z} = \mathbf{0}\}$. An important property of $\mathcal{G}(A)$ is the following.

► **Proposition 3** ([26, Lemma 3.4]). *Every integer vector $\mathbf{x} \neq \mathbf{0}$ with $A\mathbf{x} = \mathbf{0}$ is a sign-compatible sum $\mathbf{x} = \sum_{i=1}^t \alpha_i \mathbf{g}^i$, $\alpha_i \in \mathbb{N}$, $\mathbf{g}^i \in \mathcal{G}(A)$ and $t \leq 2n - 2$.*

Let \mathbf{x} be a feasible solution to (ILP). We call \mathbf{g} an *\mathbf{x} -feasible step* (or simply *feasible* if \mathbf{x} is clear) if $\mathbf{x} + \mathbf{g}$ is feasible for (ILP). Further, we call a feasible step \mathbf{g} *augmenting* if $\mathbf{w}(\mathbf{x} + \mathbf{g}) < \mathbf{w}\mathbf{x}$; note that \mathbf{g} decreases the objective by $\mathbf{w}\mathbf{g}$. An augmenting step \mathbf{g} and a *step length* $\gamma \in \mathbb{N}$ form an *\mathbf{x} -feasible step pair* with respect to a feasible solution \mathbf{x} if $\mathbf{l} \leq \mathbf{x} + \gamma\mathbf{g} \leq \mathbf{u}$. A pair $(\gamma, \mathbf{g}) \in (\mathbb{N} \times \mathcal{G}(A))$ is a *γ -Graver-best step pair* and $\gamma\mathbf{g}$ is a *γ -Graver-best step* if it is feasible and for every feasible step pair (γ, \mathbf{g}') , $\mathbf{g}' \in \mathcal{G}(A)$, we have $\mathbf{w}\gamma\mathbf{g} \leq \mathbf{w}\gamma\mathbf{g}'$. An augmenting step \mathbf{g} and a step length $\gamma \in \mathbb{N}$ form a *Graver-best step pair* if it is γ -Graver-best and it minimizes $\mathbf{w}\gamma'\mathbf{g}'$ over all $\gamma' \in \mathbb{N}$, where (γ', \mathbf{g}') is a γ' -Graver-best step pair. We say that $\gamma\mathbf{g}$ is a *Graver-best step* if (γ, \mathbf{g}) is a Graver-best step pair.

The *Graver-best augmentation procedure* for (ILP) with a given feasible solution \mathbf{x}_0 and initial value $i = 0$ works as follows:

1. If there is no Graver-best step for \mathbf{x}_i , return it as optimal.
2. If a Graver-best step $\gamma\mathbf{g}$ for \mathbf{x}_i exists, set $\mathbf{x}_{i+1} := \mathbf{x}_i + \gamma\mathbf{g}$, $i := i + 1$, and go to 1.

► **Proposition 4** (Convergence bound [26, Lemma 3.10]). *Given a feasible solution \mathbf{x}_0 for (ILP), the Graver-best augmentation procedure finds an optimum in at most $(2n - 2) \log M$ steps, where $M = \mathbf{w}(\mathbf{x}_0 - \mathbf{x}^*)$ and \mathbf{x}^* is any minimizer of $\mathbf{w}\mathbf{x}$.*

By standard techniques (detecting unboundedness etc.) we can ensure that $\log M \leq L$.

N -fold IP. The structure of $E^{(N)}$ allows us to divide the Nt variables of \mathbf{x} into N bricks of size t . We use subscripts to index within a brick and superscripts to denote the index of the brick, i.e., x_j^i is the j -th variable of the i -th brick with $j \in [t]$ and $i \in [N]$.

3 Approximate Graver-best Steps

In this section we introduce the notion of a c -approximate Graver-best step (Definition 5), show that such steps exhibit good convergence (Lemma 6), can be easily obtained (Lemma 7), and result in a significant speed-up of the N -fold IP algorithm (Theorem 2).

► **Definition 5** (c -approximate Graver-best step). Let $c \in \mathbb{R}$, $c \geq 1$. Given an instance of (ILP) and a feasible solution \mathbf{x} , we say that \mathbf{h} is a c -approximate Graver-best step for \mathbf{x} if, for every \mathbf{x} -feasible step pair $(\gamma, \mathbf{g}) \in (\mathbb{N} \times \mathcal{G}(A))$, we have $\mathbf{w}\mathbf{h} \leq \frac{1}{c} \cdot \gamma\mathbf{w}\mathbf{g}$.

Recall the Graver-best augmentation procedure. We call its analogue where we replace a Graver-best step with a c -approximate Graver-best step the *c -approximate Graver-best augmentation procedure*.

► **Lemma 6** (c -approximate convergence bound). *Given a feasible solution \mathbf{x}_0 for (ILP), the c -approximate Graver-best augmentation procedure finds an optimum of (ILP) in at most $c \cdot (2n - 2) \log M$ steps, where $M = \mathbf{w}(\mathbf{x}_0 - \mathbf{x}^*)$ and \mathbf{x}^* is any minimizer of $\mathbf{w}\mathbf{x}$.*

Proof. The proof is a straightforward adaptation of the proof of Proposition 4. Let \mathbf{x}^* be a minimizer and let $\mathbf{h} = \mathbf{x}^* - \mathbf{x}_0$. Since $A\mathbf{h} = \mathbf{0}$, by Proposition 3, $\mathbf{h} = \sum_{i=1}^{2n-2} \alpha_i \mathbf{g}^i$ for some $\alpha_i \in \mathbb{N}$, $\mathbf{g}^i \in \mathcal{G}(A)$, $i \in [2n - 2]$. Thus, an \mathbf{x} -feasible step pair (γ, \mathbf{g}) such that $\gamma\mathbf{g}$ is a Graver-best step must satisfy $\mathbf{w}\gamma\mathbf{g} \leq \frac{1}{2n-2}M$. In other words, any Graver-best step pair improves the objective function by at least a $\frac{1}{2n-2}$ -fraction of the total optimality gap M , and thus $(2n - 2) \log M$ steps suffice to reach an optimum. It is straightforward to see that a c -approximate Graver-best step satisfies $\mathbf{w}\mathbf{x} - \mathbf{w}(\mathbf{x} + \gamma\mathbf{g}) \leq \frac{c}{2n-2}M$, and thus $c(2n - 2) \log M$ steps suffice. ◀

► **Remark.** Lemma 6 extends naturally to separable objectives; see the original proof [26, Lemma 3.10].

► **Lemma 7** (Powers of c step lengths). *Let $c \in \mathbb{N}$, \mathbf{x} be a feasible solution of (ILP), and let*

$$\Gamma_{c\text{-apx}} = \{c^i \mid \exists \mathbf{g} \in \mathcal{G}(A) : \mathbf{1} \leq \mathbf{x} + c^i \mathbf{g} \leq \mathbf{u}\}.$$

Let $(\gamma, \mathbf{g}) \in (\Gamma_{c\text{-apx}} \times \mathcal{G}(A))$ be an \mathbf{x} -feasible step pair such that $\gamma\mathbf{g} \leq \gamma'\mathbf{g}'$ for any \mathbf{x} -feasible step pair $(\gamma', \mathbf{g}') \in (\Gamma_{c\text{-apx}} \times \mathcal{G}(A))$. Then $\gamma\mathbf{g}$ is a c -approximate Graver-best step.

Proof. Let (γ, \mathbf{g}) satisfy the assumptions, and let $(\tilde{\gamma}, \tilde{\mathbf{g}}) \in (\mathbb{N} \times \mathcal{G}(A))$ be a Graver-best step pair. Let γ' be a nearest smaller power of c from $\tilde{\gamma}$, and observe that $\gamma'\tilde{\mathbf{g}}$ is a c -approximate Graver-best step because $\gamma' \geq \frac{\tilde{\gamma}}{c}$. On the other hand, since $\gamma\mathbf{g}$ is a γ -Graver-best step, we have $\gamma\mathbf{g} \leq \gamma'\tilde{\mathbf{g}}$ and thus $\gamma\mathbf{g}$ is also a c -approximate Graver-best step. \blacktriangleleft

► **Theorem 2 (restated).** *Problem (ILP) with $A = E^{(N)}$ can be solved in time $\Delta^{O(r^2s+rs^2)}(Nt)^2 \log(Nt)M$, where $M = \log(\mathbf{w}\mathbf{x}^* - \mathbf{w}\mathbf{x}_0)$ for some minimizer \mathbf{x}^* of $\mathbf{w}\mathbf{x}$.*

Proof. Recall that $\Delta = \|A\|_\infty + 1$. Koutecký et al. [24, Theorem 2] show that a γ -Graver-best step can be found in time $\Delta^{r^2s+rs^2}Nt$. Moreover, Hemmecke et al. [15] prove a proximity theorem which allows the reduction of an instance of (ILP) to an equivalent instance with new bounds \mathbf{l}' , \mathbf{u}' satisfying $\|\mathbf{u}' - \mathbf{l}'\|_\infty \leq Ntg_\infty$, with

$$g_\infty = \max_{\mathbf{g} \in \mathcal{G}(A)} \|\mathbf{g}\|_\infty \leq \max_{\mathbf{g} \in \mathcal{G}(A)} \|\mathbf{g}\|_1 \leq (\Delta rs)^{O(rs)},$$

where the last inequality can be found in the proof of [24, Theorem 4]. This bound implies that $\Gamma_{2\text{-apx}}$ from in Lemma 7 satisfies $|\Gamma_{2\text{-apx}}| \leq rs \log(\Delta Ntrs)$. By Lemma 7, finding a γ -Graver-best for each $\gamma \in \Gamma_{2\text{-apx}}$ and picking the minimum results in a 2-approximate Graver-best step, and can be done in time $\Delta^{r^2s+rs^2}(Nt) \log(Nt)$. By Lemma 6, $(4n-4) \log M$ steps suffice to reach the optimum. \blacktriangleleft

4 Implementation

We first give an overview of the original algorithm, which is our starting point. Then we discuss our specific improvements and mention a few details of the software implementation.

4.1 Overview of the Original Algorithm

The key property of the N -fold product $E^{(N)}$ is that, for any $N \in \mathbb{N}$, the number of nonzero bricks of any $\mathbf{g} \in \mathcal{G}(E^{(N)})$ is bounded by some constant $g(E)$ called the *Graver complexity of E* , and, moreover, that the sum of all non-zero bricks of \mathbf{g} can be decomposed into at most $g(E)$ elements of $\mathcal{G}(E_2)$ [16, Lemma 3.1]. This facilitates the following construction. Let

$$Z(E) = \left\{ \mathbf{z} \in \mathbb{Z}^t \mid \exists \mathbf{g}^1, \dots, \mathbf{g}^k \in \mathcal{G}(E_2), k \leq g(E), \mathbf{z} = \sum_{i=1}^k \mathbf{g}^i \right\}.$$

Then, every prefix sum of the bricks of $\mathbf{g} \in \mathcal{G}(E^{(N)})$ is contained in $Z(E)$ and a γ -Graver-best step, $\gamma \in \mathbb{N}$, can be found using dynamic programming over the elements of $Z(E)$.

To ensure that a Graver-best step is found, a set of step-lengths Γ_{best} is constructed as follows. Observe that any Graver-best (and thus feasible) step pair $(\gamma, \mathbf{g}) \in (\mathbb{N} \times \mathcal{G}(E^{(N)}))$, must satisfy that in at least one brick $i \in [n]$ it is “tight”, that is, (γ, \mathbf{g}) is \mathbf{x} -feasible while $(\gamma + 1, \mathbf{g})$ is not specifically because $\mathbf{l}^i \leq \mathbf{x} + \gamma\mathbf{g} \leq \mathbf{u}^i$ holds but $\mathbf{l}^i \leq \mathbf{x} + (\gamma + 1)\mathbf{g} \leq \mathbf{u}^i$ does not. Thus, for each $\mathbf{z} \in Z(E)$ and each $i \in [n]$, we find all the potentially “tight” step lengths γ and add them to Γ_{best} , which results in a bound of $|\Gamma_{\text{best}}| \leq |Z(E)| \cdot n$.

4.2 Replacing Dynamic Programming with ILP

We have started off by implementing the algorithm exactly as it is described by Hemmecke et al. [16]. The first obstacle is encountered almost immediately and is contained in the constant $g(E)$. This constant can be computed, but the computation is extremely difficult [10, 14].

Algorithm 1: Pseudocode of the algorithm of Hemmecke, Onn, and Romanchuk.

```

input : matrices  $E_1, E_2$ , positive integer  $N$ , and vectors  $\mathbf{b}, \mathbf{l}, \mathbf{u}, \mathbf{w}$ 
output : optimal solution to (ILP) with  $A = E^{(N)}$ 
1  $g = \text{GraverComplexity}(E_1, E_2)$ ;
2  $\mathbf{x}_0 = \text{FindFeasibleSolution}(E, \mathbf{b}, \mathbf{l}, \mathbf{u}, \mathbf{w})$ ,  $i = 0$ ;
3  $\mathcal{G}(E_1) = \text{GraverBasis}(E_1, g)$ ;
4  $Z(E) = \text{DynamicProgramStates}(\mathcal{G}(E_1), g)$ ;
5 do
6    $\Gamma = \text{BuildGamma}(\mathbf{x}_i)$ ;
7    $i = i + 1$ ;
8   foreach  $\gamma \in \Gamma$  do
9      $\mathbf{g}_\gamma = \text{gammaBestStep}(\gamma, \mathbf{g})$ ;
10     $\mathbf{x}_i = \mathbf{x}_{i-1} + \text{argmin}_{\{\mathbf{g}_\gamma | \gamma \in \Gamma\}} \mathbf{w}\mathbf{g}_\gamma$ ;
11 while  $\mathbf{x}_{i-1} \neq \mathbf{x}_i$ ;
12 return  $\mathbf{x}_i$ ;

```

Another possibility is to estimate it, in which case it is almost always larger than N and thus is essentially meaningless. Finally, one can take the approach partially suggested in [16, Section 7], where we consider $g(E)$ in the construction of $Z(E)$ to be a tuning parameter and consider the approximate set $Z_{\mathbf{g}\mathbf{c}}(E)$, $\mathbf{g}\mathbf{c} \in \mathbb{N}$, obtained by taking sums of at most $\mathbf{g}\mathbf{c}$ elements of $\mathcal{G}(E_2)$. This makes the algorithm more practical, but turns it into a heuristic.

In spite of this sacrifice, already for small ($r = 3$, $s = 1$, $t = 7$, $n = 10$) instances and extremely small value of $\mathbf{g}\mathbf{c} = 3$, the dynamic programming based on the $Z_{\mathbf{g}\mathbf{c}}(E)$ construction was taking an unreasonably long time (over one minute). Admittedly this could be improved; however, already for $\mathbf{g}\mathbf{c} > 5$, it becomes infeasible to compute $Z_{\mathbf{g}\mathbf{c}}(E)$, and for larger instances ($r > 5$, $t > 12$) it becomes very difficult to compute even $\mathcal{G}(E_2)$. For these reasons we sought to completely replace the dynamic program involving $Z(E)$.

Koutecký et al. [24] show that all instances of (ILP) with the property that the so-called *dual treedepth* $\text{td}_D(A)$ of A is bounded and the largest coefficient $\|A\|_\infty$ is bounded also have the property that $g_1(A) = \max_{\mathbf{g} \in \mathcal{G}(A)} \|\mathbf{g}\|_1$ is bounded, which implies that augmenting steps can be found efficiently. This class of ILPs contains N -fold IP.

The interpretation of the above fact is that, in order to solve (ILP), it is sufficient to repeatedly (for different \mathbf{x} and γ) solve an auxiliary instance

$$\min \{ \mathbf{w}\mathbf{h} \mid A\mathbf{h} = \mathbf{0}, \mathbf{l} \leq \mathbf{x} + \gamma\mathbf{h} \leq \mathbf{u}, \|\mathbf{h}\|_1 \leq g_1(A) \} \quad (\text{AugILP})$$

in order to find good augmenting steps; we note that the constraint $\|\mathbf{h}\|_1 \leq g_1(A)$ can be linearized [24, Lemma 25]. The heuristic approach outlined above transfers easily: we replace $g_1(A)$ in (AugILP) with some integer \mathbf{g}_1 , $1 < \mathbf{g}_1 \leq g_1(A)$; this makes (AugILP) easier to solve at the cost of losing the guarantee that an augmenting step is found if one exists. In theory, solving (AugILP) should be easier than solving the original instance (ILP) due to the special structure of A [24, Lemma 25]. Our approach here is to simply invoke an industrial MILP solver on (AugILP) in order to find a γ -Graver-best step.

4.3 Augmentation Strategy: Step Lengths

Logarithmic Γ . The majority of algorithms based on Graver basis augmentation rely on the Graver-best augmentation procedure [5, 8, 16, 20, 22, 26] and thus require finding

Algorithm 2: Pseudocode of our new heuristic algorithm. The algorithm is exact if $\mathbf{g}_1 \geq g_1(A) = \max_{\mathbf{g} \in \mathcal{G}(A)} \|\mathbf{g}\|_1$.

input : matrices E_1, E_2 , positive integers N, c , and \mathbf{g}_1 , and vectors $\mathbf{b}, \mathbf{l}, \mathbf{u}, \mathbf{w}$
output : a feasible solution to (ILP) with $A = E^{(N)}$

```

1  $\mathbf{x}_0 = \text{FindFeasibleSolution}(E, \mathbf{b}, \mathbf{l}, \mathbf{u}, \mathbf{w})$ ,  $i = 0$ ;
2 do
3    $\Gamma = \emptyset$ ;  $j = 0$ ,  $i = i + 1$ ;
4   do
5      $\gamma = c^j$ ;
6      $\mathbf{g}_\gamma = \min\{\mathbf{w}\gamma\mathbf{g} \mid A\mathbf{g} = \mathbf{0}, \mathbf{l} \leq \mathbf{x} + \gamma\mathbf{g} \leq \mathbf{u}, \|\mathbf{g}\|_1 \leq \mathbf{g}_1, \mathbf{g} \in \mathbb{Z}^{Nt}\}$ ;
7      $\gamma' = \text{ExhaustDirection}(\mathbf{g}_\gamma)$ ;
8      $\Gamma = \Gamma \cup \{\gamma'\}$ ,  $j = j + 1$ ;
9   while  $\mathbf{g}_\gamma \neq \mathbf{0}$ ;
10   $\mathbf{x}_i = \mathbf{x}_{i-1} + \operatorname{argmin}_{\{\mathbf{g}_\gamma \mid \gamma \in \Gamma\}} \mathbf{w}\mathbf{g}_\gamma$ ;
11 while  $\mathbf{x}_{i-1} \neq \mathbf{x}_i$ ;
12 return  $\mathbf{x}_i$ 

```

(exact) Graver-best steps. In the aforementioned algorithms this is always done using the construction of a set Γ_{best} mentioned above, which is of size $f(k) \cdot n$ where k is the relevant parameter (e.g., $(ars)^{O(rst+st^2)}$ in the original algorithm for N -fold IP). We replace this construction with $\Gamma_{2\text{-apx}} = \{1, 2, 4, 8, \dots\}$ which, combined with the proximity technique, is only of size $O(\log n)$ (Theorem 2); in particular, independent of the function $f(k)$.

Exhausting γ . Moreover, we have noticed that sometimes the algorithm finds a step \mathbf{g} for $\gamma = 2^k$ which is not tight in any brick, and then repeatedly applies it for shorter step-lengths $\gamma' < \gamma$. In other words, the discovered direction \mathbf{g} is not *exhausted*. Thus, for each $\gamma \in \mathbb{N}$, upon finding the γ -Graver-best step \mathbf{g} , we replace γ with the largest $\gamma' \geq \gamma$ for which (γ', \mathbf{g}) is still \mathbf{x} -feasible.

Early termination. Another observation is that in any given iteration of the algorithm, if $\gamma > 1$ then *some* augmenting step has been found and if the computation is taking too long, we might terminate it and simply apply the best step found so far.

4.4 Software and Hardware

We have implemented our solver in the SageMath computer algebra system [31]. This was a convenient choice for several reasons. The SageMath system offers an interactive notebook-style web-based interface, which allows rapid prototyping and debugging. Data types for vectors and matrices, Graver basis algorithms [1], and a unified interface for MILP solvers are also readily available. We have experimented with the open-source solvers GLPK [30], Coin-OR CBC [29], and the commercial solver Gurobi [12] and have settled for using the latter since it performs the best. The downside of SageMath is that an implementation of the original dynamic program is likely much slower than a similar implementation in C; however this DP is impractical anyway as explained in Section 4.2. For random instance generation and subsequent data evaluation and graphing, we have used the Jupyter notebook environment [19] and Matplotlib library [17]. The computations were performed on a computer with an Intel® Xeon® E5-2630 v3 (2.40GHz) CPU and 128 GB RAM.

5 Evaluation

We begin our evaluation with two main questions, specifically, how is the performance of the algorithm (both in terms of the number of iterations and the quality of the returned solution) influenced by:

1. the tuning parameter g_1 and
2. the augmentation strategy?

Regarding our first question, theoretically we should see either an increase in the number of iterations, a decrease in the quality of the returned solution, or both. However, the range of the tuning parameter g_1 is quite large: any number between 2 and $g_1(A)$ is a valid choice, and in all our scenarios the true value of $g_1(A)$ exceeds 300. Thus, we are interested in the threshold values of g_1 when the algorithm no longer finds the true optimum or when its convergence rate drops significantly.

Regarding our second question, there are two main candidates for the set of step-lengths Γ . We can either use the “best step” construction Γ_{best} of the original algorithm, which assures that we always make a Graver-best step before moving to the next iteration. Or, we can use the “approximate best step” construction $\Gamma_{2\text{-apx}}$ of Theorem 2, which provides a 2-approximate Graver-best step. To make this comparison more interesting, we also consider $\Gamma_{5\text{-apx}}$ and also the trivial “any step” strategy where we always make the 1-Graver-best step, which corresponds to taking $\Gamma_{\text{any}} = \{1\}$. Recall that due to the trick of always exhausting the discovered direction, this strategy actually has a chance at quick convergence, unlike if we only made the step with $\gamma = 1$.

5.1 Instances

We choose two problems for which N -fold IP formulations were shown in the literature, namely the $Q||C_{\max}$ scheduling problem [22] and the CLOSEST STRING problem [20].

UNIFORMLY RELATED MACHINES MAKESPAN MINIMIZATION ($Q||C_{\max}$)

Input: Set of m machines M , each with a speed $s_i \in \mathbb{N}$. A set of n jobs J , each with a processing time $p_j \in \mathbb{N}$.

Find: Find an assignment of jobs to machines such that the time when last job finishes (the makespan) is minimized; a job j scheduled on a machine i takes time p_j/s_i to execute.

CLOSEST STRING

Input: A set of k strings s_1, \dots, s_k of length L over an alphabet Σ .

Find: Find a string $y \in \Sigma^L$ minimizing $\max_{i=1}^k d_H(s_i, y)$, where d_H is the Hamming distance.

For both problems we generate random instances as follows.

Scheduling. We view the problem as a decision problem where, given a number B , we ask whether a schedule of makespan at most B exists. This is equivalent to the multi-sized bin packing problem, where we have m bins of various capacities instead of m machines of different speeds, and we adopt this view as it is more convenient. We also view it as a high-multiplicity problem where the items are not given explicitly as a list of item sizes, but succinctly by a vector of item multiplicities. Because the algorithm is primarily an optimization algorithm, we follow the standard approach [16, Lemma 3.8] and turn the feasibility problem into an auxiliary optimization instance where finding a starting feasible solution is easy. Specifically for $Q||C_{\max}$ this means introducing auxiliary slack variables for “not-yet-scheduled” jobs and minimizing the total “not-yet-scheduled” length.

The input parameters of the instance generation are number of bins m , the smallest and largest capacities S and L , respectively, item sizes p_1, \dots, p_k and probability weights w_1, \dots, w_k , $W = \sum_{i=1}^k w_i$, and a slack ratio σ . The instance is then generated as follows. First, we choose m capacities from $[S, L]$ uniformly at random. This determines the total capacity of the bins C . Our goal is to generate items whose total size is roughly $\sigma \cdot C$. We do this by repeatedly picking an item length from p_1, \dots, p_k , where p_j is selected with probability w_j/W , until the total size of items picked so far exceeds $\sigma \cdot C$, when we terminate and return the generated instance.

Closest String. As before, we view the problem as a decision problem: given a number $d \in \mathbb{N}$, decide whether there is a string y with $\max_{i=1}^k d_H(s_i, y) \leq d$. The random instance is generated exactly as done by Chimani et al. [6]: first, we generate a random “target” string $y \in \Sigma^L$ and create k copies s_1, \dots, s_k of it; then, we make α random changes in s_1, \dots, s_k . This way, we have an upper bound α on the optimum. The input parameters of the instance generation are thus k, L, Σ and the distance ratio r such $\alpha = n/r$. Again, we solve an auxiliary instance where we essentially start with a string of “all blanks” and try to fill in all the blanks while staying in the specified distance d ; the objective is thus the remaining number of blanks.

5.2 Results

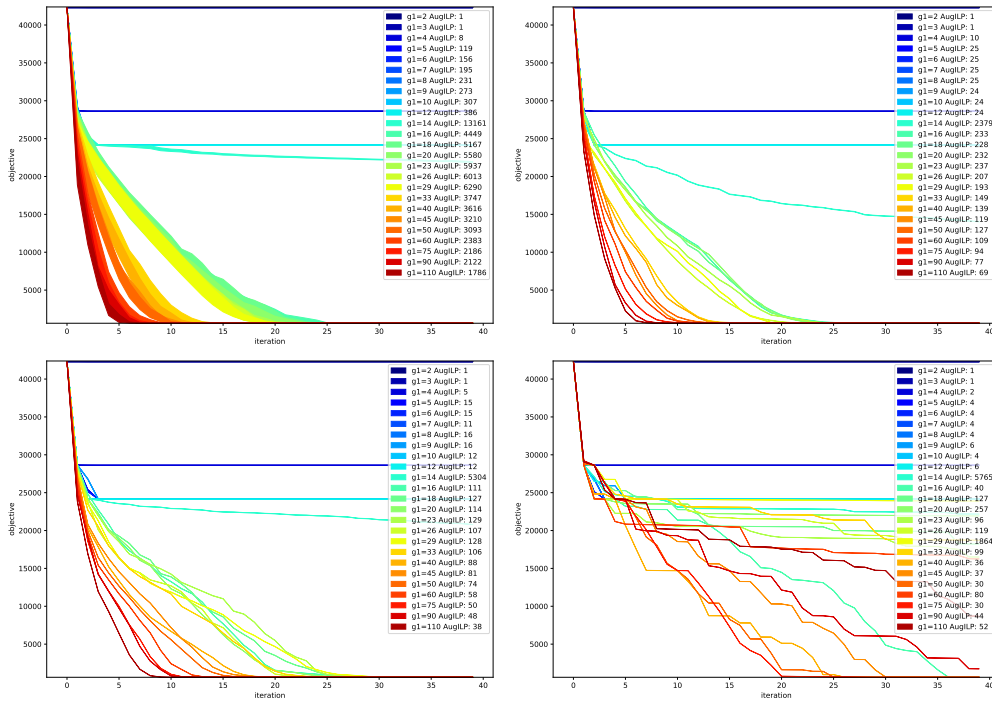
Here we demonstrate the overall behavior of the algorithm on three selected instances; we encourage the reader to see the full data (incl. plots) at <https://github.com/katealtmanova/nfoldexperiment>. We have determined that the sensible range of values of g_1 is between 2 and roughly 100 and beyond that the behavior does not change significantly. For all augmentation strategies there are values of g_1 which take much longer to converge than any other values; because of these outliers we clip our figures. To expose the behavior of the algorithm we use two types of figures.

Outer loop. In the *outer loop figure* we focus on the behavior of the algorithm with respect to the loop starting at line 2 of Algorithm 2, i.e., where each iteration corresponds to making an augmenting step. There is a line plot for each tested value of g_1 . The x axis shows the iteration, the y axis shows the objective value attained in this iteration, i.e., $\mathbf{w}\mathbf{x}_i$ for iteration i . We indicate the expensiveness of computing one augmentation by the thickness of the line in a given iteration – the thicker the line, the more times the (AugILP) has been solved in this iteration. The legend indicates the exact number of times (AugILP) has been solved for this value of g_1 .

Inner loop. In the *inner loop figure* we focus on the loop starting at line 4 of Algorithm 2, i.e., where iterations correspond to solutions of (AugILP). As before, each color corresponds to a tested value of g_1 . There is a line plot displaying the *minimum* over augmenting steps found in each outer iteration; however now there is also a semiopaque region above this line, indicating the values of all the augmenting steps (including the non-minimal ones) found in this iteration.

We chose two instances among the tested once as representative of the overall behavior:

- Our first instance is $Q||C_{\max}$ with parameters $m = 15$, $S = 2000$, $L = 10000$, item sizes $(2, 3, 13, 35)$ (so that we have an instance with nontrivial $\|A\|_\infty$), and weights $(6, 13, 2, 1)$ and $\sigma = 0.45$. The theoretical upper bound on $g_1(A)$ is $(rs\|A\|_\infty + 1)^{O(rs)}$, and here we



■ **Figure 1** Outer loop results for MAKESPAN MINIMIZATION when using (left to right) Γ_{best} , $\Gamma_{2\text{-apx}}$, $\Gamma_{5\text{-apx}}$, and Γ_{any} ; clipped to 40 outer iterations.

have $r = 4$, $s = 1$ and $\|A\|_{\infty} = 35$; thus, without computing $g_1(A)$ exactly, we should consider it to be at least $(4 \cdot 36)^4$. See Figures 1 and 2.

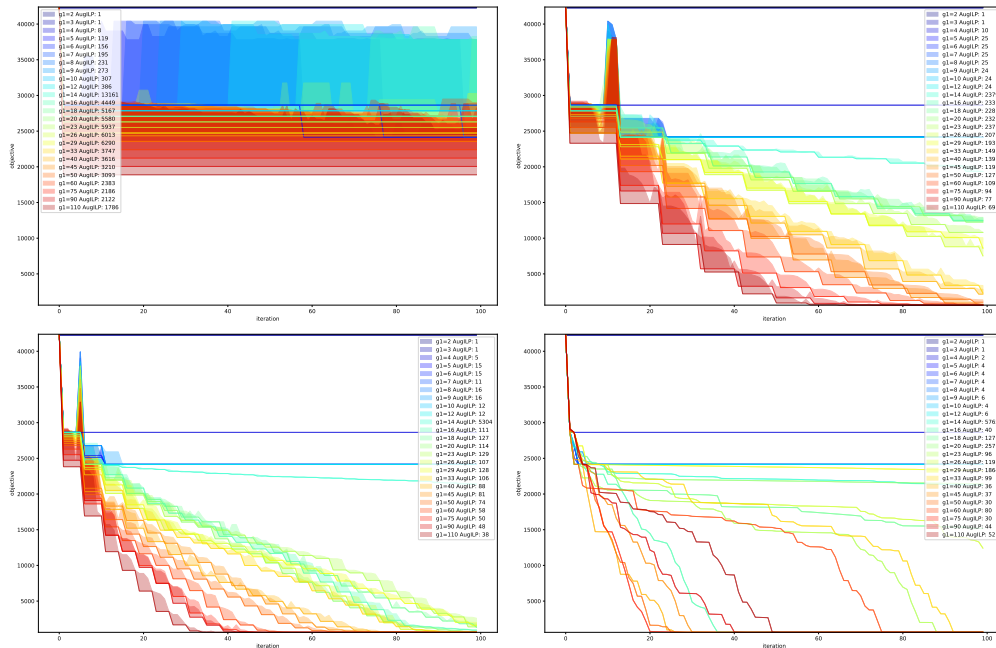
- The third instance is CLOSEST STRING with parameters $k = 5$, $|\Sigma| = 2$, $L = 10000$ and $r = 1$. The N -fold model has $r = 5$, $s = 1$ and $\|A\|_{\infty} = 1$, thus, without computing $g_1(A)$ exactly, we should consider it to be at least $(2 \cdot 5)^5$. See Figure 3.

5.3 Conclusions

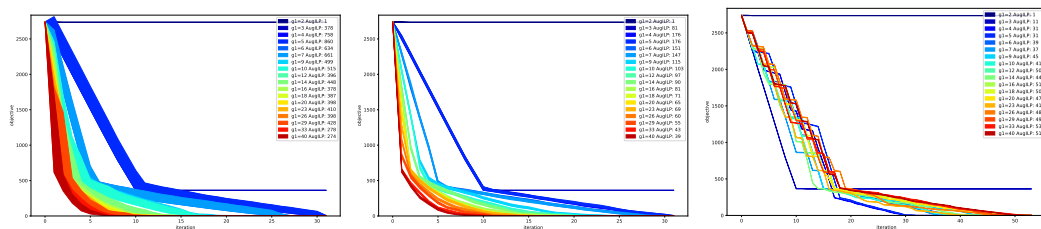
Our main takeaway regarding Question #1 is that, while the theoretical upper bounds for $g_1(A)$ are huge, already small values of g_1 ($g_1 > 5$ for CLOSEST STRING and $g_1 > 20$ for MAKESPAN MINIMIZATION) are sufficient for convergence. We remark that, in the case of CLOSEST STRING, this hints at the possibility that the maximum value of any *feasible* augmenting step $\mathbf{g} \in \mathcal{G}(A)$ is bounded by $k^{O(1)}$ rather than $k^{O(k)}$, which would imply an algorithm with runtime $k^{O(k)} \log L$ while the currently best algorithm runs in time $k^{O(k^2)} \log L$ [20].

Regarding Question #2, we see that $\Gamma_{2\text{-apx}}$ provides essentially the same convergence rate as Γ_{best} but is orders of magnitude cheaper to compute. The “any step” augmentation strategy Γ_{any} usually converges surprisingly quickly, but our results make it clear that its behavior is erratic and unpredictable. The inner loop Figure 2 reveals that a good step (close to a Graver-best step) is usually found for larger step-length γ ; this motivates adding the $\Gamma_{5\text{-apx}}$ augmentation strategy to the comparison, as it spends less time on short step-lengths than $\Gamma_{2\text{-apx}}$. Figure 1 shows that using 5-approximate Graver-best steps instead of 2-approximate does not affect the outer loop convergence much, and Figure 2 shows that in terms of the total number of (AugILP) calls it performs better.

10:12 Evaluating and Tuning n -fold Integer Programming



■ **Figure 2** Inner loop results for MAKESPAN MINIMIZATION when using (left to right) Γ_{best} , $\Gamma_{2\text{-apx}}$, $\Gamma_{5\text{-apx}}$, and Γ_{any} ; clipped to 100 inner iterations.



■ **Figure 3** Outer loop results for the CLOSEST STRING instance when using (left to right) Γ_{best} , $\Gamma_{2\text{-apx}}$, and Γ_{any} .

Furthermore, we observe that solving (AugILP) using a MILP solver such as Gurobi typically takes essentially as much time as solving (ILP) itself; in other words, current MILP solvers are (without tuning) unable to make any use neither of the extra structure of $E^{(N)}$, nor the fact that we are seeking a solution with small ℓ_1 norm and the right hand side is $\mathbf{0}$. Moreover, with a growing number of bricks N , the time to solve (AugILP) using a MILP solver grows superlinearly, suggesting that, for large enough N , a specialized dynamic programming algorithm might be competitive with generic MILP solvers.

6 Outlook

We have initiated an experimental investigation of certain subclasses of ILP with block structured constraint matrices. Our results show that, as theory suggests, for such ILPs a primal algorithm always augmenting with steps of small ℓ_1 norm converges quickly. We close with a few interesting research directions. First, is there a way to tune generic MILP solvers to solve (AugILP) significantly faster than (ILP)? Second, what is the behavior of

our algorithm on instances *other* than N -fold IP? For example, how large must be g_1 in order to attain the optimum quickly for standard benchmark instances, e.g. MIPLIB [23]? Third, the approach of Koutecký et al. [24] suggests that a key property for the efficient solvability of (AugILP) is a certain “sparsity” and “shallowness” (formally captured by the graph parameter *tree-depth*) of graphs related to A ; what are “natural” instances with small tree-depth?

References

- 1 4ti2 team. 4ti2—a software package for algebraic, geometric and combinatorial problems on linear spaces. Available at www.4ti2.de.
- 2 Livio Bertacco, Matteo Fischetti, and Andrea Lodi. A feasibility pump heuristic for general mixed-integer problems. *Discrete Optimization*, 4(1):63–76, 2007. Mixed Integer Programming. doi:10.1016/j.disopt.2006.10.001.
- 3 Timo Berthold. Measuring the impact of primal heuristics. *Operations Research Letters*, 41(6):611–614, 2013.
- 4 Ralf Borndörfer, Martin Grötschel, and Ulrich Jäger. Planning problems in public transit. In *Production Factor Mathematics*, pages 95–121. Springer, 2010.
- 5 Lin Chen and Daniel Marx. Covering a tree with rooted subtrees—parameterized and approximation algorithms. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2801–2820. SIAM, 2018.
- 6 Markus Chimani, Matthias Woste, and Sebastian Böcker. A closer look at the closest string and closest substring problem. In *2011 Proceedings of the Thirteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 13–24. SIAM, 2011.
- 7 Emilie Danna, Edward Rothberg, and Claude Le Pape. Exploring relaxation induced neighborhoods to improve mip solutions. *Mathematical Programming*, 102(1):71–90, 2005.
- 8 Jesus A. De Loera, Raymond Hemmecke, and Matthias Köppe. *Algebraic and Geometric Ideas in the Theory of Discrete Optimization*, volume 14 of *MOS-SIAM Series on Optimization*. SIAM, 2013.
- 9 Friedrich Eisenbrand, Christoph Hunkenschröder, and Kim-Manuel Klein. Faster algorithms for integer programs with block structure. *arXiv preprint arXiv:1802.06289*, 2018.
- 10 Elisabeth Finhold and Raymond Hemmecke. Lower bounds on the graver complexity of m -fold matrices. *Annals of Combinatorics*, 20(1):73–85, 2016.
- 11 Matteo Fischetti and Andrea Lodi. Local branching. *Mathematical programming*, 98(1-3):23–47, 2003.
- 12 Gurobi Optimization, Inc. Gurobi optimizer reference manual, 2016. URL: <http://www.gurobi.com>.
- 13 Stefan Heinz, Wen-Yang Ku, and Christopher J. Beck. Recent improvements using constraint integer programming for resource allocation and scheduling. In *International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 12–27. Springer, 2013.
- 14 Raymond Hemmecke. Exploiting symmetries in the computation of graver bases. *arXiv preprint math/0410334*, 2004.
- 15 Raymond Hemmecke, Matthias Köppe, and Robert Weismantel. Graver basis and proximity techniques for block-structured separable convex integer minimization problems. *Math. Program.*, 145(1-2, Ser. A):1–18, 2014.
- 16 Raymond Hemmecke, Shmuel Onn, and Lyubov Romanchuk. n -fold integer programming in cubic time. *Math. Program.*, 137(1-2, Ser. A):325–341, 2013.
- 17 John D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007. doi:10.1109/MCSE.2007.55.

- 18 Klaus Jansen, Kim-Manuel Klein, Marten Maack, and Malin Rau. Empowering the configuration-ip-new ptas results for scheduling with setups times. *arXiv preprint arXiv:1801.06460*, 2018.
- 19 Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, et al. Jupyter notebooks-a publishing format for reproducible computational workflows. In *ELPUB*, pages 87–90, 2016.
- 20 Dušan Knop, Martin Koutecký, and Matthias Mnich. Combinatorial n -fold integer programming and applications. In *Proc. ESA 2017*, volume 87 of *Leibniz Int. Proc. Informatics*, pages 54:1–54:14, 2017.
- 21 Dušan Knop, Martin Koutecký, and Matthias Mnich. Voting and bribing in single-exponential time. In *Proc. STACS 2017*, volume 66 of *Leibniz Int. Proc. Informatics*, pages 46:1–46:14, 2017.
- 22 Dušan Knop and Martin Koutecký. Scheduling meets n -fold integer programming. *Journal of Scheduling*, Nov 2017. doi:10.1007/s10951-017-0550-0.
- 23 Thorsten Koch, Tobias Achterberg, Erling Andersen, Oliver Bastert, Timo Berthold, Robert E Bixby, Emilie Danna, Gerald Gamrath, Ambros M Gleixner, Stefan Heinz, et al. Miplib 2010. *Mathematical Programming Computation*, 3(2):103, 2011.
- 24 Martin Koutecký, Asaf Levin, and Shmuel Onn. A parameterized strongly polynomial algorithm for block structured integer programs. *arXiv preprint arXiv:1802.05859*, 2018.
- 25 Andrea Lodi. Mixed integer programming computation. In *50 Years of Integer Programming 1958-2008*, pages 619–645. Springer, 2010.
- 26 Shmuel Onn. Nonlinear discrete optimization. *Zurich Lectures in Advanced Mathematics, European Mathematical Society*, 2010.
- 27 David Pisinger and Stefan Ropke. Large neighborhood search. In *Handbook of metaheuristics*, pages 399–419. Springer, 2010.
- 28 Yves Pochet and Laurence A Wolsey. *Production planning by mixed integer programming*. Springer Science & Business Media, 2006.
- 29 Matthew J. Saltzman. Coin-or: an open-source library for optimization. In *Programming languages and systems in computational economics and finance*, pages 3–32. Springer, 2002.
- 30 Tommi Sottinen. Operations research with gnu linear programming kit. *ORMS*, 1020:200, 2009.
- 31 The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 7.6)*, 2017. <http://www.sagemath.org>.

A Computational Investigation on the Strength of Dantzig-Wolfe Reformulations

Michael Bastubbe

Lehrstuhl für Operations Research, RWTH Aachen University,
Kackertstr. 7, D-52072 Aachen, Germany
michael.bastubbe@rwth-aachen.de

Marco E. Lübbecke

Lehrstuhl für Operations Research, RWTH Aachen University,
Kackertstr. 7, D-52072 Aachen, Germany
marco.luebbecke@rwth-aachen.de

Jonas T. Witt

Lehrstuhl für Operations Research, RWTH Aachen University,
Kackertstr. 7, D-52072 Aachen, Germany
jonas.witt@rwth-aachen.de

Abstract

In Dantzig-Wolfe reformulation of an integer program one convexifies a subset of the constraints, leading to potentially stronger dual bounds from the respective linear programming relaxation. As the subset can be chosen arbitrarily, this includes the trivial cases of convexifying no and all constraints, resulting in a weakest and strongest reformulation, respectively. Our computational study aims at better understanding of what happens in between these extremes. For a collection of integer programs with few constraints we compute, optimally solve, and evaluate the relaxations of all possible (exponentially many) Dantzig-Wolfe reformulations (with mild extensions to larger models from the MIPLIBs). We observe that only a tiny number of different dual bounds actually occur and that only a few inclusion-wise minimal representatives exist for each. This aligns with considerably different impacts of individual constraints on the strengthening the relaxation, some of which have almost no influence. In contrast, types of constraints that are convexified in textbook reformulations have a larger effect. We relate our experiments to what could be called a hierarchy of Dantzig-Wolfe reformulations.

2012 ACM Subject Classification Mathematics of computing → Integer programming

Keywords and phrases Dantzig-Wolfe reformulation, strength of reformulations, Lagrangean relaxation, partial convexification, column generation, hierarchy of relaxations

Digital Object Identifier 10.4230/LIPIcs.SEA.2018.11

1 Motivation

The strength of formulations is a central topic in integer programming, and expressed via the quality of the dual bound obtained from the respective linear programming relaxation. It is well-known that a Dantzig-Wolfe (DW) reformulation of an integer program, the convexification of a subset of the constraints, may yield strong dual bounds. Therefore, such reformulations have been proposed in the literature for many models stemming from various applications. Even though a DW reformulation follows a certain mechanics that exploits the structure of the model, this is by far not unique. Technically, *every subset* of constraints gives rise to its own DW reformulation, including the two trivial cases: convexifying no or all constraints, implying the weakest and strongest possible dual bounds, respectively.



© Michael Bastubbe, Marco E. Lübbecke, and Jonas T. Witt;
licensed under Creative Commons License CC-BY

17th International Symposium on Experimental Algorithms (SEA 2018).

Editor: Gianlorenzo D'Angelo; Article No. 11; pp. 11:1–11:12

Leibniz International Proceedings in Informatics



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

Because of this freedom, speaking of “the” strength of DW reformulations in general is not useful. Instead, a differentiation is necessary, but theoretical results are very scarce (we mention two exceptions later). In order to make progress on this topic our study provides some general computational intuition. To the best of our knowledge, it is the first of its kind.

Brief Background on Dantzig-Wolfe Reformulation

We sketch the basics of DW reformulating an integer linear program (including the mixed-integer case), mainly to introduce our notation and conventions. We consider

$$\begin{aligned} z_{IP} = \min \quad & c^T x \\ \text{s. t.} \quad & a_i^T x \geq b_i \quad \forall i \in I \\ & x \in \mathbb{Z}^n, \end{aligned} \quad (1)$$

where I denotes a finite index set, $n \in \mathbb{Z}_{>0}$, $c, a_i \in \mathbb{Q}^n$, and $b_i \in \mathbb{Q}$ for $i \in I$. We identify constraints with their respective index $i \in I$. When we speak of relaxations, we always refer to the linear programming relaxation, obtained by dropping the integrality requirement on the variables. We denote the optimum of the relaxation of (1) by z_{LP} . The DW reformulation (“convexification”) of a subset $I' \subseteq I$ of constraints amounts to (implicitly) additionally require in (1) that $x \in \text{conv} \{ \tilde{x} \in \mathbb{Z}^n : a_i^T \tilde{x} \geq b_i \forall i \in I' \}$. It is irrelevant here (but very well understood) how this is technically achieved (see Vanderbeck [13] for details on Dantzig-Wolfe reformulation and the relation to Lagrangean relaxation). This reformulation has the same integer feasible solutions as (1), but the relaxation

$$\begin{aligned} z_{DW}(I') = \min \quad & c^T x \\ \text{s. t.} \quad & a_i^T x \geq b_i \quad \forall i \in I \setminus I' \\ & x \in \text{conv} \{ \tilde{x} \in \mathbb{Z}^n : a_i^T \tilde{x} \geq b_i \forall i \in I' \} \\ & x \in \mathbb{R}^n \end{aligned} \quad (2)$$

is potentially stronger than that of (1), i.e.,

$$z_{IP} \geq z_{DW}(I') \geq z_{LP} \quad \forall I' \subseteq I. \quad (3)$$

This relation is a main reason for performing a DW reformulation in the first place. For convenience we identify a DW reformulation of constraints $I' \subseteq I$ with I' itself. We formally repeat that both extreme cases $z_{IP} = z_{DW}(I)$ and $z_{LP} = z_{DW}(\emptyset)$ are possible. Therefore, the notion of strength of a DW reformulation must necessarily relate to I' [14]. Geoffrion [7] gave as necessary condition for $z_{DW}(I') \geq z_{LP}$ that $\text{conv} \{ \tilde{x} \in \mathbb{Z}^n : a_i^T \tilde{x} \geq b_i \forall i \in I' \} \subsetneq \{ \tilde{x} \in \mathbb{R}^n : a_i^T \tilde{x} \geq b_i \forall i \in I' \}$. The condition is not sufficient; in particular, the actual strengthening may depend on the objective function. For the stable set (and related) problems we recently characterized DW reformulations $I' \subseteq I$ for which $z_{IP} = z_{DW}(I')$ or $z_{LP} = z_{DW}(I')$ independently of the objective function [14]. A generalization does not seem to be in reach and nothing is known about what happens “in between.”

Our Approach

For each instance, taken from a set of small models of different problem classes, we compute the dual bounds $z_{DW}(I')$ from all $2^{|I|}$ DW reformulations $I' \subseteq I$ and collect some statistics about the respective DW reformulations. We then report these statistics for each problem class. When we plot figures, this is usually only for one representative of each class, because they look similar for the other problems of this class.

2 Instances and Experimental Setup

In order to keep the task of evaluating *all* (exponentially many) DW reformulations of an integer program manageable, we only consider very small models with up to 18 constraints. We first consider instances of problems where DW reformulation classically applies well (we call these models *structured*): bin packing (bpp), vertex coloring, capacitated p -median (cpmp), single-source capacitated facility location (cflp), generalized assignment (gap), and capacitated vehicle routing problems with time windows (vrp) problems. We used an instance generator to create small bin packing problems [12] and created 3 vertex coloring instances on connected graphs with 2 or 3 vertices (yielding integer programs with up to 15 constraints). Furthermore, we created 2 instances for the capacitated vehicle routing problem with time windows consisting of a single depot, 2 customers, and 2 vehicles (yielding mixed integer programs with up to 18 constraints). For all other problems, we took instances from the literature [4, 5, 8, 10, 11] and modified them to reduce the number of constraints. We chose standard textbook formulations for all problems with as few constraints as possible (e.g., by using formulations with few but relatively weak coupling constraints). Additionally, we consider very small and easy-to-solve instances from MIPLIB 3 [3], namely `flugpl`, `mod008`, and `p0033`. Although the instances `markshare1`, `markshare2`, `mas74`, and `mas76` have only a small number of constraints, they turned out to be too hard to solve in preliminary experiments. All our instances have a positive integrality gap, i.e., $z_{LP} < z_{IP}$.

For the experiments, we use a development version of the generic branch-price-and-cut solver GCG [6], see www.or.rwth-aachen.de/gcg for the current released version 2.1.4. We implemented a so-called detector that creates, for each instance, all possible DW reformulations $I' \subseteq I$ and solve their relaxations optimally by column generation to obtain $z_{DW}(I')$. We turned off separation of cutting planes as well as the internal handling of problems with integral optimum (this would lead to only integer dual bounds $\lceil z_{DW}(I') \rceil$ for $I' \subseteq I$). For the structured models we also disabled presolving. The MIPLIB instances were presolved, i.e., the number of constraints and variables can differ from the original instance.

The total computation time spend for optimally solving over one million relaxations by column generation was approximately 100 hours. These numbers do not include the time spent for creating the decompositions and evaluating the computations.

3 Number and Frequency of Distinct Dual Bounds

This section is motivated by hierarchies of relaxations which are, roughly speaking, finite chains of (nested) stronger and stronger relaxations (of the same type), starting from the linear programming relaxation and arriving at the convex hull of integer feasible solutions. Several of these hierarchies are known, e.g., the Chvátal-Gomory procedure produces one.

Let $\mathcal{I} = 2^I$ denote the powerset of I . Consider the partially ordered set $P = (\mathcal{I}, \subseteq)$ consisting of all $2^{|I|}$ DW reformulations and their partial order induced by set inclusion. The empty set \emptyset is the unique minimal element and I is the unique maximal element of P . Every chain $\emptyset = I_0 \subseteq I_1 \subseteq \dots \subseteq I_m = I$ in P obviously defines a hierarchy of relaxations with $z_{LP} = z_{DW}(I_0) \leq z_{DW}(I_1) \leq \dots \leq z_{DW}(I_m) = z_{IP}$, where the inequalities need not be (all) strict. That is, a chain may induce fewer than $|I| + 1$ dual bounds. We call a DW reformulation I' *minimal* if there does not exist any $I'' \subsetneq I'$ with $z_{DW}(I'') = z_{DW}(I')$.

Our first experiment reveals for every given instance how many distinct dual bounds actually occur and in what frequencies. In particular, what is the distribution of dual bounds in $[z_{LP}, z_{IP}]$ and what can we learn about minimal DW reformulations?

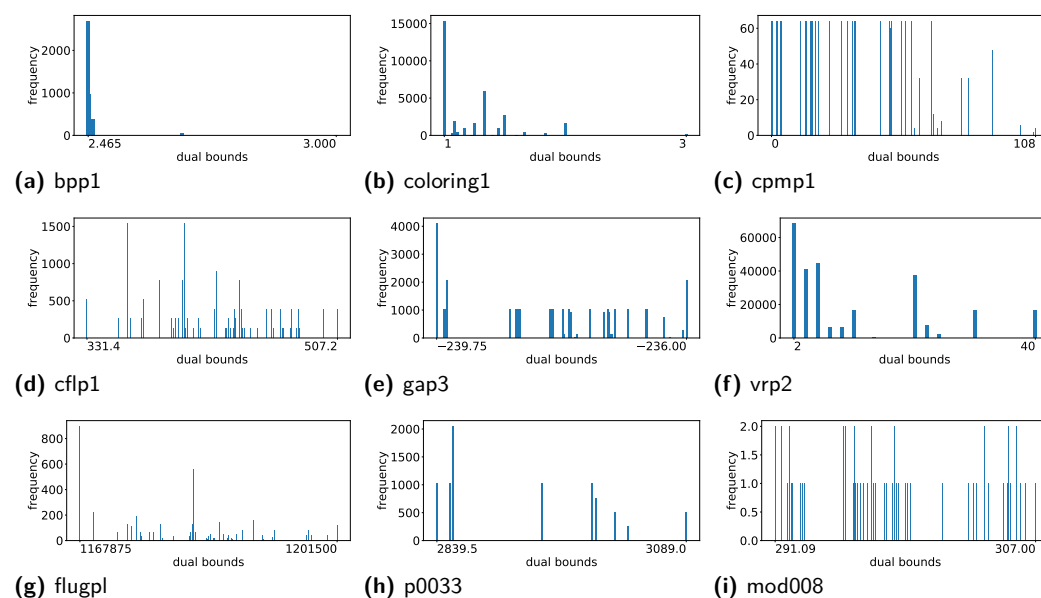
■ **Table 1** For each (toy) instance, we state the type (binary, general integer, or mixed integer variables); the number of constraints (nconss); the number of variables (nvars); the number of DW reformulations (nrefs); the number of distinct dual bounds (ndbs); the number of minimal DW reformulations (nmin); as well as the average (cavg), the minimum (cmin), and the maximum (cmax) number of distinct dual bounds in a chain.

Instance	type	nconss	nvars	nrefs	ndbs	nmin	cavg	cmin	cmax
bpp1	BP	12	42	4096	5	27	3.318	3.0	4.0
bpp2	BP	10	30	1024	5	18	3.375	3.0	4.0
bpp3	BP	14	56	16384	6	38	3.765	3.0	5.0
cflp1	BP	14	45	16384	49	66	5.283	4.0	7.0
cflp2	BP	14	44	16384	77	90	6.400	5.0	7.0
cflp3	BP	14	44	16384	83	86	6.367	5.0	7.0
coloring1	BP	15	12	32768	16	835	4.172	2.0	8.0
coloring2	BP	12	12	4096	7	75	2.399	2.0	4.0
coloring3	BP	6	6	64	4	8	1.800	1.0	3.0
cpmp1	BP	11	30	2048	42	46	5.733	5.0	9.0
cpmp2	BP	11	30	2048	46	52	5.597	5.0	9.0
cpmp3	BP	11	30	2048	31	32	4.481	4.0	6.0
gap1	BP	15	50	32768	26	44	3.122	2.0	6.0
gap2	BP	15	50	32768	28	67	3.016	3.0	7.0
gap3	BP	15	50	32768	40	56	4.441	3.0	9.0
vrp1	MIP	18	18	262144	20	3008	3.828	2.0	8.0
vrp2	MIP	18	18	262144	12	208	3.753	2.0	7.0
flugpl	IP	12	14	4096	48	65	4.050	3.0	7.0
mod008	BP	6	319	64	51	51	5.517	4.0	6.0
p0033	BP	13	28	8192	9	9	3.183	2.0	4.0

3.1 Toy Instances

In Table 1, we list some statistics on the instances and their DW reformulations. The number of distinct dual bounds (ndbs) is usually *much* smaller than the number of all DW reformulations, i.e., many $I' \subseteq I$ yield the same $z_{DW}(I')$. The only exception is MIPLIB instance `mod008` which has a much higher number of variables relative to $|I|$. Even more interestingly, also the number of minimal DW reformulations is *very* small, often not much larger than ndbs. This could hint at many chains in P with only a small number of distinct dual bounds, i.e., with many non-minimal DW reformulations. The numbers `cmin`, `cavg`, `cmax` give some distribution information about the number of distinct dual bounds in a chain.

Figure 1 shows histograms of the number of distinct dual bounds which often occur with the same frequencies (and often enough, these are powers of 2). This also supports the existence of constraints that do not have an influence on the dual bound in most DW reformulations. For example in the capacitated p -median instance `cpmp1`, there exist 5 set partitioning constraints (forcing each location to be assigned to exactly one median) and a cardinality constraint (forcing to choose exactly p locations as medians). In Section 4 we will see that often these constraints do not improve the dual bound when added to the set of convexified constraints, which explains why the frequency $2^6 = 64$ occurs multiple times in the histogram of the instance `cpmp1` in Figure 1(c).



■ **Figure 1** Histogram for the number of DW reformulations with different dual bounds. On the x -axis we see the (discrete) spectrum of potential dual bounds from weakest z_{LP} to strongest z_{IP} and the y -axis displays the respective frequencies of dual bounds.

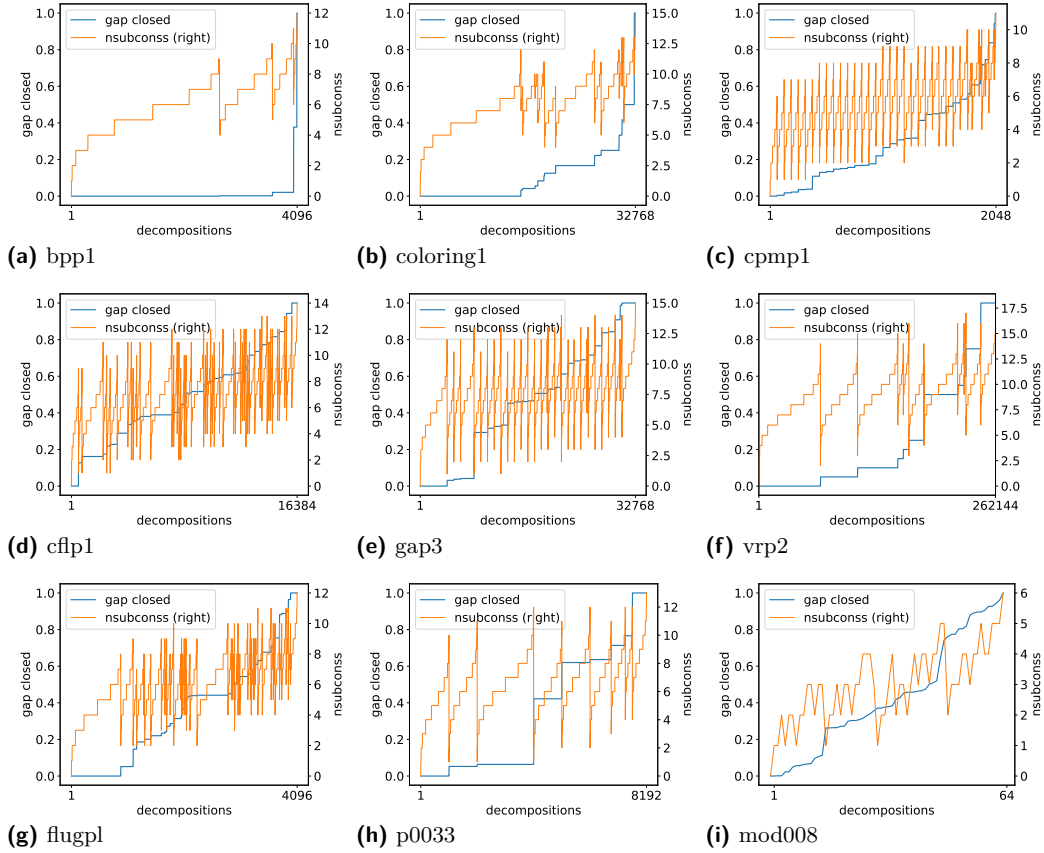
The histograms give the impression that weak(er) dual bounds are more frequent than strong(er) ones. Therefore, we analyze how many “good” DW reformulations exist. We normalize the dual bounds, which helps us to compare them across different models: We define the integrality gap that was closed by DW reformulation $I' \subseteq I$ as

$$\text{gap_cl}(I') := \frac{z_{DW}(I') - z_{LP}}{z_{IP} - z_{LP}}. \quad (4)$$

Note that $\text{gap_cl}(I') = 0 \iff z_{DW}(I') = z_{LP}$ and $\text{gap_cl}(I') = 1 \iff z_{DW}(I') = z_{IP}$.

In Figure 2, we depict the gaps that were closed for all DW reformulations of a particular instance. The plots for instances of different problems have some common features. There are (many) more DW reformulations $I' \subseteq I$ that close only a small amount of the gap than there are DW reformulations with $\text{gap_cl}(I') \approx 1$. This is particularly pronounced for bin packing, vertex coloring, and vehicle routing instances for which a huge portion of DW reformulations are weakest possible (for the considered objective function). Note that these instances are highly symmetric; the experiments suggest that this results in many symmetric DW reformulations yielding the same dual bound. This statement is endorsed by the fact that instances of the capacitated p -median and the capacitated facility location problem, which are more general, less symmetric variants of the bin packing problem, induce more distinct dual bounds than the bin packing instances.

Nevertheless, we notice that most DW reformulations are “in between” the weakest and strongest possible DW reformulations, i.e., for most subsets $I' \subseteq I$ it holds that $z_{LP} < z_{DW}(I') < z_{IP}$. Finally, we look at the number of convexified constraints in Figure 2. The plots illustrate that the pure number $|I'|$ is *not* a good indicator for the strength of the DW reformulation: in the majority of the instances there exist weak DW reformulations with a relatively large number of convexified constraints. Moreover, as the number of minimal DW reformulations is usually not much larger than the number of distinct dual bounds, for each “plateau” of the blue curve there are only few minimal DW reformulations, probably with relatively few convexified constraints (see lowest brown line for each plateau).



■ **Figure 2** The x -axis shows for each instance all DW reformulations $I' \subseteq I$, sorted by dual bound (in case of ties sorted by $|I'|$). The gap closed (4) by each DW reformulation is shown on the y -axis. The secondary y -axis displays the number of convexified constraints of each DW reformulation.

3.2 Extensions to larger Instances?

Our conjecture that $|\{z_{DW}(I') : I' \subseteq I\}| \ll 2^{|I|}$ is impractical to verify on instances with larger $|I|$. Unfortunately, there is little hope that we even obtain a statistical statement from a random sampling of DW reformulations unless we have further information about the structure of the model: the sample size would need to be too large.

The underlying statistics relates to the *distinct elements problem*, which in our context reads as: Given ℓ randomly drawn DW reformulations (and the corresponding dual bounds) from the set of all $2^{|I|}$ DW reformulations, estimate the total number of distinct dual bounds occurring among all DW reformulations. The minimum sample size ℓ needed to estimate this number (with high probability) within a given additive error tolerance $\Delta = c2^{|I|}$ for any small constant c is in $\Theta(\frac{2^{|I|}}{|I|})$ [15] which is not much smaller than $\Theta(2^{|I|})$.

4 The Influence of Individual Constraints on Dual Bounds

We have seen that the number of minimal DW reformulations is very small and the frequency of distinct dual bounds is often (close to) a power of 2. This hints at constraints that do not (or rarely) improve the dual bound when additionally convexified in particular DW reformulations. This individual impact is analyzed next.

We first introduce some notation. For each constraint $i \in I$, we investigate how the dual bound $z_{DW}(I')$ changes for subsets $I' \subseteq I$ with $i \notin I'$ if we add the constraint i to the set I' of convexified constraints, i.e., we compare $z_{DW}(I')$ and $z_{DW}(I' \cup \{i\})$. We define the *gain* of a constraint $i \in I$ when added to the set of convexified constraints $I' \subseteq I$ with $i \notin I'$ as

$$\text{gain}(i, I') := \frac{z_{DW}(I' \cup \{i\}) - z_{DW}(I')}{z_{IP} - z_{LP}}. \quad (5)$$

The normalization to the integrality gap helps again to compare the gains of constraints from different instances. Correspondingly, we define the *average gain* of constraint $i \in I$ as

$$\text{gain}(i) := \frac{\sum_{I' \subseteq I: i \notin I'} \text{gain}(i, I')}{|\{I' \subseteq I : i \notin I'\}|}. \quad (6)$$

Let DW reformulation I' belong to level $\ell = 0, \dots, |I|$ if $|I'| = \ell$. We want to analyze the gain of a constraint when added to the set of convexified constraints of a DW reformulation in a given level. We define the (average) gain of constraint $i \in I$ in level $\ell = 0, \dots, |I| - 1$ as

$$\text{gain}_\ell(i) := \frac{\sum_{I' \subseteq I: |I'| = \ell, i \notin I'} \text{gain}(i, I')}{|\{I' \subseteq I : |I'| = \ell, i \notin I'\}|}. \quad (7)$$

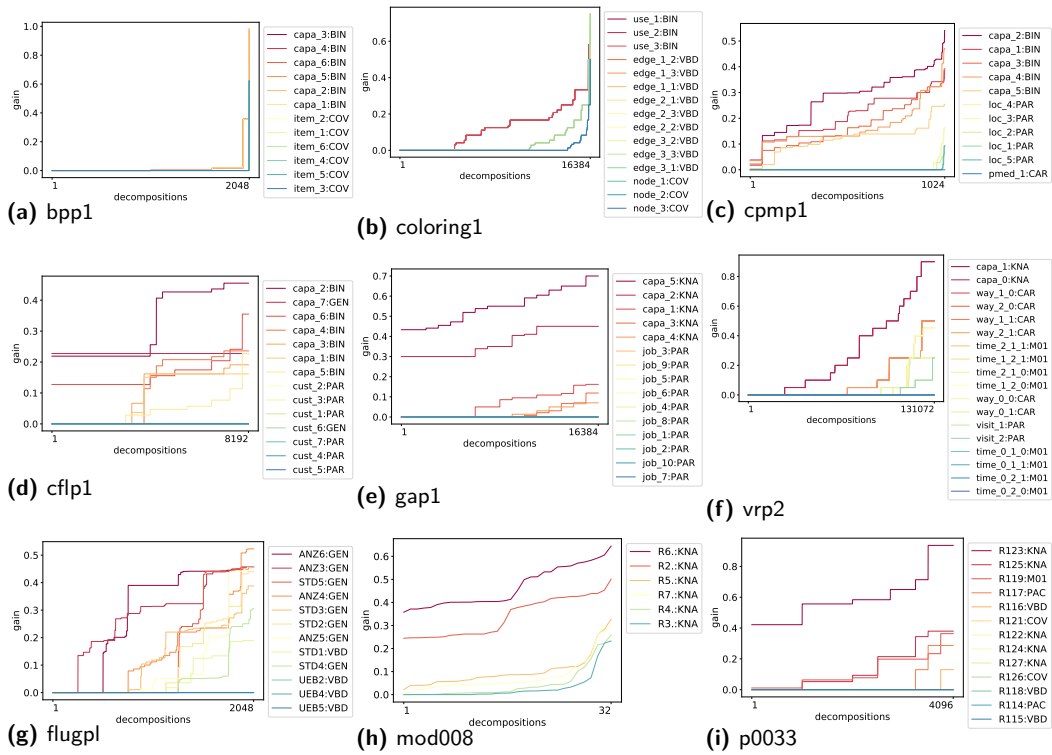
Similarly, the (average) gain of constraint $i \in I$ up to level $\ell = 0, \dots, |I| - 1$ is defined as

$$\text{gain}_{\leq \ell}(i) := \frac{\sum_{I' \subseteq I: |I'| \leq \ell, i \notin I'} \text{gain}(i, I')}{|\{I' \subseteq I : |I'| \leq \ell, i \notin I'\}|}. \quad (8)$$

Obviously, all gains defined in (5)–(8) range in $[0, 1]$. We depict the distribution of gains (defined in Equation (5)) for all constraints in Figure 3. As additional information, we include the constraint types as defined in the MIPLIB [9]. Notice that the lines corresponding to different constraints cross only occasionally. This can be interpreted as follows: Whenever the highest gain of constraint i is higher than the highest gain of constraint i' , the overall gain of constraint i is higher than the overall gain of constraint i' . Hence, the average gain should give a sufficiently accurate view on which constraints have a significant effect on the dual bounds when considering all DW reformulations. Moreover, we observed that for each constraint $i \in I$ the sum of (average) gains across all levels $\sum_{\ell=0}^{m-1} \text{gain}_\ell(i)$ behaves similarly as the average gain $\text{gain}(i)$, which is why we only depict the (average) gain per level including their sum in Figure 4.

The difference in gain of constraints corresponding to different types of constraints is remarkably huge, as can be seen in Figure 4. In particular, bin packing (BIN) and knapsack (KNA) constraints, which are convexified in textbook DW reformulations, have a much larger gain than the other types. This holds not only for the sum of (average) gains in each level, but also for the individual gain in each level as well as the overall gain distribution.

Additionally, we observe that a large gain in low levels is a good indicator for a large gain in higher levels, and hence, a good indicator for a large average gain as well. This correlation becomes visible in Figure 5. The scale of the x -axis in Figure 5 is not uniform since we are only interested in whether there is a correlation between the gain in level ℓ and the average gain independently for each level ℓ at all. The correlation in the bin packing and vertex coloring instances (not shown in Figure 5) is not as high as in the other instances: In several low levels (the exact number depends on the instance) no constraint has positive gain. We assume that this is due to the high symmetry in these instances and that in an optimal LP solution not all bins/colors are used. Apart from this, it seems that the magnitude of the gain in low levels predicts the average gain quite well, which is rather remarkable.



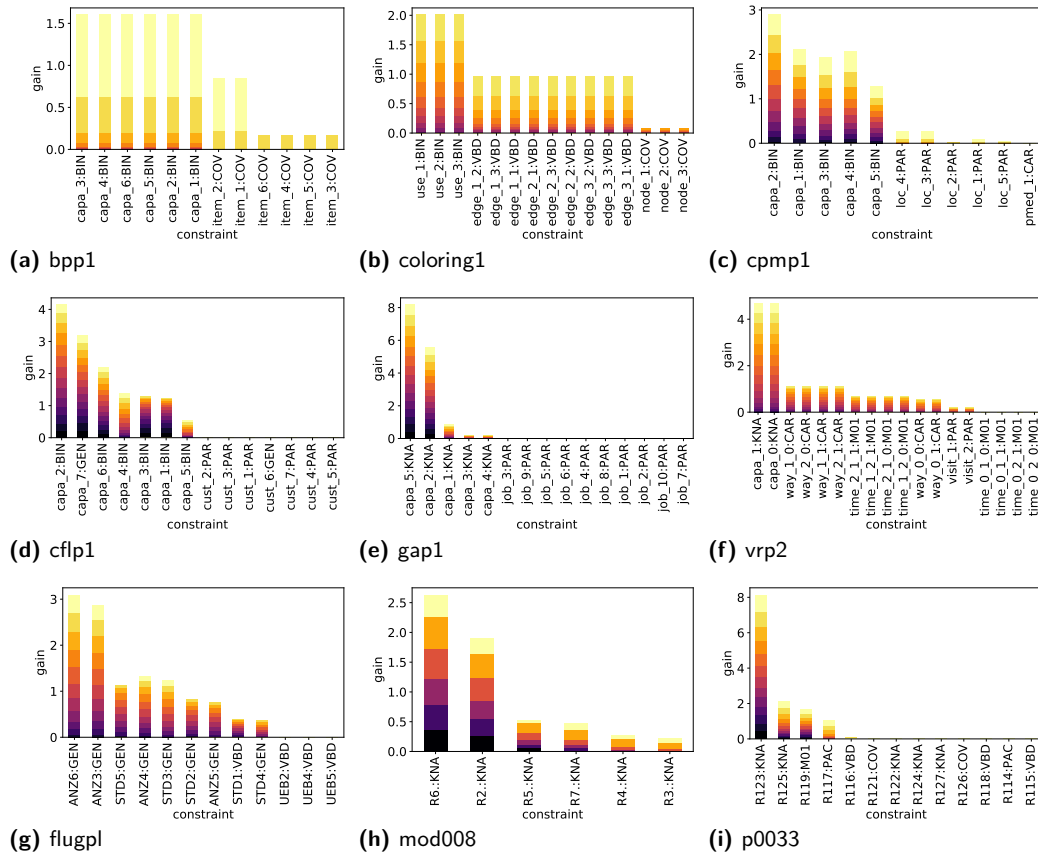
■ **Figure 3** For each constraint $i \in I$ (a line of a particular color) the DW reformulations $I' \subseteq I$ (on the x -axis) are sorted by non-decreasing $\text{gain}(i, I')$, which is shown on the y -axis. The legend lists the constraint's name and type according to the MIPLIB constraint types [9], separated by a colon. The cardinality (CAR) constraints in vehicle routing problems are actually flow conservation constraints (this is called “upgrading” in SCIP/GCG by negating variables).

4.1 Extensions to larger Instances from the MIPLIBs

Figure 5 suggests a correlation between the gain in low levels and the average gain. Since it is intractable to compute the gains in larger instances even for levels 1 or 2, we focus on the gain in level 0, i.e., on larger instances we compute $\text{gain}_0(i)$ for each constraint $i \in I$.

We investigate instances from MIPLIB 2003 and 2010 [1, 9] for which relaxations of DW reformulations were already computed by column generation in [2]. On 12 out of these 38 instances (we excluded mine-166-5 because some DW reformulations failed to solve) there exist constraints having positive $\text{gain}_0(i)$; Figure 6 depicts the number of constraints with positive $\text{gain}_0(i)$ as well as the average $\text{gain}_0(i)$ for each constraint type on these 12 instances.

First of all, we note that set partitioning/packing/covering (PAR/PAC/COV), cardinality (CAR), and invariant knapsack (IVK) constraints cannot have positive gain in level 0 due to Geoffrion's result [7], which can also be seen in Figure 6. For all other types that occur on the MIPLIB instances there exist some constraints having positive gain in level 0. Furthermore, bin packing (BIN) and knapsack (KNA) constraints often have positive, relatively large gain in level 0 compared to other constraints. As in the toy instances, this suggests that convexifying these constraints might give strong dual bounds.



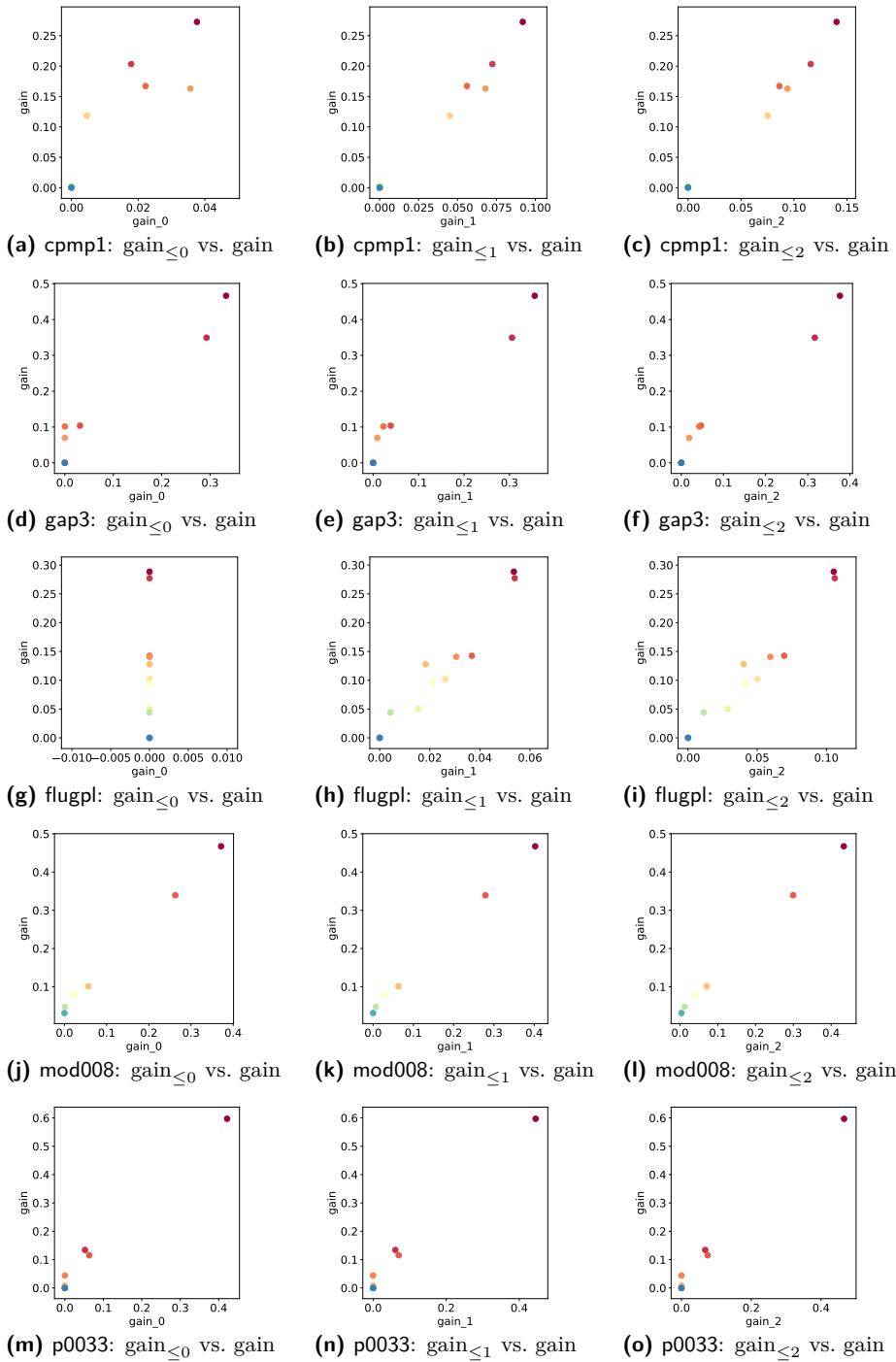
■ **Figure 4** For each constraint (x -axis) the gain summed over all levels is displayed (y -axis). More precisely, the gains of different levels are stacked by increasing level and depicted in different colors.

5 What we have learned and what lies ahead

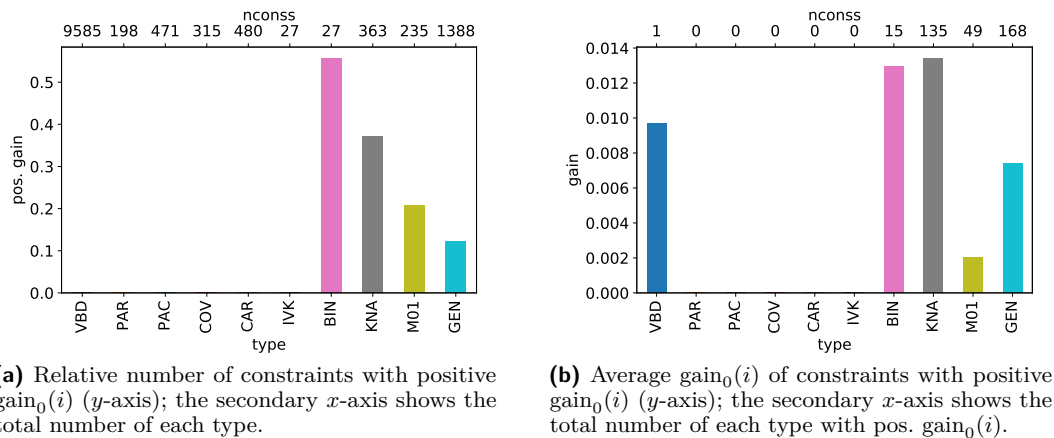
Because of Geoffrion’s result [7] one formulates carefully that a DW reformulation *potentially* gives a stronger dual bound than z_{LP} . Figure 2 suggests that for most instances a random DW reformulation *will* give *some* improvement over z_{LP} . As the actual improvement depends on the objective function, repeating our experiments with several (randomly drawn) objective functions per instance should be interesting.

This is not to say that picking a DW reformulation at random will give one with a strong dual bound; actually, Figure 2 shows that this is rather unlikely. This is particularly visible for bin packing, but also for coloring and vehicle routing. It is fair to say that for bin packing using “the correct” DW reformulation is crucial for obtaining a strong dual bound – and this is precisely a reformulation that we would find in a textbook.

We started this experiment with the sense that there should be some sort of hierarchy of DW reformulations (other than the powerset of I). This is strongly supported (much stronger than we expected) by two experimental results we got: First, only a tiny fraction of the possible $2^{|I|}$ distinct dual bounds actually occurs, which is also true for the number of minimal DW reformulation which is very small (we would love to see geometric/polyhedral explanations for this). Second, individual (types of) constraints have considerably different impacts on strengthening an existing DW reformulation. Some of them seem to have almost no influence at all.



■ **Figure 5** For each constraint $i \in I$ (colored dots), the $\text{gain}_{\leq \ell}(i)$ in levels up to $\ell = 0, 1, 2$ (x -axis) is plotted against the average $\text{gain}(i)$ (y -axis), suggesting a correlation. The colors corresponding to the constraints are identical to the ones in Figure 3.



(a) Relative number of constraints with positive $\text{gain}_0(i)$ (y -axis); the secondary x -axis shows the total number of each type.

(b) Average $\text{gain}_0(i)$ of constraints with positive $\text{gain}_0(i)$ (y -axis); the secondary x -axis shows the total number of each type with pos. $\text{gain}_0(i)$.

■ **Figure 6** Constraint types (c.f. [9]) that occur in the 12 MIPLIB instances of our testset (from [2]) which contain constraints with positive $\text{gain}_0(i)$ (x -axis); in particular, BIN is bin packing, KNA is knapsack, M01 is mixed binary, and GEN is general.

We conjecture that the poset of DW reformulations defined in Section 3 contains a (very) sparse substructure that “represents” all DW reformulations for a given instance. A starting point could consist of the set $\mathcal{I}^* \subseteq 2^I$ of minimal DW reformulations partially ordered by set inclusion (remember that in our experiments $|\mathcal{I}^*| \ll 2^{|I|}$). This again gives a poset $P^* = (\mathcal{I}^*, \subseteq)$ of height at most $|I|$ (and by Dilworth’s theorem of width at least $|\mathcal{I}^*|/|I|$), but our experiments show that the height can actually be smaller, c.f. numbers cmin , cavg , cmax in Table 1. Could studying the structure and properties of such a poset yield insights into DW reformulations and maybe explain the special behavior of bin packing and coloring instances? Even if we could characterize a meaningful substructure of P , would we be able to (efficiently) compute it? Are we able to (efficiently) recognize that a DW reformulation is minimal once we have optimally solved it? Even only answering this for particular problem classes would be valuable. It is interesting in this context whether there exist pathological instances (like the Klee-Minty cubes in linear programming) with $2^{|I|}$ DW reformulations. If so, one might seek output sensitive algorithms for computing e.g., P^* whose complexity depends on the size of the output (here $|P^*|$) to account for the cases (we observed) in which the number of minimal DW reformulations is small.

Equipped with such questions we are optimistic that our experimental work spawns mathematical, algorithmic, and computational questions that hopefully guide us to a better insight into the nature of DW reformulations in general.

References

- 1 T. Achterberg, T. Koch, and A. Martin. MIPLIB 2003. *Oper. Res. Lett.*, 34(4):361–372, 2006.
- 2 M. Bergner, A. Caprara, A. Ceselli, F. Furini, M.E. Lübbecke, E. Malaguti, and E. Traversi. Automatic Dantzig-Wolfe reformulation of mixed integer programs. *Math. Programming*, 149(1–2):391–424, 2015.
- 3 R Bixby, Sebastian Ceria, C McZeal, and M Savelsbergh. An updated mixed integer programming library: Miplib 3.0, 1996.
- 4 D.G. Cattrysse, M. Salomon, and L.N. Van Wassenhove. A set partitioning heuristic for the generalized assignment problem. *European J. Oper. Res.*, 72(1):167–174, 1994.

- 5 P.C. Chu and JE Beasley. A genetic algorithm for the generalised assignment problem. *Comput. Oper. Res.*, 24(1):17–23, 1997.
- 6 G. Gamrath and M.E. Lübbecke. Experiments with a generic Dantzig-Wolfe decomposition for integer programs. In P. Festa, editor, *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA)*, volume 6049 of *Lect. Notes Comput. Sci.*, pages 239–252, Berlin, 2010. Springer-Verlag.
- 7 A.M. Geoffrion. Lagrangean relaxation for integer programming. *Math. Programming Stud.*, 2:82–114, 1974.
- 8 Kaj Holmberg, Mikael Rönnqvist, and Di Yuan. An exact algorithm for the capacitated facility location problems with single sourcing. *European J. Oper. Res.*, 113(3):544–559, 1999.
- 9 T. Koch, T. Achterberg, E. Andersen, O. Bastert, T. Berthold, R. E. Bixby, E. Danna, G. Gamrath, A. M. Gleixner, S. Heinz, A. Lodi, H. Mittelmann, T. Ralphs, D. Salvagnin, D. E. Steffy, and K. Wolter. MIPLIB 2010. *Math. Program. Comput.*, 3(2):103–163, 2011.
- 10 Anuj Mehrotra and Michael A Trick. A column generation approach for graph coloring. *INFORMS J. Comput.*, 8(4):344–354, 1996.
- 11 Rashed Sahraeian and Payman Kaveh. Solving capacitated p-median problem by hybrid k-means clustering and fixed neighborhood search algorithm. In *Proceedings of the 2010 International Conference on Industrial Engineering and Operation Management*, pages 1–6, 2010.
- 12 P. Schwerin and G. Wäscher. The bin-packing problem: A problem generator and some numerical experiments with ffd packing and mtp. *Internat. Trans. Oper. Res.*, 4(5-6):377–389, 1997.
- 13 F. Vanderbeck and M.W.P. Savelsbergh. A generic view of Dantzig-Wolfe decomposition in mixed integer programming. *Oper. Res. Lett.*, 34(3):296–306, 2006.
- 14 J.T. Witt and M.E. Lübbecke. Dantzig-Wolfe reformulations for the stable set problem. repORt 2015–029, Lehrstuhl für Operations Research, RWTH Aachen University, Nov 2015. URL: <http://www.or.rwth-aachen.de/research/publications/2015-DW-stable.pdf>.
- 15 Y. Wu and P. Yang. Sample complexity of the distinct elements problem. *ArXiv e-prints*, Dec 2016. arXiv:1612.03375.

Experimental Evaluation of Parameterized Algorithms for Feedback Vertex Set

Krzysztof Kiljan

Institute of Informatics, University of Warsaw, Poland
krzysztof.kiljan@student.uw.edu.pl

Marcin Pilipczuk

Institute of Informatics, University of Warsaw, Poland
malcin@mimuw.edu.pl

Abstract

FEEDBACK VERTEX SET is a classic combinatorial optimization problem that asks for a minimum set of vertices in a given graph whose deletion makes the graph acyclic. From the point of view of parameterized algorithms and fixed-parameter tractability, FEEDBACK VERTEX SET is one of the landmark problems: a long line of study resulted in multiple algorithmic approaches and deep understanding of the combinatorics of the problem. Because of its central role in parameterized complexity, the first edition of the Parameterized Algorithms and Computational Experiments Challenge (PACE) in 2016 featured FEEDBACK VERTEX SET as the problem of choice in one of its tracks. The results of PACE 2016 on one hand showed large discrepancy between performance of different classic approaches to the problem, and on the other hand indicated a new approach based on half-integral relaxations of the problem as probably the most efficient approach to the problem. In this paper we provide an exhaustive experimental evaluation of fixed-parameter and branching algorithms for FEEDBACK VERTEX SET.

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

Keywords and phrases Empirical Evaluation of Algorithms, Feedback Vertex Set

Digital Object Identifier 10.4230/LIPIcs.SEA.2018.12

Funding Supported by the “Recent trends in kernelization: theory and experimental evaluation” project, carried out within the Homing programme of the Foundation for Polish Science co-financed by the European Union under the European Regional Development Fund.

1 Introduction

The FEEDBACK VERTEX SET problem asks to delete from a given graph a minimum number of vertices to make it acyclic. It is one of the classic graph optimization problems, appearing already on Karp’s list of 21 NP-hard problems [24]. In this work, we are mostly focusing on *fixed-parameter algorithms* for the problem, that is, exact (and thus exponential-time, as we are dealing with an NP-hard problem) algorithms whose exponential blow-up in the running time bound is confined by a proper parameterization. More formally, a fixed-parameter algorithm on an instance of size n with parameter value k runs in time bounded by $f(k) \cdot n^c$ for some computable (usually exponential) function f and a constant c independent of k .

FEEDBACK VERTEX SET is one of the most-studied problems from the point of view of parameterized algorithms. A long line of research [4, 15, 16, 29, 23, 12, 18, 9, 8, 3, 11] lead to a very good understanding of the combinatorics of the problem, multiple known algorithmic approaches, and a long “race” for the fastest parameterized algorithm under the



© Krzysztof Kiljan and Marcin Pilipczuk;
licensed under Creative Commons License CC-BY
17th International Symposium on Experimental Algorithms (SEA 2018).

Editor: Gianlorenzo D’Angelo; Article No. 12; pp. 12:1–12:12



Leibniz International Proceedings in Informatics

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

parameterization of the solution size (i.e., the parameter k equals the size of the solution we are looking for). Among many approaches, one can find the following:

1. A classic randomized algorithm of Becker et al. [3] with expected running time $4^k n^{\mathcal{O}(1)}$. This algorithm is based on the following observation: after performing a set of simple reductions that reduce the graph to minimum degree at least 3, at least half of the edges of the graph are incident with solution vertices, so a random endpoint of an edge chosen uniformly at random is in the solution with probability at least $1/4$.
2. A line of research of branching algorithms based on iterative compression and an intricate measure to bound the size of the branching tree [9, 8, 26], leading to a simple $3.62^k n^{\mathcal{O}(1)}$ -time deterministic algorithm [26]. These algorithms solve a “disjoint” version of the problem where, given a set $U \subseteq V(G)$ such that $G - U$ is a forest, one seeks for a solution *disjoint* with U of size at most k . A branching algorithm picks a vertex $v \in V(G) \setminus U$ and includes it in the solution (deletes and decreases k by one) or moves to U . The crucial observation is that here, apart from k , the number of connected components of $G[U]$ is a useful potential to measure the progress of a branching algorithm: if the branching pivot v has many neighbors in U , then moving it to U decreases the number of connected components significantly.
3. A polynomial-time algorithm for FEEDBACK VERTEX SET in subcubic graphs [8] (see also [26] for a simpler proof), used heavily in other approaches as a subroutine. The algorithm build on a reduction from FEEDBACK VERTEX SET in cubic graphs to the matroid matching problem in graphic matroids, which is polynomial-time solvable.
4. A Monte Carlo algorithm running in time $3^k n^{\mathcal{O}(1)}$ via the Cut&Count technique [11]. The Cut&Count technique is a generic framework for turning algorithms with running time $2^{\mathcal{O}(t \log t)} n^{\mathcal{O}(1)}$ for connectivity problems in graphs of treewidth bounded by t into Monte-Carlo algorithms with running time bounds $c^t n^{\mathcal{O}(1)}$ by changing the “connectivity” requirement into a modulo-2 counting requirement. For FEEDBACK VERTEX SET, the base of the exponent has been optimized to $c = 3$, and has been proven to be optimal under the Strong Exponential Time Hypothesis.
5. A surprisingly simple branching algorithm by Cao [7] with a running time bound $8^k n^{\mathcal{O}(1)}$. Cao [7] shown that the following algorithm has running time bounded by $8^k n^{\mathcal{O}(1)}$: after applying simple reduction rules, one either branch on maximum-degree vertex or, if the maximum degree is at most 3, solve the instance in polynomial time.
6. A branching algorithm based on intricate half-integral relaxation due to Imanishi and Iwata [19, 20] with running time bound $4^k k^{\mathcal{O}(1)} n$. The work of Iwata, Wahlström, and Yoshida [21] showed a generic framework for branching algorithms for various graph separation problems. In the case of FEEDBACK VERTEX SET, [21] shows that the following local problem has a polynomial-time solvable half-integral relaxation: given a root vertex $s \in V(G)$, find a minimum-size set $X \subseteq V(G) \setminus \{s\}$ such that the connected component of $G - X$ that contains s is a tree. Imanishi and Iwata [19, 20] showed a linear-time combinatorial algorithm that solves this half-integral relaxation. The final algorithm follows the framework of [21]: it branches on vertices for which the half-integral relaxation solution was undecided (i.e., gave it the non-integral value of $1/2$).

Another line of research concerns so-called polynomial kernels for the problem [6, 5, 30, 20].

Parameterized Algorithms and Computational Experiments challenge is an annual programming challenge started in 2016 that aims to “investigate the applicability of algorithmic ideas studied and developed in the subfields of multivariate, fine-grained, parameterized, or fixed-parameter tractable algorithms”. With two successful editions so far [13, 14] and the third one currently being conducted, PACE continues to bring together theory and practice in parameterized algorithms community.

The first edition of PACE in 2016 featured FEEDBACK VERTEX SET as the problem of choice in Track B. The winning entry by Imanishi and Iwata, implementing the aforementioned algorithm based on half-integral relaxation, turned out to outperform the second entry by the second author of this work [28], based on the algorithm of Cao [7]. The winning margin has been substantial: out of 130 test cases, the winning entry solved 84, while the second entry solved 66.

These results indicated the branching algorithms based on half-integral relaxation of the problem as potentially most efficient approach to FEEDBACK VERTEX SET in practice. Furthermore, experimental results of Akiba and Iwata [2] showed big potential in a branching algorithm based on the same principle for the VERTEX COVER problem.

In the light of the above, we see the need to rigorously experimentally evaluate different approaches to FEEDBACK VERTEX SET. While the results of PACE 2016 indicate algorithms based on half-integral relaxation as potentially fastest, a lot of differences may come from the use of different preprocessing routines, different choice of lower bounding or pruning techniques, or even simply different data structures handling basic graph operations.¹

In this work, we offer such a comparison. We implement a number of branching strategies mentioned above. Our implementations use the same data structures for handling graphs, the same implementations of basic graph operations, the same basic reduction rules such as suppressing degree-2 vertices, and the same branching framework. Most of the tested approaches differ only at a small fraction of code: they usually differ by the choice of the branching pivot, and some use one or more approach-specific reduction rules.

In our experiments, we follow up the set up of PACE 2016: we take their 230 instances (100 public and 130 hidden, on which evaluation took place) as our benchmark set, allow each algorithm to run for 30 minutes on each test instance.

The paper is organized as follows. In Section 2 we discuss the studied approaches and some technical details of the implementations. Section 3 discusses the setup of the experiment. Section 4 presents results, while Section 5 concludes the paper.

2 Studied algorithms

Most of the known branching algorithm for FEEDBACK VERTEX SET, in particular all algorithms studied in our work, follow the following general framework.

Every instance to be solved by a recursive branching algorithm consists of a multigraph G , a set $U \subseteq V(G)$ of *undeletable* vertices and allowed budget k for solution size. The goal is to find a set $X \subseteq V(G) \setminus U$ of size at most k such that $G - X$ is a forest. Each branching step consists of picking a vertex $v \in V(G) \setminus U$ and branching into two cases: either v gets picked to a solution (and the algorithm recurses on the instance $(G - \{v\}, U, k - 1)$) or moved to set U (and the algorithm recurses on the instance $(G, U \cup \{v\}, k)$).

The intuition of the progress of the algorithm is as follows. In the first branch, the budget k gets decreased. For the second branch, note that the algorithm can safely terminate for instances with $G[U]$ not being acyclic. Thus, if the branching pivot v has d neighbors in U ,

¹ A good example here is as follows. In FEEDBACK VERTEX SET, it is natural to keep the graph in the form of adjacency lists, as the considered graphs are usually of constant average degree. However, given an edge uv , it is not clear whether the vertex u in its adjacency list should only store the vertex v , or also a pointer to the position where v keeps u in its adjacency list. On one hand, such pointers greatly simplify the operations of deleting a vertex or contracting an edge. On the other hand, they effectively double the size of the graph data structure, increasing the cost of copying the graph in the branching step.

then the number of connected components of $G[U \cup \{v\}]$ decreases by $(d - 1)$ as compared to $G[U]$.

Between branching step, the algorithm is allowed to perform a number of reduction (preprocessing) steps. In the literature, a number of simple reduction steps are known that are performed by all our algorithms:

1. If k gets negative or $G[U]$ is not acyclic, stop.
2. Remove all vertices of degree at most 1.
3. If two vertices are connected by more than 2 parallel edges, reduce their multiplicity to 2.
4. If there exists a vertex v that has a self-loop, or there exists a single connected component D of $G[U]$ more than one edge incident with v and a vertex of D (i.e., there exists a cycle C in G with v being the only vertex of $V(C) \setminus U$), then greedily include it in the solution (i.e., delete it and decrease k by one).
5. Suppress vertices of degree 2. That is, if a vertex v is of degree 2 with incident edges vu and vw is present, delete v and replace it with an edge uw .
6. If there exists a vertex v with two neighbors u and w , such that vu is a single edge and vw is a double edge, greedily include w in the solution.

For efficiency, the above reduction rules are implemented in the form of a queue of vertices to reduce: when the number of distinct neighbors of a vertex drop to two or less, or a vertex gets a self-loop, it is enqueued, and preprocessing routines start by clearing up the queue.

Other preprocessing steps used by some of our algorithms are:

Split into connected components. If the instance becomes disconnected, solve each connected component independently.

Solving subcubic instances. As proven by [8], if every vertex $v \in V(G) \setminus U$ is of degree at most three, then the corresponding instance is polynomial-time solvable. Kociumaka and Pilipczuk [26] provided a simpler proof of this result via a reduction to the matroid parity problem in graphic matroids, which proceeds as follows. First, apply the known reduction rules that reduce the problem to the case when every $v \in V(G) \setminus U$ is of degree exactly three. Second, subdivide every edge $uv \in E(G - U)$ with a new vertex $x_{uv} \in U$, so that $V(G) \setminus U$ is an independent set. Third, for every $v \in V(G) \setminus U$, pick two out of the three edges incident to v as a pair P_v . Let $Q = E(G) \setminus \bigcup_{v \in V(G) \setminus U} P_v$ be the remaining edges; note that every $v \in V(G) \setminus U$ is of degree exactly 1 in the graph $(V(G), Q)$. Then, it is easy to observe that the FEEDBACK VERTEX SET problem is equivalent to picking a set $Y \subseteq V(G) \setminus U$ of maximum cardinality such that $(V(G), Q \cup \bigcup_{v \in Y} P_v)$ is a forest. This condition is the same as requiring Y to be a maximum matching in the graphic matroid of the graph G/Q (G with edges of Q contracted) with pairs $(P_v)_{v \in V(G) \setminus U}$.

In some of our algorithms, we use the approach of [26] to solve such instances. As the underlying solver to the matroid matching problem, we use the augmenting path algorithm of Gabow and Stallmann [17].

Solution lower bound. Consider an instance (G, U, k) and let v_1, v_2, \dots be the vertices of $V(G) \setminus U$ in the nonincreasing order of degrees in G . If (G, U, k) admits a solution X of size j , then $G - X$ has at least $|E(G)| - \sum_{i=1}^j \deg_G(v_i)$ edges. On the other hand, if $G - X$ is a forest, it has less than $|V(G)| - j$ edges. Consequently, we can safely stop if for all $0 \leq j \leq k$ we have that

$$|E(G)| - \sum_{i=1}^j \deg_G(v_i) \geq |V(G)| - j.$$

The above pruning strategy has been used in the entry of Imanishi and Iwata [19].

Unless otherwise noted, all our implementations split instances into connected components. We also compare a number of selected approaches without this preprocessing step to see its impact on performance.

The algorithm of Cao [7] uses all aforementioned simple reduction rules as well as the solver of subcubic instances. On a branching step, it simply chooses the vertex of highest degree. As shown by Cao, such an algorithm has running time bound $8^k n^{\mathcal{O}(1)}$. We also test a variant of the algorithm of Cao that first branches on vertices incident with double edges, a variant that does not use the subcubic instance solver, and a variant that prunes the search tree via the aforementioned lower bound.

2.1 Approximation and iterative compression

The algorithms of [9, 8, 26] operate in the framework of iterative compression. That is, their central subroutine solves a seemingly simpler problem, where additionally a slightly too large solution Y is given, and the algorithm first branches on the vertices of Y (putting each $y \in Y$ into the solution or into set U). Since at the beginning $G - Y$ is a forest, and every vertex of Y is either deleted or put into U , we obtain the property that $G - U$ is a forest. This greatly helps in the analysis.

In the literature, the set Y is traditionally taken from the iterative compression step. One picks an order $V(G) = \{v_1, v_2, \dots, v_n\}$ and solves iteratively FEEDBACK VERTEX SET on graphs $G_i = G[\{v_1, v_2, \dots, v_i\}]$. Given a solution X_{i-1} to G_{i-1} , one can set $Y = X_{i-1} \cup \{v_i\}$ for G_i .

However, such an approach leads to multiple invocation of the same algorithm, and a substantial multiplicative overhead in the running time bound. In our algorithms, we instead find Y via a simple heuristic: reduce the graph via simple reductions as long as possible and, when impossible, delete the vertex of highest degree. In Section 4 we discuss the performance of this heuristic on our test data.

Our branching framework keeps a queue of **branching hints** and, if nonempty, the algorithm always branches on a vertex from the queue. For algorithms based on iterative compression, the queue is initiated by an approximate solution found by our heuristic. This corresponds to passing the set Y to the algorithms based on iterative compression, but allows to reduce some of the vertices of the set Y by reductions after a number of branching steps. If an algorithm does not use iterative compression, the queue is empty through the entire run of the algorithm.

The algorithm of [9] implements the iterative compression framework and branches on a vertex of $V(G) \setminus U$ that is incident to maximum number of edges leading to U . As shown in [9], such an algorithm has $5^k n^{\mathcal{O}(1)}$ time bound guarantee.

The algorithm of [26] is arguably a simplification of the arguments of [8], so we implement only the first one. It modifies the algorithm of [9] in the following way:

- It leaves alone vertices $v \in V(G) \setminus U$ that are of degree 3 and all their incident edges lead to U (such vertices are called henceforth *tents*). The crux is that if every vertex $v \in V(G) \setminus U$ is a tent, then we can apply the polynomial-time algorithm of [8, 26] to the instance. In other words, tents form a “polynomial-time solvable” part of the instance.
- Given a vertex $v \in V(G) \setminus U$ of degree 3 with exactly one neighbor u in $V(G) \setminus U$ (and other 2 neighbors in U), it proceeds as follows:
 - subdivide the edge uv with a new vertex $w \in U$; note that this does not change the set of feasible solutions to the FEEDBACK VERTEX SET problem;
 - marks w irreducible for the reduction suppressing degree-2 vertices.

Note that this operation turns v into a tent, while reducing the number of vertices of $V(G) \setminus U$ that are not tents.

- It applies the solver for subcubic instances if every vertex of $V(G) \setminus U$ is a tent.

As shown in [26], such an algorithm has $3.62^k n^{\mathcal{O}(1)}$ time bound guarantee.

Inspired by the methods of choice of branching pivots of the algorithms [9, 8, 26], we also test a variant of Cao’s algorithm where the choice of the branching pivot is as in [9]: vertex with maximum number of neighbors in U (but, contrary to [9], no iterative compression).

Additionally, we check how much the algorithms can be sped up by adding pruning via the aforementioned lower bound and if the Cao’s algorithm can benefit from the use of iterative compression.

2.2 Branching based on half-integral relaxation

Iwata, Wahlström, and Yoshida showed a generic approach to numerous transversal problems via half-integral relaxations [21]. They are all based on the following principle: a half-integral relaxation of a variant of the problem is defined and shown to be polynomial-time solvable. Furthermore, the solution to the half-integral relaxation has some persistency property: it either indicates some greedy choice for the integral problem, or indicates a good branching pivot. In this approach, the time needed to find a solution of the half-integral relaxation is critical.

The algorithms of [21] run in linear time for edge deletion problems, but unfortunately for vertex-deletion problems the subroutine that finds the half-integral relaxation requires solving linear programs. The main contribution of Iwata [20] (implemented in the PACE 2016 entry by Iminishi and Iwata [19]) is a combinatorial linear-time solver for the half-integral relaxation in the special case of FEEDBACK VERTEX SET. We reimplement this solver in our branching framework.

Iwata [20] also observed that the half-integral relaxation can be used to find a polynomial kernel for the problem, improving the previous seminal kernel of Thomassé [30]. Apart from the reduction rules mentioned before, this kernel employs another involved reduction rule, applicable on vertices of degree more than $2k$. All our implementations based on a half-integral relaxation implement this preprocessing step as well.

The half-integral relaxation of [21, 20] does not solve FEEDBACK VERTEX SET directly, but rather given an undeletable vertex $u \in U$, tries to separate an acyclic connected component (i.e., a tree) containing u from the rest of the graph. On high level, the branching strategy of the algorithm is as follows. If $U = \emptyset$, we branch on any vertex; we choose one of highest degree here for efficiency. Otherwise, we use a vertex $u \in U$ as a root for the half-integral relaxation. The persistence properties of the relaxation ensure that we can perform a greedy step unless the relaxation put values 0.5 on all neighbors of u . If this is the case, we branch on a neighbor of u ; we choose highest-degree neighbor here for efficiency. Note that once a tree component with u gets separated, simple reduction rules delete it from the graph.

Since the kernelization routine of [20] is computationally expensive, it is not obvious if one should apply it at every step. We experiment with two variants: when we run the kernelization step at every step, or only at steps with $U = \emptyset$. Furthermore, we also check how much pruning with the lower bound heuristic or solver of subcubic instances helps.

■ **Table 1** Comparison of the size of the approximate and exact solution found on 127 instances.

difference approximation minus optimum	0	1	2	>2	> 10% · optimum
number of instances	89	30	4	4	5

3 Experiment setup

3.1 Hardware and code

The experiments have been performed on a cluster of 16 computers at the Institute of Informatics, University of Warsaw. Each machine was equipped with Intel Xeon E3-1240v6 3.70GHz processor and 16 GB RAM. All machines shared the same NFS drive. Since the size of the inputs and outputs to the programs is small, the network communication was negligible during the process.

The code has been written in C++ and is available at [25] or at project’s webpage [1].

3.2 Test cases

As discussed in the introduction, we repeat the setup of the PACE 2016 challenge [13]. At PACE 2016, the organizers gathered 230 graphs from different sources [27]. A subset of 100 of them has been made public prior to the competition deadline, and the final evaluation has been made on the hidden 130 instances. We run every tested algorithm on each of the 230 instances with 30 minutes timeout.

Our of the test instances, we gathered two subsets to compare actual running times of the algorithm. The first set, *A*, consists of test cases solved by all algorithms, but with substantial running time of some of them. The second set, *B*, is defined similarly, but with regards only to the algorithms that use pruning via the lower bound. More precisely, set *A* consists of the following 14 tests:

```
hidden_001  hidden_007  hidden_012  hidden_056  hidden_065  hidden_083  hidden_099
hidden_106  public_011  public_014  public_037  public_069  public_076  public_086
```

The set *B* consists of the following 7 tests:

```
hidden_022  hidden_041  hidden_068  hidden_088  public_035  public_066  public_067
```

We also gather sizes of approximate solution found by our heuristic and compare it with the optimal size found by the algorithm.

4 Results

A full table with running times of each program at each test can be found at project’s website [1].

4.1 Performance of the approximation heuristic

We compared the performance of the approximation heuristic discussed in Section 2.1 (i.e., greedily take the largest-degree vertex after applying the simple reduction rules) with the size of the optimum solution that was known to us on 127 instances. The results are in Table 1. On only 8 instances, the approximate solution was more than one vertex larger than the optimum one. Consequently, the approximate solution can serve well as the basis for iterative compression.

■ **Table 2** Comparison of different algorithms. The first column indicates the base of the algorithm: Cao’s [7], CFLLV for Chen et al. [9], KP for Kociumaka and Pilipczuk [26], and II for Imanishi and Iwata [19, 20]. II/kernel stands for the algorithm of [19, 20] that runs the kernelization step more often, namely at every branching step. Cao/double stands for the algorithm of [7] that first branches on double edges. Cao/undel stands for the algorithm of [7] that chooses branching pivot with regards to maximum degree to undeletable vertices. In the optimizations columns, CC stands for splitting into connected components, deg3 for the use of solver of subcubic instances, LB for the use of pruning with the lower bound, and IC stands for iterative compression.

algorithm	optimizations				solved instances			total time (MM:SS.ms)	
	CC	deg3	LB	IC	all	public	hidden	set <i>A</i>	set <i>B</i>
Cao		+			100	48	52	35:17.21	-
CFLLV				+	91	44	47	35:16.06	-
KP		+		+	101	49	52	36:22.48	-
Cao	+	+			101	48	53	37:31.83	-
CFLLV	+			+	91	44	47	31:23.63	-
KP	+	+		+	101	49	52	32:00.49	-
Cao	+				91	43	48	38:00.78	-
II	+				92	46	46	34:51.17	-
II/kernel	+				90	45	45	75:11.16	-
Cao	+	+	+		123	62	61	0:19.28	10:38.88
Cao/double	+	+	+		122	62	60	3:22.52	21:31.54
Cao/undel	+	+	+		118	58	60	0:35.67	8:39.05
Cao	+	+	+	+	123	62	61	0:19.99	10:37.88
Cao/double	+	+	+	+	122	62	60	3:13.71	14:31.66
Cao/undel	+	+	+	+	118	58	60	0:36.88	8:12.80
CFLLV	+		+	+	117	57	60	0:37.61	8:12.88
KP	+	+	+	+	118	58	60	0:38.95	8:28.46
Cao	+		+		122	61	61	0:22.94	10:28.32
II	+		+		117	58	59	1:36.79	25:31.95
II	+	+	+		118	59	59	1:37.08	25:33.76
II/kernel	+		+		109	55	54	7:24.42	-

4.2 Comparison

We have run 21 different algorithms on the whole test data. A CSV file with full results is available at the project’s webpage [1]. Table 2 contains aggregated values: number of solved test instances within the time limit (30 minutes per instance) and the total running time on sets *A* and *B*. Please see the caption of Table 2 for description of the notation used in the table.

The first nine algorithms do not use pruning via the lower bounding technique, and the first three do not use splitting into connected components. They are mostly meant to compare basic approaches.

Without the lower bound pruning, the best approaches are Cao’s [7] and Kociumaka-Pilipczuk [26], and they seem to be rather incomparable. The other algorithm based on iterative compression of Chen et al. [9] is clearly outperformed by the other two, and the same holds for the branching algorithm based on half-integral relaxation [19, 20].

The first three rows differ from rows 4-6 by the usage of splitting into connected components. They show that the effect of this improvement is small, and even hurt a bit Cao’s algorithm [7].

The 7th row treats Cao’s algorithm [7] without the solver of the subcubic instances. It indicates that this solver is essential for the performance of Cao’s algorithm.

The 9th row treats the Imanishi-Iwata algorithm [19, 20] with more often application of the kernelization step, namely at every branching step (not only at the ones with $U = \emptyset$). It shows that the step is too expensive to execute it that often.

Let us now discuss the algorithms with the lower bound pruning step. First, the results show that the pruning step greatly improves performance for all algorithms.

With regards to the variants of Cao’s algorithm [7], the results show that any mutation of the branching pivot rule here is undesirable. Also, adding the iterative compression step does not seem to have any particular impact on the performance. Interestingly, the negative effect of dropping the solver of the subcubic instances mostly disappear if one adds the pruning step.

For the branching algorithm based on the half-integral relaxation [19, 20], the last three rows of Table 2 again confirm the corollary that more often application of the kernelization step is undesirable. There is little difference with addition of the solver of subcubic instances (the main difference comes from the fact that the test set contains one huge cubic graph, `public_84`, which is not amenable to any branching technique we tested).

Finally, in our experiments the best variant of Cao’s algorithm [7] with the pruning slightly outperforms the best variant of the branching algorithm based on half-integral relaxations [19, 20]. However, the difference (5 tests more and roughly $3\times$ speed-up on sets A and B) is not big enough to decisively conjecture its advantage.

First, it is possible that a $3\times$ speed-up can be gained by low-level optimizations of the solver of the half-integral relaxations. Arguably, Cao’s algorithm [7] is much simpler and thus easier to optimize. Second, it may be also an artifact of the chosen test data: there are 5 tests in the data set that were solved by the penultimate algorithm in Table 2, but not by the 10th (and 10 tests vice-versa). That is, there are types of instances solved significantly faster by one of the approaches but not by another.

We conclude with a remark about comparison with PACE’16 results [13], where the entry of Imanishi and Iwata [19] solved 84 hidden instances while the entry of the second author [28], based on Cao’s algorithm [7], solved 66. While this data seemingly stands in contradiction with the results in Table 2, one should note that the two entries differ significantly in other internals. Most importantly, the first entry used pruning with the lower bound, while the second one did not. Other difference includes: removal of bridges in the first entry vs splitting into 2-connected components in the second, and the use of bounded treewidth subroutine in the second entry. In other words, our results indicate that the big difference in the performance of the first two entries at PACE 2016 were mainly caused by the difference in preprocessing routines and pruning heuristics (and possibly low-level optimizations) rather than in the underlying base branching algorithm.

5 Conclusions

We have conducted a thorough experimental evaluation of various parameterized algorithms for the FEEDBACK VERTEX SET problem, following the setup of Parameterized Algorithms and Computational Experiments Challenge from 2016 [13]. Our results does not confirm greater advantage of the approach based on half-integral relaxations that was suggested by PACE’16 results, but rather suggest that lower bounding techniques and low-level optimizations were decisive at PACE’16.

On the other hand, the approach via half-integral relaxation turned out to be almost on par with the best variant of Cao’s algorithm [7]. This still indicates big potential in this approach, in particular in the light of the recent (theoretical) generalization of this approach to other problems [22]. In particular, it would be interesting to see an experimental evaluation of parameterized algorithms for MULTIWAY CUT with the comparison of the approach of [22] with the more classic ones via so-called important separators (see Chapter 8 of [10]). Other interesting problem to study is ODD CYCLE TRANSVERSAL and its edge-deletion variant EDGE BIPARTIZATION, where again both classic and half-integral relaxation-based approaches are known.

References

- 1 Recent trends in kernelization: theory and experimental evaluation — project website, 2018. URL: <http://kernelization-experiments.mimuw.edu.pl>.
- 2 Takuya Akiba and Yoichi Iwata. Branch-and-reduce exponential/fpt algorithms in practice: A case study of vertex cover. *Theor. Comput. Sci.*, 609:211–225, 2016. doi:10.1016/j.tcs.2015.09.023.
- 3 Ann Becker, Reuven Bar-Yehuda, and Dan Geiger. Randomized algorithms for the loop cutset problem. *J. Artif. Intell. Res. (JAIR)*, 12:219–234, 2000. URL: <http://www.cs.washington.edu/research/jair/abstracts/becke00a.html>.
- 4 Hans L. Bodlaender. On disjoint cycles. *Int. J. Found. Comput. Sci.*, 5(1):59–68, 1994.
- 5 Hans L. Bodlaender and Thomas C. van Dijk. A cubic kernel for feedback vertex set and loop cutset. *Theory Comput. Syst.*, 46(3):566–597, 2010. doi:10.1007/s00224-009-9234-2.
- 6 Kevin Burrage, Vladimir Estivill-Castro, Michael R. Fellows, Michael A. Langston, Shev Mac, and Frances A. Rosamond. The undirected feedback vertex set problem has a $poly(k)$ kernel. In Hans L. Bodlaender and Michael A. Langston, editors, *IWPEC*, volume 4169 of *Lecture Notes in Computer Science*, pages 192–202. Springer, 2006. doi:10.1007/11847250_18.
- 7 Yixin Cao. A naive algorithm for feedback vertex set. In Raimund Seidel, editor, *1st Symposium on Simplicity in Algorithms, SOSA 2018, January 7-10, 2018, New Orleans, LA, USA*, volume 61 of *OASICS*, pages 1:1–1:9. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018. doi:10.4230/OASICS.SOSA.2018.1.
- 8 Yixin Cao, Jianer Chen, and Yang Liu. On feedback vertex set new measure and new structures. In Haim Kaplan, editor, *SWAT*, volume 6139 of *Lecture Notes in Computer Science*, pages 93–104. Springer, 2010. doi:10.1007/978-3-642-13731-0_10.
- 9 Jianer Chen, Fedor V. Fomin, Yang Liu, Songjian Lu, and Yngve Villanger. Improved algorithms for feedback vertex set problems. *J. Comput. Syst. Sci.*, 74(7):1188–1198, 2008. doi:10.1016/j.jcss.2008.05.002.
- 10 Marek Cygan, Fedor V Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized algorithms*. Springer, 2015.
- 11 Marek Cygan, Jesper Nederlof, Marcin Pilipczuk, Michał Pilipczuk, Johan M. M. van Rooij, and Jakub Onufry Wojtaszczyk. Solving connectivity problems parameterized by treewidth in single exponential time. In Rafail Ostrovsky, editor, *FOCS*, pages 150–159. IEEE, 2011. URL: <http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6108120>, doi:10.1109/FOCS.2011.23.
- 12 Frank K. H. A. Dehne, Michael R. Fellows, Michael A. Langston, Frances A. Rosamond, and Kim Stevens. An $O(2^{O(k)}n^3)$ FPT algorithm for the undirected feedback vertex set problem. *Theory Comput. Syst.*, 41(3):479–492, 2007. doi:10.1007/s00224-007-1345-z.

- 13 Holger Dell, Thore Husfeldt, Bart M. P. Jansen, Petteri Kaski, Christian Komusiewicz, and Frances A. Rosamond. The First Parameterized Algorithms and Computational Experiments Challenge. In Jiong Guo and Danny Hermelin, editors, *11th International Symposium on Parameterized and Exact Computation, IPEC 2016, August 24-26, 2016, Aarhus, Denmark*, volume 63 of *LIPICs*, pages 30:1–30:9. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPICs.IPEC.2016.30.
- 14 Holger Dell, Christian Komusiewicz, Nimrod Talmon, and Mathias Weller. The PACE 2017 Parameterized Algorithms and Computational Experiments Challenge: The Second Iteration. In Daniel Lokshtanov and Naomi Nishimura, editors, *12th International Symposium on Parameterized and Exact Computation (IPEC 2017)*, Leibniz International Proceedings in Informatics (LIPIcs), pages 30:1–30:12, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPICs.IPEC.2017.30.
- 15 Rodney G. Downey and Michael R. Fellows. Fixed parameter tractability and completeness. In *Complexity Theory: Current Research*, pages 191–225, 1992.
- 16 Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity*. Springer, 1999.
- 17 Harold N. Gabow and Matthias F. M. Stallmann. An augmenting path algorithm for linear matroid parity. *Combinatorica*, 6(2):123–150, 1986. doi:10.1007/BF02579169.
- 18 Jiong Guo, Jens Gramm, Falk Hüffner, Rolf Niedermeier, and Sebastian Wernicke. Compression-based fixed-parameter algorithms for feedback vertex set and edge bipartization. *J. Comput. Syst. Sci.*, 72(8):1386–1396, 2006.
- 19 Kensuke Imanishi and Yoichi Iwata. Feedback Vertex Set solver, entry to PACE 2016, 2016. <http://github.com/wata-orz/fvs>.
- 20 Yoichi Iwata. Linear-time kernelization for feedback vertex set. In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl, editors, *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, volume 80 of *LIPICs*, pages 68:1–68:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPICs.ICALP.2017.68.
- 21 Yoichi Iwata, Magnus Wahlström, and Yuichi Yoshida. Half-integrality, LP-branching, and FPT algorithms. *SIAM J. Comput.*, 45(4):1377–1411, 2016. doi:10.1137/140962838.
- 22 Yoichi Iwata, Yutaro Yamaguchi, and Yuichi Yoshida. Linear-time FPT algorithms via half-integral non-returning a-path packing. *CoRR*, abs/1704.02700, 2017. arXiv:1704.02700.
- 23 Iyad A. Kanj, Michael J. Pelsmayer, and Marcus Schaefer. Parameterized algorithms for feedback vertex set. In Rodney G. Downey, Michael R. Fellows, and Frank K. H. A. Dehne, editors, *IWPEC*, volume 3162 of *Lecture Notes in Computer Science*, pages 235–247. Springer, 2004. doi:10.1007/978-3-540-28639-4_21.
- 24 Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972. URL: <http://www.cs.berkeley.edu/~simluca/cs172/karp.pdf>.
- 25 Krzysztof Kiljan and Marcin Pilipczuk. Experimental evaluation of parameterized algorithms for Feedback Vertex Set: code repository, 2018. https://bitbucket.org/marcin_pilipczuk/fvs-experiments.
- 26 Tomasz Kociumaka and Marcin Pilipczuk. Faster deterministic feedback vertex set. *Inf. Process. Lett.*, 114(10):556–560, 2014. doi:10.1016/j.ipl.2014.05.001.
- 27 Christian Komusiewicz. PACE 2016: tests for Feedback Vertex Set, 2016. <https://github.com/ckomus/PACE-fvs>.
- 28 Marcin Pilipczuk. Feedback Vertex Set solver, entry to PACE 2016, 2016. http://bitbucket.com/marcin_pilipczuk/fvs-pace-challenge.

12:12 Experimental Evaluation of Parameterized Algorithms for FVS

- 29 Venkatesh Raman, Saket Saurabh, and C. R. Subramanian. Faster fixed parameter tractable algorithms for finding feedback vertex sets. *ACM Transactions on Algorithms*, 2(3):403–415, 2006.
- 30 Stéphan Thomassé. A $4k^2$ kernel for feedback vertex set. *ACM Transactions on Algorithms*, 6(2):32:1–32:8, 2010. doi:10.1145/1721837.1721848.

An Efficient Local Search for the Minimum Independent Dominating Set Problem

Kazuya Haraguchi

Otaru University of Commerce, Midori 3-5-21/Otaru, Hokkaido, Japan

haraguchi@res.otaru-uc.ac.jp

Abstract

In the present paper, we propose an efficient local search for the minimum independent dominating set problem. We consider a local search that uses k -swap as the neighborhood operation. Given a feasible solution S , it is the operation of obtaining another feasible solution by dropping exactly k vertices from S and then by adding any number of vertices to it. We show that, when $k = 2$, (resp., $k = 3$ and a given solution is minimal with respect to 2-swap), we can find an improved solution in the neighborhood or conclude that no such solution exists in $O(n\Delta)$ (resp., $O(n\Delta^3)$) time, where n denotes the number of vertices and Δ denotes the maximum degree. We develop a metaheuristic algorithm that repeats the proposed local search and the plateau search iteratively, where the plateau search examines solutions of the same size as the current solution that are obtainable by exchanging a solution vertex and a non-solution vertex. The algorithm is so effective that, among 80 DIMACS graphs, it updates the best-known solution size for five graphs and performs as well as existing methods for the remaining graphs.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms

Keywords and phrases Minimum independent dominating set problem, local search, plateau search, metaheuristics

Digital Object Identifier 10.4230/LIPIcs.SEA.2018.13

Related Version A full version of the paper is available at <https://arxiv.org/abs/1802.06478>.

Supplement Material The source code of the proposed algorithm is written in C++ and available at <http://puzzle.haraguchi-s.otaru-uc.ac.jp/minids/>.

1 Introduction

Let $G = (V, E)$ be a graph such that V is the vertex set and E is the edge set. Let $n = |V|$ and $m = |E|$. A vertex subset S ($S \subseteq V$) is *independent* if no two vertices in S are adjacent, and *dominating* if every vertex in $V \setminus S$ is adjacent to at least one vertex in S . Given a graph, the *minimum independent dominating set* (*MinIDS*) problem asks for a smallest vertex subset that is dominating as well as independent. The MinIDS problem has many practical applications in data communication and networks [13].

There is much literature on the MinIDS problem in the field of discrete mathematics [8]. The problem is NP-hard [6] and also hard even to approximate; there is no constant $\varepsilon > 0$ such that the problem can be approximated within a factor of $n^{1-\varepsilon}$ in polynomial time, unless $P=NP$ [11].

For algorithmic perspective, Liu and Song [15] and Bourgeois et al. [4] proposed exact algorithms with polynomial space. The running times of Liu and Song's algorithms are bounded by $O^*(2^{0.465n})$ and $O^*(2^{0.620n})$, and the running time of Bourgeois et al.'s algorithm is bounded by $O^*(2^{0.417n})$, where $O^*(\cdot)$ is introduced to ignore polynomial factors. Laforest



© Kazuya Haraguchi;

licensed under Creative Commons License CC-BY

17th International Symposium on Experimental Algorithms (SEA 2018).

Editor: Gianlorenzo D'Angelo; Article No. 13; pp. 13:1–13:13

Leibniz International Proceedings in Informatics



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

and Phan [14] proposed an exact algorithm based on clique partition, and made empirical comparison with one of the Liu and Song’s algorithms, in terms of the computation time. Davidson et al. [5] proposed an integer linear optimization model for the weighted version of the MinIDS problem (i.e., weights are given to edges as well as vertices, and the weight of an edge vx is counted as cost if the edge vx is used to assign a non-solution vertex v to a solution vertex x ; every non-solution vertex v is automatically assigned to an adjacent solution vertex x such that the weight of vx is the minimum) and performed experimental validation for random graphs. Recently, Wang et al. [20] proposed a tabu-search based memetic algorithm and Wang et al. [21] proposed a metaheuristic algorithm based on GRASP (greedy randomized adaptive search procedure). They showed their effectiveness on DIMACS instances, in comparison with CPLEX12.6 and LocalSolver5.5.

A vertex subset S is an IDS iff it is a maximal independent set with respect to set-inclusion [2]. Then one can readily see that the MinIDS problem is equivalent to the maximum minimal vertex cover (MMVC) problem and the minimum maximal clique problem. Zehavi [22] studied the MMVC problem, which has applications to wireless ad hoc networks, from the viewpoint of fixed-parameter-tractability.

For a combinatorially hard problem like the MinIDS problem, it is practically meaningful to develop a heuristic algorithm to obtain a nearly-optimal solution in reasonable time. In the present paper, we propose an efficient local search for the MinIDS problem. By the term “efficient”, we mean that the proposed local search has a better time bound than one naïvely implemented. The local search can serve as a key tool of local improvement in a metaheuristic algorithm, or can be used in an initial solution generator for an exact algorithm. We may also expect that it is extended to the weighted version of the MinIDS problem in the future work.

Our strategy is to search for a smallest maximal independent set. Hereafter, we may call a maximal independent set simply a *solution*. In the proposed local search, we use k -swap for the neighborhood operation. Given a solution S , k -swap refers to the operation of obtaining another solution by dropping exactly k vertices from S and then by adding any number of vertices to it. The k -neighborhood of S is the set of all solutions that can be obtained by performing k -swap on S . We call S k -minimal if its k -neighborhood contains no S' such that $|S'| < |S|$.

To speed up the local search, one should search the neighborhood for an improved solution as efficiently as possible. For this, we propose k -neighborhood search algorithms for $k = 2$ and 3. When $k = 2$ (resp., $k = 3$ and a given solution is 2-minimal), the algorithm finds an improved solution or decides that no such solution exists in $O(n\Delta)$ (resp., $O(n\Delta^3)$) time, where Δ denotes the maximum degree in the input graph.

Furthermore, we develop a metaheuristic algorithm named *ILPS (Iterated Local & Plateau Search)* that repeats the proposed local search and the plateau search iteratively. ILPS is so effective that, among 80 DIMACS graphs, it updates the best-known solution size for five graphs and performs as well as existing methods for the remaining graphs.

The paper is organized as follows. Making preparations in Section 2, we present k -neighborhood search algorithms for $k = 2$ and 3 in Section 3 and describe ILPS in Section 4. We show computational results in Section 5 and then give concluding remark in Section 6.

2 Preliminaries

2.1 Notation and Terminologies

For a vertex $v \in V$, we denote by $\deg(v)$ the degree of v , and by $N(v)$ the set of neighbors of v , i.e., $N(v) = \{u \mid vu \in E\}$. For $S \subseteq V$, we define $N(S) = (\bigcup_{v \in S} N(v)) \setminus S$. We denote by $G[S]$ the subgraph induced by S . The S is called a k -subset if $|S| = k$.

Suppose that S is an independent set. The *tightness* of $v \notin S$ is the number of neighbors of v that belong to S , i.e., $|N(v) \cap S|$. We call the v t -tight if its tightness is t . In particular, a 0-tight vertex is called *free*. We denote by T_t the set of t -tight vertices. Then V is partitioned into $V = S \cup T_0 \cup \dots \cup T_{n-1}$, where T_t may be empty. Let $T_{\geq t}$ denote the set of vertices that have the tightness no less than t , that is, $T_{\geq t} = T_t \cup T_{t+1} \cup \dots \cup T_{n-1}$.

An independent set S is a solution (i.e., a maximal independent set) iff $T_0 = \emptyset$. We call $x \in S$ a *solution vertex* and $v \notin S$ a *non-solution vertex*. When a solution vertex $x \in S$ and a t -tight vertex $v \notin S$ are adjacent, x is a *solution neighbor* of v , or equivalently, v is a *t-tight neighbor* of x .

A k -swap on a solution S is the operation of obtaining another solution $(S \setminus D) \cup A$ such that D is a k -subset of S and that A is a non-empty subset of $V \setminus S$. We call D a *dropped subset* and A an *added subset*. The k -neighborhood of S is the set of all solutions obtained by performing a k -swap on S . A solution S is k -minimal if the k -neighborhood contains no improved solution S' such that $|S'| < |S|$. Note that every solution is 1-minimal.

If a k -subset D is dropped from S , then trivially, the k solution vertices in D become free, and some non-solution vertices may also become free. Observe that a non-solution vertex becomes free if the solution neighbors are completely contained in D . We denote by $F(D)$ the set of such vertices and it is defined as $F(D) = \{v \in V \setminus S \mid N(v) \cap S \subseteq D\}$. Clearly the added subset A should be a maximal independent set in $G[D \cup F(D)]$. We have $F(D) \subseteq N(D)$, and the tightness of any vertex in $F(D)$ is at most k (at the time before dropping D from S).

2.2 Data Structure

We store the input graph by means of the typical adjacency list. We maintain a solution based on the data structure that Andrade et al. [1] invented for the maximum independent set problem. For the current solution S , we have an ordering $\pi : V \rightarrow \{1, \dots, n\}$ on all vertices in V such that;

- $\pi(x) < \pi(v)$ whenever $x \in S$ and $v \notin S$;
- $\pi(v) < \pi(v')$ whenever $v \in T_0$ and $v' \in T_{\geq 1}$;
- $\pi(v') < \pi(v'')$ whenever $v' \in T_1$ and $v'' \in T_{\geq 2}$;
- $\pi(v'') < \pi(v''')$ whenever $v'' \in T_2$ and $v''' \in T_{\geq 3}$.

Note that the ordering is partitioned into five sections; S , T_0 , T_1 , T_2 and $T_{\geq 3}$. In each section, the vertices are arranged arbitrarily. We also maintain the number of vertices in each section and the tightness $\tau(v)$ for every non-solution vertex $v \notin S$.

Let us describe the time complexities of some elementary operations. We can scan each vertex section in linear time. We can pick up a free vertex (if exists) in $O(1)$ time. We can drop (resp., add) a vertex v from (resp., to) the solution in $O(\deg(v))$ time. See [1] for details.

Before closing this preparatory section, we mention the time complexities of two essential operations.

► **Proposition 1.** *Let D be a k -subset of S . We can list all vertices in $F(D)$ in $O(k\Delta)$ time.*

Proof. We let every $v \in V$ have an integral counter, which we denote by $c(v)$. It suffices to scan vertices in $N(D)$ twice. In the first scan, we initialize the counter value as $c(u) \leftarrow 0$ for every neighbor $u \in N(x)$ of every solution vertex $x \in D$. In the second, we increase the counter of u by one (i.e., $c(u) \leftarrow c(u) + 1$) when u is searched in the adjacency list of $x \in D$. Then, if $c(u) = \tau(u)$ holds, we output u as a member of $F(D)$ since the equality represents that every solution neighbor of u is contained in D . Obviously the time bound is $O(k\Delta)$. ◀

► **Proposition 2.** *Let D be a k -subset of S . For any non-solution vertex $v \in F(D)$, we can decide whether v is adjacent to all vertices in $F(D) \setminus \{v\}$ in $O(k\Delta)$ time.*

Proof. We use the algorithm of Proposition 1. As preprocessing of the algorithm, we set the counter $c(u)$ of each $u \in N(v)$ to 0, i.e., $c(u) \leftarrow 0$, which can be done in $O(\deg(v))$ time. After we acquire $F(D)$ by running the algorithm of Proposition 1, we can see if v is adjacent to all other vertices in $F(D)$ in $O(\deg(v))$ time by counting the number of vertices $u \in N(v)$ such that $\tau(u) \in \{1, \dots, k\}$ and $c(u) = \tau(u)$. If the number equals to (resp., does not equal to) $|F(D)| - 1$, then we can conclude that it is true (resp., false). ◀

3 Local Search

Assume that, for some $k \geq 2$, a given solution S is k' -minimal for every $k' \in \{1, \dots, k-1\}$. Such k always exists, e.g., $k = 2$. In this section, we consider how we find an improved solution in the k -neighborhood of S or conclude that S is k -minimal efficiently.

Let us describe how time-consuming naïve implementation is. In naïve implementation, we search all k -subsets of S as candidates of the dropped subset D , where the number of them is $O(n^k)$. Furthermore, for each D , there are $O(n^{k-1})$ candidates of the added subset A . The number of possible pairs (D, A) is up to $O(n^{2k-1})$.

In the proposed neighborhood search algorithm, we do not search dropped subsets but added subsets; we generate a dropped subset from each added subset. When $k \in \{2, 3\}$, the added subsets can be searched more efficiently than the dropped subsets. This search strategy stems from Proposition 3, a necessary condition of a k -subset D that the improvement is possible by a k -swap that drops D . We introduce the condition in Section 3.1.

Then in Section 3.2 (resp., 3.3), we present a k -neighborhood search algorithm that finds an improved solution or decides that no such solution exists for $k = 2$ (resp., 3), which runs in $O(n\Delta)$ (resp., $O(n\Delta^3)$) time.

3.1 A Necessary Condition for Improvement

Let D be a k -subset of S . If there is a subset $A \subseteq F(D)$ such that A is maximal independent in $G[D \cup F(D)]$ and $|A| < |D|$, then we have an improved solution $(S \setminus D) \cup A$. The connectivity of $G[D \cup F(D)]$ is necessary for the existence of such A , as stated in the following proposition.

► **Proposition 3.** *Suppose that a solution S is k' -minimal for every $k' \in \{1, \dots, k-1\}$ for some integer $k \geq 2$. Let D be a k -subset of S . There is a maximal independent set A in $G[D \cup F(D)]$ such that $A \subseteq F(D)$ and $|A| < |D|$ only when the subgraph is connected.*

Proof. Suppose that $G[D \cup F(D)]$ is not connected. Let q be the number of connected components and $D^{(p)} \cup F^{(p)}(D)$ be the subset of vertices in the p -th component ($q \geq 2$, $p = 1, \dots, q$, $D^{(p)} \subseteq D$, $F^{(p)}(D) \subseteq F(D)$). Each $D^{(p)}$ is not empty since otherwise there would be an isolated vertex in $F^{(p)}(D)$. It is a free vertex with respect to S , which contradicts that S is a solution. Then we have $1 \leq |D^{(p)}| < k$.

The maximal independent set A is a subset of $F(D)$. We partition A into $A = A^{(1)} \cup \dots \cup A^{(q)}$, where $A^{(p)} = A \cap F^{(p)}(D)$. Each $A^{(p)}$ is maximal independent for the p -th component. As $|A| < |D|$, $|A^{(p)}| < |D^{(p)}|$ holds for some p . Then we can construct an improved solution $(S \setminus D^{(p)}) \cup A^{(p)}$, which contradicts the k' -minimality of S . ◀

3.2 2-Neighborhood Search

Applying Proposition 3 to the case of $k = 2$, we have the following proposition.

▶ **Proposition 4.** *Let D be a 2-subset of S . There is a non-solution vertex v in $F(D)$ such that $(S \setminus D) \cup \{v\}$ is a solution only when there is a 2-tight vertex in $F(D)$.*

We can say more on Proposition 4. The vertex v should be 2-tight since, if not so (i.e., v is 1-tight), $\{v\}$ would not be maximal independent for $G[D \cup F(D)]$; v is adjacent to only one of $D = \{x, y\}$ from the definition of 1-tightness.

In summary, if there is an improved solution $(S \setminus D) \cup \{v\}$, then v is 2-tight and has x and y as the solution neighbors. Instead of searching all 2-subsets of S , we scan all 2-tight vertices, and for each 2-tight vertex v , we take $D = \{x, y\}$ as the candidate of the dropped set. We have the following theorem.

▶ **Theorem 5.** *Given a solution S , we can find an improved solution in the 2-neighborhood or conclude that S is 2-maximal in $O(n\Delta)$ time.*

Proof. Since we maintain the solution by means of the vertex ordering, we can scan all the 2-tight vertices in $O(|T_2|)$ time. For each 2-tight v , we can detect the two solution neighbors, say x and y , in $O(\deg(v))$ time.

Let $D = \{x, y\}$. The singleton $\{v\}$ is maximal independent for $G[D \cup F(D)]$ and thus we have an improved solution $(S \setminus D) \cup \{v\}$ iff v is adjacent to all other vertices in $F(D)$. Whether v is adjacent to all other vertices in $F(D)$ is decided in $O(\Delta)$ time, as we stated in Proposition 2. If it is the case, then we can construct an improved solution $(S \setminus D) \cup \{v\}$ in $O(\deg(x) + \deg(y) + \deg(v)) = O(\Delta)$ time as the vertex ordering takes $O(\deg(x))$ time to drop x from S and $O(\deg(v))$ time to add v to it [1]. Otherwise, we can conclude that $(S \setminus D) \cup \{v\}$ is not a solution because some vertices in $F(D)$ are not dominated.

We have seen that, for each 2-tight vertex v , it takes $O(\Delta)$ time to find an improved solution $(S \setminus D) \cup \{v\}$ or to conclude that it is not a solution. Therefore, the overall running time is bounded by $O(|T_2|\Delta) = O(n\Delta)$. ◀

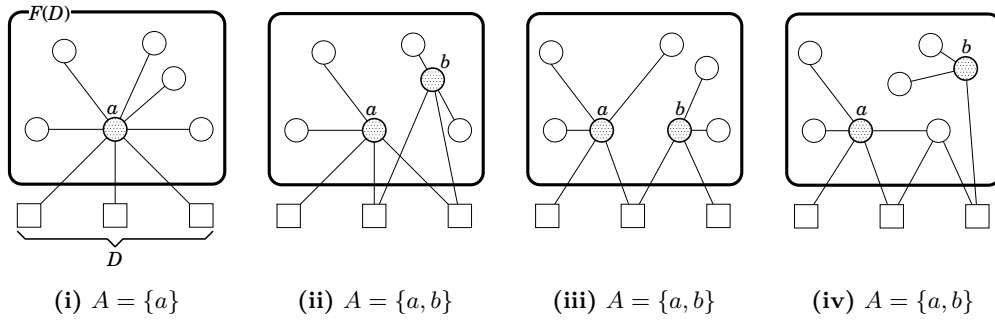
3.3 3-Neighborhood Search

We have the following proposition by applying Proposition 3 to the case of $k = 3$.

▶ **Proposition 6.** *Suppose that S is a 2-minimal solution and that $D = \{x, y, z\}$ is a 3-subset of S . There is a subset A of $F(D)$ such that A is maximal independent in $G[D \cup F(D)]$ and $|A| < |D|$ only when either of the followings holds:*

- (a) *there is a 3-tight vertex in $F(D)$ that has x , y and z as the solution neighbors;*
- (b) *there are two 2-tight vertices in $F(D)$ such that one has x and y as the solution neighbors and the other has x and z as the solution neighbors.*

Let us make observation on the added subset. Suppose that, for an arbitrary 3-subset $D \subseteq S$, there is $A \subseteq F(D)$ such that A is maximal independent in $G[D \cup F(D)]$ and $|A| < |D|$. When $|A| = 1$, the only vertex in A is 3-tight since otherwise some vertex in D would not be dominated. When $|A| = 2$, at least one of the two vertices in A is either 2-tight or 3-tight; if



■ **Figure 1** Illustration of a dropped set D and an added set A for (i) to (iv) in Section 3.3: For clarity of the figure, we draw only edges that are incident to the vertices a and b . Note that every vertex in $F(D)$ is adjacent to at least one vertex in D .

both are 1-tight, one vertex of D would not be dominated. Concerning the tightness, the following four situations are possible:

- (i) $A = \{a\}$ and a is 3-tight;
- (ii) $A = \{a, b\}$, a is 3-tight, and b is t -tight such that $t \in \{1, 2, 3\}$;
- (iii) $A = \{a, b\}$, a is 2-tight, and b is 2-tight;
- (iv) $A = \{a, b\}$, a is 2-tight, and b is 1-tight.

From (ii) to (iv), the vertices a and b are not adjacent. We illustrate (i) to (iv) in Figure 1.

Based on the above, we summarize the search strategy as follows. In order to generate all 3-subsets D of S such that $F(D)$ satisfies either (a) or (b) of Proposition 6, we scan all 3-tight vertices u (Proposition 6 (a)) and all pairs of 2-tight vertices, say v and w , such that $|(N(v) \cup N(w)) \cap S| = 3$ (Proposition 6 (b)). For (a), we take $D = N(u) \cap S$ and search $F(D)$ for a 1- or 2-subset A that is maximal independent in $G[D \cup F(D)]$, regarding the 3-tight vertex u as the vertex a in (i) and (ii). Similarly, for (b), we take $D = (N(v) \cup N(w)) \cap S$ and search $F(D)$ for a 2-subset A that is maximal independent in $G[D \cup F(D)]$, regarding the 2-tight vertex v as the vertex a in (iii) and (iv).

We have the following theorem on the time complexity of 3-neighborhood search. We omit the proof due to space limitation.

► **Theorem 7.** *Given a 2-minimal solution S , we can find an improved solution in the 3-neighborhood or conclude that S is 3-maximal in $O(n\Delta^3)$ time.*

4 Iterated Local & Plateau Search

In this section, we present a metaheuristic algorithm named ILPS (Iterated Local & Plateau Search) that repeats the proposed local search and the plateau search iteratively.

We show the pseudo code of ILPS in Algorithm 1. The ILPS has four parameters, that is S , k , δ and ν , where S is an initial solution, k is the order of the local search (i.e., a k -minimal solution is searched by LOCALSEARCH(S, k) in Line 6), and δ and ν are integers. The roles of the last two parameters are mentioned in Section 4.2.

The LOCALSEARCH(S, k) in Line 6 is the subroutine that returns a k -minimal solution from an initial solution S , where k is set to either two or three. When $k = 2$, it determines a 2-minimal solution by moving to an improved solution repeatedly as long as the 2-neighborhood search algorithm delivers one. When $k = 3$, it first finds a 2-minimal solution, and then runs the 3-neighborhood search algorithm. If an improved solution is delivered, then the local search moves to the improved solution and seeks a 2-minimal one again since the solution is not necessarily 2-minimal. Otherwise, the current solution is 3-minimal.

Algorithm 1 Iterated Local & Plateau Search (ILPS).

```

1: function ILPS( $S, k, \delta, \nu$ )
2:    $S^* \leftarrow S$  ▷  $S^*$  is used to store the incumbent solution
3:    $\rho \leftarrow$  a penalty function such that  $\rho(v) = 0$  for all  $v \in V$ 
4:    $\rho \leftarrow$  UPDATEPENALTY( $S, \rho, \delta$ )
5:   while termination condition is not satisfied do
6:      $S \leftarrow$  LOCALSEARCH( $S, k$ ) ▷ The local search returns a  $k$ -minimal solution
7:      $S \leftarrow$  PLATEAUSEARCH( $S, k$ ) ▷ The plateau search returns a  $k$ -minimal solution
8:     if  $|S| \leq |S^*|$  then
9:        $S^* \leftarrow S$ 
10:    end if
11:     $S \leftarrow$  KICK( $S^*, \rho, \nu$ ) ▷ The initial solution of the next iteration is generated
12:     $\rho \leftarrow$  UPDATEPENALTY( $S, \rho, \delta$ ) ▷ The penalty function is updated
13:  end while
14:  return  $S^*$ 
15: end function

```

Below we explain two key ingredients: the plateau search and the vertex penalty. We describe these in Sections 4.1 and 4.2 respectively. We remark that they are inspired by *Dynamic Local Search* for the maximum clique problem [19] and *Phased Local Search* for the unweighted/weighted maximum independent set and minimum vertex cover [18].

4.1 Plateau Search

In the plateau search (referred to as PLATEAUSEARCH(S, k) in Line 7), we search solutions of the size $|S|$ that can be obtained by swapping a solution vertex $x \in S$ and a non-solution vertex $v \notin S$. Let $\mathcal{P}(S)$ be the collection of all solutions that are obtainable in this way. The size of any solution in $\mathcal{P}(S)$ is $|S|$. We execute LOCALSEARCH(S', k) for every solution $S' \in \mathcal{P}(S)$, and if we find an improved solution S'' such that $|S''| < |S'| = |S|$, then we do the same for S'' , i.e., we execute LOCALSEARCH(P, k) for every solution $P \in \mathcal{P}(S'')$. We repeat this until no improved solution is found and employ a best solution among those searched as the output of the plateau search.

We emphasize the efficiency of the plateau search; all solutions in $\mathcal{P}(S)$ can be listed in $O(|T_1|\Delta)$ time. Observe that $(S \setminus \{x\}) \cup \{v\}$ is a solution iff v is 1-tight such that x is the only solution neighbor of v , and v is adjacent to all vertices in $F(\{x\})$ other than v . We can scan all 1-tight vertices in $O(|T_1|)$ time. For each 1-tight vertex v , the solution neighbor x is detected in $O(\deg(v))$ time, and whether the last condition is satisfied or not is identified in $O(\Delta)$ time from Proposition 2. Dropping x from S and adding v to $S \setminus \{x\}$ can be done in $O(\Delta)$ time.

4.2 Vertex Penalty

In order to avoid the search stagnation, one possible approach is to apply a variety of initial solutions. To realize this, we introduce a penalty function $\rho : V \rightarrow \mathbb{Z}^+ \cup \{0\}$ on the vertices. The penalty function ρ is initialized so that $\rho(v) = 0$ for all $v \in V$ (Line 3). During the algorithm, ρ is managed by the subroutine UPDATEPENALTY (Lines 4 and 12). When the initial solution S of the next local search is determined, it increases the penalty $\rho(v)$ of every vertex $v \in S$ by one, i.e., $\rho(v) \leftarrow \rho(v) + 1$. Furthermore, to “forget” the search history long ago, it reduces $\rho(v)$ to $\lfloor \min\{\rho(v), \delta\}/2 \rfloor$ for all $v \in V$ in every δ iterations. This δ is the third parameter of ILPS and called the *penalty delay*.

The ρ is used in the subroutine KICK (Line 11), the initial solution generator, so that vertices with fewer penalties are more likely to be included in the initial solution. KICK generates an initial solution by adding non-solution vertices (with respect to the incumbent solution S^*) “forcibly” to S^* . The added vertices are chosen one by one as follows; in one trial, KICK picks up one non-solution vertex. It then goes on to the next trial with the probability $(\nu - 1)/\nu$ or stops the selection with the probability $1/\nu$, where ν is the fourth parameter of ILPS. Observe that ν specifies the expected number of added vertices. In the first trial, KICK randomly picks up a non-solution vertex that has the fewest penalty. In a subsequent r -th trial ($r = 2, 3, \dots$), let $R = \{v_1, \dots, v_{r-1}\}$ be the set of vertices chosen so far. KICK samples three vertices randomly from $V \setminus (S^* \cup R \cup N(R))$, and picks up the one that has the fewest penalty among the three. Suppose that $R = \{v_1, \dots, v_r\}$ has been picked up as the result of r trials. Then we construct an independent set $S = (S^* \setminus N(R)) \cup R$. The S may not be a solution as there may remain free vertices. If so, we repeatedly pick up free vertices by the maximum-degree greedy method until S becomes a solution. We use the acquired S as the initial solution of the next local search.

5 Computational Results

We report some experimental results in this section. In Section 5.1, to gain insights into what kind of instance is difficult, we examine the phase transition of difficulty with respect to the edge density. The next two subsections are devoted to observation on the behavior of the proposed method. In Section 5.2, we show how a single run of LOCALSEARCH(S, k) improves a given initial solution. In Section 5.3, we show how the penalty delay δ affects the search. Finally in Section 5.4, we compare ILPS with the memetic algorithm [20], GRASP+PC [21], CPLEX12.6 [12] and LocalSolver5.5 [16] in terms of the solution size, using DIMACS graphs.

All the experiments are conducted on a workstation that carries an Intel Core i7-4770 Processor (up to 3.90GHz by means of Turbo Boost Technology) and 8GB main memory. The installed OS is Ubuntu 16.04. Under this environment, it takes 0.25s, 1.54s and 5.90s approximately to execute `dmcliq` (<http://dimacs.rutgers.edu/pub/dsj/cliq/>) for instances `r300.5.b`, `r400.5.b` and `r500.5.b`, respectively. The ILPS algorithm is implemented in C++ and compiled by the g++ compiler (ver. 5.4.0) with `-O2` option.

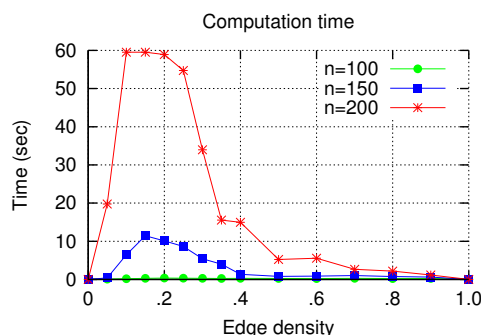
5.1 Phase Transition of Difficulty

The phase transition has been observed for many combinatorial problems [7, 9, 10]. Roughly, it is said that over-constrained and under-constrained instances are relatively easy, and that intermediately constrained ones tend to be more difficult.

In the MinIDS problem, the amount of constraints is proportional to the edge density p . We examine the change of difficulty with respect to p . We estimate the difficulty of an instance by how long CPLEX12.8 takes to solve it.

For each $(n, p) \in \{100, 150, 200\} \times \{0.00, 0.05, \dots, 1.00\}$, we generate 100 random graphs (Erdős-Rényi model) with n vertices and the edge density p , i.e., an edge is drawn between two vertices with probability p . We solve the 100 instances by CPLEX12.8 and take the averaged computation time. We set the time limit of each run to 60s. If CPLEX12.8 terminates by the time limit, then we regard the computation time as 60s.

Figure 2 shows the result. We may say that instances with the edge densities from 0.1 to 0.4 are likely to be more difficult than others. In fact, the experiments in [5, 14] mainly deal with random graphs with the edge densities in this range.



■ **Figure 2** Computation time of CPLEX12.8 for random graphs.

■ **Table 1** Averaged sizes of random, 2-minimal and 3-minimal solutions in random graphs with 10^3 vertices.

	$p = .1$.2	.3	.4	.5	.6	.7	.8	.9	.95	.99
random	44.57	24.42	16.70	12.50	9.66	7.70	6.12	4.84	3.62	3.00	2.12
2-minimal	37.37	20.36	13.84	10.18	7.86	6.12	4.95	3.95	2.99	2.00	1.95
3-minimal	35.44	19.04	12.74	9.28	7.01	5.64	4.06	3.02	2.15	2.00	1.95

5.2 A Single Run of Local Search

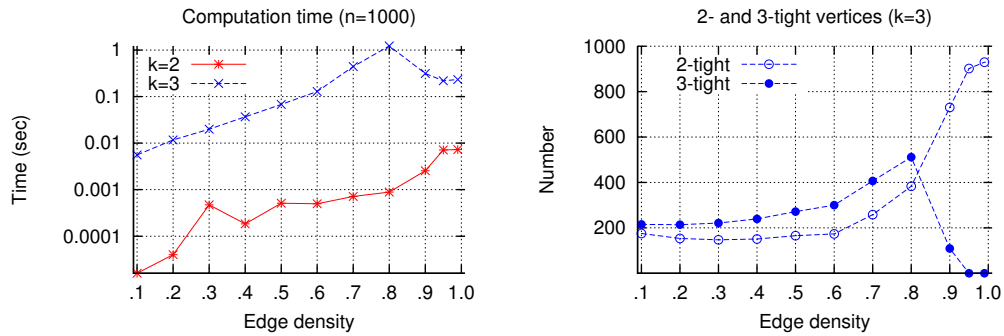
We show how a single run of $\text{LOCALSEARCH}(S, k)$ improves an initial solution S . Again we take a random graph. We fix the number n of vertices to 10^3 . For every $p \in \{0.1, \dots, 0.9, 0.95, 0.99\}$, we generate 100 random graphs. Then for each graph, we run $\text{LOCALSEARCH}(S, k)$ five times, where we use different random seeds in each time and construct the initial solution S randomly.

We show the averaged sizes of random, 2-minimal and 3-minimal solutions in Table 1. We see that, the larger the edge density p is, the fewer the solution size becomes. The local search improves a random solution to some extent. $\text{LOCALSEARCH}(S, 3)$ improves the solution more than $\text{LOCALSEARCH}(S, 2)$. The difference between the two local searches is the largest when $p = 0.1$, that is $37.37 - 35.44 = 1.93$. The difference gets smaller when p gets larger. In particular, when $p > 0.9$, we see no difference.

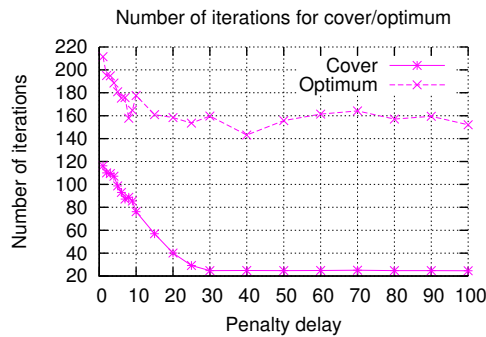
Let us discuss computation time. In the left of Figure 3, we show how the averaged computation time changes with respect to p . We see that the computation time of $\text{LOCALSEARCH}(S, 3)$ is tens to thousands of times the computation time of $\text{LOCALSEARCH}(S, 2)$. However, it does not necessarily diminish the value of the 3-neighborhood search. As will be shown in Section 5.4, when $k = 3$, ILPS can find such a good solution that is not obtained by $k = 2$.

In general, for a fixed k , it takes more computation time when p is larger. Recall Theorem 5 (resp., 7); when $k = 2$ (resp., 3), the k -neighborhood search algorithm finds an improved solution for the current solution S or concludes that S is k -minimal in $O(n\Delta)$ (resp., $O(n\Delta^3)$) time. Roughly, Δ is increasing as p gets larger.

For $k = 3$, we attribute the peak at $p = 0.8$ to the number of 3-tight vertices. In the right of Figure 3, We show the averaged numbers of 2- and 3-tight vertices with respect to 3-minimal solutions. The 3-neighborhood search algorithm searches 2- and 3-tight vertices. The numbers of both vertices are generally non-decreasing from $p = 0.1$ to 0.8, but when $p > 0.8$, the number of 3-tight vertices decreases dramatically. This is due to the solution



■ **Figure 3** (Left) averaged computation time that $\text{LOCALSEARCH}(S, k)$ takes to decide a k -minimal solution (Right) numbers of 2- and 3-tight vertices with respect to 3-minimal solutions.



■ **Figure 4** Averaged numbers of iterations to cover all vertices (solid line) and to find the optimum (dashed line).

size. The solution size gives an upper bound on the tightness of any non-solution vertex, and when $p > 0.8$, the averaged size of a 3-minimal solution is less than three; see Table 1. Since most of the non-solution vertices are either 1- or 2-tight, we hardly handle the situations (i) and (ii) in Section 3.3.

5.3 Penalty Delay

We introduced the notion of vertex penalty to control the search diversification. When the penalty delay δ is larger, more varieties of initial solutions are expected to be tested in ILPS.

To illustrate the expectation, we evaluate how many iterations ILPS takes until all vertices are covered by the initial solutions, that is, used in the initial solutions at least once. The solid line in Figure 4 shows the number of iterations taken to cover all vertices. The graph we employ here is a 10×10 grid graph such that each vertex is associated with a 2D integral point $(i, j) \in \{1, \dots, 10\}^2$, and that two vertices (i, j) and (i', j') are adjacent iff $|i - i'| + |j - j'| = 1$. For each δ , the number of iterations is averaged over 500 runs of ILPS with different random seeds, where we fix $(k, \nu) = (2, 1)$ and construct the first initial solution S by the maximum-degree greedy algorithm.

The observed phenomenon meets our expectation; The number is non-increasing with respect to δ and saturated for $\delta \geq 30$. In other words, when δ is larger, more varieties of initial solutions are generated in a given number of iterations.

However, setting δ to a large value does not necessarily lead to discovery of better solutions.

The dashed line in Figure 4 shows the averaged number of iterations that ILPS takes to find an optimal solution; we know that the optimal size is 24 since we solve the instance optimally by CPLEX. When $\delta \leq 40$, the number is approximately decreasing and takes the minimum at $\delta = 40$, but a larger δ does not make any improvement. Hence, given an instance, we need to choose an appropriate value of δ carefully.

5.4 Performance Validation

We run ILPS algorithm for 80 DIMACS instances that are downloadable from [17]. We generate the first initial solution S by the maximum-degree greedy method, and fix the parameter ν to three. For (k, δ) , all pairs in $\{2, 3\} \times \{2^0, \dots, 2^6\}$ are tested. For each instance and each (k, δ) , we run ILPS algorithm 10 times, using different random seeds. We terminate the algorithm by the time limit. The time limit is set to 200 s. When $k = 3$, we modify Algorithm 1 so that `PLATEAUSEARCH(S, k)` in Line 7 is called only when $|S| \leq |S^*| + 2$ as the plateau search is rather time-consuming.

We take four competitors from [20] and [21]. The first is MEM, a tabu-search based memetic algorithm in [20]. The second is GP, the GRASP+PC algorithm in [21]. The third is CP, which stands for CPLEX12.6 [12] that solves an integer optimization model of the MinIDS problem. The fourth is LS, which stands for LocalSolver5.5 [16], a general discrete optimization solver based on local search. MEM is run on a computer with a 2.0GHz CPU and a 4GB memory, whereas the other competitors are run on computers with a 2.3GHz CPU and an 8GB memory. The time limit of MEM and GP is set to 200 s, and that of CP and LS is set to 3600 s.

In Table 2, we show the results on selected instances. The columns “ n ” and “ p ” indicate the number of vertices and the edge density, respectively. The edge density is between 0.1 and 0.5 in all instances except hamming8-2. In our context, the instances are expected to be difficult. For ILPS, we show the results for $(k, \delta) = (2, 2^6)$ in detail, regarding this pair as the representative. The columns “Min” and “Max” indicate the minimum/maximum solution size over 10 runs, and the column “Avg” indicates the average. The column “TTB” indicates the time to best (in seconds), that is, the average of the computation time that ILPS takes to find the solution of the size “Min”. The symbol ε represents that the time is less than 0.1 s. The column “Best” indicates the minimum solution size attained over all $(k, \delta) \in \{2, 3\} \times \{2^0, \dots, 2^6\}$. The rightmost four columns indicate the solution size attained by the competitors. The symbol * before the instance name indicates that the solution size attained by CPLEX is optimal.

The table contains only results on the 13 selected instances such that the solutions sizes attained by “Best”, “MEM” and “GP” are not-all-equal, except hamming8-2. We guarantee that, for the remaining 67 instances, ILPS’s “Best” is as good as any competitor. The boldface indicates that the solution size is strictly smaller than those of the competitors. Then we update the best-known solution size in five graphs. These show the effectiveness of the proposed local search and the ILPS algorithm.

For hamming8-2, when $k = 2$, ILPS cannot find a solution of the optimal size 32 for any penalty delay $\delta \in \{2^0, \dots, 2^6\}$. However, when $k = 3$, ILPS finds an optimal solution with $\delta = 2^0, 2^1$ and 2^2 .

Before closing this section, let us report our preliminary results briefly.

- A preliminary version of ILPS happened to find a solution of the size 31 for C2000.9 and a solution of the size 15 for keller6.
- Let us consider a finer swap operation, (j, k) -swap, that obtains another j vertices by dropping exactly k vertices from the current one and then by adding exactly j vertices

■ **Table 2** Selected results from the validation experiments on DIMACS graphs.

	n	p	ILPS					[20]	[21]		
			$(k = 2, \nu = 2^6)$				Best	MEM	GP	CP	LS
			Min	Avg	Max	TTB					
brock400_2	400	.25	10	10.0	10	1.1	9	9	10	10	11
C1000.9	1000	.10	27	27.8	29	0.0	26	27	27	29	30
*C125.9	125	.10	14	14.0	14	0.1	14	14	15	14	14
C2000.9	2000	.10	32	33.6	35	12.1	32	33	33	48	36
C4000.5	4000	.50	7	7.9	8	49.7	7	8	8	-	-
C500.9	500	.10	22	22.2	23	92.3	21	22	23	23	22
gen400_p0.9_55	400	.10	20	20.1	21	39.4	20	20	21	22	22
gen400_p0.9_65	400	.10	20	20.7	21	99.0	20	20	21	21	22
*hamming8-2	256	.03	36	36.0	36	0.0	32	N/A	32	32	32
keller6	3361	.18	18	18.0	18	26.1	16	18	18	32	19
*san200_0.7_1	200	.30	6	6.1	7	85.9	6	6	7	6	7
*san200_0.9_1	200	.10	15	15.0	15	16.7	15	15	16	15	16
san400_0.7_3	400	.30	7	7.8	8	106.6	7	7	8	8	9

to it. One can prove that, given a solution S and a constant k , we can improve S by $(1, k)$ -swap or conclude that it is not possible in $O(n\Delta)$ time. We implemented $(1, k)$ -swap in a preliminary version of ILPS, but it does not yield significant improvement even when k is set to a constant larger than three.

- We tested Laforest and Phan’s exact algorithm [14], and found that the algorithm is not suitable for a task of finding a good solution quickly. The source code is available at <http://todo.lamsade.dauphine.fr/spip.php?article42>.
- BHOSLIB [3] is another well-known collection of benchmark instances. It contains 36 instances such that n is between 450 and 4000 and that p is no less than 0.82. Hence, the BHOSLIB instances are expected to be easy in our context. The ILPS with $(k, \delta) = (2, 2^6)$ finds a solution of the size three for all the instances. We also run CPLEX12.8 for 200 s, generating an initial solution by the maximum-degree greedy algorithm. CPLEX12.8 finds a solution of the size five for frb100-40, and a solution of the size three for the other instances. In addition, the solution of the size three is proved to be optimal for 15 instances whose names start with frb30, frb35 and frb40.

6 Concluding Remark

We have considered an efficient local search for the MinIDS problem. We proposed fast k -neighborhood search algorithms for $k = 2$ and 3, and developed a metaheuristic algorithm named ILPS that repeats the local search and the plateau search iteratively. ILPS is so effective that it updates the best-known solution size in five DIMACS graphs.

The proposed local search is applicable to other metaheuristics such as genetic algorithms, as a key tool of local improvement. The future work includes an extension of the local search to a weighted version of the MinIDS problem.

References

- 1 D.V. Andrade, M.G.C. Resende, and R.F. Werneck. Fast local search for the maximum independent set problem. *Journal of Heuristics*, 18:525–547, 2012.

- 2 C. Berge. *Theory of Graphs and its Applications*. Methuen, London, 1962.
- 3 BHOSLIB: Benchmarks with hidden optimum solutions for graph problems. <http://sites.nlsde.buaa.edu.cn/~kexu/benchmarks/graph-benchmarks.htm>. accessed on February 1, 2018.
- 4 N. Bourgeois, F.D. Croce, B. Escoffier, and V.Th. Paschos. Fast algorithms for MIN independent dominating set. *Discrete Applied Mathematics*, 161(4):558–572, 2013.
- 5 P.P. Davidson, C. Blum, and J. Lozano. The weighted independent domination problem: ILP model and algorithmic approaches. In *Proc. EvoCOP 2017*, pages 201–214, 2017. doi:10.1007/978-3-319-55453-2_14.
- 6 M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Company, 1979.
- 7 I.P. Gent and T. Walsh. The SAT phase transition. In *Proc. ECAI-94*, pages 105–109, 1994.
- 8 W. Goddard and M.A. Henning. Independent domination in graphs: A survey and recent results. *Discrete Mathematics*, 313:839–854, 2013.
- 9 C.P. Gomes and B. Selman. Problem structure in the presence of perturbations. In *Proc. AAAI-97*, pages 221–227, 1997.
- 10 C.P. Gomes and D.B. Shmoys. Completing quasigroups or latin squares: a structured graph coloring problem. In *Proc. Computational Symposium on Graph Coloring and Generalizations*, 2002.
- 11 M.M. Halldórsson. Approximating the minimum maximal independence number. *Information Processing Letters*, 46(4):169–172, 1993.
- 12 IBM ILOG CPLEX. <https://www.ibm.com/analytics/data-science/prescriptive-analytics/cplex-optimizer>. accessed on February 1, 2018.
- 13 F. Kuhn, T. Nieberg, T. Moscibroda, and R. Wattenhofer. Local approximation schemes for ad hoc and sensor networks. In *Proc. the 2005 Joint Workshop on Foundations of Mobile Computing*, pages 97–103, 2005.
- 14 C. Laforest and R. Phan. Solving the minimum independent domination set problem in graphs by exact algorithm and greedy heuristic. *RAIRO-Operations Research*, 47(3):199–221, 2013.
- 15 C. Liu and Y. Song. Exact algorithms for finding the minimum independent dominating set in graphs. In *Proc. ISAAC 2006*, LNCS 4288, pages 439–448, 2006.
- 16 LocalSolver. <http://www.localsolver.com/>. accessed on February 1, 2018.
- 17 F. Mascia. dimacs benchmark set. http://iridia.ulb.ac.be/~fmascia/maximum_clique/DIMACS-benchmark. accessed on February 1, 2018.
- 18 W. Pullan. Optimisation of unweighted/weighted maximum independent sets and minimum vertex covers. *Discrete Optimization*, 6(2):214–219, 2009.
- 19 W. Pullan and H.H. Hoos. Dynamic local search for the maximum clique problem. *Journal of Artificial Intelligence Research*, 25:159–185, 2006.
- 20 Y. Wang, J. Chen, H. Sun, and M. Yin. A memetic algorithm for minimum independent dominating set problem. *Neural Computing and Applications*, in press. doi:10.1007/s00521-016-2813-7.
- 21 Y. Wang, R. Li, Y. Zhou, and M. Yin. A path cost-based grasp for minimum independent dominating set problem. *Neural Computing and Applications*, 28(1):143–151, 2017. doi:10.1007/s00521-016-2324-6.
- 22 M. Zehavi. Maximum minimal vertex cover parameterized by vertex cover. *SIAM Journal on Discrete Mathematics*, 31(4):2440–2456, 2017.

Empirical Evaluation of Approximation Algorithms for Generalized Graph Coloring and Uniform Quasi-Wideness

Wojciech Nadara¹

Institute of Informatics, University of Warsaw, Poland
wn341489@students.mimuw.edu.pl

Marcin Pilipczuk²

Institute of Informatics, University of Warsaw, Poland
malcin@mimuw.edu.pl

Roman Rabinovich³

Lehrstuhl für Logic und Semantik, Technische Universität Berlin, Germany
roman.rabinovich@tu-berlin.de

Felix Reidl

Department of Computer Science, Royal Holloway University of London, UK
felix.reidl@rhul.ac.uk

Sebastian Siebertz⁴

Institute of Informatics, University of Warsaw, Poland
siebertz@mimuw.edu.pl

Abstract

The notions of *bounded expansion* and *nowhere denseness* not only offer robust and general definitions of uniform sparseness of graphs, they also describe the tractability boundary for several important algorithmic questions. In this paper we study two structural properties of these graph classes that are of particular importance in this context, namely the property of having bounded *generalized coloring numbers* and the property of being *uniformly quasi-wide*. We provide experimental evaluations of several algorithms that approximate these parameters on real-world graphs. On the theoretical side, we provide a new algorithm for uniform quasi-wideness with polynomial size guarantees in graph classes of bounded expansion and show a lower bound indicating that the guarantees of this algorithm are close to optimal in graph classes with fixed excluded minor.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis

Keywords and phrases Empirical Evaluation of Algorithms, Sparse Graph Classes, Generalized Coloring Numbers, Uniform Quasi-Wideness

¹ Supported by the “Recent trends in kernelization: theory and experimental evaluation” project, carried out within the Homing programme of the Foundation for Polish Science co-financed by the European Union under the European Regional Development Fund.

² Supported by the “Recent trends in kernelization: theory and experimental evaluation” project, carried out within the Homing programme of the Foundation for Polish Science co-financed by the European Union under the European Regional Development Fund.

³ Supported by the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (ERC Consolidator Grant DISTRUCT, grant agreement No 648527).

⁴ Supported by the National Science Centre of Poland via POLONEZ grant agreement UMO-2015/19/P/ST6/03998, which has received funding from the European Union’s Horizon 2020 research and innovation programme (Marie Skłodowska-Curie grant agreement No. 665778).



© Wojciech Nadara, Marcin Pilipczuk, Roman Rabinovich, Felix Reidl, and Sebastian Siebertz; licensed under Creative Commons License CC-BY

17th International Symposium on Experimental Algorithms (SEA 2018).

Editor: Gianlorenzo D’Angelo; Article No. 14; pp. 14:1–14:16

Leibniz International Proceedings in Informatics



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

Digital Object Identifier 10.4230/LIPIcs.SEA.2018.14

Related Version A full version of the paper is available at <https://arxiv.org/abs/1802.09801>.

Acknowledgements We thank Christoph Dittmann for providing us with his code for the `mfcs` algorithm, which we partially used for our implementation. We thank Michał Pilipeczuk for many hours of fruitful discussions.

1 Introduction

The exploitation of structural properties found in sparse graphs has a long and fruitful history in the design of efficient algorithms. Besides the long list of results on planar graphs and graphs of bounded degree (which are too numerous to fairly represent here), the celebrated structure theory of graphs with excluded minors, developed by Robertson and Seymour [57] falls into this category. It not only had an immense influence on the design of efficient algorithms (see e.g. [18, 19]) it further introduced the now widely used notion of treewidth (see e.g. [8]) and gave rise to the field of parameterized complexity: “In the beginning, all we did was graph minors” (M. Fellows, pers. comm.). As such, the impact of the theory of sparse graphs on algorithmic research cannot be overstated.

Many of the algorithmic results concerning classes excluding a minor or a topological minor are in some way based on topological arguments, depending on the structure theorems (e.g. decompositions) for the class under consideration. A complete paradigm shift was initiated by Nešetřil and Ossona de Mendez with their foundational work and introduction of the notions of *bounded expansion* [42, 43, 44] and *nowhere denseness* [46]. These graph classes extend and properly contain all the aforementioned sparse classes and many arguments based on topology can be replaced by more general, and surprisingly often much simpler, arguments based on *density*. We refer to the textbook [47] for extensive background on the theory of sparse graph classes.

The rich structural theory for bounded expansion and nowhere dense graph classes has been successfully applied to design efficient algorithms for hard computational problems on specific sparse classes of graphs, see e.g. [6, 16, 21, 22, 23, 24, 25, 28, 30, 63]. On the other hand, several results indicate that nowhere dense graph classes form a natural limit for algorithmic methods based on sparseness arguments, see e.g. [21, 23].

One core strength of the bounded expansion/nowhere dense framework is that there exists a multitude of equivalent definitions that provide complementing perspectives. Here, we study two structural properties of these classes that are of particular importance in the algorithmic context, namely the property of having bounded *generalized coloring numbers* and the property of being *uniformly quasi-wide*. The generalized coloring numbers intuitively measure reachability properties in a linear vertex ordering of a given graph. Such an ordering yields a very weak and local form of a graph decomposition which can be exploited combinatorially [24, 54] and algorithmically [6, 21, 22, 30]. Uniform quasi-wideness was originally introduced in finite model theory [15], and soon found combinatorial and algorithmic applications on nowhere dense classes [16, 24, 28, 35, 45, 52, 60].

Even though the above results render many problems tractable in theory, many of the known algorithms have worst-case running times that involve huge constant factors and combinatorial explosions with respect to the discussed parameters. The central question of our work here is to investigate how the generalized coloring numbers and uniform quasi-wideness behave on real-world graphs, an endeavor which so far has only been conducted

for a single notion of bounded expansion and on a smaller scale [20]. Controllable numbers would be a prerequisite for practical implementations of these algorithms based on such structural approaches. We provide an experimental evaluation of several algorithms that approximate these parameters on real world graphs.

On the theoretical side, we provide a new algorithm for uniform quasi-wideness with polynomial size guarantees in graph classes of bounded expansion and show a lower bound indicating that the guarantees of this algorithm are close to optimal in graph classes with fixed excluded minor.

2 Basic definitions

Graphs. All graphs in this paper are finite, undirected and simple, that is, they do not have loops or multiple edges between the same pair of vertices. For a graph G , we denote by $V(G)$ the vertex set of G and by $E(G)$ its edge set. The *distance* between a vertex v and a vertex w is the length (that is, the number of edges) of a shortest path between v and w . For a vertex v of G , we write $N^G(v)$ for the set of all neighbors of v , $N^G(v) = \{u \in V(G) \mid \{u, v\} \in E(G)\}$, and for $r \in \mathbb{N}$ we denote by $N_r^G[v]$ the *closed r -neighborhood of v* , that is, the set of vertices of G at distance at most r from v . Note that we always have $v \in N_r^G[v]$. The radius of a connected graph G is the minimum integer r such that there exists $v \in V(G)$ with the property that all vertices of G have distance at most r to v . A set A is *r -independent* if all distinct vertices of A have distance greater than r .

Bounded expansion and nowhere denseness. A *minor model* of a graph H in a graph G is a family $(I_u)_{u \in V(H)}$ of pairwise vertex-disjoint connected subgraphs of G , called *branch sets*, such that whenever uv is an edge in H , there are $u' \in V(I_u)$ and $v' \in V(I_v)$ for which $u'v'$ is an edge in G . The graph H is a *depth- r minor* of G , denoted $H \preceq_r G$, if there is a minor model $(I_u)_{u \in V(H)}$ of H in G such that each I_u has radius at most r . A class \mathcal{C} of graphs is *nowhere dense* if there is a function $t: \mathbb{N} \rightarrow \mathbb{N}$ such that for all $r \in \mathbb{N}$ it holds that $K_{t(r)} \not\preceq_r G$ for all $G \in \mathcal{C}$, where $K_{t(r)}$ denotes the clique on $t(r)$ vertices. The class \mathcal{C} has *bounded expansion* if there is a function $d: \mathbb{N} \rightarrow \mathbb{N}$ such that for all $r \in \mathbb{N}$ and all $H \preceq_r G$ with $G \in \mathcal{C}$, the *edge density* of H , i.e. $|E(H)|/|V(H)|$, is bounded by $d(r)$.

Weak coloring numbers. The *weak coloring numbers* wcol_r were introduced by Kierstead and Yang [31] and intuitively measure reachability properties in a linear vertex ordering of a given graph. Formally, they are a series of numbers, parameterized by a positive integer r , which denotes the radius of the considered ordering. Let $\Pi(G)$ be the set of all linear orders of the vertices of the graph G , and let $L \in \Pi(G)$. Let $u, v \in V(G)$. For a positive integer r , we say that u is *weakly r -reachable* from v with respect to L , if there exists a path P of length ℓ , $0 \leq \ell \leq r$, between u and v such that u is minimum among the vertices of P (with respect to L). Let $\text{WReach}_r[G, L, v]$ be the set of vertices that are weakly r -reachable from v with respect to L . Note that $v \in \text{WReach}_r[G, L, v]$. The *weak r -coloring number* $\text{wcol}_r(G)$ of G is defined as

$$\text{wcol}_r(G) := \min_{L \in \Pi(G)} \max_{v \in V(G)} |\text{WReach}_r[G, L, v]|.$$

As proved by Zhu [67], the weak coloring numbers can be used to characterize bounded expansion and nowhere dense classes of graphs.

Uniform quasi-widness. Intuitively, a class of graphs is *wide* if for every graph G from the class, every radius $r \in \mathbb{N}$ and every large subset $A \subseteq V(G)$ of vertices one can find a large r -independent subset $B \subseteq A$. The notion of uniform quasi-widness allows to additionally delete a small number of vertices to make B r -independent. The following definition formalizes the meaning of “large” and “small”.

► **Definition 2.1.** A class \mathcal{C} of graphs is *uniformly quasi-wide* if for every $m \in \mathbb{N}$ and every $r \in \mathbb{N}$ there exist numbers $N(m, r)$ and $s(r)$ such that the following holds.

Let $G \in \mathcal{C}$ and let $A \subseteq V(G)$ with $|A| \geq N(m, r)$. Then there exists a set $S \subseteq V(G)$ with $|S| \leq s(r)$ and a set $B \subseteq A \setminus S$ of size at least m such that for all distinct $u, v \in B$ we have $\text{dist}_{G-S}(u, v) > r$.

Uniform quasi-widness was introduced by Dawar in [15] and it was proved by Nešetřil and Ossona de Mendez in [45] that uniform quasi-widness is equivalent to nowhere denseness.

3 Weak coloring numbers

We experiment with the following approximation algorithms of weak coloring numbers. We here only briefly list them and give necessary definitions to discuss studied variants; a more exhaustive presentation can be found in the full version of the paper.

Distance-constrained transitive fraternal augmentations. We can approximate the weak coloring numbers by orienting the input graph G and iteratively inserting arcs according to certain rules. Such *transitive-fraternal augmentations* (tf-augmentations) were studied first in [43]. We work with an optimized version, called *distance-constrained tf-augmentations* (dtf-augmentations) which were introduced in [53].

Flat decompositions. The following algorithm was introduced in [62]. It provides a way of constructing an order with bounded $wcol_r$ numbers on class of graphs with excluded minors.

Consider the following procedure for computing a vertex ordering of G . At each step, we maintain a family of *blobs* $B_1, B_2, \dots, B_p \subseteq V(G)$, which are pairwise disjoint and connected, and we let $U := V(G) \setminus \bigcup_{i=1}^p B_i$ be the vertices which are not yet contained in any blob. We call vertices in U *unprocessed* and vertices in $V(G) \setminus U$ *processed*. To create the next blob, we let u be any vertex of U and let C be the connected component of $G[U]$ that contains u . Create blob B_{p+1} as follows: start with $\{u\}$, and for every blob B_i that is adjacent to C , pick any vertex $v \in C$ adjacent to B_i , and add to B_{p+1} any shortest path from u to v within C . Finally, when all vertices are subsumed in the blobs, order vertices from different blobs according to the creation time of their blobs, and vertices from the same blob arbitrarily.

As shown in [62], if $K_t \not\leq G$, then the above procedure produces an order that certifies that $wcol_r(G) \in O(r^{t-1})$. Note that this algorithm leaves a lot of room for heuristic optimizations: we can first vary the order of vertices within the blobs and we can vary the choice of the vertex u . As it is not clear which choices would be the best, we decided to create a few sets of rules for both choices and evaluate every combination of them. Within one blob we can order vertices (1) according to a BFS, (2) according to a DFS, (3) in the order of descending degree (motivated by the results of another heuristic). In the tables presented in Section 6, these rules will be abbreviated as BFS, DFS and SORT, respectively. Moreover, each of these orders can be reversed; reversed orders are denoted with an overline over their acronym.

As the next unprocessed vertex u we can choose a vertex (1) with the largest number of processed neighbours, (2) with the largest degree among all unprocessed vertices, (3) with the largest degree among all unprocessed vertices with a processed neighbor. Later, we refer to these rules by their numbers.

Treedepth heuristic. Since the ‘limit’ of weak-coloring numbers is exactly the treedepth of a graph, i.e. $\text{wcol}_\infty(G) = \text{td}(G)$, we consider computing a treedepth decomposition and using an ordering derived from the decomposition. Our algorithm of choice, developed by Sanchez [59] and implemented by Oelschlägel [51], recursively extracts separators from the graph.

Treewidth heuristic. A well-known approach to compute a treewidth decomposition of a graph is to find a linear order of the vertices, an elimination order, of possibly small maximum so-called “back-degree”. There is a number of heuristics to produce good elimination orders. We chose one that is simple, fast and that gives rather good results for treewidth: the so-called minimum-degree heuristic [9].

Other simple heuristics. Apart from algorithms with theoretical guarantees we also compared several naive heuristics.

- For $r = 1$ an optimal order is a degeneracy order, which can be easily computed. We can check if this order produces reasonable results for higher values of r as well.
- Intuitively, it makes sense to sort vertices by descending degree (ties are broken arbitrarily) because from vertices of high degree more vertices can be reached in one step.
- A simple idea of generalizing the above heuristics to bigger values of r is to apply them to the r th power G^r of G (G^r is defined as the graph with $V(G^r) = V(G)$ and $uv \in E(G^r) \Leftrightarrow \text{dist}_G(u, v) \leq r$).
- As a baseline we also included random ordering of vertices.

The intuition behind using a degree-ordering is further supported by a popular network model: *Chung–Lu* random graphs which sample graphs with a fixed degree distribution and successfully replicate several statistics exhibited by real-world networks [12, 13]. In this model, vertices are assigned weights (corresponding to their expected degree) and edges are sampled independently but biased according to the endpoints weights. Therefore vertices of the same degree are exchangeable and the one ordering we can choose to minimize the number of r -reachable vertices is simply the descending degree ordering. It follows that if Chung–Lu graphs are a reasonable approximation of real-world networks, then the degree ordering should be a good choice.

3.1 Local search

In addition to all these approaches we can try to improve their results by local search, a technique where we make small changes to a candidate solution. We applied the following local changes and tested whether they caused improvements to the current order L .

- Swap a vertex v that has biggest $\text{WReach}_r[G, L, v]$ with a random vertex that is smaller with respect to L .
- Swap a vertex v that has biggest $\text{WReach}_r[G, L, v]$ with its direct predecessor u in L .

Both heuristics try to place a vertex with many weakly reachable vertices earlier in the order and thus to make them non-weakly reachable. The advantage of the second rule is that $\text{WReach}_r[G, L, v]$ is trivial to recompute and the only computationally heavy update is for the new $\text{WReach}_r[G, L, u]$. For the first rule, recomputing WReach sets is more expensive. However, the disadvantage of the second rule is that it does not lead to further improvements quickly, hence applications of only the first rule give better results than applications of the second rule only. In our implementation we did a few optimizations in order to improve the results of second rule, but we refrain from describing them in detail. The final algorithm conducting local search firstly performs a round of applications of the first rule and when they no longer improve results it performs a round of applications of the second rule. Such combination turned out to be empirically most effective.

4 Uniform quasi-widness

We experiment with the following algorithms for uniform quasi-widness. We here only briefly list them and give necessary definitions to discuss studied variants; more exhaustive presentation can be found in the full version of the paper.

Distance trees. [52] introduced a method for showing uniform quasi-widness of nowhere dense graphs by iteratively building r -independent sets for increasing values of r . The critical part is an algorithm that, given an (1) -independent set A in a graph G , finds a (small) set S and a 2 -independent set $B \subseteq A$ in $G - S$. An involved combinatorial argument shows the following: either such set B can be already found for the tentative S , or there exists a vertex $v \in V(G)$ with many neighbors in A ; then one includes v in S and restrict A to $N(v) \cap A$. The final restriction is critical for the proof of the bound on the final set S .

We have implemented three variants of this algorithm, denoted later `tree1`, `tree2`, and `ld_it`. `tree2` is the original algorithm of [52], while `tree1` is a variant that, in the step when the set A is restricted to $N(v) \cap A$, tries to preserve some vertices of $A \setminus N(v)$ for future use. Finally, `ld_it` is a variant that replaces every execution of the method of [52] with greedy approach to search for large 2 -independent set $B \subseteq A$.

From weak coloring numbers to uniform quasi-widness. First, we implemented an approach of [34] which is designed for classes of bounded expansion and combines the weak coloring numbers with uniform quasi-widness. This algorithm is later referred to as `mfcs`.

Second, motivated by the rather conservative character of the algorithm of [34], we propose here a new algorithm (albeit inspired by [34]), proving the following.

► **Theorem 4.1.** *Assume we are given a graph G , a set $A \subseteq V(G)$, integers $r \geq 1$ and $m \geq 2$, and an ordering L of $V(G)$ with $c = \max_{v \in V(G)} |\text{WReach}_r[G, L, v]|$. Furthermore, assume that $|A| \geq 4 \cdot (2cm)^c$. Then in polynomial time, one can compute sets $S \subseteq V(G)$ and $B \subseteq A \setminus S$ such that $|S| \leq c$, $|B| \geq m$, and B is r -independent in $G - S$.*

We implemented three variants of the above algorithm, `new1`, `new2`, and `new_ld`. The first two differ in some minor internal details, whereas `new_ld` extends `new2` as follows: at every step it attempts to complete the currently handled partial r -independent set in a greedy manner, and at the end returns the best solution found during the entire execution.

Other naive approaches and heuristic optimizations. Since computing uniform quasi-widness for $r = 1$ is equivalent to finding independent sets, it is sensible to include independent set heuristics as a baseline. Moreover, the approach based on distance trees computes independent sets as a subroutine. We used a simple greedy algorithm to find independent sets: As long as our graph is nonempty, take a vertex of minimum degree, add it to the independent set and remove its closed neighborhood from the graph.

The following algorithm is what we came up with as a naive but reasonable heuristic for larger values of r . For every number $k \in \{0, 1, \dots, K\}$ (where K is some hardcoded constant) compute the biggest independent set in graph $(G - S_k)^r[A]$ using the greedy procedure described above, where S_k is a set of k vertices with biggest degrees. This heuristic is based on the fact that independent sets in G^r correspond to r -independent sets in G . Without any further knowledge about the graph, vertices with the biggest degree seem to be the best candidates to be removed. In the end, we output the best solution obtained in this manner. In the following, we abbreviate this approach as `ld` (least degree on power graph).

4.1 Score: Comparing different results

Uniform quasi-wideness is a two-dimensional measure: we have to measure both the size m of the r -independent set B which we desire to find, as well as the size $s(r)$ of vertices to be deleted. In order to compare the performance of our studied methods we propose the following approach that arises from applications of uniform quasi-wideness in several algorithms [16, 21, 24, 52, 60].

Let $G, A \subseteq V(G), r \in \mathbb{N}$ be an input to any of our algorithms (note that none of our algorithms takes the target size m of the r -independent set as input, we rather try to maximize its size) and let $S \subseteq V(G)$ and $B \subseteq A \setminus S$ such that B is r -independent in $G - S$ be its output. Let us define $\pi_r[v, S]$ – the r -distance profile of v on S – as the function from S to $\{0, 1, \dots, r, \infty\}$ so that $\pi_r[v, S](a) = \text{dist}_G(v, a)$ if this distance is at most r , and $\pi_r[v, S](a) = \infty$ otherwise. The performance of the algorithms [16, 21, 24, 52, 60] strongly depends on the size of the largest equivalence class on B defined by $u \sim v$ if $\pi_r[u, S] = \pi_r[v, S]$ for $u, v \in B$.

We hence decided to use the size of the largest equivalence class in the above relation as the scoring function to measure the performance of our algorithms. Note that number of different r -distance profiles is bounded by $(r + 2)^{|S|}$, so if r is fixed and $|S|$ is bounded then the number of different r -distance profiles is also bounded, so having a big r -independent set implies having a big subset of this set with equal r -distance profiles on S .

This well defined scoring function makes it possible to compare the results of the algorithms. Furthermore, in our code the implementation of the scoring function can be easily exchanged, so if different scoring functions are preferred, re-evaluation is easily possible.

5 Experimental setup

5.1 Hard- and Software

The experiments on generalized coloring numbers has been performed on an Asus K53SC laptop with Intel® Core™ i3-2330M CPU @ 2.20GHz x 2 processor and with 7.7GiB of RAM. Weak coloring numbers of a larger number of graphs for the statistics in Section 6.4 (presented without running times) were produced on a cluster at the Logic and Semantics Research Group, Technische Universität Berlin. The experiments on uniform quasi-wideness have been performed on a cluster of 16 computers at the Institute of Informatics, University of Warsaw. Each machine was equipped with Intel Xeon E3-1240v6 3.70GHz processor and 16 GB RAM. All machines shared the same NFS drive. Since the size of the inputs and outputs to the programs is relatively small, the network communication was negligible for tests with substantial running times. The dtf implementation has been done in Python, while all other code in C++ or C. The code is available at [41, 3].

5.2 Test data

Our dataset consists of a number of graphs from different sources.

Real-world data. We collected appropriately-sized networks from several collections [1, 33, 39, 7, 58, 36]. Our selection contains classic social networks [66, 11], collaboration networks [38, 49, 48] contact networks [61, 40], communication patterns [38, 56, 32, 37, 55, 4], protein-protein interaction [10], gene expression [27], infrastructure [64], tournament data [26], and neural networks [65]. We kept the names assigned to these files by the respective source.

PACE 2016 Feedback Vertex Set. The Parameterized Algorithms and Computational Experiments Challenge is an annual programming challenge started in 2016 that aims at *investigate the applicability of algorithmic ideas studied and developed in the subfields of multivariate, fine-grained, parameterized, or fixed-parameter tractable algorithms* (from the PACE webpage). In the first edition, one of the tracks focused on the FEEDBACK VERTEX SET problem [17], providing 230 instances from various sources and of different sizes. We have chosen a number of instances with small feedback vertex set number, guaranteeing their very strong sparsity properties (in particular, low treewidth). In our result tables, they are named `fvs???`, where `???` is the number in the PACE 2016 dataset.

Random planar graphs. In their seminal paper, Alber, Fellows, and Niedermeier [5] initiated the very fruitful direction of developing of polynomial kernels (preprocessing routines rigorously analyzed through the framework of parameterized complexity) in sparse graph classes by providing a linear kernel for DOMINATING SET in planar graphs. In [5], an experimental evaluation is conducted on random planar graphs generated by the LEDA library [2]. We followed their setup and included a number of random planar graphs with various size and average degree. In our result tables, they are named `planarN`, where `N` stands for the number of vertices.

Random graphs with bounded expansion. A number of random graph models has been shown to produce almost surely graphs of bounded expansion [20]. We include a number of graphs generated by O'Brien and Sullivan [50] using the following models: the stochastic block model (`sb-?` in our dataset) [29] and the Chung-Lu model with households (`c1h-?`) and without households (`c1-?`) [14]. We refer to [20, 50] for more discussion on these sources.

The graphs have been partitioned into four groups, depending on their size: the `small` group gathers graphs up to 1 000 edges, `medium` between 1 000 and 10 000 edges, `big` between 10 000 and 48 000 edges, and `huge` above 48 000 edges. The random planar graphs in every test group have respectively 900, 3 900, 21 000, and 150 000 edges. The whole dataset is available for download at [3].

6 Weak coloring numbers: results

6.1 Fine-tuning flat decompositions

As discussed in Section 3, we have experimented with a number of variants of the flat decompositions approach, with regards to the choice of the next root vertex and the internal order of the vertices of the next B_i . The results for the `big` dataset are presented in Table 1. They clearly indicate that (a) all reversed orders performed much worse, and (b) among other options, the best is to sort the vertices of a new B_i nonincreasingly by degree and choose as the next root the vertex of maximum degree. In the subsequent tests, we use this best configuration for comparison with other approaches.

6.2 Comparison of all approaches

Table 2 presents the results of our experiments on all test instances and all approaches, summarized as follows:

- dtf** dtf-augmentations with the respective radius r supplied as the distance bound;
- flat** the best configuration of the flat decompositions approach (see previous section);
- treedepth** the treedepth approximation heuristic;
- treewidth** the treewidth heuristic;

■ **Table 1** Comparison of different flat decomposition variants: sorting vertices of the new blobs B_i by the BFS, DFS, by degree (nonincreasing), or these orders reversed; the second coordinate refers to the choice of the root vertex: (1) maximizing the number of neighbors already processed, (2) maximizing degree in U , (3) as previous, but only among neighbors of already processed vertices. The value is the average of the approximation ratios to the best generalized coloring numbers found by all versions of this algorithm.

option	average appx. ratio	option	average appx. ratio	option	average appx. ratio
BFS/(1)	1.159	DFS/(1)	1.156	SORT/(1)	1.072
BFS/(2)	1.131	DFS/(2)	1.117	SORT/(2)	1.039
BFS/(3)	1.147	DFS/(3)	1.135	SORT/(3)	1.054
$\overline{\text{BFS}}/(1)$	1.363	$\overline{\text{DFS}}/(1)$	1.368	$\overline{\text{SORT}}/(1)$	1.41
$\overline{\text{BFS}}/(2)$	1.277	$\overline{\text{DFS}}/(2)$	1.291	$\overline{\text{SORT}}/(2)$	1.329
$\overline{\text{BFS}}/(3)$	1.309	$\overline{\text{DFS}}/(3)$	1.324	$\overline{\text{SORT}}/(3)$	1.36

■ **Table 2** *Gray columns:* Comparison of the main approaches and their average approximation ratio to the best found coloring number. Some of the approaches did not finish in time on larger graphs or ran out of memory. *White columns:* Total running time of the main approaches. Note that for some approaches the ordering (and thus running time) is independent of the radius.

tests	r	dtf	flat	treedepth	treewidth	degree sort	
small	2	1.19	0:04.20	1.2	1.408	1.12	1.179
	3	1.439	0:05.08	1.239	1.438	1.124	1.211
	4	1.558	0:05.74	1.288	1.384	1.135	1.213
	5	1.718	0:06.55	1.353	1.414	1.167	1.263
medium	2	1.177	0:27.97	1.362	2.171	1.524	1.142
	3	1.258	1:02.31	1.43	1.918	1.283	1.102
	4	1.499	1:53.21	1.451	1.698	1.159	1.113
	5	1.595	2:15.04	1.469	1.612	1.093	1.149
big	2	1.107	0:32.82	1.43	—	2.278	1.183
	3	1.243	—	1.419	—	1.895	1.088
	4	—	—	1.414	—	1.434	1.079
	5	—	—	1.415	—	1.189	1.065
huge	2	—	—	1.727	—	—	1.152
	3	—	—	2.156	—	—	1.031
	4	—	—	2.13	—	—	1.032
	5	—	—	2.095	—	—	1.029

degree sort the heuristic which sorts the vertices nonincreasing by degree.

Out of all simple heuristics (c.f. Section 3) the degree sorting was supreme and we skip the results of inferior heuristics (see [41, 3] for full data). Interestingly, this heuristic also outperformed all other (much more involved) approaches on larger graphs. On small graphs, the treewidth heuristic takes the lead. An explanation why the treewidth heuristic is better on smaller graphs G might be that $\text{tw}(G) = \text{col}_\infty(G)$ and on small graphs the difference between $\text{col}_\infty(G)$ and $\text{col}_r(G)$ for the considered r is not that big. However, this does not explain why treedepth does not perform better than treewidth. (Recall that $\text{td}(G) = \text{wcol}_\infty(G)$.) It is worth observing that on larger graphs (the **big** group) the performance of the flat

■ **Table 3** *Gray columns*: Comparison of average approximation ratio after local search. *White columns*: Relative improvement of local search for ordering output by the studied approaches.

tests	radius	dtf		flat		treedepth		treewidth		degree sort	
small	2	1.126		1.032		1.142		1.059		1.025	
	3	1.227	16.7%	1.076	16.9%	1.235	15.2%	1.098	7.0%	1.044	16.2%
	4	1.327		1.091		1.281		1.131		1.053	
	5	1.466		1.135		1.311		1.154		1.088	
medium	2	1.192		1.138		1.206		1.135		1.011	
	3	1.204	13.9%	1.115	21.4%	1.303	30.9%	1.121	15.3%	1.023	17.1%
	4	1.444		1.28		1.349		1.139		1.017	
	5	1.482		1.325		1.401		1.134		1.034	
big	2	1.12		1.142		—		1.201		1.045	
	3	1.218	—	1.14	24.4%	—	—	1.29	24.3%	1.015	18.3%
	4	—		1.223		—		1.27		1.017	
	5	—		1.257		—		1.212		1.022	

decomposition matches or outperforms the one of the treewidth heuristic for radii $r = 2, 3, 4$. However, the treewidth heuristic outperforms all approaches with proved guarantees for $r = 5$ on test sets up to the **big** group.

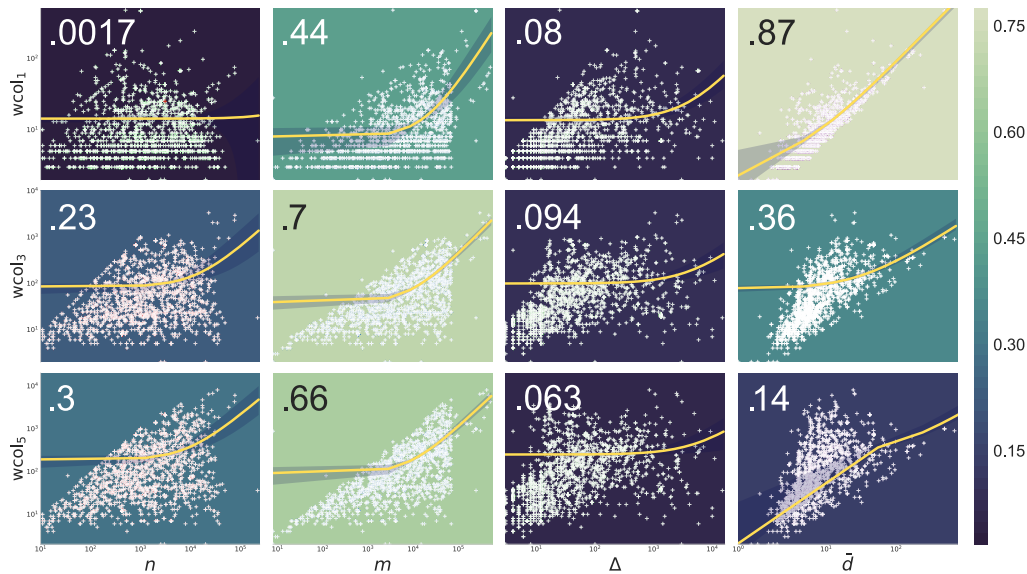
Table 2 gathers total running time of our programs on discussed data sets. These results clearly indicate large discrepancy between consumed resources for different approaches. Out of the approaches with provable guarantees on the output coloring number, the flat decompositions approach is clearly the most efficient.

Note that we applied different timeout policies for generating different data. For generating time of execution and for applying local search we set timeout to be 1 minute, however for generating orders and wcol numbers we set timeout to be 5 minutes, but for the sake of completeness we sometimes allowed some programs to run longer.

In summary, on our data sets the simple heuristic is consistently the fastest and produces the best results, save for the smallest graphs on which the treewidth heuristic won. We remark here that it is simple to “fool” the degree-sorting heuristic by adding multiple pendant vertices of degree one and thus forcing it to take arbitrarily bad ordering, but such adversarial obstacles seem to be absent in real-world graphs. If one is to choose an algorithm with provable guarantees, the discussed variant of the flat decompositions approach appears to be the best choice.

6.3 Local search

In a second round of experiments we applied a simple local-search routine that, given an ordering output by one of the approaches, tries to improve it by moving vertices with the largest weakly reachable sets earlier in the ordering. The white columns in Table 3 show how local search improved orderings output by discussed approaches, and the gray columns show average approximation ratios of orderings improved by local search. Two remarks are in place. First, regardless of how the ordering was computed, a local search step always significantly improves the ordering (we have no good explanation on why local search is significantly less effective on the orderings output by the treewidth heuristic for bigger radii). Second, the local search step does *not* improve the orderings enough to change the relative order of the performance of the base approaches except for one remarkable case. On **medium** group the treewidth heuristic gave best results on $r = 5$, however degree sort regained the lead after



■ **Figure 1** Correlation of $wcol$ (computed using the degree sort heuristic) with graph size, maximum degree and average degree of 1675 real-world graphs. The background shade and number reflect the correlation of the two respective measures, superimposed is a log-log plot of the measurements. The yellow lines are linear regressions with dark shaded confidence intervals.

application of local search due to its low performance on larger radii for treewidth heuristic. We therefore recommend the local search improvement as a relatively cheap post-processing improvement to any existing algorithm.

6.4 Correlation of weak coloring numbers with other parameters

While it is undeniable that weak coloring numbers have immense algorithmic power from a theoretical perspective, the efficient computation of such weak coloring orders is only one component to leverage them in practice: we also need these numbers to be reasonably low. So far, this had only been established on a smaller scale [20, 53] for a related measure. Here, we computed the weak coloring number for $r \in \{1, \dots, 5\}$ for 1675 real-world networks from various sources [36, 39, 58, 7, 1]. Figure 1 summarizes our findings: for $r \in \{1, \dots, 3\}$ we find a modest correlation with n and a significant correlation with m . The correlation with n becomes quite pronounced for $r = 5$; the probable reason being that for all networks involved $\log n \leq 10$. Still, even in the worst examples $wcol_5$ is at least one order of magnitude smaller than n or m . We further see a high correlation between $wcol_1$ and the average degree \bar{d} which vanishes for larger radii. It is no big surprise that \bar{d} and the degeneracy $wcol_1$ are highly correlated since these values are only far apart in graphs with highly inhomogeneous densities.

The low dependence on the maximum degree confirms the findings of [20]: the exact shape of the degree distribution's tail is much more relevant than the singular value of the maximum degree. Finally, note that in our graphs the degeneracy $wcol_1$ practically does not grow with n .

■ **Table 4** Aggregated results of uniform quasi-widness on **medium** set for $r = 3$ and $r = 5$ (values for $r = 2$ and $r = 4$ can be found in the full version of the paper): total size of all deleted and independent sets, total score (total size of largest equivalence classes w.r.t. deleted vertices), and total running time.

r	algorithm	start with whole $V(G)$				start with 20% of $V(G)$			
		deleted	independent	score	time	deleted	independent	score	time
3	mfcs	5076	11471	2153	0:01.25	1922	3459	1135	0:00.48
	new1	78	2345	2211	0:37.53	49	1192	1159	0:29.96
	new2	84	3820	3673	0:34.34	49	2132	2096	0:23.36
	new_ld	—	—	—	—	5	2926	2873	11:10.63
	tree1	7	6072	5686	0:02.77	4	2652	2598	0:00.48
	tree2	5	5645	5645	0:01.00	4	2603	2603	0:00.38
	ld_it	7	6136	5748	0:01.71	4	2741	2688	0:00.39
	ld	5	6471	6296	0:08.13	6	2972	2871	0:02.01
5	mfcs	7946	15773	1164	0:01.93	4057	4396	594	0:00.67
	new1	115	1623	1445	4:38.57	84	709	676	3:20.15
	new2	122	2079	1888	4:19.50	103	1036	982	3:07.82
	new_ld	—	—	—	—	—	—	—	—
	tree1	11	2988	2643	0:02.85	4	1325	1282	0:00.53
	tree2	5	2603	2603	0:01.05	4	1284	1284	0:00.45
	ld_it	12	3102	2752	0:01.84	5	1380	1336	0:00.64
	ld	7	3192	3043	0:29.32	5	1517	1473	0:07.15

7 Uniform quasi widness: results

Table 4 gathers aggregated data from our experiments on **medium** dataset. (Full data can be downloaded from [41, 3].) Every tested algorithm has been run on every test with timeout 10 minutes and with radii $r \in \{2, 3, 4, 5\}$ and with the starting set either $A = V(G)$ or a random subset of 20% of vertices of $V(G)$.

Data indicate the simple heuristic, **ld**, as the best choice in most scenarios, as it has always best or nearly-best total score and runs relatively quickly. The third variant of the new algorithm **new_ld** has comparable results, but is inefficient and does not finish within the timeout. Other variants **new1** and **new2** as well as **mfcs** are significantly outperformed by other approaches. Out of other approaches with provable guarantees, the variants **tree1**, **tree2**, and **ld_it** provide results in most cases less than 10% worse than the heuristic **ld**, with **tree2** being consistently worse.

To sum up, our experiments show that the simple heuristic **ld** gives best results, but if one is interested in algorithm with provable guarantees, one should choose one of the variant **tree1** over **mfcs** or **new1/new2**.

8 Conclusions

We have conducted a thorough empirical evaluation of algorithms for computing generalized coloring numbers and uniform quasi-widness. In both cases, one of the simplest heuristics, without any theoretical guarantees, outperformed the rest. In particular, our new algorithm for uniform quasi-widness, whose development was motivated by conservativeness of the previous approach of [34], performed rather poorly in the experiments. From the algorithms

with provable guarantees, the experiments indicated a variant of the algorithm of [62] as the algorithm of choice for generalized coloring numbers and a variant of the algorithm of [52] as the algorithm of choice for uniform quasi-wideness.

References

- 1 Gephi datasets. URL: <https://github.com/gephi/gephi/wiki/Datasets>.
- 2 LEDA. URL: <http://www.algorithmic-solutions.com/leda/index.htm>.
- 3 Recent trends in kernelization: theory and experimental evaluation — project website, 2018. URL: <http://kernelization-experiments.mimuw.edu.pl>.
- 4 Lada A Adamic and Natalie Glance. The political blogosphere and the 2004 US election: divided they blog. In *Proceedings of the 3rd International Workshop on Link Discovery*, pages 36–43. ACM, 2005.
- 5 Jochen Alber, Michael R. Fellows, and Rolf Niedermeier. Polynomial-time data reduction for dominating set. *Journal of the ACM*, 51(3):363–384, 2004. doi:10.1145/990308.990309.
- 6 Saeed Akhoondian Amiri, Patrice Ossona de Mendez, Roman Rabinovich, and Sebastian Siebertz. Distributed domination on graph classes of bounded expansion. *CoRR*, abs/1702.02848, 2017.
- 7 Vladimir Batagelj and Andrej Mrvar. Pajek datasets. <http://vlado.fmf.uni-lj.si/pub/networks/data/>, 2006.
- 8 Hans L. Bodlaender. Treewidth: Algorithmic techniques and results. In *International Symposium on Mathematical Foundations of Computer Science*, pages 19–36. Springer, 1997.
- 9 Hans L. Bodlaender and Arie M. C. A. Koster. Treewidth computations I. Upper bounds. *Information and Computation*, 208(3):259–275, 2010. doi:10.1016/j.ic.2009.03.008.
- 10 Dongbo Bu, Yi Zhao, Lun Cai, Hong Xue, Xiaopeng Zhu, Hongchao Lu, Jingfen Zhang, Shiwei Sun, Lunjiang Ling, Nan Zhang, et al. Topological structure analysis of the protein–protein interaction network in budding yeast. *Nucleic acids research*, 31(9):2443–2450, 2003.
- 11 Eunjoon Cho, Seth A. Myers, and Jure Leskovec. Friendship and mobility: user movement in location-based social networks. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Diego, CA, USA, August 21–24, 2011*, pages 1082–1090, 2011. doi:10.1145/2020408.2020579.
- 12 Fan Chung and Linyuan Lu. The average distances in random graphs with given expected degrees. *Proceedings of the National Academy of Sciences*, 99(25):15879–15882, 2002.
- 13 Fan Chung and Linyuan Lu. Connected components in random graphs with given expected degree sequences. *Annals of combinatorics*, 6(2):125–145, 2002.
- 14 Fan R. K. Chung and Linyuan Lu. The average distance in a random graph with given expected degrees. *Internet Mathematics*, 1(1):91–113, 2003. doi:10.1080/15427951.2004.10129081.
- 15 Anuj Dawar. Homomorphism preservation on quasi-wide classes. *Journal of Computer and System Sciences*, 76(5):324–332, 2010.
- 16 Anuj Dawar and Stephan Kreutzer. Domination problems in nowhere-dense classes. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS*, pages 157–168, 2009.
- 17 Holger Dell, Thore Husfeldt, Bart M. P. Jansen, Petteri Kaski, Christian Komusiewicz, and Frances A. Rosamond. The First Parameterized Algorithms and Computational Experiments Challenge. In *11th International Symposium on Parameterized and Exact Computation (IPEC 2016)*, volume 63 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 30:1–30:9, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.IPEC.2016.30.

- 18 Erik D. Demaine and MohammadTaghi Hajiaghayi. The bidimensionality theory and its algorithmic applications. *The Computer Journal*, 51(3):292–302, 2007.
- 19 Erik D. Demaine, MohammadTaghi Hajiaghayi, and Ken-ichi Kawarabayashi. Algorithmic graph minor theory: Improved grid minor bounds and wagner’s contraction. *Algorithmica*, 54(2):142–180, 2009.
- 20 Erik D. Demaine, Felix Reidl, Peter Rossmanith, Fernando Sánchez Villaamil, Somnath Sikdar, and Blair D. Sullivan. Structural sparsity of complex networks: Random graph models and linear algorithms. *CoRR*, abs/1406.2587, 2014. [arXiv:1406.2587](https://arxiv.org/abs/1406.2587).
- 21 Pål Grønås Drange, Markus Sortland Dregi, Fedor V. Fomin, Stephan Kreutzer, Daniel Lokshtanov, Marcin Pilipczuk, Michał Pilipczuk, Felix Reidl, Fernando Sánchez Villaamil, Saket Saurabh, Sebastian Siebertz, and Somnath Sikdar. Kernelization and sparseness: the case of dominating set. In *33rd Symposium on Theoretical Aspects of Computer Science, STACS*, pages 31:1–31:14, 2016.
- 22 Zdenek Dvorak. Constant-factor approximation of the domination number in sparse graphs. *European Journal of Combinatorics*, 34(5):833–840, 2013.
- 23 Zdenek Dvorak, Daniel Král, and Robin Thomas. Testing first-order properties for subclasses of sparse graphs. *Journal of the ACM*, 60(5):36:1–36:24, 2013.
- 24 Kord Eickmeyer, Archontia C. Giannopoulou, Stephan Kreutzer, O-joung Kwon, Michał Pilipczuk, Roman Rabinovich, and Sebastian Siebertz. Neighborhood complexity and kernelization for nowhere dense classes of graphs. In *44th International Colloquium on Automata, Languages, and Programming, ICALP*, pages 63:1–63:14, 2017.
- 25 Jakub Gajarský, Petr Hliněný, Jan Obdržálek, Sebastian Ordyniak, Felix Reidl, Peter Rossmanith, Fernando Sánchez Villaamil, and Somnath Sikdar. Kernelization using structural parameters on sparse graph classes. *Journal of Computer and System Sciences*, 84:219–242, 2017.
- 26 Michelle Girvan and Mark E. J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826, 2002.
- 27 Kwang-Il Goh, Michael E. Cusick, David Valle, Barton Childs, Marc Vidal, and Albert-László Barabási. The human disease network. *Proceedings of the National Academy of Sciences*, 104(21):8685–8690, 2007.
- 28 Martin Grohe, Stephan Kreutzer, and Sebastian Siebertz. Deciding first-order properties of nowhere dense graphs. *Journal of the ACM*, 64(3):17:1–17:32, 2017.
- 29 Paul W. Holland, Kathryn B. Laskey, and Samuel Leinhardt. Stochastic blockmodels: First steps. *Social networks*, 5(2):109–137, 1983.
- 30 Wojciech Kazana and Luc Segoufin. Enumeration of first-order queries on classes of structures with bounded expansion. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS*, pages 297–308, 2013.
- 31 Henry A. Kierstead and Daqing Yang. Orderings on graphs and game coloring number. *Order*, 20:255–264, 2003.
- 32 Bryan Klimt and Yiming Yang. Introducing the Enron corpus. In *CEAS*, 2004.
- 33 Donald Ervin Knuth. *The Stanford GraphBase: a platform for combinatorial computing*, volume 37. Addison-Wesley Reading, 1993.
- 34 Stephan Kreutzer, Michał Pilipczuk, Roman Rabinovich, and Sebastian Siebertz. The generalised colouring numbers on classes of bounded expansion. In *41st International Symposium on Mathematical Foundations of Computer Science, MFCS*, 2016.
- 35 Stephan Kreutzer, Roman Rabinovich, and Sebastian Siebertz. Polynomial kernels and wideness properties of nowhere dense graph classes. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 1533–1545, 2017.
- 36 Jérôme Kunegis. KONECT - the Koblenz network collection. In *Proc. Int. Web Observatory Workshop*, pages 1343–1350, 2013. URL: <http://konect.uni-koblenz.de>.

- 37 Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. Signed networks in social media. In *Proceedings of the SIGCHI conference on human factors in computing systems*, pages 1361–1370. ACM, 2010.
- 38 Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 177–187. ACM, 2005.
- 39 Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. <http://snap.stanford.edu/data>, 2014.
- 40 David Lusseau, Karsten Schneider, Oliver J. Boisseau, Patti Haase, Elisabeth Slooten, and Steve M. Dawson. The bottlenose dolphin community of doubtful sound features a large proportion of long-lasting associations. *Behavioral Ecology and Sociobiology*, 54(4):396–405, 2003.
- 41 Wojciech Nadara, Marcin Pilipczuk, Felix Reidl, Roman Rabinovich, and Sebastian Siebertz. Empirical evaluation of approximation algorithms for generalized graph coloring and uniform quasi-widness. code repository, 2018. https://bitbucket.org/marcin_pilipczuk/wcol-uw-experiments.
- 42 Jaroslav Nešetřil and Patrice Ossona de Mendez. Grad and classes with bounded expansion I. decompositions. *European Journal of Combinatorics*, 29(3):760–776, 2008.
- 43 Jaroslav Nešetřil and Patrice Ossona de Mendez. Grad and classes with bounded expansion II. algorithmic aspects. *European Journal of Combinatorics*, 29(3):777–791, 2008.
- 44 Jaroslav Nešetřil and Patrice Ossona de Mendez. Grad and classes with bounded expansion III. restricted graph homomorphism dualities. *European Journal of Combinatorics*, 29(4):1012–1024, 2008.
- 45 Jaroslav Nešetřil and Patrice Ossona de Mendez. First order properties on nowhere dense structures. *The Journal of Symbolic Logic*, 75(3):868–887, 2010.
- 46 Jaroslav Nešetřil and Patrice Ossona de Mendez. On nowhere dense graphs. *European Journal of Combinatorics*, 32(4):600–617, 2011.
- 47 Jaroslav Nešetřil and Patrice Ossona de Mendez. *Sparsity - Graphs, Structures, and Algorithms*, volume 28 of *Algorithms and combinatorics*. Springer, 2012.
- 48 Mark EJ Newman. The structure of scientific collaboration networks. *Proceedings of the national academy of sciences*, 98(2):404–409, 2001.
- 49 Mark EJ Newman. Finding community structure in networks using the eigenvectors of matrices. *Physical review E*, 74(3):036104, 2006.
- 50 Michael P. O’Brien and Blair D. Sullivan. Experimental evaluation of counting subgraph isomorphisms in classes of bounded expansion. *CoRR*, abs/1712.06690, 2017. [arXiv:1712.06690](https://arxiv.org/abs/1712.06690).
- 51 Tobias Oelschlägel. *Treewidth from Treedepth*. Bachelor’s thesis, RWTH Aachen University, Germany, 2014.
- 52 Michał Pilipczuk, Sebastian Siebertz, and Szymon Toruńczyk. On the number of types in sparse graphs. *arXiv preprint arXiv:1705.09336*, 2017.
- 53 Felix Reidl. *Structural sparseness and complex networks*. Dr., Aachen, Techn. Hochsch., Aachen, 2016. Aachen, Techn. Hochsch., Diss., 2015. URL: <http://publications.rwth-aachen.de/record/565064>.
- 54 Felix Reidl, Fernando Sánchez Villaamil, and Konstantinos Stavropoulos. Characterising bounded expansion by neighbourhood complexity. *CoRR*, abs/1603.09532, 2016.
- 55 Matthew Richardson, Rakesh Agrawal, and Pedro Domingos. Trust management for the semantic web. In *International semantic Web conference*, pages 351–368. Springer, 2003.
- 56 Matei Ripeanu, Ian Foster, and Adriana Iamnitchi. Mapping the Gnutella Network. *IEEE Internet Computing*, 6(1):50–57, 2002.

- 57 Neil Robertson and Paul D. Seymour. Graph minors I–XXII, 1982–2010.
- 58 Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015. URL: <http://networkrepository.com>.
- 59 Fernando Sanchez Villaamil. *About Treedepth and Related Notions*. Dissertation, RWTH Aachen University, Aachen, 2017. doi:10.18154/RWTH-2017-09829.
- 60 Sebastian Siebertz. Reconfiguration on nowhere dense graph classes. *CoRR*, abs/1707.06775, 2017.
- 61 Juliette Stehlé, N. Voirin, Alain Barrat, Ciro Cattuto, Lorenzo Isella, Jean-François Pinton, Marco Quaggiotto, Wouter Van den Broeck, C. Régis, B. Lina, and P. Vanhems. High-resolution measurements of face-to-face contact patterns in a primary school. *PLOS ONE*, 6(8):e23176, 08 2011.
- 62 Jan van den Heuvel, Patrice Ossona de Mendez, Daniel A. Quiroz, Roman Rabinovich, and Sebastian Siebertz. On the generalised colouring numbers of graphs that exclude a fixed minor. *European Journal of Combinatorics*, 66:129–144, 2017.
- 63 Jan van den Heuvel, Stephan Kreutzer, Michał Pilipczuk, Daniel A. Quiroz, Roman Rabinovich, and Sebastian Siebertz. Model-checking for successor-invariant first-order formulas on graph classes of bounded expansion. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS*, pages 1–11, 2017.
- 64 Duncan J Watts and Steven H Strogatz. Collective dynamics of ‘small-world’ networks. *nature*, 393(6684):440, 1998.
- 65 John G. White, Eileen Southgate, J. Nichol Thomson, and Sydney Brenner. The structure of the nervous system of the nematode *caenorhabditis elegans*: the mind of a worm. *Phil. Trans. R. Soc. Lond.*, 314:1–340, 1986.
- 66 Wayne W. Zachary. An information flow model for conflict and fission in small groups. *Journal of anthropological research*, pages 452–473, 1977.
- 67 Xuding Zhu. Colouring graphs with bounded generalized colouring number. *Discrete Math.*, 309:5562–5568, 2009.


Multi-Level Steiner Trees

Reyan Ahmed

University of Arizona, Tucson, United States


Patrizio Angelini

Universität Tübingen, Tübingen, Germany

 <https://orcid.org/0000-0002-7602-1524>

Faryad Darabi Sahneh

University of Arizona, Tucson, United

 <https://orcid.org/0000-0003-2521-4331>

Alon Efrat

University of Arizona, Tucson, United States

David Glickenstein

University of Arizona, Tucson, United States

Martin Gronemann


Universität zu Köln, Cologne, Germany

Niklas Heinsohn

Universität Tübingen, Tübingen, Germany

Stephen G. Kobourov

University of Arizona, Tucson, United States

 <https://orcid.org/0000-0002-0477-2724>

Richard Spence

University of Arizona, Tucson, United States


rspence@email.arizona.edu

Joseph Watkins

University of Arizona, Tucson, United States

Alexander Wolff

Universität Würzburg, Würzburg, Germany

 <https://orcid.org/0000-0001-5872-718X>

Abstract

In the classical Steiner tree problem, one is given an undirected, connected graph $G = (V, E)$ with non-negative edge costs and a set of *terminals* $T \subseteq V$. The objective is to find a minimum-cost edge set $E' \subseteq E$ that spans the terminals. The problem is APX-hard; the best known approximation algorithm has a ratio of $\rho = \ln(4) + \varepsilon < 1.39$. In this paper, we study a natural generalization, the *multi-level Steiner tree* (MLST) problem: given a nested sequence of terminals $T_1 \subset \dots \subset T_k \subseteq V$, compute nested edge sets $E_1 \subseteq \dots \subseteq E_k \subseteq E$ that span the corresponding terminal sets with minimum total cost.

The MLST problem and variants thereof have been studied under names such as Quality-of-Service Multicast tree, Grade-of-Service Steiner tree, and Multi-Tier tree. Several approximation results are known. We first present two natural heuristics with approximation factor $O(k)$. Based on these, we introduce a composite algorithm that requires 2^k Steiner tree computations. We determine its approximation ratio by solving a linear program. We then present a method that guarantees the same approximation ratio and needs at most $2k$ Steiner tree computations. We compare five algorithms experimentally on several classes of graphs using four types of graph generators. We also implemented an integer linear program for MLST to provide ground truth.



© Reyan Ahmed, Patrizio Angelini, Faryad Darabi Sahneh, Alon Efrat, David Glickenstein, Martin Gronemann, Niklas Heinsohn, Stephen Kobourov, Richard Carlton Spence, Joe Watkins, and Alexander Wolff;

licensed under Creative Commons License CC-BY

17th International Symposium on Experimental Algorithms (SEA 2018).

Editor: Gianlorenzo D'Angelo; Article No. 15; pp. 15:1–15:14



Leibniz International Proceedings in Informatics

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

Our combined algorithm outperforms the others both in theory and in practice when the number of levels is small ($k \leq 22$), which works well for applications such as designing multi-level infrastructure or network visualization.

2012 ACM Subject Classification Theory of computation \rightarrow Design and analysis of algorithms, Theory of computation \rightarrow Approximation algorithms analysis, Theory of computation \rightarrow Routing and network design problems

Keywords and phrases Approximation algorithm, Steiner tree, multi-level graph representation

Digital Object Identifier 10.4230/LIPIcs.SEA.2018.15

Related Version An extended version of this manuscript is available at arXiv:1804.02627 [1].

Funding This work is partially supported by NSF grants CCF-1423411 and CCF-1712119.

Acknowledgements The authors wish to thank the organizers and participants of the First Heiligkreuztal Workshop on Graph and Network Visualization for the stimulating atmosphere.

1 Introduction

Let $G = (V, E)$ be an undirected, connected graph with non-negative edge costs $c: E \rightarrow \mathbb{R}^+$, and let $T \subseteq V$ be a set of vertices called *terminals*. A *Steiner tree* is a tree in G that spans T . The *network (graph) Steiner tree problem* (ST) is to find a minimum-cost Steiner tree $E' \subseteq E$, where the cost of E' is $c(E') = \sum_{e \in E'} c(e)$. ST is one of Karp's initial NP-hard problems [13]; see also a survey [23], an online compendium [12], and a textbook [20].

Due to its practical importance in many domains, there is a long history of exact and approximation algorithms for the problem. The classical 2-approximation algorithm for ST [11] uses the *metric closure* of G , i.e., the complete edge-weighted graph G^* with vertex set T in which, for every edge uv , the cost of uv equals the length of a shortest $u-v$ path in G . A minimum spanning tree of G^* corresponds to a 2-approximate Steiner tree in G .

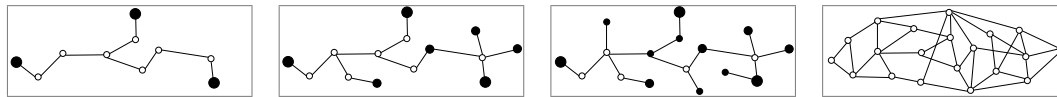
Currently, the last in a long list of improvements is the LP-based approximation algorithm of Byrka et al. [6], which has a ratio of $\ln(4) + \varepsilon < 1.39$. Their algorithm uses a new iterative randomized rounding technique. Note that ST is APX-hard [5]; more concretely, it is NP-hard to approximate the problem within a factor of $96/95$ [8]. This is in contrast to the geometric variant of the problem, where terminals correspond to points in the Euclidean or rectilinear plane. Both variants admit polynomial-time approximation schemes (PTAS) [2, 16], while this is not true for the general metric case [5].

In this paper, we consider a natural generalization of ST where the terminals appear on “levels” and must be connected by edges of appropriate levels. We propose new approximation algorithms and compare them to existing ones both theoretically and experimentally.

► **Definition 1** (Multi-Level Steiner Tree (MLST) Problem). Given a connected, undirected graph $G = (V, E)$ with edge weights $c: E \rightarrow \mathbb{R}^+$ and k nested terminal sets $T_1 \subseteq \dots \subseteq T_k \subseteq V$, a *multi-level Steiner tree* consists of k nested edge sets $E_1 \subseteq \dots \subseteq E_k \subseteq E$ such that E_1 spans T_1 , \dots , E_k spans T_k . The cost of an MLST is defined by $c(E_1) + c(E_2) + \dots + c(E_k)$. The MLST problem is to find an MLST $E_{\text{OPT},1} \subseteq \dots \subseteq E_{\text{OPT},k} \subseteq E$ with minimum cost.

Since the edge sets are nested, we can also express the cost of an MLST as follows:

$$kc(E_1) + (k-1)c(E_2 \setminus E_1) + \dots + c(E_k \setminus E_{k-1}).$$



■ **Figure 1** An illustration of a 3-level MLST for the graph at the right. Solid and open circles represent terminal and non-terminal nodes, respectively. Note that the level 1 tree (left) is contained in the level 2 tree (mid), which is in turn contained in the level 3 tree (right).

This emphasizes that the total cost $c(e)$ of an edge that appears at level ℓ is $(k - \ell + 1)c(e)$.

We denote the cost of an optimal MLST by OPT . We can write

$$\text{OPT} = k\text{OPT}_1 + (k - 1)\text{OPT}_2 + \cdots + \text{OPT}_k$$

where $\text{OPT}_1 = c(E_{\text{OPT},1})$ and $\text{OPT}_\ell = c(E_{\text{OPT},\ell} \setminus E_{\text{OPT},\ell-1})$ for $2 \leq \ell \leq k$. Thus OPT_ℓ represents the cost of edges on level ℓ but not on level $\ell - 1$ in the minimum cost MLST. Figure 1 shows an example of an MLST for $k = 3$.

Applications. This problem has natural applications in designing multi-level infrastructure of low cost. Apart from this application in network design, multi-scale representations of graphs are useful in applications such as network visualization, where the goal is to represent a given graph at different levels of detail.

Previous Work. Variants of the MLST problem have been studied previously under various names, such as *Multi-Level Network Design (MLND)* [3], *Multi-Tier Tree (MTT)* [15], *Quality-of-Service (QoS) Multicast Tree* [7], and *Priority-Steiner Tree* [9].

In MLND, the vertices of the given graph are partitioned into k levels, and the task is to construct a k -level network. For $1 \leq \ell \leq k$, let $c^\ell(e)$ be the cost of edge e if it is in level ℓ . The vertices on each level must be connected by edges of the corresponding level or higher, and edges of higher level are more costly, that is, $0 \leq c^k(e) \leq \cdots \leq c^1(e)$ for any edge e . The cost of an edge partition is the sum of all edge costs, and the task is to find a partition of minimum cost. Let ρ be the ratio of the best approximation algorithm for (single-level) ST, that is, currently $\rho = \ln(4) + \varepsilon < 1.39$. Balakrishnan et al. [3] gave a $4/3\rho$ -approximation algorithm for 2-level MLND with proportional edge costs, that is, $c^\ell(e) = c^k(e)(k - \ell + 1)$. Note that the definitions of MLND and MLST treat the bottom level differently. While MLND requires that *all* vertices are connected eventually, this is not the case for MLST. In this respect, MLST is more general than MLND, which makes it harder to approximate. On the other hand, MLND is more flexible in terms of edge costs. Whereas the Steiner tree problem is a special case of the MLST problem for $k = 1$, the same problem is a special case of MLND for $k = 2$, by setting $c^2(e) = 0$.

For MTT, which is equivalent to MLND, Mirchandani [15] presented a recursive algorithm that involves 2^k Steiner tree computations. For $k = 3$, the algorithm achieves an approximation ratio of 1.522ρ independently of the edge costs $c^1, \dots, c^k: E \rightarrow \mathbb{R}^+$. For proportional edge costs, Mirchandani's analysis yields even an approximation ratio of 1.5ρ for $k = 3$. Recall, however, that this assumes $T_k = V$, and setting the edge costs on the bottom level to zero means that edge costs are *not* proportional.

In the QoS Multicast Tree problem [7] one is given a graph, a source vertex s , and a level between 1 and k for each terminal (1 meaning important). The task is to find a minimum-cost Steiner tree that connects all terminals to s . The level of an edge e in this tree is the minimum over the levels of the terminals that are connected to s via e . The cost

of the edges and of the tree are as above. As a special case, Charikar et al. [7] study the *rate model*, where edge costs are proportional, and show that the problem remains NP-hard if all vertices (except the source) are terminals (at some level). Note that if we choose as source any vertex at the top level T_1 , then MLST can be seen as an instance of the rate model.

Charikar et al. [7] gave a simple 4ρ -approximation algorithm for the rate model. Given an instance φ , their algorithm constructs an instance φ' where the levels of all vertices are rounded up to the nearest power of 2. Then the algorithm simply computes a Steiner tree at each level of φ' and prunes the union of these Steiner trees into a single tree. The ratio can be improved to $e\rho$, where e is the base of the natural logarithm, using randomized doubling.

Instead of taking the union of the Steiner trees on each rounded level, Karpinski et al. [14] contract them into the source in each step, which yields a 2.454ρ -approximation. They also gave a $(1.265 + \varepsilon)\rho$ -approximation for the 2-level case. (Since these results are not stated with respect to ρ , but depend on several Steiner tree approximation algorithms – among them the best approximation algorithm with ratio 1.549 [21] available at the time – we obtained the numbers given here by dividing their results by 1.549 and stating the factor ρ .)

For the more general Priority-Steiner Tree problem, where edge costs are not necessarily proportional, Charikar et al. [7] gave a $\min\{2\ln |T|, k\rho\}$ -approximation algorithm. Chuzhoy et al. [9] showed that Priority-Steiner Tree does not admit an $O(\log \log n)$ -approximation algorithm unless $\text{NP} \subseteq \text{DTIME}(n^{O(\log \log \log n)})$. For Euclidean MLST, Xue et al. [24] gave a recursive algorithm that uses any algorithm for Euclidean Steiner Tree (EST) as a subroutine. With a PTAS [2, 16] for EST, the approximation ratio of their algorithm is $4/3 + \varepsilon$ for $k = 2$ and $(5 + 4\sqrt{2})/7 + \varepsilon \approx 1.5224 + \varepsilon$ for $k = 3$.

Our Contribution. We introduce and analyze two intuitive approximation algorithms for MLST – bottom-up and top-down; see Section 2.1. The bottom-up heuristic uses a Steiner tree at the bottom level for the higher levels after pruning unnecessary edges at each level. The top-down heuristic first computes a Steiner tree on the top level. Then it passes edges down from level to level until the bottom level terminals are spanned.

We then propose a composite heuristic that generalizes these and examines all possible 2^{k-1} (partial) top-down and bottom-up combinations and returns the one with the lowest cost; see Section 2.2. We propose a linear program that finds the approximation ratio of the composite heuristic for any fixed value of k . We compute the explicit approximation ratios for up to 22 levels, which turn out to be better than those of previously known algorithms. The composite heuristic requires, however, 2^k ST computations.

Therefore, we propose a procedure that achieves the same approximation ratio as the composite heuristic but needs only $2k$ ST computations. In particular, it achieves a ratio of 1.5ρ for $k = 3$ levels, which settles a question posed by Karpinski et al. [14] who were asking whether the $1.5224 + \varepsilon$ -approximation of Xue et al. [24] can be improved for $k = 3$. Note that Xue et al. treated the Euclidean case, so their ratio does not include the factor ρ . We generalize an integer linear programming (ILP) formulation for ST [19] to obtain an exact algorithm for MLST; see Section 3. We experimentally evaluate several approximation and exact algorithms on a wide range of problem instances; see Section 4. The results show that the new algorithms are also surprisingly good in practice. We conclude in Section 5.

2 Approximation Algorithms

In this section we propose several approximation algorithms for MLST. In Section 2.1, we show that the natural approach of computing edge sets either from top to bottom or vice versa, already give $O(k)$ -approximations; we call these two approaches *top-down* and *bottom-up*,

and denote their cost by TOP and BOT, respectively. Then, we show that running the two approaches and selecting the solution with minimum cost produces a better approximation ratio than either top-down or bottom-up.

In Section 2.2, we propose a composite approach that mixes the top-down and bottom-up approaches by solving ST on a certain subset of levels, then propagating the chosen edges to higher and lower levels in a way similar to the previous approaches. We then run the algorithm for each of the 2^{k-1} possible subsets, and select the solution with minimum cost. For relatively small values of k ($k \leq 22$), our results improve over the state of the art.

2.1 Top-Down and Bottom-Up Approaches

We present top-down and bottom-up approaches for computing approximate multi-level Steiner trees. The approaches are similar to the MST and Forward Steiner Tree (FST) heuristics by Balakrishnan et al. [3]; however, we generalize the analysis to an arbitrary number of levels.

In the top-down approach, we compute an exact or approximate Steiner tree $E_{\text{TOP},1}$ spanning T_1 . Then we modify the edge weights by setting $c(e) := 0$ for every edge $e \in E_{\text{TOP},1}$. In the resulting graph, we compute a Steiner tree $E_{\text{TOP},2}$ spanning T_2 . This extends $E_{\text{TOP},1}$ in a greedy way to span the terminals in T_2 not already spanned by $E_{\text{TOP},1}$. Iterating this procedure for all levels yields a solution $E_{\text{TOP},1} \subseteq \dots \subseteq E_{\text{TOP},k} \subseteq E$ with cost TOP.

In the bottom-up approach, we compute a Steiner tree $E_{\text{BOT},k}$ spanning the terminals T_k in level k . Then, for each level ℓ , we obtain $E_{\text{BOT},\ell}$ as the smallest subtree of $E_{\text{BOT},k}$ that spans all the terminals in T_ℓ , giving a solution with cost BOT.

A natural approach is to run both top-down and bottom-up approaches and select the solution with minimum cost. This yields an approximation ratio better than those from top-down or bottom-up. Let $\rho \geq 1$ denote the approximation ratio for ST (that is, $\rho = 1$ corresponds to using an exact ST subroutine).

► **Theorem 2.** *For $k \geq 2$ levels, the top-down approach is a $\frac{k+1}{2}\rho$ -approximation to MLST, the bottom-up approach is a $k\rho$ -approximation, and taking the minimum of TOP and BOT is a $\frac{k+2}{3}\rho$ -approximation.*

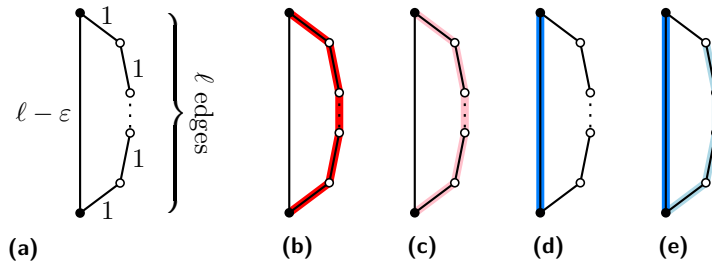
Proof. We give the proof for an arbitrary number of levels in the full version [1]; here we treat only the case $k = 2$. We have $\text{OPT} = 2\text{OPT}_1 + \text{OPT}_2$. Let TOP be the total cost produced by the top-down approach, and let $\text{TOP}_\ell = c(E_{\text{TOP},\ell} \setminus E_{\text{TOP},\ell-1})$ denote the cost of edges on level ℓ but not level $\ell - 1$, produced by the top-down approach, so that $\text{TOP} = 2\text{TOP}_1 + \text{TOP}_2$. Define BOT and BOT_ℓ analogously. Let MIN_ℓ denote the cost of a minimum Steiner tree over terminals T_ℓ with original edge weights, independently of other levels, so that $\text{MIN}_1 \leq \text{MIN}_2 \leq \dots \leq \text{MIN}_k$.

► **Lemma 3.** *The following inequalities relate TOP with OPT:*

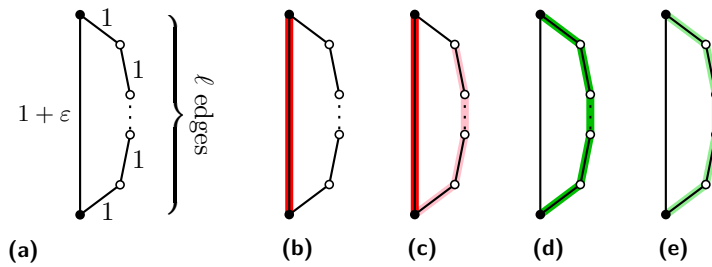
$$\text{TOP}_1 \leq \rho \text{OPT}_1 \tag{1}$$

$$\text{TOP}_2 \leq \rho(\text{OPT}_1 + \text{OPT}_2) \tag{2}$$

Proof. (1) follows from the fact that $E_{\text{TOP},1}$ is a ρ -approximation for ST over T_1 , that is, $\text{TOP}_1 \leq \rho \text{MIN}_1 \leq \rho \text{OPT}_1$. To show (2), note that TOP_2 is at most ρ times the cost (denote MIN'_2) of a minimum Steiner tree over T_2 in the instance obtained by setting $c(e) = 0$ for each $e \in E_{\text{TOP},1}$. Thus, $\text{TOP}_2 \leq \rho \text{MIN}'_2 \leq \rho \text{MIN}_2$. Additionally, since $E_{\text{OPT},2}$ spans T_2 by definition, we have $\text{MIN}_2 \leq \text{OPT}_1 + \text{OPT}_2$, so $\text{TOP}_2 \leq \rho(\text{OPT}_1 + \text{OPT}_2)$ as desired. ◀



■ **Figure 2** The analysis of the top-down approach (light and dark blue) is asymptotically tight for two layers (optimal solution in light and dark red). The dark vertices and edges are on the top level, the white vertices and light edges are on the bottom level. Here, $\text{OPT} = 2\ell$, while $\text{TOP} = 2(\ell - \varepsilon) + \ell - 1 = 3\ell - 2\varepsilon - 1$.



■ **Figure 3** The analysis of the bottom-up approach (light and dark green) is asymptotically tight for two layers (optimal solution in light and dark red). Here, $\text{OPT} = \ell + 1 + 2\varepsilon$, while $\text{BOT} = 2\ell$.

Combining (1) and (2), we have $\text{TOP} = 2\text{TOP}_1 + \text{TOP}_2 \leq 3\rho\text{OPT}_1 + \rho\text{OPT}_2 \leq 3\rho\text{OPT}_1 + \frac{3}{2}\rho\text{OPT}_2 = \frac{3}{2}\rho\text{OPT}$, and hence the top-down approach provides a $\frac{3}{2}\rho$ -approximation when $k = 2$. In Fig. 2 we provide an example showing that our analysis is tight for $\rho = 1$.

► **Lemma 4.** *The following inequality relates BOT with OPT:*

$$\text{BOT}_1 + \text{BOT}_2 \leq \rho(\text{OPT}_1 + \text{OPT}_2)$$

Proof. This follows from the fact that $\text{BOT}_1 + \text{BOT}_2 \leq \rho\text{MIN}_2$, and that the tree with cost $\text{OPT}_1 + \text{OPT}_2$ spans T_2 with cost at least MIN_2 . ◀

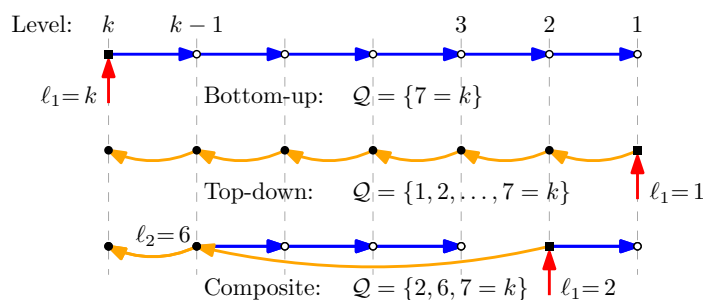
Hence, $\text{BOT} = 2\text{BOT}_1 + \text{BOT}_2 \leq 2(\text{BOT}_1 + \text{BOT}_2) \leq 2\rho(\text{OPT}_1 + \text{OPT}_2) \leq 2\rho(2\text{OPT}_1 + \text{OPT}_2) = 2\rho\text{OPT}$. Again, the approximation ratio of 2 (for $\rho = 1$) is asymptotically tight; see Figure 3.

We show that taking the better of the two solutions returned by the top-down and the bottom-up approach provides a $\frac{4}{3}\rho$ -approximation to MLST for $k = 2$. To prove this, we use the fact that $\min\{x, y\} \leq \alpha x + (1 - \alpha)y$ for any real numbers x, y , and $\alpha \in [0, 1]$. Thus,

$$\begin{aligned} \min\{\text{TOP}, \text{BOT}\} &\leq \alpha(3\rho\text{OPT}_1 + \rho\text{OPT}_2) + (1 - \alpha)(2\rho\text{OPT}_1 + 2\rho\text{OPT}_2) \\ &= (2 + \alpha)\rho\text{OPT}_1 + (2 - \alpha)\rho\text{OPT}_2 \end{aligned}$$

Setting $\alpha = \frac{2}{3}$ gives $\min\{\text{TOP}, \text{BOT}\} \leq \frac{8}{3}\rho\text{OPT}_1 + \frac{4}{3}\rho\text{OPT}_2 = \frac{4}{3}\rho\text{OPT}$. Combining the graphs in Figures 2 and 3, we can show that, asymptotically, the ratio $\frac{4}{3}$ is tight.

For $k > 2$ levels, the inequalities in Lemmas 3 and 4 generalize; we provide the proof in the full version [1]. ◀



■ **Figure 4** Illustration of a composite heuristic for an arbitrary choice of $\mathcal{Q} = \{\ell_1, \ell_2, \dots, \ell_m\}$. Blue arrows pointing right indicate bottom-up propagations (prune E_{ℓ_i} to get $E_{\ell_{i-1}}$). Orange curved arrows pointing left indicate top-down propagations (set to 0 the cost of edges in E_{ℓ_i} when computing $E_{\ell_{i+1}}$). Red arrows indicate where the algorithms starts. Bottom-up and top-down heuristics are special cases with $\mathcal{Q} = \{k\}$, and $\mathcal{Q} = \{1, 2, \dots, k\}$, respectively.

2.2 Composite Algorithm

We describe an approach that generalizes the above approaches in order to obtain a better approximation ratio for $k > 2$ levels. The main idea behind this composite approach is the following: In the top-down approach, we choose a set of edges $E_{\text{TOP},1}$ that spans T_1 , and then propagate this choice to levels $2, \dots, k$ by setting the cost of these edges to 0. On the other hand, in the bottom-up approach, we choose a set of edges $E_{\text{BOT},k}$ that spans T_k , which is propagated to levels $k-1, \dots, 1$. The idea is that for $k > 2$, we can choose a set of intermediate levels and propagate our choices between these levels in a top-down manner, and to the levels lying in between them in a bottom-up manner.

Formally, let $\mathcal{Q} = \{\ell_1, \ell_2, \dots, \ell_m\}$ with $1 \leq \ell_1 < \ell_2 < \dots < \ell_m = k$ be a subset of levels sorted in increasing order. We first compute a Steiner tree $E_{\ell_1} = ST(G, T_{\ell_1})$ for level ℓ_1 , and then use it to construct trees $E_{\ell_{i-1}}, \dots, E_1$ similarly to the bottom-up approach. Then, we set the weights of E_{ℓ_1} to zero (as in the top-down approach) and compute a Steiner tree $E_{\ell_2} = ST(G', T_{\ell_2})$ for level ℓ_2 in the reweighed graph. Again, we can use E_{ℓ_2} to construct the trees $E_{\ell_{i-1}}$ to $E_{\ell_{i+1}}$. Repeating this procedure until spanning $E_{\ell_m} = E_k$ results in a solution to MLST. Note that the top-down and bottom-up heuristics are special cases of this approach, with $\mathcal{Q} = \{1, 2, \dots, k\}$ and $\mathcal{Q} = \{k\}$, respectively. Figure 4 provides an illustration of the propagations in the top-down, in the bottom-up, and in a general heuristic. Let $\text{CMP}(\mathcal{Q})$ be the cost of the MLST returned by the composite approach over some set \mathcal{Q} .

For any choice of \mathcal{Q} , we have $\text{CMP}(\mathcal{Q}) \leq \rho \sum_{i=1}^m (k - \ell_{i-1}) \text{MIN}_{\ell_i}$, with the convention $\ell_0 = 0$. The proof of this claim is similar to that of Lemma 3: when we compute E_{ℓ_1} and propagate its edges to all levels, we incur a cost of at most $\rho k \text{MIN}_{\ell_1}$. When we compute E_{ℓ_2} , we also construct the trees $E_{\ell_{i-1}}, \dots, E_{\ell_{i+1}}$. Using the lower bound $\text{OPT} \geq \sum_{\ell=1}^k \text{MIN}_{\ell}$, we can find an upper bound for the approximation ratio t . Without loss of generality, assume $\sum_{\ell=1}^k \text{MIN}_{\ell} = 1$, so that $\text{OPT} \geq 1$. Also, since all the equations and inequalities scale by ρ , we let $\rho = 1$. Hence, we have

$$t = \frac{\text{CMP}(\mathcal{Q})}{\text{OPT}} \leq \frac{\rho \sum_{i=1}^m (k - \ell_{i-1}) \text{MIN}_{\ell_i}}{\sum_{\ell=1}^k \text{MIN}_{\ell}} = \sum_{i=1}^m (k - \ell_{i-1}) \text{MIN}_{\ell_i}.$$

As observed above, both the top-down and the bottom-up approaches (which, due to Theorem 2, are $\frac{k+1}{2}$ - and k -approximations, respectively) are two of the 2^{k-1} heuristics possible in the composite approach. For the top-down heuristic, $\text{TOP} = \text{CMP}(\{1, 2, \dots, k\}) \leq$

$k\text{MIN}_1 + (k-1)\text{MIN}_2 + \dots + \text{MIN}_k \leq \frac{k+1}{2}$, with equality when $\text{MIN}_1 = \text{MIN}_2 = \dots = \text{MIN}_k = \frac{1}{k}$. For the bottom-up heuristic, $\text{BOT} = \text{CMP}(\{k\}) \leq k\text{MIN}_k \leq k$.

An important choice of \mathcal{Q} is $\mathcal{Q} = \{k-2^q+1 : 0 \leq q \leq q_{\max} = \lfloor \log_2 k \rfloor\}$. For $k = 2^{q_{\max}+1}-1$, the weakest upper bound occurs when $\text{MIN}_1 = \dots = \text{MIN}_{k-2^{q_{\max}}} = 0$ and $\text{MIN}_{k-2^{q_{\max}+1}} = \dots = \text{MIN}_k = 1/2^{q_{\max}}$ resulting in $t \leq \sum_{q=0}^{q_{\max}} 2^{q+1} - 1/2^{q_{\max}} \leq 2^{q_{\max}+2}/2^{q_{\max}} = 4$. Indeed, this choice of \mathcal{Q} produces the 4ρ -approximation (QoS) given by Charikar et al. [7].

When $k = 2$, the only $2^{2-1} = 2$ composite heuristics are top-down and bottom-up (see Section 2.1). For $k \geq 2$, the set $\{1, \dots, k\}$ has 2^{k-1} subsets that contain k , so there are 2^{k-1} different choices of \mathcal{Q} . The composite algorithm executes all of them and picks the solution with minimum cost (denoted CMP):

$$\text{CMP} = \min_{\substack{\mathcal{Q} \subseteq \{1, \dots, k\} \\ k \in \mathcal{Q}}} \text{CMP}(\mathcal{Q}).$$

More generally, for $k \geq 2$, the composite heuristic produces a t -approximation, where t is the largest real number that simultaneously satisfies the 2^{k-1} inequalities

$$t \leq \sum_{i=1}^m (k - \ell_{i-1}) \text{MIN}_{\ell_i},$$

for all subsets $\{\ell_1, \dots, \ell_m\} \subseteq \{1, 2, \dots, k\}$ that contain k and for all choices of $\text{MIN}_1, \dots, \text{MIN}_k$ such that $\text{MIN}_1 \leq \text{MIN}_2 \leq \dots \leq \text{MIN}_k$ and $\sum_{\ell=1}^k \text{MIN}_{\ell} = 1$. The system of 2^{k-1} inequalities can be expressed in matrix form as

$$M_k \mathbf{s} \geq t \cdot \mathbf{1}_{2^{k-1} \times 1},$$

where $\mathbf{s} = [\text{MIN}_1, \text{MIN}_2, \dots, \text{MIN}_k]^T$ and M_k is a $(2^{k-1} \times k)$ -matrix that can be constructed recursively as

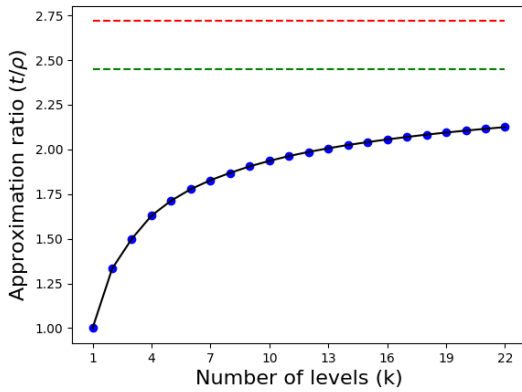
$$M_k = \begin{bmatrix} k \cdot \mathbf{1}_{2^{k-2} \times 1} & M_{k-1} \\ \mathbf{0}_{2^{k-2} \times 1} & P_{k-1} + M_{k-1} \end{bmatrix} \text{ with } P_k = \begin{bmatrix} \mathbf{1}_{2^{k-2} \times 1} & \mathbf{0}_{2^{k-2} \times (k-1)} \\ \mathbf{0}_{2^{k-2} \times 1} & P_{k-1} \end{bmatrix},$$

starting with the 1×1 matrices $M_1 = [1]$ and $P_1 = [1]$. Therefore, for each value of k , we can find the approximation ratio of the composite algorithm by solving a linear program (LP). We summarize our discussion as follows.

► **Theorem 5.** *For any $k = 2, \dots, 22$, the composite algorithm yields a t -approximation to MLST, where the values of t are listed in Figure 5.*

Neglecting the factor ρ for now, the approximation ratio $t = 3/2$ for $k = 3$ is better than the ratio of $(5 + 4\sqrt{2})/7 + \varepsilon \approx 1.5224 + \varepsilon$ guaranteed by Xue et al. [24] for the Euclidean case. (The additive constant ε in their ratio stems from using Arora's PTAS as a subroutine for Euclidean ST, which corresponds to the multiplicative constant ρ for using an ST algorithm as a subroutine for MLST.) Recall that an improvement for $k = 3$ was posed as an open problem by Karpinski et al. [14]. Also, for each of the cases $4 \leq k \leq 22$ our results in Theorem 5 improve the approximation ratios of $e\rho \approx 2.718\rho$ and 2.454ρ guaranteed by Charikar et al. [7] and by Karpinski et al. [14], respectively. On the other hand, our ratios increase with k , while their results hold for every k . The graph of the approximation ratio of the composite algorithm (see Figure 5) for $k = 1, \dots, 22$ suggests that it will stay below 2.454ρ for values of k much larger than 22.

Since the number of heuristics in the composite algorithm grows exponentially with k , it is computationally efficient only for small k . Indeed, for k levels, the composite heuristic requires 2^k ST computations. In the following, we show that we can achieve the same approximation guarantee with at most $2k$ ST computations.



k	t/ρ	k	t/ρ
1	1.000	12	1.986
2	1.333	13	2.007
3	1.500	14	2.025
4	1.630	15	2.041
5	1.713	16	2.056
6	1.778	17	2.070
7	1.828	18	2.083
8	1.869	19	2.094
9	1.905	20	2.106
10	1.936	21	2.116
11	1.963	22	2.125

■ **Figure 5** Approximation ratios for the composite algorithm for $k = 1, \dots, 22$ (blue curve), compared to the ratio $t/\rho = e$ (red dashed line) guaranteed by the algorithm of Charikar et al. [7] and $t/\rho = 2.454$ (green dashed line) guaranteed by the algorithm of Karpinski et al. [14]. The table to the right lists the exact values for the ratio t/ρ .

► **Theorem 6.** *For a given instance of the MLST problem, a specific choice of Q^* can be found through k ST computations for which $\text{CMP}(Q^*)$ is guaranteed the theoretical approximation ratio of the composite heuristic.*

Proof. Given a graph $G = (V, E)$ with cost function c , and terminal sets $T_1 \subset T_2 \subset \dots \subset T_k \subseteq V$, compute a Steiner tree on each level and set $\text{MIN}_\ell = c(\text{ST}(G, T_\ell))$. Since $s = [\text{MIN}_1, \dots, \text{MIN}_k]^T$ is not necessarily the optimal solution to the LP for computing the approximation ratio t , there must be at least one constraint for which $\sum_{i=1}^m (k - \ell_{i-1}) \text{MIN}_{\ell_i} \leq t \sum_{\ell=1}^k \text{MIN}_\ell$. The minimum entry in the vector $M_k s$ corresponds to such a constraint. Let $q \in \{1, \dots, 2^{k-1}\}$ be the index of this entry, and let $Q^* \subseteq \{1, \dots, k\}$ be the index set corresponding to non-zero entries in the q^{th} row of M_k . Then we have $\text{CMP}(Q^*)/\text{OPT} \leq (\sum_{i=1}^m (k - \ell_{i-1}) \text{MIN}_{\ell_i}) / (\sum_{\ell=1}^k \text{MIN}_\ell)$, which yields $\text{CMP}(Q^*) \leq t \cdot \text{OPT}$. ◀

3 Exact Algorithm

Recall the well-known flow formulation for ST [3, 19]. It assumes that the input graph is directed, which we can achieve by simply replacing each undirected edge by two directed edges in opposite directions of the same cost. Recall that T is the set of terminals. Let s be a fixed terminal node, the *source*. Then the ILP formulation for ST is as follows.

$$\begin{aligned}
 & \text{Minimize} && \sum_{(u,v) \in E} c(u,v) \cdot y_{uv} \\
 & \text{subject to} && \sum_{vw \in E} x_{vw} - \sum_{uv \in E} x_{uv} = \begin{cases} |T| - 1 & \text{if } v = s \\ -1 & \text{if } v \in T \setminus \{s\} \\ 0 & \text{else} \end{cases} \quad \text{for } v \in V \\
 & && 0 \leq x_{uv} \leq (|T| - 1) \cdot y_{uv}, \text{ and } y_{uv} \in \{0, 1\}
 \end{aligned}$$

15:10 Multi-Level Steiner Trees

In MLST, if an edge is selected on level ℓ , it must be selected on all levels below, that is, on levels $\ell + 1, \dots, k$. The flow variables x_{uv}^ℓ and the binary variables y_{uv}^ℓ are now additionally indexed by the level ℓ . The intended meaning of $y_{uv}^\ell = 1$ is that edge uv is selected on level ℓ . We constrain the graph on level ℓ to be a subgraph of the graph on level $\ell + 1$ as follows:

$$y_{uv}^{\ell+1} \geq y_{uv}^\ell \quad \text{for } \ell \in \{1, 2, \dots, k-1\} \text{ and } (u, v) \in E$$

We also modify the objective function in the natural way:

$$\text{Minimize } \sum_{\ell=1}^k \sum_{uv \in E} c(u, v) \cdot y_{uv}^\ell$$

In the full version of our paper [1], we provide two further ILP formulations of MLST. Among the three, the above formulation uses the smallest number of constraints.

4 Experimental Results

Graph Data Synthesis. The graph data we used in our experiment are synthesized from graph generative models. In particular, we used four random network generation models: Erdős–Renyi [10], random geometric [18], Watts–Strogatz [22], and Barabási–Albert [4]. These networks are very well studied in the literature [17].

In each graph instance, we assign integer edge weights $c(e)$ randomly and uniformly between 1 and 10 inclusive. Even though the generated graphs are almost surely connected, it is possible to get a disconnected graph. Therefore, in our experiment, we only use connected graphs and discard the rest. Computational challenges of solving an ILP limit the size of the graphs to a few hundred in practice.

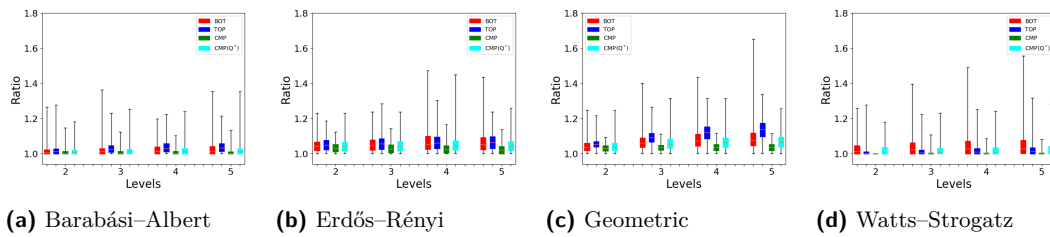
Selection of Levels and Terminal Nodes. For each generated graph, we generated MLST instances with $k = 2, 3, 4, 5$ levels. We adopted two strategies for selecting the terminals on the k levels: *linear* vs. *exponential*. In the linear scenario, we select the terminals on each level by randomly sampling $\lfloor |V|(\ell + 1)/(k + 1) \rfloor$ nodes on level ℓ so that $|T_{\ell+1}| - |T_\ell| \approx |T_\ell| - |T_{\ell-1}|$. In the exponential case, we select the terminals at each layer by sampling uniformly randomly $\lfloor |V|/2^{k-\ell} \rfloor$ nodes so that $|T_{i+1}|/|T_i| \approx |T_i|/|T_{i-1}|$.

To summarize, a single instance of an input to MLST is characterized by four parameters: network generation model $\text{NGM} \in \{\text{ER}, \text{RG}, \text{WS}, \text{BA}\}$, number of nodes $|V|$, number of levels k , and the terminal selection method $\text{TSM} \in \{\text{LINEAR}, \text{EXPONENTIAL}\}$.

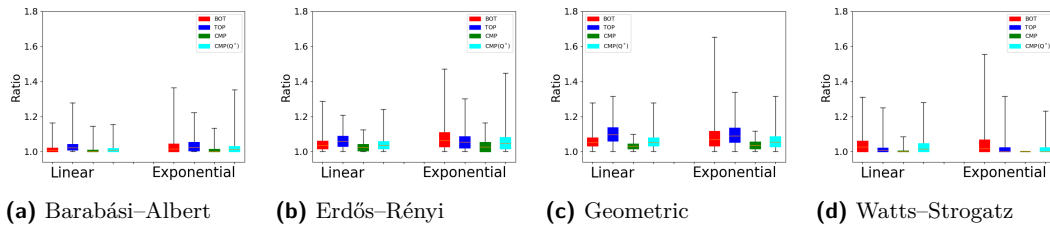
Algorithms and Outputs. We implemented the bottom-up, top-down, and composite heuristics described in Section 2 and the simple 4ρ -approximation algorithm by Charikar et al. [7] for the QoS Multicast Tree problem, all in Python.

For evaluating the heuristics, we also implemented the ILP described in Section 3 using CPLEX 12.6.2 as ILP solver. We distributed the experiment on a high performance computer (HPC) into multiple tasks. A single task performs the computation of 5 to 50 graphs. The number of graphs varies because for smaller graphs we can combine more graphs in a single task. For larger graphs, however, the time limit for a single task is not enough if the number of graphs is too large.

For each instance of MLST, we compute the costs of the MLST from the ILP solution (OPT), the bottom-up solution (BOT), the top-down solution (TOP), the composite heuristic



■ **Figure 6** Performance of BOT, TOP, CMP, and $CMP(Q^*)$ w.r.t. the number k of levels.



■ **Figure 7** Performance of BOT, TOP, CMP, and $CMP(Q^*)$ w.r.t. the terminal selection method.

(CMP), the guaranteed performance heuristic ($CMP(Q^*)$) heuristic, and the simple 4ρ -approximate Quality-of-Service heuristic (QoS) of Charikar et al. [7]. For the ST computation we used the 2-approximation algorithm of Gilbert and Pollak [11].

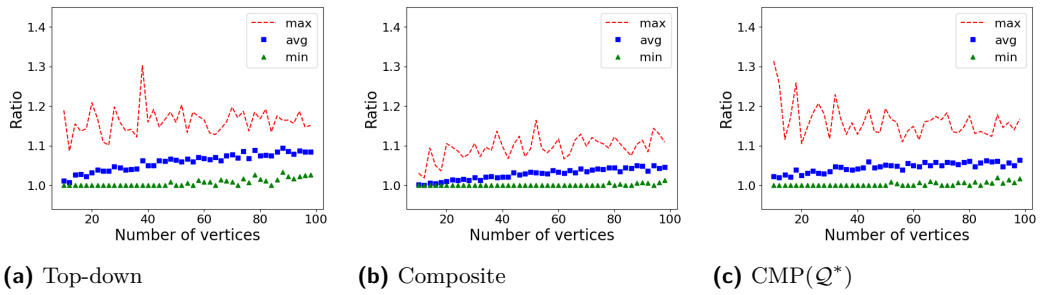
After completing the experiment, we compared the results of the heuristics with exact solutions. We show the performance ratio APP/OPT for each heuristic, and how they depend on parameters of the experiment setup. For example, we investigate how the performance ratio changes as $|V|$ increases. Since each instance of the experiment setup involves randomness at different steps, we generated 5 instances for any fixed setup (e.g., Geometric graph, $|V| = 100$, 5 levels, linear terminal selection).

We did not compare the running times of our implementations in detail since our Python code is not optimized in this respect. As a rough measure, however, we list the number of Steiner tree computations performed by each algorithm in the worst case – BOT: 1, TOP: k , CMP: 2^k , $CMP(Q^*)$: $2k$, and QoS: k .

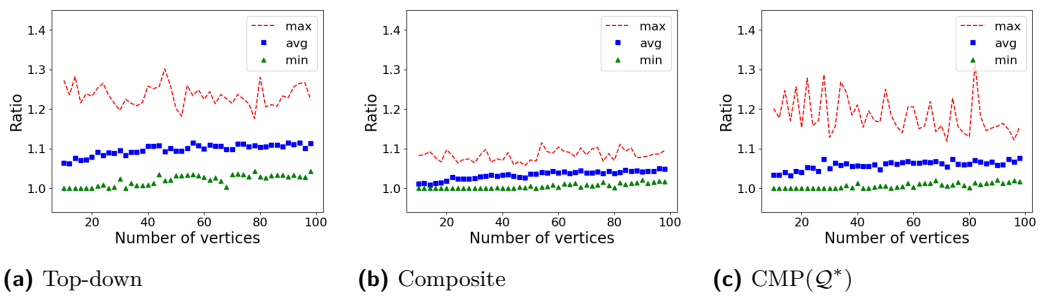
Results. First, we examined how the performance of the heuristics compared with the exact solution as the number of the levels k changed. In our experiments, k varies between 2 and 5. We show the results using box plots in Figure 6. As expected, the performance of the heuristics gets slightly worse as k increases. The bottom-up approach had the worst performance, while the composite heuristic performed very well in practice.

Second, we examined how the performance of the heuristics compared with the exact solution for different terminal selection methods, either LINEAR or EXPONENTIAL. We show the results using box plots in Figure 7. Overall, the heuristics performed worse when the sizes of the terminal sets decrease exponentially.

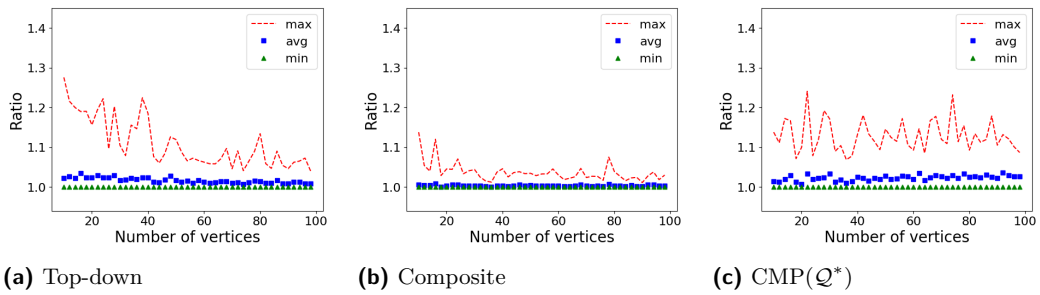
Third, we investigated how the heuristics perform with respect to the graph size $|V|$, for each of the network models ER, RG, WS, and BA; see Figures 8–11. Note that the y-axes of the graphs in these figures have a different scale than the graphs in Figures 6 and 7. Since several instances share the same network size, we show minimum, maximum, and mean values. Overall, the performance of the heuristics slightly deteriorated as $|V|$ increased. Due to lack of space, we omit the bottom-up heuristic here, which tends to be comparable to or slightly worse than the top-down heuristic. Again, the composite heuristic yielded the



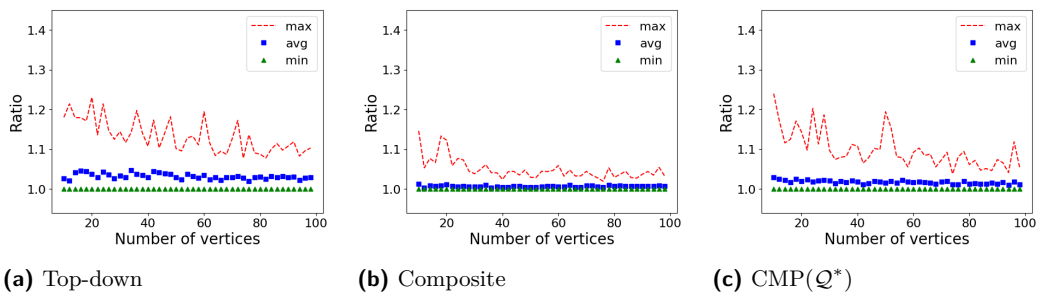
■ **Figure 8** Performance of TOP, CMP, and $CMP(Q^*)$ on Erdős–Rényi graphs.



■ **Figure 9** Performance of TOP, CMP, and $CMP(Q^*)$ on Geometric graphs.



■ **Figure 10** Performance of TOP, CMP, and $CMP(Q^*)$ on Watts–Strogatz graphs.



■ **Figure 11** Performance of TOP, CMP, and $CMP(Q^*)$ on Barabási–Albert graphs.

best performance; top-down and $\text{CMP}(\mathcal{Q}^*)$ were comparable. Data for the other heuristics is available in the full version [1].

5 Conclusions

We presented several heuristics for the MLST problem and analyzed them both theoretically and experimentally. Natural open problems include determining inapproximability results for MLST, determining a closed-form expression for the approximation ratio of the composite heuristic (Section 2.2), and generalizing the notion of multi-level graphs to related problems (such as the node-weighted Steiner tree problem).

References

- 1 Reyan Ahmed, Patrizio Angelini, Faryad Darabi Sahneh, Alon Efrat, David Glickenstein, Martin Gronemann, Niklas Heinsohn, Stephen G. Kobourov, Richard Spence, Joseph Watkins, and Alexander Wolff. Multi-level Steiner trees, 2018. [arXiv:1804.02627](https://arxiv.org/abs/1804.02627).
- 2 Sanjeev Arora. Polynomial time approximation schemes for Euclidean Traveling Salesman and other geometric problems. *J. ACM*, 45(5):753–782, 1998. doi:10.1145/290179.290180.
- 3 Anantaram Balakrishnan, Thomas L. Magnanti, and Prakash Mirchandani. Modeling and heuristic worst-case performance analysis of the two-level network design problem. *Management Sci.*, 40(7):846–867, 1994. doi:10.1287/mnsc.40.7.846.
- 4 Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *science*, 286(5439):509–512, 1999.
- 5 Marshall Bern and Paul Plassmann. The Steiner problem with edge lengths 1 and 2. *Inform. Process. Lett.*, 32(4):171–176, 1989. doi:10.1016/0020-0190(89)90039-2.
- 6 Jaroslav Byrka, Fabrizio Grandoni, Thomas Rothvoß, and Laura Sanità. Steiner tree approximation via iterative randomized rounding. *J. ACM*, 60(1):6:1–6:33, 2013. doi:10.1145/2432622.2432628.
- 7 Moses Charikar, Joseph (Seffi) Naor, and Baruch Schieber. Resource optimization in QoS multicast routing of real-time multimedia. *IEEE/ACM Trans. Networking*, 12(2):340–348, 2004. doi:10.1109/TNET.2004.826288.
- 8 Miroslav Chlebík and Janka Chlebíková. The Steiner tree problem on graphs: Inapproximability results. *Theoret. Comput. Sci.*, 406(3):207–214, 2008. doi:10.1016/j.tcs.2008.06.046.
- 9 Julia Chuzhoy, Anupam Gupta, Joseph (Seffi) Naor, and Amitabh Sinha. On the inapproximability of some network design problems. *ACM Trans. Algorithms*, 4(2):23:1–23:17, 2008. doi:10.1145/1361192.1361200.
- 10 Paul Erdős and Alfréd Rényi. On random graphs I. *Publicationes Mathematicae (Debrecen)*, 6:290–297, 1959.
- 11 Edgar N. Gilbert and Henry O. Pollak. Steiner minimal trees. *SIAM J. Appl. Math.*, 16(1):1–29, 1968. doi:10.1137/0116001.
- 12 Mathias Hauptmann and Marek Karpinski (eds.). A compendium on Steiner tree problems, 2015. URL: <http://theory.cs.uni-bonn.de/info5/steinercompendium/>.
- 13 Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972. doi:10.1007/978-1-4684-2001-2_9.
- 14 Marek Karpinski, Ion I. Mandoiu, Alexander Olshevsky, and Alexander Zelikovsky. Improved approximation algorithms for the quality of service multicast tree problem. *Algorithmica*, 42(2):109–120, 2005. doi:10.1007/s00453-004-1133-y.

- 15 Prakash Mirchandani. The multi-tier tree problem. *INFORMS J. Comput.*, 8(3):202–218, 1996.
- 16 Joseph S. B. Mitchell. Guillotine subdivisions approximate polygonal subdivisions: A simple polynomial-time approximation scheme for geometric TSP, k -MST, and related problems. *SIAM J. Comput.*, 28(4):1298–1309, 1999. doi:10.1137/S0097539796309764.
- 17 Mark E.J. Newman. The structure and function of complex networks. *SIAM Review*, 45(2):167–256, 2003. doi:10.1137/S003614450342480.
- 18 Mathew Penrose. *Random geometric graphs*, volume 5 of *Oxford Studies in Probability*. Oxford University Press, 2003.
- 19 Tobias Polzin and Siavash Vahdati Daneshmand. A comparison of Steiner tree relaxations. *Discrete Appl. Math.*, 112(1):241–261, 2001. doi:10.1016/S0166-218X(00)00318-8.
- 20 Hans Jürgen Prömel and Angelika Steger. *The Steiner Tree Problem*. Vieweg and Teubner Verlag, 2002.
- 21 Gabriel Robins and Alexander Zelikovsky. Tighter bounds for graph Steiner tree approximation. *SIAM J. Discrete Math.*, 19(1):122–134, 2005. doi:10.1137/S0895480101393155.
- 22 Duncan J. Watts and Steven H. Strogatz. Collective dynamics of ‘small-world’ networks. *Nature*, 393(6684):440–442, 1998. doi:10.1038/30918.
- 23 Pawel Winter. Steiner problem in networks: A survey. *Networks*, 17(2):129–167, 1987. doi:10.1002/net.3230170203.
- 24 Guoliang Xue, Guo-Hui Lin, and Ding-Zhu Du. Grade of service Steiner minimum trees in the Euclidean plane. *Algorithmica*, 31(4):479–500, 2001. doi:10.1007/s00453-001-0050-6.

Dictionary Matching in Elastic-Degenerate Texts with Applications in Searching VCF Files On-line

Solon P. Pissis

Department of Informatics, King's College London, London, UK
solon.pissis@kcl.ac.uk

Ahmad Retha¹

Department of Informatics, King's College London, London, UK
ahmad.retha@kcl.ac.uk

Abstract

An *elastic-degenerate string* is a sequence of n sets of strings of total length N . It has been introduced to represent multiple sequence alignments of closely-related sequences in a compact form. For a standard pattern of length m , pattern matching in an elastic-degenerate text can be solved on-line in time $\mathcal{O}(nm^2 + N)$ with pre-processing time and space $\mathcal{O}(m)$ (Grossi et al., CPM 2017). A fast bit-vector algorithm requiring time $\mathcal{O}(N \cdot \lceil \frac{m}{w} \rceil)$ with pre-processing time and space $\mathcal{O}(m \cdot \lceil \frac{m}{w} \rceil)$, where w is the size of the computer word, was also presented. In this paper we consider the same problem for a set of patterns of total length M . A straightforward generalization of the existing bit-vector algorithm would require time $\mathcal{O}(N \cdot \lceil \frac{M}{w} \rceil)$ with pre-processing time and space $\mathcal{O}(M \cdot \lceil \frac{M}{w} \rceil)$, which is prohibitive in practice. We present a new on-line $\mathcal{O}(N \cdot \lceil \frac{M}{w} \rceil)$ -time algorithm with pre-processing time and space $\mathcal{O}(M)$. We present experimental results using both synthetic and real data demonstrating the performance of the algorithm. We further demonstrate a real application of our algorithm in a pipeline for discovery and verification of *minimal absent words* (MAWs) in the human genome showing that a significant number of previously discovered MAWs are in fact false-positives when a population's variants are considered.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases on-line algorithms, algorithms on strings, dictionary matching, elastic-degenerate string, Variant Call Format

Digital Object Identifier 10.4230/LIPIcs.SEA.2018.16

Acknowledgements We warmly thank Panagiotis Charalampopoulos (King's College London), Roberto Grossi (University of Pisa) and Nadia Pisanti (University of Pisa) for their insightful comments.

1 Introduction

A set of closely-related sequences can be represented in different ways to reduce its size and improve search performance. DNA sequences of the same species or closely-related species can be combined into a *pan-genome* [17, 24, 13, 20], the result of a *multiple sequence alignment* (MSA) of these sequences. Most regions in the DNA sequences are in consensus but they exhibit differences at some positions consisting of letter substitutions, insertions or deletions. Various data structures have been proposed for storing pan-genomes [8, 2, 22, 24] – many

¹ AR is supported by the Graduate Teaching Assistant scheme of the Department of Informatics at King's College London



designs are realized from observing the result of aligning the related sequences in an MSA fashion. Consider the following example.

```

ATGCAACGGGTA--TTTAA
ATGCAACGGGTATATTTAA
ATGCACCTGG----TTTAA

```

The first five columns of the MSA all match, so when this is compacted, it creates a deterministic segment with a single string: ATGCA. The next letter in the MSA is A for the first and second sequence but C for the third, making it the site of a variant. The second variant in the example is similarly a single-base substitution variant. But the third variant site consists of insertions and deletions. These are represented differently in state-of-the-art data structures for the purpose of storage and indexing for on-line pattern searches.

Some researchers choose to represent the variants in the combined sequence in the form of *De Bruijn* graphs [24] or specialized implementations of the data structure – *Variation Graphs* [22]. Other researchers use *Trie*-based data structures such as the *Bloom Filter Trie* in [8] or compressed *Suffix Tree* data structures [2]. All of these are indexes constructed mainly for fast searching of patterns; and thus much effort has gone into solving this *off-line* version of the problem through pre-processing the set of similar sequences [9, 14, 22, 15]. Less research to-date has gone into the *on-line* version of this problem [19, 12, 7, 3, 18].

A text-based, on-line searchable representation for a set of similar sequences was suggested in [12], namely, the notion of *elastic-degenerate string* (ED string). Specifically, aligned sequences can be compacted into one sequence made up of *deterministic* and *non-deterministic* (or *degenerate*) segments. Deterministic segments contain letters that are in perfect conformity among the different sequences, meaning all the letters match, while degenerate segments mark polymorphic sites containing substitutions, insertions or deletions.

The MSA above can be converted into an ED string by representing the first deterministic segment with the single string ATGCA, and representing the next segment, which happens to be degenerate, with the set {A,C}. An empty string marker ε is used to represent a deletion, as is done for the third degenerate site in the MSA, consisting of the set {TA,TATA, ε }. The resulting ED string of the MSA above example is as follows.

$$\tilde{T} = \{ \text{ATGCA} \} \cdot \left\{ \begin{array}{c} \text{A} \\ \text{C} \end{array} \right\} \cdot \{ \text{C} \} \cdot \left\{ \begin{array}{c} \text{G} \\ \text{T} \end{array} \right\} \cdot \{ \text{GG} \} \cdot \left\{ \begin{array}{c} \text{TA} \\ \text{TATA} \\ \varepsilon \end{array} \right\} \cdot \{ \text{TTTTA} \}$$

The motivation for solving the on-line version of the problem is to remove the burden of building disk-based indexes or rebuilding them with every update in the sequences. Indexes are often cumbersome, take a lot of time and space to build, and require lots of disk space to be stored. Their usage carries the assumption that the data is static or changes very infrequently. Solutions to the on-line version can be beneficial for a number of reasons: (a) efficient on-line solutions can be used in combination with partial indexes as practical trade-offs; (b) efficient on-line solutions for exact pattern matching can be applied for fast average-case approximate pattern matching, similar to standard strings; (c) on-line solutions can be useful when one wants to search for a few patterns in many ED texts.

Variant Call Format (VCF) is a file format that has become the standard way of storing variants for pan-genomes and in next-generation sequencing. These specially-formatted, often compressed text files, in combination with a reference genome, are able to document all insertions, deletions and substitutions that occur in a population. While it is possible – for the purpose of searching for patterns – to recreate the genome of all individuals (samples) in the pan-genome as deterministic strings, it is very impractical and requires a lot of

processing power and disk space. It also defeats the purpose of storing the information in the VCF format in the first place. We were motivated to make it possible to do on-line searching of one or more patterns in a pan-genome without extracting sample sequences. Our solution takes the position of variants in the VCF file and encodes them as degenerate segments of an ED text. In this way, we are able to search a pan-genome on-line given the reference sequence and the associated VCF files. We created a tool EDSO (available at <https://github.com/webmasterar/edso>) for creating ED files which are directly searchable.

Our Contributions. Our focus in this paper is extending existing solutions for exact on-line pattern matching in ED texts, specifically, the algorithm of [7] through adding the ability to search for *multiple* patterns simultaneously. Grossi et al. [7] presented an algorithm requiring time $\mathcal{O}(N \cdot \lceil \frac{m}{w} \rceil)$ with pre-processing time and space $\mathcal{O}(m \cdot \lceil \frac{m}{w} \rceil)$, where m is the length of the *single* pattern and w is the size of the computer word. A straightforward generalization of the existing bit-vector algorithm for a set of patterns of total length M would require time $\mathcal{O}(N \cdot \lceil \frac{M}{w} \rceil)$ with pre-processing time and space $\mathcal{O}(M \cdot \lceil \frac{M}{w} \rceil)$, which is prohibitive in practice. In this paper we present a new algorithm requiring time $\mathcal{O}(N \cdot \lceil \frac{M}{w} \rceil)$ with pre-processing time and space $\mathcal{O}(M)$. We present experimental results using both synthetic and real data demonstrating the performance of our algorithm. Finally, we present a real application of our algorithm's use as part of identifying and verifying *minimal absent words* (MAWs) in the *Homo sapiens* pan-genome with data in the VCF taken from the 1000 Genomes Project [23]. Specifically, we show that a significant number of previously discovered MAWs are in fact false-positives when a population of genomes is considered.

2 Definitions and Notation

2.1 Strings

We begin with a few definitions from [4]. An *alphabet* Σ is a non-empty finite set of letters of size $\sigma = |\Sigma|$. A (*deterministic*) *string* on a given alphabet Σ is a finite sequence of letters of Σ . For this work, we assume that the alphabet is fixed, i.e. $\sigma = \mathcal{O}(1)$. The *length* of a string x is denoted by $|x|$. For two positions i and j on x , we denote by $x[i..j] = x[i]..x[j]$ the *factor* (sometimes called *substring*) of x that starts at position i and ends at position j (it is empty if $j < i$), and by ε we denote the *empty string*. The set of all strings on an alphabet Σ (including the empty string ε) is denoted by Σ^* . For any string $y = uv$, where u and v are strings, if $u = \varepsilon$ then x is a *prefix* of y . Similarly, if $v = \varepsilon$ then x is a *suffix* of y . If u and v are non-empty strings, we call x an *infix* of y . We say that x is a *proper factor* of y if x is a factor (resp. prefix/suffix) of y distinct from y .

We say that the string x is an *absent word* of string y if x does not occur in y . We consider absent words of length at least 2 only. An absent word x of length m , $m \geq 2$, of y is *minimal* if and only if all its proper factors occur in y . This is equivalent to saying that a minimal absent word (MAW) of y is of the form aub , $a, b \in \Sigma, u \in \Sigma^*$, such that au and ub are factors of y but aub is not.

► **Example 1.** Let $y = \text{ABAACA}$. Its factors of lengths 1 and 2 are A, B, C, AA, AB, AC, BA, and CA. The set of MAWs of y is obtained by combining the aforementioned factors: $\{\text{BB, BC, CB, CC, AAA, AAB, BAB, BAC, CAA, CAB, CAC}\}$.

2.2 Elastic-Degenerate Strings

An *elastic-degenerate string* (ED string) $\tilde{X} = \tilde{X}[0]\tilde{X}[1]\dots\tilde{X}[n-1]$, of length n , on an alphabet Σ , is a finite sequence of n *degenerate letters*. Every *degenerate letter* $\tilde{X}[i]$, for all $0 \leq i < n$, is a non-empty set of strings $\tilde{X}[i][j]$, with $0 \leq j < |\tilde{X}[i]|$, where each $\tilde{X}[i][j]$ is a deterministic string on Σ . The *total size* of \tilde{X} is defined as

$$N = \sum_{i=0}^{n-1} \sum_{j=0}^{|\tilde{X}[i]|-1} |\tilde{X}[i][j]|.$$

Only for the purpose of computing N , $|\varepsilon| = 1$. We remark that, for an ED string \tilde{X} , the size and the length are two distinct concepts.

We say that a string y *matches* an ED string $\tilde{X} = \tilde{X}[0]\dots\tilde{X}[m'-1]$ of length $m' > 1$, denoted by $y \approx \tilde{X}$, if and only if string y can be decomposed into $y_0\dots y_{m'-1}$, $y_i \in \Sigma^*$, such that:

1. there exists a string $s \in \tilde{X}[0]$ such that a suffix of s is $y_0 \neq \varepsilon$;
2. if $m' > 2$, there exists $s \in \tilde{X}[i]$, for all $1 \leq i \leq m' - 2$, such that $s = y_i$;
3. there exists a string $s \in \tilde{X}[m' - 1]$ such that a prefix of s is $y_{m'-1} \neq \varepsilon$.

Note that, in the above definition, we require that both y_0 and $y_{m'-1}$ are non-empty to avoid spurious matches at the beginning or end of an occurrence. A string y is said to have an *occurrence* ending at position j in an ED string \tilde{T} if there exist $i < j$ such that $\tilde{T}[i]\dots\tilde{T}[j] \approx y$, or, if there exists $s \in \tilde{T}[j]$ such that y occurs in s .

► **Example 2.** Suppose we have a pattern $p = \text{ACACA}$ of length $m = 5$ and an ED string \tilde{T} of length $n = 6$ and total size $N = 18$; the first occurrence of p starts at position 1 and ends at position 2 of \tilde{T} ; and the second one starts at position 2 and ends at position 4.

$$\tilde{T} = \{ \text{C} \} \cdot \left\{ \begin{array}{c} \text{A} \\ \text{C} \end{array} \right\} \cdot \left\{ \begin{array}{c} \text{AC} \\ \text{ACC} \\ \text{CACA} \end{array} \right\} \cdot \left\{ \begin{array}{c} \text{C} \\ \varepsilon \end{array} \right\} \cdot \left\{ \begin{array}{c} \text{A} \\ \text{AC} \end{array} \right\} \cdot \{ \text{C} \}$$

We are now in a position to formally define the main problem of this paper.

MULTIPLE ELASTIC-DEGENERATE STRING MATCHING (*MEDSM*)
Input: A set P of strings of total length M and an ED string \tilde{T} of length n and total size N .
Output: All pairs (p, j) : an occurrence of string $p \in P$ ends at position j in \tilde{T} .

3 Algorithmic Toolbox

3.1 Suffix Tree

Let x be a string of length $n > 0$. The *suffix tree* \mathcal{ST}_x of string x is a compacted trie representing all suffixes of x . The nodes of the trie which become nodes of the suffix tree are called *explicit* nodes, while the other nodes are called *implicit*. Each edge of the suffix tree can be viewed as an upward maximal path of implicit nodes starting with an explicit node. Moreover, each node belongs to a unique path of that kind. Thus, each node of the trie can be represented in the suffix tree by the edge it belongs to and an index within the corresponding path. The label of an edge is its first letter. We let $\mathcal{L}(v)$ denote the *path-label* of a node v , i.e. the concatenation of the edge labels along the path from the root to v . We say that v is

path-labelled $\mathcal{L}(v)$. Additionally, $\mathcal{D}(v) = |\mathcal{L}(v)|$ is used to denote the *string-depth* of node v . Node v is a *terminal* node if its path-label is a suffix of x , that is, $\mathcal{L}(v) = x[i..n-1]$ for some $0 \leq i < n$; here v is also labelled with index i . It should be clear that each factor of x is uniquely represented by either an explicit or an implicit node of \mathcal{ST}_x . Once \mathcal{ST}_x is constructed, it can be traversed in a depth-first manner to compute $\mathcal{D}(v)$ for each node v .

► **Fact 3** ([5, 4]). *Given a string x of length n , \mathcal{ST}_x can be constructed in time and space $\mathcal{O}(n)$. Finding all Occ_p occurrences of a string p of length m in x can be performed in time $\mathcal{O}(m + \text{Occ}_p)$ using \mathcal{ST}_x .*

3.2 The Shift-And Algorithm

The *Shift-And* algorithm is an exact pattern matching algorithm that takes advantage of the parallelism of bitwise operations performed on a computer word [16]. It works by simulating a *Nondeterministic Finite Automaton* (NFA) and uses bit-level operations to simultaneously update the states of the NFA in a single CPU cycle. This offers speed-ups bounded by the number of bits in a computer word w , where, typically, on modern computer architectures, we have that $w = 64$. For short patterns, where $m = \mathcal{O}(w)$, searching a text of length n runs in $\mathcal{O}(n)$ time but for longer patterns, the search takes $\mathcal{O}(n \cdot \lceil \frac{m}{w} \rceil)$ time. The pre-processing time of the algorithm is $\mathcal{O}(\sigma \cdot \lceil \frac{m}{w} \rceil + m) = \mathcal{O}(m)$ thus making it suitable for small-sized alphabets and short patterns. The *Shift-And* algorithm can be easily generalized for a set of patterns; it is then known as the *Multiple Shift-And* algorithm [16].

► **Fact 4** ([16]). *Given a set P of strings of total length M , a string x of length N , and a computer word of size w , finding all occurrences of the patterns in P in x takes time $\mathcal{O}(N \cdot \lceil \frac{M}{w} \rceil)$ after pre-processing time $\mathcal{O}(M)$.*

In addition to using the *Shift-And* algorithm for searching whole patterns, we take advantage of its ability to compute suffix/prefix overlaps between string $s \in \tilde{T}[i]$, where $\tilde{T}[i]$ is the i th set arriving on-line, and the set P of patterns we are searching for. Given s , we can find all the prefixes of a pattern $p \in P$ of length m by searching $s[|s| - m + 1..|s| - 1]$ if $|s| \geq m$ or $s[0..|s| - 1]$, otherwise. This updates the NFA to mark any prefixes of the patterns occurring as suffixes of s . We store the states in a bit vector which we bitwise-OR to itself for each string $s \in \tilde{T}[i]$. The resulting state bit vector memorizes all the prefixes ending at position i in \tilde{T} , and we then use *Shift-And* in the searching stage of the algorithm to search the $(i + 1)$ th set to find a suffix, that either completes the match for some pattern in P , or further extends some prefix of a pattern, in which case the algorithm updates the search state. We summarize the above description in the following fact.

► **Fact 5.** *Given a set S of strings of total length $N = \sum_{s \in S} |s|$ and a set P of strings of total length $M = \sum_{p \in P} |p|$, computing the suffix/prefix overlaps of S and P can be done in time $\mathcal{O}(N \cdot \lceil \frac{M}{w} \rceil)$.*

4 The Multi-EDSM Algorithm

4.1 Our Data Structure

We define the following auxiliary problem of independent interest.

OCCURRENCES VECTOR DATA STRUCTURE (*OccVec*)

Input: A string x of length n .

Query: Given a string α on-line, return a pointer to a bit vector B , with $B[i] = 1$ if and only if α occurs at starting position i of x , and otherwise $B[i] = 0$.

In what follows, we show the following lemma.

► **Lemma 6.** *Given a parameter $1 \leq \tau \leq \lceil n/w \rceil$, a data structure of size $\mathcal{O}(\lceil n/\tau \rceil \cdot \lceil n/w \rceil)$ can be constructed in time and space $\mathcal{O}(\lceil n/\tau \rceil \cdot \lceil n/w \rceil)$ answering *OccVec* queries in time $\mathcal{O}(|\alpha| + \tau)$.*

Let us denote the data structure of Lemma 6 over string x by OCC-VECTOR_x . Observe that, if we set $\tau = 1$, we essentially have the $\mathcal{O}(n \cdot \lceil n/w \rceil)$ -sized data structure proposed by Grossi et al in [7], which is constructible in time and space $\mathcal{O}(n \cdot \lceil n/w \rceil)$.

Construction. We start by constructing the suffix tree \mathcal{ST}_x of x . By Fact 3, this can be done in time and space $\mathcal{O}(n)$.

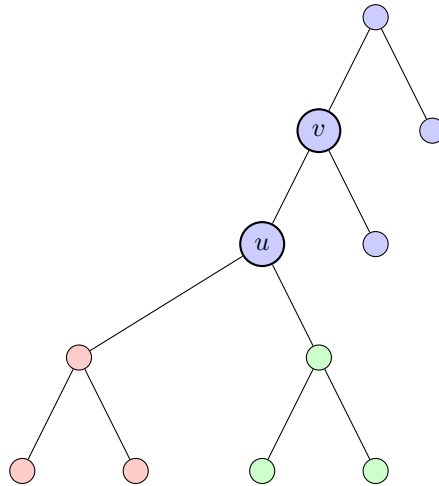
We next convert \mathcal{ST}_x to a binary tree using a standard procedure (see [6], for instance). We process each explicit node of \mathcal{ST}_x with out-degree $k > 2$ as follows. Let v be a node with children u_1, \dots, u_k , and $k \geq 3$. We replace v with $k - 1$ new nodes v_1, \dots, v_{k-1} ; make u_1 and u_2 be the left and right children of v_1 , respectively; and for each $\ell = 2, \dots, k - 1$, we make $v_{\ell-1}$ and $u_{\ell+1}$ be the left and right children of v_ℓ , respectively. If v is not the root of \mathcal{ST}_x then we set the parent of v_{k-1} to be the parent of v ; otherwise, v_{k-1} is the root. This procedure can at most double the size of \mathcal{ST}_x so still is in $\mathcal{O}(n)$. For clarity of presentation, in what follows, we use \mathcal{ST}_x to refer to the resulting binary tree. (Note that we can keep a copy of the original \mathcal{ST}_x and a pointer for each node from the original to its binary version).

We next rely on the classic notion of *micro-macro* tree decomposition [1]. We apply this decomposition on (the binary version of) \mathcal{ST}_x . Let τ be some input parameter, $1 \leq \tau \leq \lceil n/w \rceil$. We decompose \mathcal{ST}_x in $\mathcal{O}(n/\tau)$ disjoint subgraphs called *micro trees*. Each micro tree is of size *at most* τ and contains *at most two* boundary nodes that are adjacent to nodes in other micro trees. The topmost of these boundary nodes is the *root* of the whole micro tree, and the other one is called the *bottom* boundary node. Such a decomposition is always possible and can be found in time $\mathcal{O}(n)$ (see [1] for more details).

For each boundary node v of a micro tree, we store a bit vector b_v , where $b_v[i] = 1$, if the terminal node representing the i th suffix of x is a descendant of v , and otherwise $b_v[i] = 0$. For the bottom boundary node v of micro tree t , b_v can be computed by merging the bit vectors from the roots of the micro trees that are adjacent to v and then add manually the terminal nodes within t for the root boundary node of t . By the above description and the fact that we have $\mathcal{O}(n/\tau)$ micro trees, the total size of OCC-VECTOR_x , and therefore the time to construct it, are bounded by $\mathcal{O}(n + \lceil n/\tau \rceil \cdot \lceil n/w \rceil) = \mathcal{O}(\lceil n/\tau \rceil \cdot \lceil n/w \rceil)$, for $1 \leq \tau \leq \lceil n/w \rceil$. OCC-VECTOR_x also includes a linked-list \mathcal{L} of integers from $[0, n - 1]$ used to maintain the bit vectors when a new query arrives. This completes the construction.

Querying. Given a pattern α , we spell the pattern from the root of \mathcal{ST}_x until we reach the last explicit node v . This takes time $\mathcal{O}(|\alpha|)$ for constant-sized alphabets. There are then two cases to consider:

- If v is a boundary node of some micro tree, we simply return a pointer to b_v ; this takes constant time.
- If v is not a boundary node, we first need to obtain the starting positions (labels) of all terminal nodes of the micro tree in the subtree rooted at v , and set the corresponding



■ **Figure 1** Three micro trees: the topmost in light blue and the bottommost ones in light red and light green. If the query reaches node v , then the terminal node in the subtree rooted at v in the topmost micro tree must be combined with the bit vector stored in the bottom boundary node u .

bits on in the bit vector b_u , where u is the bottom boundary node of the micro tree. We then return a pointer to the updated b_u . (If no such node u exists, we simply set these bits on in an empty bit vector.) The whole process takes time $\mathcal{O}(\tau)$: traverse the micro tree and set the bits on. We also need to store these starting positions in \mathcal{L} for the next query. In the beginning of the next query we will need to set the bits at these indices off from b_u and empty \mathcal{L} . This maintenance cost requires time $\mathcal{O}(\tau)$ due to the size of the micro trees and so we charge it to this query. Inspect Figure 1 in this regard.

Hence any query requires time $\mathcal{O}(|\alpha| + \tau)$. By the above description we ultimately arrive at Lemma 6.

4.2 Pre-Processing Stage

The pre-processing stage of our algorithm consists in pre-processing the set P of our patterns. We view the set P as the concatenation of its elements to form a new string y of length M .

The first step is the pre-processing of the pattern set P of combined length M in the *Multiple Shift-And* algorithm [16]. We create σ bit vectors of size $\lceil \frac{M}{w} \rceil$ and for each letter a in Σ we set $I_a[i] = 1$ if $y[i] = a$. Therefore this first step requires time and extra space $\mathcal{O}(M + \sigma \cdot \lceil \frac{M}{w} \rceil) = \mathcal{O}(M)$, for constant-sized alphabets.

The second step is a simple application of Lemma 6 over string y of length M with the additional steps of filtering out non-infix positions and subtracting 1 from the index positions. This makes it possible to maintain infix extensions during the search stage. We build the OCC-VECTOR $_P$ data structure by setting $\tau = \lceil \frac{M}{w} \rceil$, thus restricting its size and construction time to $\mathcal{O}(M)$, resulting also in $\mathcal{O}(|\alpha| + \lceil \frac{M}{w} \rceil)$ query time for the search stage.

The total time and space for the pre-processing stage are thus in $\mathcal{O}(M)$.

4.3 On-line Searching Stage

After the pre-processing stage, every degenerate letter S of text \tilde{T} can be searched one after the other in an on-line manner by passing them to the SEARCH function (see Algorithm 1). We maintain the state of the search in bit vector B in between searches and use temporary

Algorithm 1 MULTIPLE ELASTIC-DEGENERATE STRING MATCHING search function.

```

1: procedure SEARCH( $S$ )
2:   if isFirstSegment( $S$ ) then
3:     for  $s \in S$  do
4:       if  $|s| \geq m_{\min}$  and  $s \neq \varepsilon$  then
5:          $B_3 \leftarrow 0$ 
6:         MULTI-SHIFT-AND-SEARCH( $s, B_3$ )
7:         Report any matches found
8:        $B \leftarrow \text{OVERLAPS}(S)$ 
9:     else
10:       $B_1 \leftarrow \text{OVERLAPS}(S)$ 
11:      if  $\varepsilon \in S$  then
12:         $B_1 \leftarrow B_1 \mid B$ 
13:      for  $s \in S$  and  $s \neq \varepsilon$  do
14:         $\triangleright$  Pattern suffix completion / full pattern searching
15:         $B_2 \leftarrow B$ 
16:        MULTI-SHIFT-AND-SEARCH( $s, B_2$ )
17:        Report any matches found
18:         $\triangleright$  Maintain valid infix positions
19:        if  $|s| \leq m_{\max} - 2$  then
20:           $B_3 \leftarrow B \ \& \ \text{OCC-VECTOR}_P(s)$ 
21:           $B_1 \leftarrow B_1 \mid \text{LEFT-SHIFT}(B_3, |s|)$ 
22:       $B \leftarrow B_1$ 

```

bit vectors B_1, B_2 and B_3 to update the state during processing. By m_{\min} and m_{\max} we denote the length of the shortest and longest patterns in pattern set P , respectively.

In the first degenerate letter, for every string $s \in S$ of length $|s| \geq m_{\min}$, we call the MULTI-SHIFT-AND-SEARCH function with a fresh state to find any patterns that occur in s . In any case, the OVERLAPS function is called for computing the suffix/prefix *overlaps* between every string $s \in S$ and the set P of patterns we are searching for. The function essentially memorises the prefixes starting in the current segment using B . By Facts 4 and 5, lines 2-8 require $\mathcal{O}(|S| \cdot \lceil \frac{M}{w} \rceil)$ time. For subsequent degenerate letters, we use the state of B from the previously searched letter to continue the search with MULTI-SHIFT-AND-SEARCH. This time the function is called regardless of the length of s because it is used to find whole patterns as well as prefixes of patterns that began in the previously searched letters and whose suffixes end in the current letter. By Facts 4 and 5, this requires $\mathcal{O}(|S| \cdot \lceil \frac{M}{w} \rceil)$ time.

Then we consider how to handle infixes (see line 19). We only need to process strings short enough to be considered as infixes of a pattern and we query them with the OCC-VECTOR $_P$ data structure to mark the positions where each infix starts. Querying OCC-VECTOR $_P$ for a string s requires $\mathcal{O}(|s| + \tau) = \mathcal{O}(|s| + \lceil \frac{M}{w} \rceil)$ time by Lemma 6. We bitwise-AND the bit vector it returns with B to maintain only the states started or continued from the previous letter. On the next line, we use the LEFT-SHIFT function to update the position of the bits to reflect the state of the search while ensuring that the bits are not shifted past the end of a pattern. What bits remain as 1s are bitwise-ORed with B_1 to update the state to maintain partial search states. This takes time $\mathcal{O}(\lceil \frac{M}{w} \rceil)$.

The final line of the algorithm saves the final search state of the segment to bit vector B , ready for the next on-line letter to be sent to the SEARCH function. A full illustrative example of the search stage is provided below.

With the above description we arrive at the main result of this paper.

► **Theorem 7.** *Algorithm Multi-EDSM solves the MEDSM problem in an on-line manner in time $\mathcal{O}(N \cdot \lceil \frac{M}{w} \rceil)$. Algorithm Multi-EDSM requires pre-processing time and space $\mathcal{O}(M)$.*

Notably, our algorithm improves on the pre-processing time and space of [7] by a factor of $\mathcal{O}(\lceil \frac{m}{w} \rceil)$; namely, it improves on the algorithm for a single pattern (EDSM problem [7]).

► **Corollary 8.** *Algorithm Multi-EDSM solves the EDSM problem in an on-line manner in time $\mathcal{O}(N \cdot \lceil \frac{m}{w} \rceil)$. Algorithm Multi-EDSM requires pre-processing time and space $\mathcal{O}(m)$.*

► **Example 9.** Given an ED string \tilde{T} as shown below, we wish to search for the patterns in set $P = \{\text{ATAT}, \text{TAGA}\}$ of total length $M = 8$. A collection I of σ bit vectors are created during the Shift-And pre-processing stage marking the positions of the letters in Σ of the concatenated patterns. We also build OCC-VECTOR $_P$. Note that the bit vectors are read from right to left and recall that OCC-VECTOR $_P$ subtracts 1 from the index positions.

$$\tilde{T} = \left\{ \begin{array}{c} \text{AT} \\ \underline{\text{A}} \end{array} \right\} \cdot \left\{ \begin{array}{c} \text{AT} \\ \underline{\text{TA}} \end{array} \right\} \cdot \left\{ \begin{array}{c} \text{T T T A} \\ \underline{\text{A G A}} \end{array} \right\}$$

I_A	1010 0101
I_C	0000 0000
I_G	0100 0000
I_T	0001 1010

The algorithm starts with $\tilde{T}[0]$, skips the Shift-And search of the strings in the segment because they are too short, and computes bit vector $B = 0001\ 0011 = \text{OVERLAPS}(\tilde{T}[0])$ on line 8. The OVERLAPS function memorises the prefixes starting in the current segment using B .

Then, the next segment is considered; on line 10 we compute the bit vector $B_1 = 0011\ 0011 = \text{OVERLAPS}(\tilde{T}[1])$. The next step is to check each string $s \in \tilde{T}[1]$ and after doing MULTI-SHIFT-AND-SEARCH($\underline{\text{AT}}$, B_2) with state B (from $\tilde{T}[0]$) it discovers a match for $P[0]$ and reports it. This time the Shift-And search function is called regardless of the length of s because it is used to find whole patterns as well as suffixes of patterns that began in the previous segments. The function completes the suffix by matching $\underline{\text{AT}}$ at positions $I_A[2]$ and $I_T[3]$ to spell out $P[0]$.

Then we consider how to handle infixes on line 19. We only need to process strings short enough to be considered as infixes of a pattern and we query them with the OCC-VECTOR $_P$ data structure to mark the positions where each infix starts. Calling OCC-VECTOR $_P(\underline{\text{AT}})$ finds infix position 2 and returns 0000 0010. So $B_3 = 0000\ 0010 = B \ \& \ 0000\ 0010$, thus maintaining the active search state. The LEFT-SHIFT function does bitwise left-shift of B_3 by $|s|$ positions whilst ensuring no 1s end up at or beyond the ending position of each pattern in the set. What bits remain as 1s are bitwise-ORed with B_1 to update the state to maintain partial search states, but in this case, the 1 is shifted too far and B_1 remains unchanged.

In the next iteration, we do MULTI-SHIFT-AND-SEARCH($\underline{\text{TA}}$, B_2) yielding no match. Then we call OCC-VECTOR $_P(\underline{\text{TA}})$ which finds infix position 1 and returns 0000 0001 which we bitwise-AND with B to take $B_3 = 0000\ 0001$. Then LEFT-SHIFT is performed on B_3 and bitwise-ORing its result with B_1 makes $B_1 = 0011\ 0111$ because no boundaries are crossed. Having searched all the strings in the segment, we save the state of the search to B on line 22 and observe that we have thus far matched $\underline{\text{ATA}}$ of $P[0]$ spanning across $\tilde{T}[0]$ and $\tilde{T}[1]$.

Now we go ahead and search the final segment $\tilde{T}[2]$ of our example. We compute $B_1 = 0010\ 0001 = \text{OVERLAPS}(\tilde{T}[2])$ first and then by calling MULTI-SHIFT-AND-SEARCH($\underline{\text{T T T A}}$, B_2) with the state $B_2 = B$ from the previous segment, we complete the partial match and report finding $P[0]$ in this segment. Calling this function again for the next string in the segment, MULTI-SHIFT-AND-SEARCH($\underline{\text{A G A}}$, B_2) also completes the suffix for $P[1]$, and we report it.

On the very last line of the algorithm we save the final search state of the segment to bit vector B , ready for the next on-line letter to be sent to the SEARCH function.

5 Experiments

Multi-EDSM code was written in C++ and compiled with g++ version 5.4.0 at optimization level 3 (-O3) and scripts were written in Python 2.7. A simpler version of the OCC-VECTOR_P data structure has been implemented in which the user can set a memory limit to be used by the program and then the analogous number of bit vectors are stored in the explicit nodes of the suffix tree that are closer to the root. This is because the vast majority of the variation strings to be queried in real datasets are rather short.

The following experiments were conducted on a desktop computer using one core of Intel® Core™ i7-2600S CPU at 2.8GHz and 8GB of RAM under 64-bit GNU/Linux. The Multi-EDSM application code and generated experimental datasets and scripts are licensed under the GNU General Public License (GPL-3.0); they are all freely available from <https://github.com/webmasterar/multi-edsm>.

5.1 Time Performance

To test the time performance of Multi-EDSM application, we devised two experiments. In both experiments we set the memory limit of Multi-EDSM to 4GB for the program to use as much memory as necessary.

In Figure 2a we measured the processing time when searching a randomly-generated fixed ED text of length $n = 1600000$ ($N = 5700610$) over the DNA alphabet with randomly-generated pattern sets doubling in length from length $M = 1600$ to $M = 102400$. The text searched contains 10% degenerate segments and within each degenerate segment there are 2 to 10 random strings of length 1 to 10 each. Similarly, in Figure 2b we measured the processing time when searching randomly generated ED texts doubling in size from $N = 100000$ to $N = 6400000$ with a fixed set of randomly-generated patterns of length $M = 3000$. The text had 10% degenerate positions representing single-base substitutions. (Uniform distribution has been used in all randomizations.)

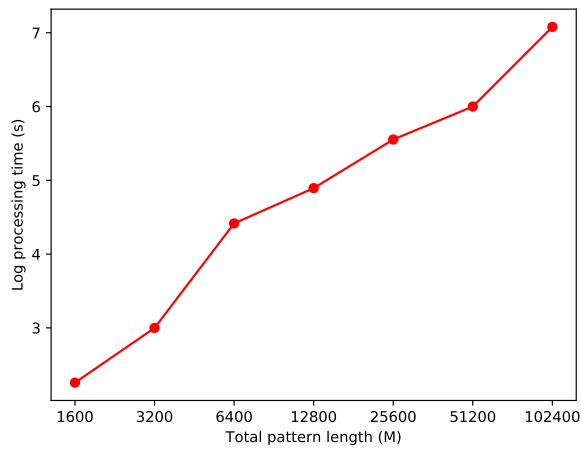
As can be seen from the charts, the change in performance, whether it be an increase in the patterns total length M or the total text size N , causes a linear increase in processing time, which conforms to our theoretical findings (Theorem 7).

5.2 Comparison to EDSM-BV

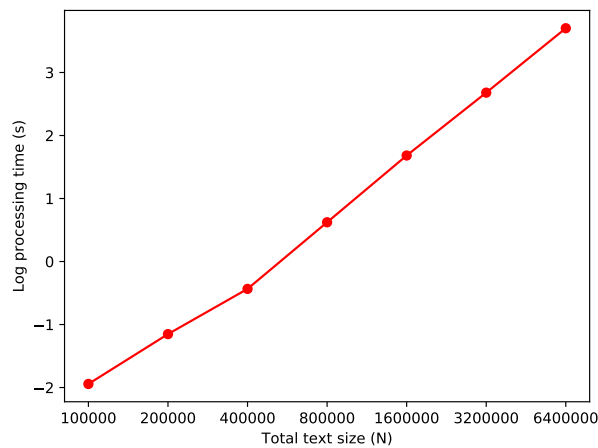
To test the performance of Multi-EDSM compared against EDSM-BV [7] we searched the same randomly-generated ED text of length $n = 1600000$ ($N = 5700610$) mentioned above against multiple sets of randomly-generated patterns of length 40 each. First we tested with a single pattern of length 40 and then the number of patterns in each set was incremented in steps of 10 from 10 to 100 patterns of length 40 each. EDSM-BV is only able to search one pattern at a time so we searched each pattern in a set individually and summed-up the total time spent. We see from the chart in Figure 3 that for a single pattern the EDSM-BV algorithm is fast, but it becomes immediately clear that for dictionary searching of even a handful of patterns, Multi-EDSM becomes orders of magnitude faster.

5.3 Real Application

We designed a three stage pipeline for determining the validity of MAWs discovered in the human genome. We obtained the GRCh37 chromosome sequences from *Ensembl* [10]



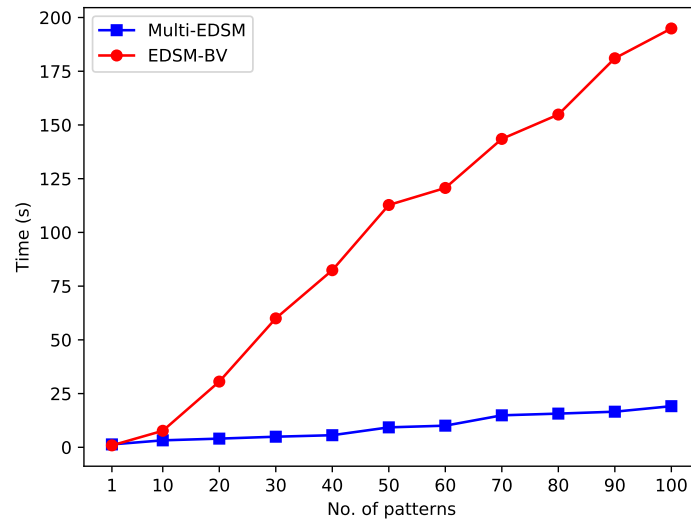
(a) Processing time with increasing patterns total length on a fixed ED text of length $n = 1600000$ ($N = 5700610$).



(b) Processing time with increasing ED total text size on a fixed set of patterns of total length $M = 3000$.

■ **Figure 2** Time performance of Multi-EDSM.

and the associated phase 3 VCF files were obtained from the *1000 Genomes project* [23] on-line repositories. Phase 3 of the 1000 Genomes project contains 2504 individuals from 26 populations whose variants are encoded in VCF files. The first step in the pipeline was to use *emMAW* [11] to extract MAWs of length between 3 and 12 from a concatenated file of the 22 autosomes and two sex chromosomes. The filtered list of MAWs contained 161565 patterns with combined length $M = 1937789$. The second stage was to use the tool *EDSO* to combine the reference chromosome sequences and the variants in the VCF files into searchable ED string (EDS) format files. Then Multi-EDSM was used to search each of the EDS files against the MAW patterns to produce a list of tuples marking the position of the match and pattern id. The final stage was validation. A script was written to validate each match, verifying a MAW genuinely exists for an individual at the identified position in the chromosome. We have found that for each chromosome more than half the MAWs discovered



■ **Figure 3** Elapsed-time comparison of Multi-EDSM and EDSM-BV with an ED text of total size $N = 5700610$ and sets of randomly-generated patterns of length 40 each.

■ **Table 1** Ebola sequences *absent* from human reference genome but *present* in human pan-genome.

id	sequence	position	variant id	sample id	ethnicity
RAW1	TTTCGCCGACT	6:93819539	rs569027564	NA18606	Han Chinese
RAW2	TACGCCCTATCG	1:74075482	rs578167440	HG02146	Peruvian
RAW3	CCTACGCGCAA	15:71003880	rs564150197	HG03598	Bengali

using Multi-EDSM exist in one or more individuals and are subsequently disqualified, i.e. they are not really MAWs. Our compiled summary of the results show that 73% of MAWs were disqualified, leaving only 43722 of 161565 potential MAWs remaining.

We applied the results of this pipeline to validate the work of Silva et. al. in [21] to identify MAWs in Ebola virus genomes that are absent from the human genome. They identify three MAWs of length 12, called RAW1, RAW2 and RAW3, that are not present in the *reference* human genome sequence. These MAWs could be used to verify an Ebola infection in a patient. However, we discovered from our results that each of the three MAWs do in fact occur in one or more individuals in the 1000 Genomes dataset, although indeed they are not that common. This means that they cannot be used as perfect identifiers for Ebola virus infection and perhaps longer unique MAWs should be used instead. In Table 1 we list the position of the discovery of each MAW as well as information about the variant and the id of one individual they occur in.

6 Final Remarks

It would be relevant [9] to investigate the problem of dictionary matching in elastic-degenerate texts under the Hamming or edit distance models (see [3] for a single pattern).

References

- 1 Stephen Alstrup, Jens P. Secher, and Maz Spork. Optimal on-line decremental connectivity in trees. *Inf. Process. Lett.*, 64(4):161–164, 1997. doi:10.1016/S0020-0190(97)00170-1.
- 2 Uwe Baier, Timo Beller, and Enno Ohlebusch. Graphical pan-genome analysis with compressed suffix trees and the Burrows-Wheeler transform. *Bioinformatics*, 32(4):497–504, 2016.
- 3 Giulia Bernardini, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Pattern matching on elastic-degenerate text with errors. In Gabriele Fici, Marinella Sciortino, and Rossano Venturini, editors, *String Processing and Information Retrieval - 24th International Symposium, SPIRE 2017, Palermo, Italy, September 26-29, 2017, Proceedings*, volume 10508 of *Lecture Notes in Computer Science*, pages 74–90. Springer, 2017. doi:10.1007/978-3-319-67428-5_7.
- 4 M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007.
- 5 Martin Farach. Optimal suffix tree construction with large alphabets. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*, pages 137–143. IEEE Computer Society, 1997. URL: <http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=5208>, doi:10.1109/SFCS.1997.646102.
- 6 Travis Gagie, Danny Hermelin, Gad M. Landau, and Oren Weimann. Binary jumbled pattern matching on trees and tree-like structures. *Algorithmica*, 73(3):571–588, 2015. doi:10.1007/s00453-014-9957-6.
- 7 Roberto Grossi, Costas S. Iliopoulos, Chang Liu, Nadia Pisanti, Solon P. Pissis, Ahmad Retha, Giovanna Rosone, Fatima Vayani, and Luca Versari. On-Line Pattern Matching on Similar Texts. In Juha Kärkkäinen, Jakub Radoszewski, and Wojciech Rytter, editors, *28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017)*, volume 78 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:14, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.CPM.2017.9.
- 8 Guillaume Holley, Roland Wittler, and Jens Stoye. Bloom Filter Trie: an alignment-free and reference-free data structure for pan-genome storage. *Algorithms for Molecular Biology*, 11:3, 2016.
- 9 Lin Huang, Victoria Popic, and Serafim Batzoglou. Short read alignment with populations of genomes. *Bioinformatics*, 29(13):361–370, 2013.
- 10 T. Hubbard, D. Barker, E. Birney, G. Cameron, Y. Chen, L. Clark, T. Cox, J. Cuff, V. Curwen, T. Down, R. Durbin, E. Eyras, J. Gilbert, M. Hammond, L. Huminiacki, A. Kasprzyk, H. Lehvaslaiho, P. Lijnzaad, C. Melsopp, E. Mongin, R. Pettett, M. Pocock, S. Potter, A. Rust, E. Schmidt, S. Searle, G. Slater, J. Smith, W. Spooner, A. Stabenau, J. Stalker, E. Stupka, A. Ureta-Vidal, I. Vastrik, and M. Clamp. The Ensembl genome database project. *Nucleic Acids Research*, 30(1):38–41, 2002. doi:10.1093/nar/30.1.38.
- 11 Alice Héliou, Solon P. Pissis, and Simon J. Puglisi. emMAW: computing minimal absent words in external memory. *Bioinformatics*, 33(17):2746–2749, 2017. doi:10.1093/bioinformatics/btx209.
- 12 Costas S. Iliopoulos, Ritu Kundu, and Solon P. Pissis. Efficient pattern matching in elastic-degenerate texts. In Frank Drewes, Carlos Martín-Vide, and Bianca Truthe, editors, *Language and Automata Theory and Applications - 11th International Conference, LATA 2017, Umeå, Sweden, March 6-9, 2017, Proceedings*, volume 10168 of *Lecture Notes in Computer Science*, pages 131–142, 2017. doi:10.1007/978-3-319-53733-7_9.
- 13 Paul Julian Kersey, James E. Allen, Irina Armean, Sanjay Boddu, Bruce J. Bolt, Denise Carvalho-Silva, Mikkel Christensen, Paul Davis, Lee J. Falin, Christoph Grabmueller,

- Jay C. Humphrey, Arnaud Kerhornou, Julia Khobova, Naveen K. Aranganathan, Nicholas Langridge, Ernesto Lowy, Mark D. McDowall, Uma Maheswari, Michael Nuhn, Chuang Kee Ong, Bert Overduin, Michael Paulini, Helder Pedro, Emily Perry, Giulietta Spudich, Electra Tapanari, Brandon Walts, Gareth Williams, Marcela K. Tello-Ruiz, Joshua C. Stein, Sharon Wei, Doreen Ware, Daniel M. Bolser, Kevin L. Howe, Eugene Kulesha, Daniel Lawson, Gareth Maslen, and Daniel M. Staines. Ensembl genomes 2016: more genomes, more complexity. *Nucleic Acids Research*, 44(Database-Issue):574–580, 2016.
- 14 Sorina Maciuca, Carlos del Ojo Elias, Gil McVean, and Zamin Iqbal. A natural encoding of genetic variation in a Burrows-Wheeler transform to enable mapping and genome inference. In Martin C. Frith and Christian Nørgaard Storm Pedersen, editors, *Algorithms in Bioinformatics - 16th International Workshop, WABI 2016, Aarhus, Denmark, August 22-24, 2016. Proceedings*, volume 9838 of *Lecture Notes in Computer Science*, pages 222–233. Springer, 2016. doi:10.1007/978-3-319-43681-4_18.
- 15 Joong Chae Na, Hyunjoon Kim, Heejin Park, Thierry Lecroq, Martine Léonard, Laurent Mouchard, and Kunsoo Park. FM-index of alignment: A compressed index for similar strings. *Theor. Comput. Sci.*, 638:159–170, 2016.
- 16 Gonzalo Navarro and Mathieu Raffinot. *Flexible Pattern Matching in Strings: Practical On-line Search Algorithms for Texts and Biological Sequences*. Cambridge University Press, 2002.
- 17 Ngan Nguyen, Glenn Hickey, Daniel R. Zerbino, Brian J. Raney, Dent Earl, Joel Armstrong, W. James Kent, David Haussler, and Benedict Paten. Building a pan-genome reference for a population. *Journal of Computational Biology*, 22(5):387–401, 2015.
- 18 Nadia Ben Nsira, Mourad Elloumi, and Thierry Lecroq. On-line string matching in highly similar DNA sequences. *Mathematics in Computer Science*, 11(2):113–126, 2017. doi:10.1007/s11786-016-0280-2.
- 19 Nadia Ben Nsira, Thierry Lecroq, and Mourad Elloumi. A fast Boyer-Moore type pattern matching algorithm for highly similar sequences. *IJDMB*, 13(3):266–288, 2015. doi:10.1504/IJDMB.2015.072101.
- 20 Siavash Sheikhezadeh, M. Eric Schranz, Mehmet Akdel, Dick de Ridder, and Sandra Smit. Pantools: representation, storage and exploration of pan-genomic data. *Bioinformatics*, 32(17):487–493, 2016.
- 21 Raquel M. Silva, Diogo Pratas, Luísa Castro, Armando J. Pinho, and Paulo J. S. G. Ferreira. Three minimal sequences found in Ebola virus genomes and absent from human DNA. *Bioinformatics*, 31(15):2421–2425, 2015. doi:10.1093/bioinformatics/btv189.
- 22 Jouni Sirén. Indexing variation graphs. In Sándor P. Fekete and Vijaya Ramachandran, editors, *Proceedings of the Nineteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2017, Barcelona, Spain, Hotel Porta Fira, January 17-18, 2017.*, pages 13–27. SIAM, 2017. doi:10.1137/1.9781611974768.2.
- 23 The 1000 Genomes Project Consortium. A global reference for human genetic variation. *Nature*, 526(7571):68–74, 2015.
- 24 The Computational Pan-Genomics Consortium. Computational pan-genomics: status, promises and challenges. *Briefings in Bioinformatics*, pages 1–18, 2016.

Fast matching statistics in small space

Djamal Belazzougui

DTISI-CERIST
Algiers, Algeria.

Fabio Cunial

MPI-CBG and CSBD
Dresden, Germany.
cunial@mpi-cbg.de

Olgert Denas

Adobe Inc.
San Jose, California, USA.

Abstract

Computing the matching statistics of a string S with respect to a string T on an alphabet of size σ is a fundamental primitive for a number of large-scale string analysis applications, including the comparison of entire genomes, for which space is a pressing issue. This paper takes from theory to practice an existing algorithm that uses just $O(|T| \log \sigma)$ bits of space, and that computes a compact encoding of the matching statistics array in $O(|S| \log \sigma)$ time. The techniques used to speed up the algorithm are of general interest, since they optimize queries on the existence of a Weiner link from a node of the suffix tree, and parent operations after unsuccessful Weiner links. Thus, they can be applied to other matching statistics algorithms, as well as to any suffix tree traversal that relies on such calls. Some of our optimizations yield a matching statistics implementation that is up to three times faster than a plain version of the algorithm, depending on the similarity between S and T . In genomic datasets of practical significance we achieve speedups of up to 1.8, but our fastest implementations take on average twice the time of an existing code based on the LCP array. The key advantage is that our implementations need between one half and one fifth of the competitor's memory, and they approach comparable running times when S and T are very similar.

2012 ACM Subject Classification Theory of computation \rightarrow Data structures design and analysis, Theory of computation \rightarrow Pattern matching

Keywords and phrases Matching statistics, maximal repeat, Burrows-Wheeler transform, wavelet tree, suffix tree topology

Digital Object Identifier 10.4230/LIPIcs.SEA.2018.17

Supplement Material Source code at https://github.com/odenas/indexed_ms.

Acknowledgements We thank German Tischler for help with adapting the implementation in [15]. We thank Oscar Gonzalez and Peter Steinbach for help setting up the experiments on the MPI-CBG cluster.

1 Introduction

The *matching statistics* of a string S , called the *query*, with respect to another string T , called the *text*, is an array $\text{MS}_{S,T}[1..|S|]$ such that $\text{MS}_{S,T}[i]$ is the length of the longest prefix of $S[i..|S|]$ that occurs in T . MS is almost as old as the suffix tree itself, with applications that include file transmission [26], the detection of sequencing errors and single-nucleotide



© Djamal Belazzougui, Fabio Cunial, and Olgert Denas;
licensed under Creative Commons License CC-BY

17th International Symposium on Experimental Algorithms (SEA 2018).

Editor: Gianlorenzo D'Angelo; Article No. 17; pp. 17:1–17:14



Leibniz International Proceedings in Informatics

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

variations in read collections [17], the computation of substring kernels between two sequences by indexing just one of them [15, 22], and the construction of whole-genome phylogenies [5, 25] using the average value of MS as a proxy for the cross-entropy of random sources [6]. The effectiveness of matching statistics in alignment-free phylogenetics has also motivated variants that allow for a user-specified number of mismatches (see e.g. [1, 11, 18, 23, 24] and references therein). Despite the typical input being of genomic scale, however, few of these methods are explicitly designed for space efficiency.

Computing $MS_{S,T}$ is a classical problem in string processing. In most practical cases, a large number of queries S are asked to a constant T , motivating the construction of an index on T . The textbook solution scans S from left to right while traversing suffix links and child links in the *suffix tree* of T , spending a total $O(|S| \log \sigma)$ time¹. Symmetrically, S can be scanned from right to left, while taking Weiner links and parent links in the (compressed) suffix tree of T [15]. Both ways require access to the string depth of a suffix tree node, which can be encoded implicitly in $\Theta(|T| \log |T|)$ bits of total space in the *longest common prefix* (LCP) array of T , and decoded in constant time per node using range-minimum queries on the array [15]. In some practical datasets, most LCP values are small and can be encoded in less than one byte. Alternatively, each depth value can be extracted from a compressed suffix array. If such array is encoded in $O(|T| \log \sigma)$ bits, the best known time complexity for this operation is $O(\log^\epsilon |T|)$, which is often fast enough in practice [19]. Asymptotic complexity increases when the compressed suffix array is encoded in $|T| \log \sigma + o(|T|) + O(\min(\sigma, |T| \log \sigma))$ bits or in $|T| \log \sigma(1 + o(1)) + O(\min(\sigma, |T| \log \sigma))$ bits. A third approach does not require access to string depths at all, and achieves $O(|T| \log \sigma)$ bits of peak space while keeping running time to $O(|S| \log \sigma)$ given the indexes [3]. Such asymptotic performance is attractive for datasets, like whole genomes or collections of genomes, with large overall size and long repeats. The algorithm in [3] has an additional feature that is useful for large datasets: it does not output $MS_{S,T}$ itself, which takes $|S| \log |S|$ bits, but an encoding that takes just $2|S|$ bits, and that allows one to later retrieve $MS[i]$ in constant time for any i , using just $o(|S|)$ more bits.

This paper studies a number of practical variants of the algorithm described in [3], which use the same data structures as in the original paper, but improve on its speed. As customary, since indexing T is performed only once and requires standard data structures, we disregard index construction and focus just on the time for computing $MS_{S,T}$ given the indexes.

2 Preliminaries

2.1 Strings

Let $\Sigma = [1..\sigma]$ be an integer alphabet, let $\# = 0 \notin \Sigma$ be a separator, and let $T = [1..\sigma]^{n-1}\#$ be a string. Given a string $W \in [1..\sigma]^k$, we call the *reverse of W* the string \overline{W} obtained by reading W from right to left. For a string $W \in [1..\sigma]^k\#$ we abuse notation, denoting by \overline{W} the string $\overline{W}[1..k]\#$. We call *repeat* a substring W of T that occurs more than once in T , and we call *left-extensions* (respectively, *right-extensions*) of W the set of distinct characters that precede (respectively, follow) the occurrences of W in T . A repeat W is *left-maximal* (respectively, *right-maximal*) iff it has more than one left-extension (respectively, right-extension). A *maximal repeat* of T is a repeat that is both left- and right-maximal.

¹ Here σ denotes the size of the alphabet from which strings S and T are drawn. We assume throughout this paper that σ is at most polynomial in $\max(|T|, |S|)$.

For reasons of space we assume the reader to be familiar with the notion of *suffix tree* $\text{ST}_T = (V, E)$ of T , which we do not define here. We denote by $\ell(v)$ the string label of node $v \in V$. It is well known that a substring W of T is right-maximal iff $W = \ell(v)$ for some internal node v of the suffix tree. It is also known that the maximal repeats of T form a subset of the internal nodes of ST_T that is closed under ancestor operation, in the sense that if the label of an internal node v is a maximal repeat of T , then the labels of all the ancestors of v are also maximal repeats of T . We assume the reader to be familiar with the notion of *suffix link* connecting a node v with $\ell(v) = aW$ for some $a \in [0..\sigma]$, to a node w with $\ell(w) = W$. Here we just recall that inverting the direction of all suffix links yields the so-called *explicit Weiner links*. Given an internal node v of ST_T and a symbol $a \in [0..\sigma]$, it might happen that string $a\ell(v)$ does occur in T , but that it is not right-maximal, i.e. it is not the label of any internal node: all such left extensions of internal nodes that end in the middle of an edge or at a leaf are called *implicit Weiner links*. Note that an internal node of ST_T can have more than one outgoing Weiner link, and that all such Weiner links have distinct labels.

We assume the reader to be familiar also with the Burrows-Wheeler transform of T (denoted BWT_T in what follows), including the notions of lexicographic interval of the label of a node of ST_T , backward step, rank and select queries on bitvectors, wavelet trees, and rank queries on wavelet trees. Finally, we call *suffix tree topology* any data structure that supports the following operations on ST_T : $\text{parent}(\text{id}(v))$, which returns the identifier of the parent of a node v with identifier $\text{id}(v)$; $\text{lca}(\text{id}(u), \text{id}(v))$, which returns the identifier of the lowest common ancestor of nodes u and v ; $\text{leftmostLeaf}(\text{id}(v))$ and $\text{rightmostLeaf}(\text{id}(v))$, which compute the identifier of the leftmost (respectively, rightmost) leaf in the subtree rooted at node v ; $\text{selectLeaf}(i)$, which returns the identifier of the i -th leaf in preorder traversal; $\text{leafRank}(\text{id}(v))$, which computes the number of leaves that occur before leaf v in preorder traversal; $\text{isAncestor}(\text{id}(u), \text{id}(v))$, which tells whether u is an ancestor of v . For brevity, we write just v rather than $\text{id}(v)$ in all such operations in what follows. It is known that the topology of an ordered tree with n nodes can be represented using $2n + o(n)$ bits as a sequence of $2n$ balanced parentheses, and that $2n + o(n)$ more bits suffice to support the operations described above in constant time [12, 21]. We drop subscripts whenever the reference strings are clear from the context.

2.2 Matching statistics in small space

$\text{MS}_{S,T}$ can be represented as a bitvector ms of $2|S|$ or $2|S| - 1$ bits, which is built by appending, for each $i \in [0..|S| - 1]$ in increasing order, $\text{MS}_{S,T}[i] - \text{MS}_{S,T}[i - 1] + 1$ zeros followed by a one [3] ($\text{MS}_{S,T}[-1]$ is set to one for convenience). Since the number of zeros before the i -th one in ms equals $i + \text{MS}_{S,T}[i]$, one can compute $\text{MS}_{S,T}[i]$ for any $i \in [0..|S| - 1]$ using select operations on ms . The algorithm described in [3] computes ms using both a backward and a forward scan over S , and it needs in each scan just BWT_T with rank support, and the topology of ST_T , or just $\text{BWT}_{\bar{T}}$ with rank support, and the topology of $\text{ST}_{\bar{T}}$. The two phases are connected via a bitvector $\text{runs}[1..|T| - 1]$, such that $\text{runs}[i] = 1$ iff $\text{MS}[i] = \text{MS}[i - 1] - 1$, i.e. iff there is no zero between the i -th and the $(i - 1)$ -th ones in ms .

First, we scan S from right to left, using BWT_T with rank support, and the suffix tree topology of T , to determine the runs of consecutive ones in ms . Assume that we know the interval $[i..j]$ in BWT_T that corresponds to substring $W = S[k..k + \text{MS}[k] - 1]$, as well as the identifier of the proper locus v of W in the topology of ST_T . We try to perform a backward step using character $a = S[k - 1]$: if the resulting interval $[i'..j']$ is nonempty, we set $\text{runs}[k] = 1$ and we reset $[i..j]$ to $[i'..j']$. Otherwise, we set $\text{runs}[k] = 0$, we update the BWT interval to the interval of $\text{parent}(v)$ using the topology, and we try another backward step with character a .

In the second phase we scan S from left to right, using $\text{BWT}_{\overline{T}}$ with rank support, a representation of the suffix tree topology of \overline{T} , and bitvector **runs**, to build **ms**. Assume that we know the interval $[i..j]$ in $\text{BWT}_{\overline{T}}$ that corresponds to substring $W = S[k..h-1]$ such that $\text{MS}[k-1] = h-k$ but $\text{MS}[k] \geq h-k$. We try to perform a backward step with character $S[h]$: if the backward step succeeds, we continue issuing backward steps with the following characters of S , until we reach a position h^* in S such that a backward step with character $S[h^*]$ from the interval $[i^*..j^*]$ of substring $W^* = S[k..h^*-1]$ in $\text{BWT}_{\overline{T}}$ fails. At this point we know that $\text{MS}[k] = h^* - k$, so we append $h^* - k - \text{MS}[k-1] + 1 = h^* - h + 1$ zeros and a one to **ms**. Moreover, we iteratively reset the current interval in $\text{BWT}_{\overline{T}}$ to the interval of $\text{parent}(v^*)$, where v^* is the proper locus of W^* in $\text{ST}_{\overline{T}}$, and we try another backward step with character $S[h^*]$, until we reach an interval $[i'..j']$ for which the backward step succeeds. Let this interval correspond to substring $W' = S[k'..h^*-1]$. Note that $\text{MS}[k'] > \text{MS}[k'-1] - 1$ and $\text{MS}[x] = \text{MS}[x-1] - 1$ for all $x \in [k+1..k'-1]$, thus k' is the position of the first zero to the right of position k in **runs**, and we can append $k' - k - 1$ ones to **ms**. Finally, we repeat the whole process from substring $S[k'..h^*]$ and its interval in $\text{BWT}_{\overline{T}}$.

Note that S is read sequentially in both phases, **runs** and **ms** are built by iteratively appending bits at one of their ends, and **runs** is read sequentially in the second phase. Thus, there is no need to keep any of these vectors fully in memory, and they can be streamed to or from disk in practice.

3 Fast matching statistics in small space

We assume that each BWT is represented as a wavelet tree, and that each suffix tree topology is represented as a sequence of balanced parentheses. Thus, a node of a suffix tree is described by two intervals: its interval in the BWT, and the pair of open and closed parentheses in the sequence of balanced parentheses. We can derive the identifier in the topology from the BWT interval, by issuing two `selectLeaf` operations followed by one `lca` operation. Vice versa, we can derive the BWT interval from the identifier in the topology, by issuing one `leftmostLeaf` and one `rightmostLeaf` operation, followed by two `leafRank` operations. As a baseline, we consider an implementation of the algorithm in Section 2.2 in which both the BWT interval and the identifier in the topology are updated at every step; in such implementation, taking a Weiner link from a node involves calling nine functions provided by the wavelet tree or the topology.

The two phases of the algorithm in Section 2.2 are symmetrical, thus the optimizations described in this section apply equally to both and have similar effects in practice.

3.1 Faster Weiner links

Following a Weiner link with label c from a node v of the suffix tree requires issuing two rank queries, for the same character, on the BWT interval $[i..j]$ of v . Such queries traverse the same nodes of the wavelet tree and, in the bitvector of each node, they access positions whose distance is at most $j - i$, and potentially decreases as the search goes deeper in the wavelet tree. Our first optimization consists in merging the two rank queries into a single `doubleRank` query, which, when the distance between its arguments is small, invokes just once some instructions (like counting the number of ones in some memory words) that would be executed twice by two distinct rank calls.

Note that the algorithm uses rank queries on the BWT interval of v also to decide whether a Weiner link with character c starts from v or not. The other advantage of merging the two rank queries is that we can detect whether the current interval is empty at each level

of the wavelet tree, and if so we can quit the traversal immediately. We call this further optimization `doubleRankAndFail`.

If node v has indeed a Weiner link labeled by c , we need to derive, from the BWT interval that results from taking such Weiner link, the identifier of the corresponding node v' in the suffix tree topology. However, the identifier of v' in the topology is used by our algorithm only if, in the following iteration, we need to move to the parent of v' , and in turn this happens only when taking a Weiner link from v' fails. In other words, inside a maximal sequence of successful Weiner links, we need just to update one BWT interval per iteration, and this costs just one `doubleRank` operation per iteration; we call this optimization `lazyWeinerLink`.

Knowing the repeat structure of T allows one to optimize Weiner links even further. Specifically, if v is not a maximal repeat of T , its BWT interval contains exactly one distinct character, thus it suffices to check whether e.g. the last position of the interval is equal to c , and if so to issue just one rank operation. We call this optimization `maxrepWeinerLink`. Even more aggressively, we can implement a `rankAndCheck` query which, in a single operation, checks whether the last position of the BWT interval matches c , stops the traversal of the wavelet tree as soon as it detects an empty interval, and returns the rank in case of success. The same optimization can be applied to intervals of size one. We detect whether a node is a maximal repeat by building a bitvector `marked[1..|T|]`, where we set `marked[i] = 1` and `marked[j] = 1` for every interval $[i..j]$ of a maximal repeat of T . We build `marked` by traversing the suffix tree topology depth-first, avoiding to explore the subtree of a node that is not a maximal repeat since it cannot contain other maximal repeats. To determine whether a given interval $[i..j]$ with $j > i$ corresponds to a maximal repeat, we just check whether `marked[i] = 1` and `marked[j] = 1`. Recall that, if a node of the suffix tree is a maximal repeat, then all its ancestors in the suffix tree are also maximal repeats. This implies that the interval of a node that is not a maximal repeat is properly contained inside the interval of a maximal repeat, and it does not contain the interval of any maximal repeat, thus it must have either its first bit or its last bit equal to zero. If $i = j$, the interval is not a repeat.

3.2 Faster parent operations

Recall that, when a Weiner link labeled by character c fails from a node v , we iteratively issue a parent operation on the topology, we convert the node identifier in the topology to its BWT interval, and we check whether c occurs inside such interval. Knowing whether a node is a maximal repeat or not enables once again a number of optimizations. First, if v is not a maximal repeat, then none of its ancestors that are not themselves maximal repeats have a Weiner link labeled with character c , so we can avoid trying the Weiner link from an ancestor of v that is not a maximal repeat. Moreover, as soon as an ancestor u of v is a maximal repeat, every one of the ancestors of u is a maximal repeat as well, so we do not need to check whether the current node is a maximal repeat after the following parent operations, if any.

Even more aggressively, when a Weiner link fails, we could directly move from node v with BWT interval $[i..j]$, to its lowest ancestor that is a maximal repeat, by taking the LCA between the rightmost one up to i in the `marked` bitvector, and the leftmost one starting from j . A further step along this line consists in moving directly to the lowest ancestor v_c of v whose BWT interval contains c : such ancestor, if any, is necessarily a maximal repeat. This technique, which we call `parentShortcut`, can be implemented as follows. We move to the rightmost occurrence of c before the interval of v (if any), by issuing a rank and a select query on the BWT, then we move to the corresponding leaf w of the suffix tree by issuing a `selectLeaf` query on the topology, and we compute the identifier of the ancestor

$p = \text{lca}(v, w)$ of v . We do the same for the leftmost occurrence of c to the right of the interval of v (if any), computing the identifier of an ancestor q of v . Finally, we set v_c to the lowest of p and q , by issuing one `isAncestor` query provided by the topology. The implementation of `parentShortcut` can be further optimized in practice. For example, if the interval of p ends at the same position as the interval of v , we don't even need to compute the leftmost occurrence of c after the interval of v . And if the interval of p does not contain the leftmost occurrence of c after the interval of v , we can just return p .

We briefly note that, in addition to being useful in practice, the `parentShortcut` technique implies also an on-line algorithm for matching statistics:

► **Lemma 1.** *There is an online algorithm that computes $\text{MS}_{S,T}$ in $O(\log \sigma)$ time per character of S , and in $|T| \log \sigma(1 + o(1)) + O(\min(\sigma, |T| \log \sigma))$ bits of space, by scanning S from right to left.*

Proof. The algorithm has the same structure as the one described in [15], but it spends a bounded amount of time per character of S . Recall that rank queries can be implemented in $O(\log \sigma)$ time with a wavelet tree, that select queries can be implemented in $O(\log \sigma)$ time as well using additional $|T|o(\log \sigma)$ bits, and that `parentShortcut` takes constant time in addition to select queries. Since the node reached by `parentShortcut` is a maximal repeat, we just need to compute the string depth of maximal repeats throughout the algorithm. This can be done in constant time, by storing also the topology of the suffix-link tree of \bar{T} and suitable bitvectors to commute between maximal repeats in the two topologies, as described in [4, Section 6]. ◀

3.2.1 Selecting the next occurrence of a character

Selecting the previous or the next occurrence of a character from a given position of the BWT of T lies at the core of the `parentShortcut` operation of Section 3.2, and it is a primitive of independent interest. We implement a generalization `selectNextT(i, j, c)`, which returns the j -th occurrence of character c strictly to the right of position i , in a string T that is assumed to be represented as a wavelet tree. For brevity we focus here just on the case in which $T[i] = c$, since the case in which $T[i] \neq c$ can be handled similarly.

We move from the root of the wavelet tree to a leaf, as in the rank operation at position i , remembering the position i_k that we reach in the bitvector of each level k . We invoke $j_k = \text{selectNext}(i_k, j, c_k)$ on the bitvector of the leaf, where c_k is the k -th bit in the binary representation of c , then we move up to the bitvector of the parent and we invoke $j_{k-1} = \text{selectNext}(i_{k-1}, j_k - i_k, c_{k-1})$; we continue in this way until we reach the root. A version of `selectNext` for bitvectors that uses a sequential, word-based scan, was already described in [9]. We take a similar approach, issuing a select operation on the bits of the computer word that contains position i_k , and on the following word if necessary (using class `bits` in SDSL). In case such selects fail, we issue a standard select operation on the whole bitvector. Note that the number of words that we check explicitly, in addition to the one containing i_k , could be bigger for bitvectors that are closer to the root of the wavelet tree: we leave such variant to the full version of the paper.

A more advanced strategy could consist in dividing the bitvector of length $|T|$ into blocks of size B each, keeping an auxiliary bitvector `pure[1..|T|/B]` to mark the blocks that contain just one distinct bit. Given a position i in the original bitvector, we could compute the block b that contains it, and we could check `pure[b]` to see if such block is pure. If this is the case we could return $i + 1$, or keep scanning `pure[b + 1..|T|/B]` until we find either a pure block with the correct value, or an impure block. If block b is impure, we could scan it

starting from position i and, in case of failure, we could continue as above. To speed up the search even further, we could introduce block clusters of size B^2 , storing for each cluster the position of the next pure cluster and of the next impure cluster. We leave the study of these and other variants to the full version of the paper.

Recall that we use `selectNext` when a Weiner link fails from a BWT interval $[i..j]$. Thus, one could create a `doubleRankAndSelectNext` operation that reuses the work performed by `doubleRank` when going down the wavelet tree in `selectNext`. Specifically, the only overhead of this operation, compared to a successful `doubleRankAndFail`, is saving the positions to which i and j are projected in each bitvector. In case of failure, the procedure does not abort but keeps going down to a leaf, from which it finally moves up, selecting both the next occurrence (from j) and the previous occurrence (from i). As with `doubleRank`, performing both selections at each level might save time by executing some instructions common to them only once. In case of success, the bottom-up phase is not executed. For reasons of time, we leave a full experimental study of `doubleRankAndSelectNext` to the full version of the paper.

Finally, note that a fast implementation of `selectNext` can be useful for answering *range matching statistics queries* on bitvector `ms`, i.e. queries that ask for all `MS[k]` values for k in a user-specified $[i..j]$, or for the average or maximum of such values, since one could replace $j - i + 1$ select operations with one select operation and $j - i$ `selectNext` operations. We leave the experimental study of range MS queries to the full version of the paper.

4 Implementation

Our C++ implementation, available at https://github.com/odenas/indexed_ms, is based on the SDSL library [8], which we adapt to our purposes. Specifically, we modify the following parts of SDSL: (1) the rank data structure `rank_support_v`; (2) the implementation `wt_pc` of the wavelet tree for byte sequences (using other wavelet tree variants, e.g. Huffman-shaped, is beyond the scope of this paper); (3) the implementation `csa_wt` of the compressed suffix array based on a wavelet tree; (4) the code for select operations on bitvectors `select_support_mcl`. We build the BWT with `dbwt` [16], and the other parts of the index by using `csa_wt` to represent a compressed suffix array without samples, and by stripping down the compressed suffix tree code in `cst_sct3` (among other things, by removing the LCP array). We represent the suffix tree topology explicitly with balanced parentheses, using a simplified version of `bp_support_sada` that supports just the `parent` operation. Recall that `bp_support_sada` is based on [20, 21], and that SDSL provides other representations `bp_support_gg` and `bp_support_g` based instead on [7, 14]. We use `bp_support_sada` because it turned out to be the fastest in preliminary experiments. We use `cst_sct3`, based on [13], rather than the other compressed suffix tree representation provided by SDSL, because its topology takes less space ($3|T|$ bits, rather than $4|T|$ bits of e.g. `cst_sada`), at the cost of slower `parent` operations. One could use more efficient algorithms to build the indexes, for example [2] for the suffix tree topology; we do not try to optimize construction in this paper.

5 Experimental results

5.1 Artificial strings

We use a number of artificial strings to evaluate the effect of the optimizations in Section 3. We resort to artificial strings because our optimizations are not designed for a particular class of inputs, and because the magnitude of their speedups might be affected by the similarity

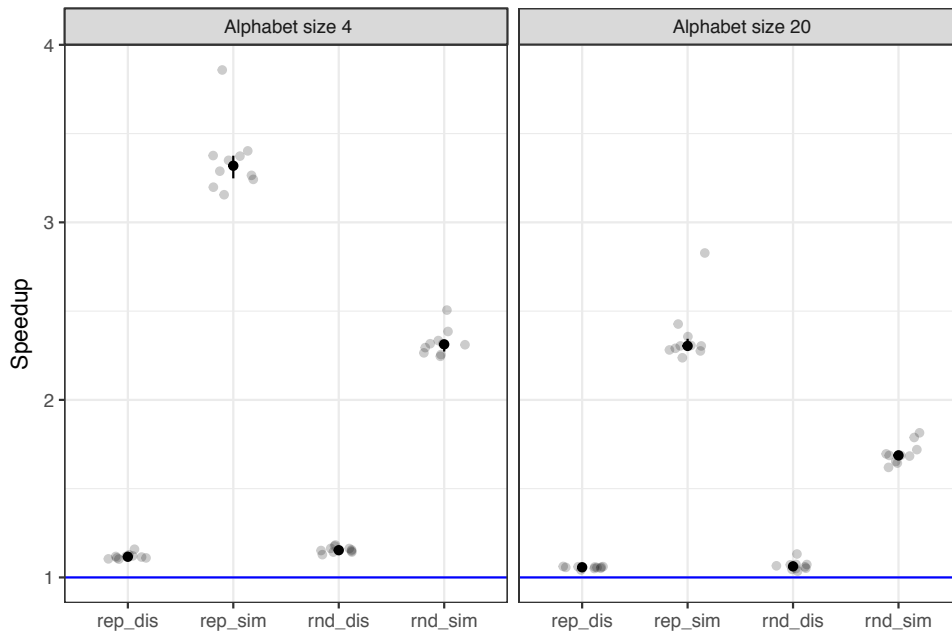
between S and T , by the repetitiveness of T , and by the alphabet size, over which we need control. In Section 5.2 we study performance on a dataset that approximates a real use case in phylogeny reconstruction by average matching statistics. Since our artificial strings do not cover the space of all possible input classes, the speedups we report should not be interpreted as upper bounds.

To measure the effect of the Weiner link optimizations in Section 3.1, we consider four types of (S, T) pairs, corresponding to repetitive or random T , and to a query S that is either similar to T or random (by random we mean a string generated by a source that assigns uniform probability to every character of the alphabet). A repetitive string is generated as follows: we take an initial random string W of length 10^4 , and we concatenate it with 10^4 copies of itself, containing each 10 random positions that mismatch with W . We create a string S that is similar to T by introducing mismatches in $5 \cdot 10^3$ random positions of a copy of T , and by taking a prefix of such copy of length $5 \cdot 10^5$, and we experiment with alphabet size 4 and 20. We measure the total time to compute `ms` and `runs` on a machine with one Intel Core i7-6600U processor with two cores. Its L3 cache is 4MB, thus T does not fit in cache. Throughout the section we call *speedup* the ratio between the time to build the `ms` and `runs` bitvectors with a baseline implementation that is clear from the context, and the time to build the same vectors with an implementation with specific optimizations turned on.

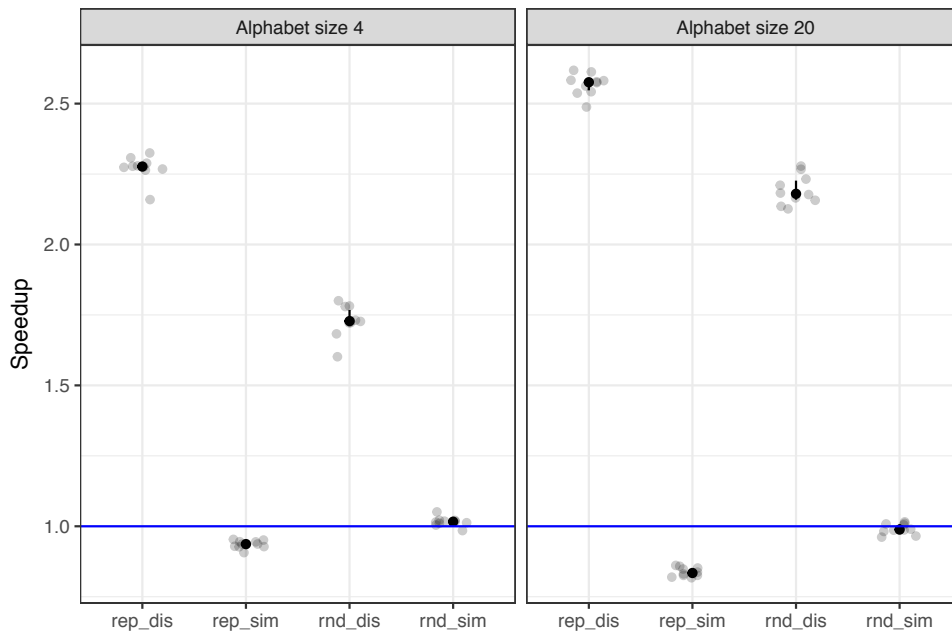
The `doubleRank` optimization tends to give median speedups smaller than 1.05 with respect to an implementation that uses two rank operations per Weiner link, both for small and for large alphabet, and it rarely degrades performance. Using `doubleRankAndFail` gives further speedups between 1.01 and 1.02 with respect to `doubleRank`, mostly when S is not similar to T ; when S is similar to T instead, performance tends to degrade by similar amounts. This is expected, since this optimization targets cases in which Weiner link calls are unsuccessful, and this is more likely to happen when S is not similar to T . If a Weiner link is successful, `doubleRankAndFail` is actually introducing an overhead. The early failure strategy might also be useful for representations in which the wavelet tree is not full, e.g. when it has a Huffman shape: we leave this study to the full version of the paper.

As expected, `lazyWeinerLink` is most effective when S is similar to T , since it optimizes successful Weiner links. When the alphabet is small, we observe median speedups of 3.2 for repetitive T and of 2.4 for random T (Figure 1). For large alphabets the median speedups become approximately 2.4 and 1.6, respectively. When S is not similar to T , the speedups are approximately equal to 1.1 and never smaller than one.

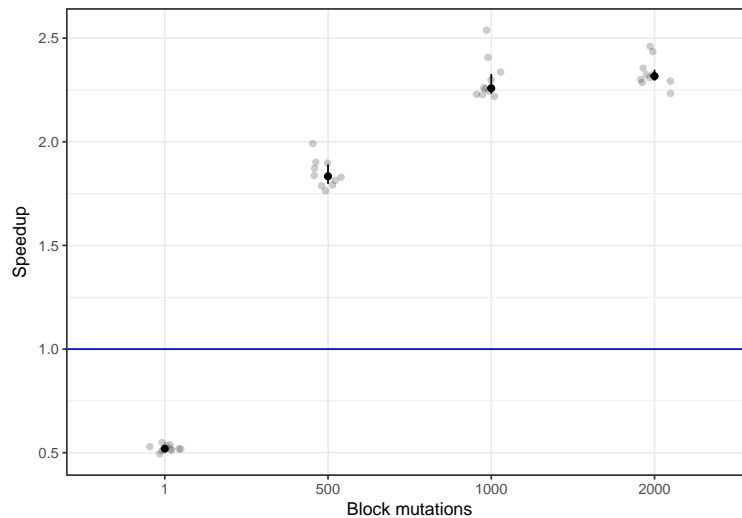
Conversely, it turns out that exploiting maximal repeat information in Weiner links is beneficial when S is not similar to T , and especially so when the alphabet is large. `maxrepWeinerLink` provides median speedups of 2.1 and 2.6 when T is repetitive (depending on alphabet size), and of 1.7 and 2.2 when T is random (Figure 2), compared to an implementation with just `doubleRankAndFail`. If S is similar to T , `maxrepWeinerLink` tends to degrade performance, and the median speedup can drop down to approximately 0.8. With respect to `maxrepWeinerLink`, `rankAndCheck` provides median speedups of approximately 1.01 when T is repetitive, and of 1.04 and 1.07 when T is random, depending on alphabet size, but only when S is not similar to T . The fact that maximal repeats are not useful when S is similar to T might be explained by the fact that the overhead of checking whether a node is a maximal repeat or not is compensated by a faster fail when a Weiner link is not successful, but it is not compensated in case of a successful Weiner link. One might try to reduce the cost of accessing the `marked` bitvector by interleaving it with the bitvector of the root of the wavelet tree: we leave such extension to the full version of the paper.



■ **Figure 1** Speedup of `lazyWeinerLink` with respect to an implementation with no optimization, for different input types (horizontal axis). Gray points are single measurements. Black points are medians. Black bars indicate the first and third quartiles of ten experiments.



■ **Figure 2** Speedup of `doubleRankAndFail` and `maxrepWeinerLink`, with respect to an implementation with just `doubleRankAndFail`. The plots follow the conventions of Figure 1.



■ **Figure 3** Speedup of `parentShortcut` with respect to a baseline with all optimizations turned off (vertical axis), for different settings of k (horizontal axis). The plot follows the conventions of Figure 1.

To measure the effect of the `parentShortcut` optimization of Section 3.2, we create pairs (S, T) in which the baseline algorithm is forced to issue long sequences of `parent` operations. Specifically, we build a repetitive T as described above, starting from a random string W of length 200 on alphabet size 6, and concatenating to it $5000 - 1$ copies of itself, containing each k random positions that mismatch with W . The resulting T takes 10MB and does not fit in cache. We build T repetitive to have nodes with large tree depth in ST_T and $ST_{\bar{T}}$. We build S by concatenating labels of nodes of ST_T with large tree depth (at least 10) but not too large string length (at most 170), separated by a character that does not occur in T , to enforce enough parent operations per successful Weiner link. We experiment with values of k ranging from one to 200, and at low values of k a large fraction of parent sequences has indeed large length. Using `parentShortcut` gives overall speedups between 1.8 and 2.3, depending on the value of k , with respect to an implementation with no optimization (Figure 3).

5.2 Biological strings

We use a dataset of 24 eukaryotic species with relatively large genomes that has been used for whole-genome phylogeny reconstruction by average matching statistics before [5]. The dataset contains both pairs of very similar and of very different genomes². We experiment with both genomes and proteomes to study the effect of alphabet size (4 and 20, respectively).

² We download the latest assemblies from NCBI, and we concatenate all sequences that belong to the genome or proteome of the same species using a separator character not in the alphabet, replacing runs of undetermined characters with a single occurrence of the separator. We run our experiments on a machine with 256GB of RAM and with one Intel Xeon E5-2680v3 processor, which has two sockets and 12 cores per socket. We pin our sequential program to a single socket using `numactl -N0 -m0`. The L3 cache of the processor is 30MB, and its L2 cache is 256KB, thus the indexes for genomes and bacteria do not fit in cache, the indexes for proteomes do not fit in the L2 cache, and the indexes of some proteomes can fit in the L3 cache. We measure wall clock time with standard C++ methods, and we measure peak memory by reading the maximum resident set size reported by `/usr/bin/time`.

The resulting genome files range from approximately 87 million to 5.8 billion characters, and proteome files range from approximately 4 million to 74 million characters. For reasons of time, we compute the matching statistics just between the genome of a fixed species, chosen arbitrarily (*Oryzias latipes*), and every other genome, and just between the proteome of a fixed species, chosen arbitrarily (*Homo sapiens*), and every other proteome. We call this setup *Experiment 1*. We also study the performance of our algorithms when T is repetitive and S is similar to T . Specifically, we download the genomes and proteomes of all the 8658 bacteria currently in NCBI: we build a query that consists of the concatenation of the genomes (respectively, proteomes) of 839 bacteria randomly sampled from the set, and a text that consists of the concatenation of the genomes (respectively, proteomes) of all remaining bacteria. The resulting text files contain approximately 30 and 8.5 billion characters, respectively, and the resulting query files contain approximately 3 billion and 980 million characters, respectively. We call this setup *Experiment 2*. As customary, we measure peak memory and running time just for computing matching statistics given the indexes. We define speedup as in Section 5.1, with respect to a version of our code with no optimization.

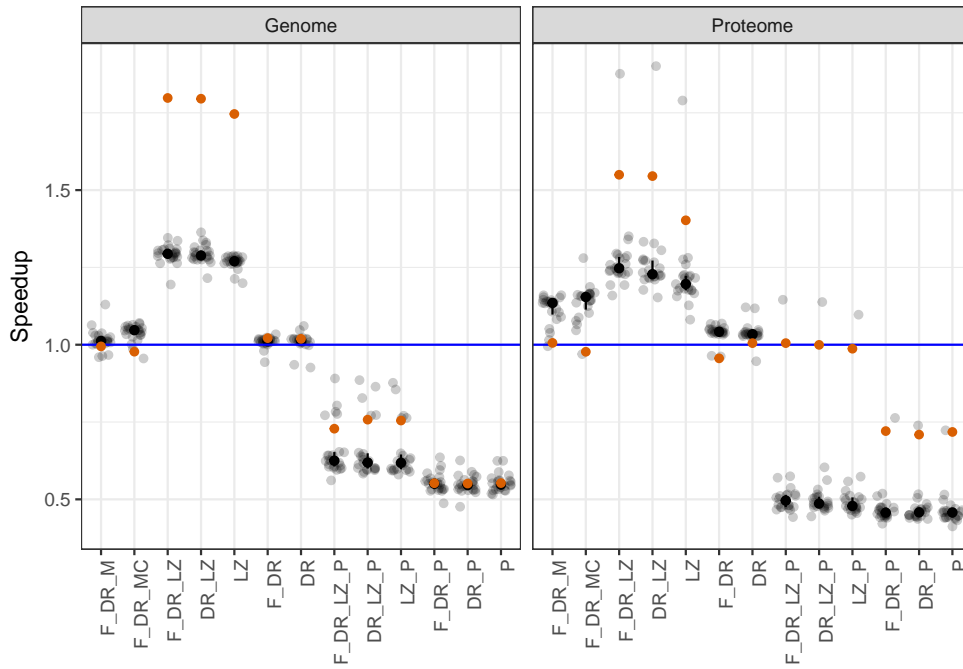
It turns out that `lazyWeinerLink` is the most effective optimization in both Experiment 1 and Experiment 2, providing approximately a 1.25 speedup for both genomes and proteomes in the first case, and a 1.75 speedup for genomes and a 1.5 speedup for proteomes in the second case. Using `doubleRankAndFail` in addition to `lazyWeinerLink` yields minor improvements and the fastest implementation. Speedups are greater when S is similar to T , with the largest speedups observed in Experiment 2 and for the pair of proteomes *Homo sapiens* and *Pan troglodytes* (see Figure 4).

We compare our implementations to the one described in [15], which builds essentially the same compressed suffix tree as SDSL’s `cst_sct3`, with `csa_wt` representing the compressed suffix array, but which includes a compressed LCP array³ storing all values that are at most 254 in one byte, and longer values in $\log |T|$ bits [10]. As expected, our implementation is more space-efficient, taking from approximately half to one-fifth of the space of the competitor in both genomes and proteomes (Figure 5). Not surprisingly, however, our implementation is also slower than the competitor, taking e.g. approximately twice its time for proteomes (Figure 5). Recall that the fact that the algorithm in [15] performs just one pass over S might contribute to this slowdown. However, for approximately half of the pairs of genomes in Experiment 1, and for the pair of genomes in Experiment 2, our implementation is faster than the competitor, with speedups up to 1.4.

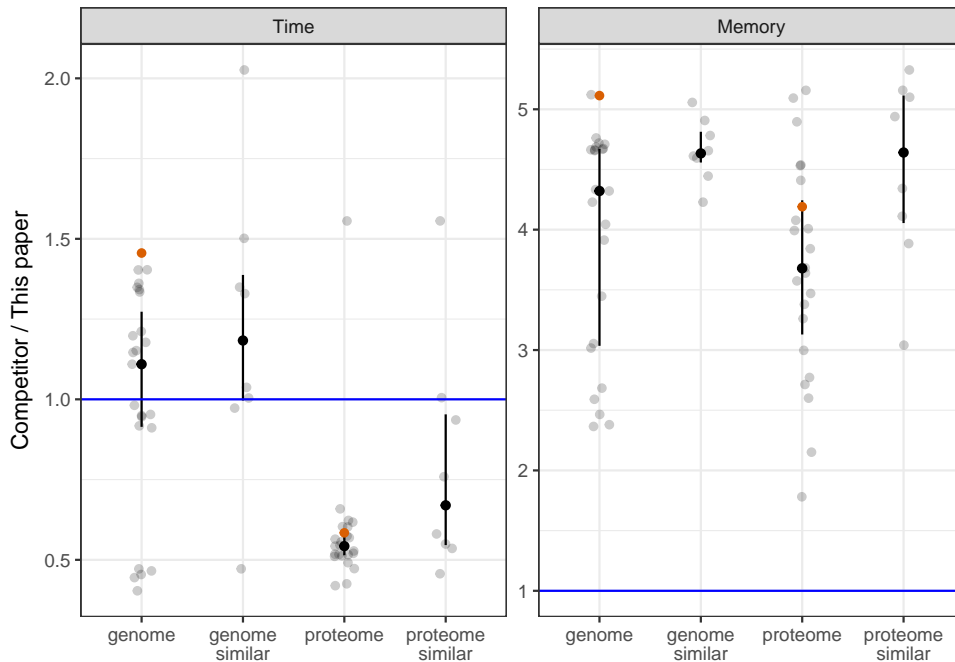
The eight pairs of sibling leaves⁴ in the phylogenetic tree of [5, Figure 1] correspond to similar species with short divergence times, and we observed before that our implementation gives larger speedups for similar strings. We call *Experiment 3* the comparison of our implementation with `doubleRankAndFail` and `lazyWeinerLink` enabled, to the one in [15], limited to such pairs. It turns out that our implementation is faster than or as fast as the competitor for all but one such pairs of genomes, with speedups up to two, while still taking between one fourth and one fifth of the competitor’s space (Figure 5). For pairs of proteomes, however, our implementation remains slower.

³ The index contains also (unused) samples of the suffix array of T and of \bar{T} , but the sampling rate (10^4) makes their space negligible in practice. The code of [15] does not work with long strings, thus we take prefixes of length 800 million of the strings it cannot handle. We use a version of the two implementations in which they do not write any output. For each (S, T) pair, we compare the fastest of our implementations to the competitor.

⁴ *Arabidopsis lyrata*-*Arabidopsis thaliana*; *Zea mays*-*Oryza sativa*; *Rattus norvegicus*-*Mus musculus*; *Gallus gallus*-*Taeniopygia guttata*; *Brugia malayi*-*Caenorhabditis elegans*; *Drosophila melanogaster*-*Anopheles gambiae*.



■ **Figure 4** Effect of our optimizations (horizontal axis) on Experiment 1 (gray circles) and on Experiment 2 (red circles). DR: `doubleRank`, F_DR: `doubleRankAndFail`, M: `maxrepWeinerLink`, MC: `rankAndCheck`, LZ: `lazyWeinerLink`, P: `parentShortcut`. The largest speedups in the plot on the right correspond to pair *Homo sapiens* and *Pan troglodytes*.



■ **Figure 5** The fastest of our implementations, compared to the implementation in [15]. “Genome”, “Proteome”: Experiment 1 (gray circles) and Experiment 2 (red circles). “Genome similar”, “Proteome similar”: Experiment 3. The plot follows the conventions of Figure 1.

References

- 1 Alberto Apostolico, Concettina Guerra, Gad M. Landau, and Cinzia Pizzi. Sequence similarity measures based on bounded Hamming distance. *Theoretical Computer Science*, 638:76–90, 2016.
- 2 Uwe Baier, Timo Beller, and Enno Ohlebusch. Space-efficient parallel construction of succinct representations of suffix tree topologies. *Journal of Experimental Algorithmics (JEA)*, 22:1–1, 2017.
- 3 Djamal Belazzougui and Fabio Cunial. Indexed matching statistics and shortest unique substrings. In *International Symposium on String Processing and Information Retrieval*, pages 179–190. Springer, 2014.
- 4 Djamal Belazzougui and Fabio Cunial. A framework for space-efficient string kernels. *Algorithmica*, 79(3):857–883, 2017.
- 5 Eyal Cohen and Benny Chor. Detecting phylogenetic signals in eukaryotic whole genome sequences. *Journal of Computational Biology*, 19(8):945–956, 2012.
- 6 Martin Farach, Michiel Noordewier, Serap Savari, Larry Shepp, Abraham Wyner, and Jacob Ziv. On the entropy of DNA: algorithms and measurements based on memory and rapid convergence. In *Proceedings of the sixth annual ACM-SIAM symposium on discrete algorithms*, pages 48–57, 1995.
- 7 Richard F Geary, Naila Rahman, Rajeev Raman, and Venkatesh Raman. A simple optimal representation for balanced parentheses. *Theoretical Computer Science*, 368(3):231–246, 2006.
- 8 Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014.
- 9 R. González, G. Navarro, and H. Ferrada. Locally compressed suffix arrays. *ACM Journal of Experimental Algorithmics*, 19(1):article 1, 2014.
- 10 Stefan Kurtz. Reducing the space requirement of suffix trees. *Software-Practice and Experience*, 29(13):1149–71, 1999.
- 11 Chris-Andre Leimeister and Burkhard Morgenstern. Kmacs: the k -mismatch average common substring approach to alignment-free sequence comparison. *Bioinformatics*, 30(14):2000–2008, 2014.
- 12 Gonzalo Navarro and Kunihiro Sadakane. Fully functional static and dynamic succinct trees. *ACM Transactions on Algorithms (TALG)*, 10(3):16, 2014.
- 13 Enno Ohlebusch, Johannes Fischer, and Simon Gog. CST++. In *International Symposium on String Processing and Information Retrieval*, pages 322–333. Springer, 2010.
- 14 Enno Ohlebusch and Simon Gog. A compressed enhanced suffix array supporting fast string matching. In *International Symposium on String Processing and Information Retrieval*, pages 51–62. Springer, 2009.
- 15 Enno Ohlebusch, Simon Gog, and Adrian Kügel. Computing matching statistics and maximal exact matches on compressed full-text indexes. In *SPIRE*, pages 347–358, 2010.
- 16 Daisuke Okanohara and Kunihiro Sadakane. A linear-time Burrows-Wheeler transform using induced sorting. In *International Symposium on String Processing and Information Retrieval*, pages 90–101. Springer, 2009.
- 17 Nicolas Philippe, Mikaël Salson, Thérèse Combes, and Eric Rivals. CRAC: an integrated approach to the analysis of RNA-seq reads. *Genome Biology*, 14(3):R30, 2013.
- 18 Cinzia Pizzi. Missmax: alignment-free sequence comparison with mismatches through filtering and heuristics. *Algorithms for Molecular Biology*, 11(1):6, 2016.
- 19 Kunihiro Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.

17:14 Fast matching statistics in small space

- 20 Kunihiro Sadakane. The ultimate balanced parentheses. Technical report, Kyushu University, 2008.
- 21 Kunihiro Sadakane and Gonzalo Navarro. Fully-functional succinct trees. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 134–149. SIAM, 2010.
- 22 Choon Hui Teo and S.V.N. Vishwanathan. Fast and space efficient string kernels using suffix arrays. In *Proceedings of the 23rd international conference on Machine learning*, pages 929–936. ACM, 2006.
- 23 Sharma V Thankachan, Alberto Apostolico, and Srinivas Aluru. A provably efficient algorithm for the k -mismatch average common substring problem. *Journal of Computational Biology*, 23(6):472–482, 2016.
- 24 Sharma V. Thankachan, Sriram P. Chockalingam, Yongchao Liu, Ambujam Krishnan, and Srinivas Aluru. A greedy alignment-free distance estimator for phylogenetic inference. *BMC bioinformatics*, 18(8):238, 2017.
- 25 Igor Ulitsky, David Burstein, Tamir Tuller, and Benny Chor. The average common substring approach to phylogenomic reconstruction. *Journal of Computational Biology*, 13(2):336–350, 2006.
- 26 Peter Weiner. Linear pattern matching algorithms. In *Switching and Automata Theory, 1973*, pages 1–11. IEEE, 1973.

Practical lower and upper bounds for the Shortest Linear Superstring

Bastien Cazaux


Department of Computer Science, University of Helsinki, Helsinki, Finland;
L.I.R.M.M., CNRS, Université Montpellier, Montpellier, France
Institute of Computational Biology, Montpellier, France
bastien.cazaux@cs.helsinki.fi

Samuel Juhel

L.I.R.M.M., CNRS, Université Montpellier, Montpellier, France
Institute of Computational Biology, Montpellier, France
samuel.juhel@zaclys.net

Eric Rivals

L.I.R.M.M., CNRS, Université Montpellier, Montpellier, France
Institute of Computational Biology, Montpellier, France
rivals@lirmm.fr

 <https://orcid.org/0000-0003-3791-3973>

Abstract

Given a set P of words, the **Shortest Linear Superstring** (SLS) problem is an optimisation problem that asks for a superstring of P of minimal length. SLS has applications in data compression, where a superstring is a compact representation of P , and in bioinformatics where it models the first step of genome assembly. Unfortunately SLS is hard to solve (**NP-hard**) and to closely approximate (**MAX-SNP-hard**). If numerous polynomial time approximation algorithms have been devised, few articles report on their practical performance. We lack knowledge about how closely an approximate superstring can be from an optimal one in practice. Here, we exhibit a linear time algorithm that reports an upper and a lower bound on the length of an optimal superstring. The upper bound is the length of an approximate superstring. This algorithm can be used to evaluate beforehand whether one can get an approximate superstring whose length is close to the optimum for a given instance. Experimental results suggest that its approximation performance is orders of magnitude better than previously reported practical values. Moreover, the proposed algorithm remains efficient even on large instances and can serve to explore in practice the approximability of SLS.

2012 ACM Subject Classification Mathematics of computing → Combinatorial optimization, Theory of computation → Discrete optimization

Keywords and phrases greedy, approximation, overlap, Concat-Cycles, cyclic cover, linear time, text compression

Digital Object Identifier 10.4230/LIPIcs.SEA.2018.18

Acknowledgements This work is supported by the Institut de Biologie Computationnelle (ANR-11-BINF-0002) and Défi MASTODONS C3G from CNRS.



© Bastien Cazaux, Samuel Juhel, and Eric Rivals;
licensed under Creative Commons License CC-BY
17th International Symposium on Experimental Algorithms (SEA 2018).
Editor: Gianlorenzo D'Angelo; Article No. 18; pp. 18:1–18:14



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

1 Introduction

Let $P := \{s_1, \dots, s_{|P|}\}$ be a set of input words, whose sum of lengths is denoted by $\|P\|$. A superstring of P is a string that contains each of the input words as substrings. Without loss of generality, we assume that P is factor free, *i.e.*, that no word of P is substring of another word of P . The *Shortest Linear Superstring* (SLS) problem – also known as Shortest Common Superstring –, asks for a superstring of P of minimal length.

A recent survey gives an idea of the variety of applications of SLS: from the most known ones, DNA assembly or text compression, to job scheduling or viral genomes compression [10]. Several variations of SLS have also been investigated in theory, *e.g.*, with reversals [13, 9], with strings of DNA [14, 4], with multiplicities [8, 7]. SLS, which is studied since the 80's, has been proven NP-hard even for instances containing only words of length 3, and difficult to approximate (MAX-SNP-hard) [10]. Several polynomial time approximation algorithms with constant ratios have been designed for SLS, and among them, the **Greedy** algorithm, which, unlike most other approximation algorithms for SLS, admits a linear time implementation [21]. Currently, the approximation ratio of the **Greedy** algorithm is proven to be 3.5 [23] (see Algorithm **Greedy** in Appendix). The 28 years old, so called, **Greedy** conjecture states that the **Greedy** algorithm achieves an approximation ratio of 2, which is better than the best known approximation ratio of $2 + 11/30$ [16, 17], the latter being achieved by a polynomial, but not linear time algorithm. Another example of approximation algorithm is **Concat-Cycles**, which linearises and concatenates the cyclic words obtained by solving the SHORTEST CYCLIC COVER OF STRINGS problem (SCCS) on the instance; **Concat-Cycles** has an approximation ratio of 4 [2].

Importantly, algorithm **Greedy** for SLS breaks ties randomly, and is thus not deterministic. Example 1 illustrates the consequences of this non determinism in terms of approximation ratio.

► **Example 1.** On the classical instance (with $k > 0$) $P := \{ab^k, b^{k+1}, b^k c\}$, **Greedy** can output either $w_b := ab^k cb^{k+1}$ or $w_g := ab^{k+1}c$ as a superstring of P . The second one is optimal, while the first is the worst greedy superstring. This instance is the one used in [20] to bound the approximation ratio of **Greedy** by 2 (which tends to 2 when $k \rightarrow \infty$).

Some recent works have developed theoretical arguments suggesting that the **Greedy** algorithm achieves good approximation in general [15]. Experimental assessments on instances up to 1,000 words of length up to 50 have shown that two approximation algorithms for SLS with ratio 3 and 4 return solutions within 1.28 times the optimal superstring length [19]. To our knowledge, this article gives the only experimental results published so far, and clearly emphasises the gap between lower and upper bounds, as well as between theory and practice. Although the algorithms used in [19] ran in short time on relatively small instances, their running times seem to increase non linearly with the instance size [19, Figure 5], indicating their limited scalability.

It would be useful to be able to determine rapidly, and before hand, whether an approximation algorithm would return a good approximate solution for a given instance. Obviously, such an algorithm should have a reasonable worst case approximation ratio, the best possible approximation in practice, should take linear time and be efficient enough to process large instances.

We propose an algorithm to compute a lower and an upper-bound on the size of an optimal solution for SLS. These two bounds, denoted respectively ℓ_{\min} and ℓ_{\max} , are defined in Section 3.

We shall obtain the following theorem.

► **Theorem 2.** *Let P be a set of strings and let w_{opt} denote an optimal solution of SLS of P . We can compute in linear time in $\|P\|$ the values ℓ_{min} and ℓ_{max} such that:*

$$\ell_{min} \leq |w_{opt}| \leq \ell_{max} \quad \text{and} \quad \frac{\ell_{max}}{\ell_{min}} \leq 4.$$

Contributions. Here, we exhibit a linear time algorithm to compute a lower and an upper bound, respectively ℓ_{min} and ℓ_{max} , on the size of a shortest superstring of P . Then we present experimental results of this algorithm on a series of instances of increasing sizes. These results show that ℓ_{min} and ℓ_{max} are extremely close to each other in practice. For more details, please see the web appendix at <http://www.lirmm.fr/~rivals/res/superstring/>.

Notation. We consider finite words over a finite alphabet Σ . The set of all finite words over Σ is denoted by Σ^* , and ϵ denotes the empty word. For a word x , $|x|$ denotes the *length* of x . Given two words x and y , we denote by xy the *concatenation* of x and y .

Let s, t, u be three strings of Σ^* . We say that s overlaps t if and only if a suffix of s also is a prefix of t . We denote by $ov(s, t)$ the longest overlap from s over t (also termed *maximum overlap*); let $pr(s, t)$ be the prefix of s such that $s = pr(s, t)ov(s, t)$, and let $su(s, t)$ be suffix of t such that $t = ov(s, t)su(s, t)$. The *merge of s over t* is the word $pr(s, t)t$. Note that neither the overlap nor the agglomeration are symmetrical.

► **Example 3.** Consider two strings $S := actgct$ and $T := tgcttac$. Then the longest overlap $ov(S, T) = tgct$, but the substring t also is an overlap from S over T . Then $pr(S, T) = ac$ and $su(S, T) = tac$. Moreover, we see that $ov(T, S) = ac$, which differs from $ov(S, T)$.

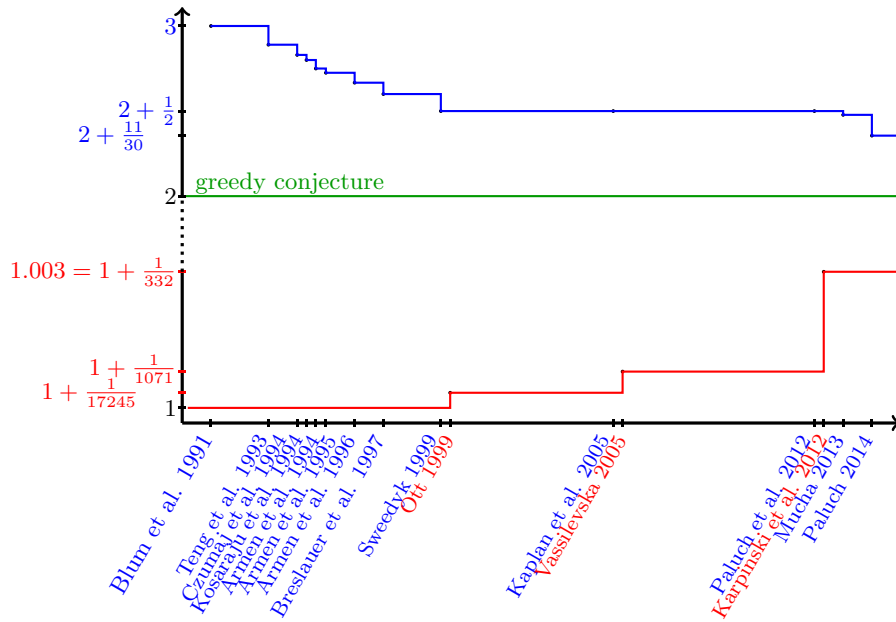
Throughout the article, the input is $P := \{s_1, \dots, s_{|P|}\}$ a set of input words, and without loss of generality, we assume that P is substring free, *i.e.*, no word of P is substring of another word of P .

2 Related Works

Significant research effort has been dedicated to designing approximation algorithms for SLS and to finding the best theoretical approximation ratios (see [11] for a list of algorithms). Both upper and lower bounds of approximation ratios have been studied [22] (see Figure 1).

A crucial result regarding the design approximation algorithms for SLS is that a variant of SLS called, SHORTEST CYCLIC COVER OF STRINGS (SCCS), can be solved exactly and returns a set of cycling strings covering the words of P . This set of cyclic strings can in turn be linearised and combined in various ways to form good linear superstrings [2]. A *cover* C is a set of strings such that any s_i is a substring of at least one string of C . An optimal cover can be obtained by computing a cyclic cover on the distance graph, a complete digraph representing the words of P and their maximum overlaps, using the Hungarian algorithm in $O(\|P\| + |P|^3)$ time once the graph is built [18]. Blum *et al.* also state in their seminal article that a greedy algorithm computes a minimal cover of strings of P [2]. Recently, it was shown how to implement this greedy algorithm for SCCS in linear time in $\|P\|$ [6, Theorem 6]; see Algorithm 1. Algorithm 1, called **CGreedy**, minimises the norm of the Cyclic Cover of Strings, but also its cardinality, that is its number of cyclic words [6, Theorem 7].

Cyclic cover based approximation algorithms. The first approximation algorithm based on a shortest cyclic cover is **Concat-Cycles** from [2]. **Concat-Cycles** computes C a Shortest Cyclic Cover of P . For $1 \leq i \leq |C|$, each cyclic string c_i of C covers a subset of words of



■ **Figure 1** All the ratios of approximation (in blue) and inapproximation (in red) for the problem SLS by year.

Algorithm 1: Algorithm **CGreedy**. We denote any cyclic string w by $\langle w \rangle$.

- 1 **Input:** a set of strings P ; **Output:** C , a Cyclic Cover of Strings of P ;
 - 2 $C := \emptyset$;
 - 3 **while** $|P| > 0$ **do**
 - 4 u and v in P (not necessarily distinct) such that $ov(u, v)$ is maximised;
 - 5 **if** $u = v$ **then** $C := C \cup \{\langle pr(u, v) \rangle\}$;
 - 6 **else** $P := P \setminus \{u, v\} \cup \{\langle pr(u, v)ov(u, v)su(u, v) \rangle\}$;
 - 7 **return** C
-

P ; let us denote this subset $P_i := \{s_{j_1}, \dots, s_{j_{|c_i|}}\}$. For each c_i , it derives a linear string w_i , which is a partial superstring of P_i , by breaking c_i between two words of P_i , say s_{j_k} and $s_{j_{k+1}}$, by concatenating $pr(s_{j_k}, s_{j_{k+1}})$. Hence, $|w_i| \leq |c_i| + |s_{j_{k+1}}|$. Then, **Concat-Cycles** concatenates the words w_i for $1 \leq i \leq |C|$ in an arbitrary order, which yields a superstring of P . **Concat-Cycles** achieves an approximation ratio of 4 for SLS [2, Theorem 8].

Blum *et al.* also proposes an improvement of this strategy: each cycle can be broken at an optimal point so as to create the shortest w_i for c_i . As the cycle word c_i defines an order of occurrence for each word of P_i in c_i , this only requires to test any pair of successive words which is linear in $\|P_i\|$. They show that a variant of the greedy algorithm for SLS, which they call **MGreedy**, does exactly that [2]. In fact, we view **MGreedy** (see the web appendix) as an application of Algorithm **LCGreedy**, followed by a concatenation. In other words, **MGreedy** builds a linear cover of P (which is made of linear, rather than cyclic, strings), and concatenate those linear strings arbitrarily into a single linear superstring of P . Blum *et al.* show that this linear superstring is shorter than the one output by **Concat-Cycles** [2].

In these two algorithms **Concat-Cycles** and **MGreedy**, each cycle contributes to adding some symbols to the final superstring. We propose to optimise such procedure by minimising the number of cycles in the Shortest Cyclic Cover obtained by a greedy algorithm for SCCS.

Remark on non-determinism. As indicated in introduction, all mentioned greedy algorithms – **Greedy**, **SCGreedy**, **LCGreedy** or **MGreedy** – break ties randomly when choosing the next overlap to use. Hence, none of these algorithms are deterministic, implying that two distinct executions may produce superstrings of different lengths or cyclic covers with different number of cycles. To our knowledge, most approximation algorithms designed to date use at least a greedy solution for **SCCS** to start with, and inherit from non-determinism.

Lower and upper bounds. Among others, Vassilevska has proven new lower bounds for the approximation ratio of **SLS**. She noticed the huge gap separating the best upper bounds and lower bounds [22].

3 Algorithm LCGreedyMin

Overview. Compared to **Concat-Cycles** or **MGreedy** [2], our algorithm builds a superstring based on a Shortest Cyclic Cover of P having a minimal number of cycles. Our algorithm proceeds as follows. First, it builds the *Extended Hierarchical Overlap Graph* (EHOG), a graph that encodes all overlaps between words of P but takes linear space. Embedded in the EHOG, it computes the Superstring Graph of P , which encodes the paths of all greedy solutions for **SCCS**. By finding an Eulerian path on each connected component of the Superstring Graph, it determines the node of minimal word depth of the component, and the shortest linearisation of each cyclic string. Moreover, this set of Eulerian paths constitutes an optimal Shortest Cyclic Cover of P ; more precisely, we get the permutation indicating in which order the words of P are merged in each component to form the cyclic strings. Then, we then compute ℓ_{\min} and ℓ_{\max} . We call our algorithm **LCGreedyMin**.

Below, we describe the graphs needed by **LCGreedyMin** and the algorithm.

3.1 EHOG

We denote by $\mathcal{O}v^+(P)$ the set of all overlaps between two (not necessarily distinct) strings of P , *i.e.* $\mathcal{O}v^+(P) := \{w \mid \exists u \text{ and } v \in P \text{ such that } w \text{ is a prefix of } u \text{ and } w \text{ is a suffix of } v\}$.

► **Definition 4.** The *Extended Hierarchical Overlap Graph* of P , denoted by $\text{EHOG}(P)$, is the directed graph $(V_E, P_E \cup S_E)$ where $V_E = P \cup \mathcal{O}v^+(P)$, while P_E is the set: $\{(x, y) \in (P \cup \mathcal{O}v^+(P))^2 \text{ such that } x \text{ is the longest proper prefix of } y\}$ and S_E is the set: $\{(x, y) \in (P \cup \mathcal{O}v^+(P))^2 \text{ such that } y \text{ is the longest proper suffix of } x\}$.

The EHOG has a node for each word of P and a node for any string that is an overlap between words of P . It can be seen that both types of nodes are also nodes of the Generalised Suffix Tree of P [12] – a Suffix Tree is a data structure that indexes all substrings of a text, while the Generalised Suffix Tree is the version that indexes several texts concatenated. Additionally, there are two types of arcs: one for recording the longest suffix relationship between nodes of V_E , the other for the longest prefix relationship. The first type can be seen as the arcs of the generalised suffix tree, while the second type corresponds to its Suffix Links. It follows that the EHOG occupies less space than the Generalised Suffix Tree of P . Examples of EHOG can be viewed in [5].

Rationale of the EHOG. The words of P and all their overlaps (*i.e.*, $\mathcal{O}v^+(P)$) are nodes of the EHOG. Consider u, v two words of P . Following arcs of S_E from u , one visits all its right overlaps in order of decreasing length. The first of such nodes that is an ancestor of v represents $ov(u, v)$. Hence, the merge of (u, v) is (bijectively) associated to the shortest

path from u to v through $ov(u, v)$ in $EHO(P)$. Call this the *merging path* from u to v . As any superstring (that does not waste any symbol) is determined by the order in which words of P are merged (solely using maximum overlaps between successive words), we see that it corresponds to a unique succession of merging paths in the EHO. Similarly, any cyclic cover of strings of P is uniquely associated with a collection of merging cycles that visit all nodes of P once in the EHO. In fact, $EHO(P)$ encode all possible, interesting superstrings and cyclic covers of P .

3.2 Superstring Graph

Consider a shortest cyclic cover of strings of P found by algorithm **CGreedy**. Its cyclic strings induce merging cycles in $EHO(P)$, and hence a permutation of P representing the order in which words are merged. The Superstring Graph is **the subgraph** of $EHO(P)$ visited by such a shortest cyclic cover of strings of P (since it is shown in [6, Proposition 3] that all greedy shortest cyclic covers of P visit the same subgraph). This is the intuitive rationale of the Superstring Graph, for which now we provide a formal definition.

► **Definition 5.** The *Superstring Graph* of a set of strings P is the sub-graph of $EHO(P) = (V_E, P_E, S_E)$ represented by the weight functions n and d on the nodes of V_E such that:

$$(n(u), d(u)) = \begin{cases} (1, 1) & \text{If } u \in P, \\ (0, -\text{dif}_{n,d}(u)) & \text{If } u \notin P \text{ and } \text{dif}_{n,d}(u) \leq 0, \\ (\text{dif}_{n,d}(u), 0) & \text{If } u \notin P \text{ and } \text{dif}_{n,d}(u) > 0, \end{cases}$$

where

$$\text{dif}_{n,d}(u) = \sum_{(v,u) \in S_E} n(v) - \sum_{(u,v) \in P_E} d(v).$$

Among all overlaps stored in the EHO, a shortest cyclic cover of P will use some overlaps to merge words, eventually more than once. An overlap is used if the cycles traversed the corresponding EHO node. While building the SG, we compute a function Ov_{SG} that indicates how many times a shortest cyclic cover use an overlap. Precisely, we define Ov_{SG} as the function from the set of nodes of $EHO(P) = (V_E, P_E, S_E)$ to \mathbb{N} , such that

$$Ov_{SG}(u) = \min\left(\sum_{(v,u) \in S_E} n(v), \sum_{(u,v) \in P_E} d(v)\right).$$

Algorithm 2 computes the superstring graph as well as the function Ov_{SG} . The idea of the algorithm is to traverse the EHO in reverse depth order and to compute the different weight functions (n , d , and Ov_{SG}). Indeed, the weights (n , d , and Ov_{SG}) of a node only depends on the weights of deeper nodes in the EHO. Each node represents a string: the substring built by concatenating the labels from the root to that node. With *deeper*, we refer to the string depth of a node.

In [6], we gave a proof that the Superstring Graph is a graph that represents all greedy solutions of *SCCS*. Because the Superstring Graph is Eulerian, it has the following property:

► **Proposition 6** ([6]). *Let P be a set of strings. One can compute in $O(|P|)$ time a greedy solution of *SCCS* with the least number of cyclic strings by computing an Eulerian path on each connected component of the Superstring Graph.*

Indeed, taking a single cyclic path to cover each of its connected component is possible (a component could be covered by combining several cycles instead of only one); finding those paths takes a time that is linear in the number of nodes of the Superstring Graph.

3.3 Linearisation of cycles and computation of the bounds

Algorithm **MGreedy** [2] first computes an optimal cycle cover of P , linearises each cycle optimally, and then concatenates the resulting linear strings. As above mentioned, it is not deterministic and instances like the one given in Example 1 shows that the resulting superstring may vary a lot. Indeed, the linearisation of each cycle increases the size of the final superstring. We introduce a variant of **MGreedy**, called **MGreedyMin**, which chooses a greedy (and thus optimal) solution of **SCCS** with the least number of cycles. We compute the bounds of Theorem 2 (ℓ_{\min} and ℓ_{\max}) based on such a cyclic cover of minimal cardinality.

Computation of ℓ_{\min} . The norm of a set Z of cyclic strings, denoted $\|Z\|$, is the sum of the length of strings in Z .

► **Proposition 7.** *Let C be a solution of the greedy algorithm for **SCCS** on P :*

$$\|C\| = \|P\| - \sum_{u \in V_E} Ov_{SG}(u) \times |u|.$$

Proof. Given a string v of P , we denote by $next_C(v)$ the string of P which follows directly v in the cyclic cover of strings C . As each greedy solution of **SCCS** is embedded in the Superstring Graph, we have

$$\begin{aligned} \|C\| &= \sum_{v \in P} |v| - |ov(v, next_C(v))| \\ &= \sum_{v \in P} |v| - \sum_{v \in P} |ov(v, next_C(v))| \\ &= \|P\| - \sum_{u \in V_E} |u| \times |\{v \in P \mid u = ov(v, next_C(v))\}| \\ &= \|P\| - \sum_{u \in V_E} |u| \times Ov_{SG}(u). \end{aligned} \quad \blacktriangleleft$$

By nature, the norm of C is smaller than an optimal shortest superstring of P . But for some instances, their difference can be as large as desired (can tend to infinity when the norm of the input tends to infinity). Thus defining ℓ_{\min} as the norm of C would not guarantee that ℓ_{\min} and ℓ_{\max} are close. We define ℓ_{\min} as the maximum between $1/4$ of ℓ_{\max} and the norm of C , which is an optimal cyclic cover for P .

Computation of ℓ_{\max} . By definition, the Superstring Graph is a sub-graph of the EHOg. Denoting by G_1, \dots, G_m the different connected components of the Superstring Graph, we get that G_1, \dots, G_m partition the node set of the Superstring Graph. We define $Cut(P)$ as the sum of the string depths (*i.e.*, the length of the string represented by a node) of the smallest node of each connected component, *i.e.*, $Cut(P) = \sum_{i=1}^m \min_{u \in G_i} |u|$.

► **Proposition 8.** *Let w a solution of **MGreedyMin**. We have that :*

$$|w| = \|P\| - \sum_{u \in V_E} Ov_{SG}(u) \times |u| + Cut(P).$$

Proof. Let w_g be a solution of **MGreedyMin** given by a greedy solution c_{min} of **SCCS** with the least number of cycles. By the property of the Superstring Graph, $c_{min} = \{c_1, \dots, c_m\}$, where for all i between 1 and m , c_i is the cyclic string representing a Eulerian cycle in G_i and c_i is a cyclic superstring of a subset of P_i of P . By the definition of **MGreedyMin**, we take w_i the minimal linearisation of c_i , *i.e.*

$$w_i \in \underset{(s_{j_k}, s_{j_{k+1}}) \in P_i \times P_i}{\mathbf{Arg\,min}} \left| \text{Linearisation}(c_i, s_{j_k}, s_{j_{k+1}}) \right|$$

where $Linearisation(c_i, s_{j_k}, s_{j_{k+1}})$ is the string obtain by breaking c_i between s_{j_k} and $s_{j_{k+1}}$ where s_{j_k} and $s_{j_{k+1}}$ are successive in c_i .

Hence, we have

$$\begin{aligned}
|w_g| &= \sum_{i=1}^m |w_i| \\
&= \sum_{i=1}^m \min_{(s_{j_k}, s_{j_{k+1}}) \in P_i \times P_i} |Linearisation(c_i, s_{j_k}, s_{j_{k+1}})| \\
&= \sum_{i=1}^m \min_{(s_{j_k}, s_{j_{k+1}}) \in P_i \times P_i} (|c_i| + |ov(s_{j_k}, s_{j_{k+1}})|) \\
&= \sum_{i=1}^m |c_i| + \min_{(s_{j_k}, s_{j_{k+1}}) \in P_i \times P_i} |ov(s_{j_k}, s_{j_{k+1}})| \\
&= \sum_{i=1}^m |c_i| + \min_{u \in G_i} |u| \\
&= \sum_{i=1}^m |c_i| + \sum_{i=1}^m \min_{u \in G_i} |u| \\
&= \|c_{min}\| + Cut(P) \\
&= \|P\| - \sum_{u \in V_E} Ov_{SG}(u) \times |u| + Cut(P).
\end{aligned}$$

Indeed, by Proposition 7, we have $\|c_{min}\| = \|P\| - \sum_{u \in V_E} Ov_{SG}(u) \times |u|$. \blacktriangleleft

By Proposition 8, we get that all solutions of **MGreedyMin** have the same length; we denote this length by ℓ_{max} .

Clearly, as a solution of **MGreedyMin** is also a solution of **MGreedy**, it follows that $|w_{opt}| \leq \ell_{max} \leq 4 \times |w_{opt}|$, where w_{opt} denotes any optimal solution of SLS. This yields Theorem 2.

Difference between ℓ_{min} and ℓ_{max} . We have defined ℓ_{max} as the length of a solution of the algorithm **MGreedyMin**, *i.e.*

$$\ell_{max} = \|P\| - \sum_{u \in V_E} Ov_{SG}(u) \times |u| + Cut(P).$$

The value of ℓ_{min} is the maximum between the norm of an optimal solution of SCCS and $\ell_{max}/4$, *i.e.*

$$\ell_{min} = \max\left(\frac{\ell_{max}}{4}, \|P\| - \sum_{u \in V_E} Ov_{SG}(u) \times |u|\right).$$

With these definitions, we obtain the following proposition.

► **Proposition 9.** *Let P be a set of strings. The bounds ℓ_{min} and ℓ_{max} are invariant and $\ell_{max} - \ell_{min} \leq Cut(P)$.*

By invariant, we mean that their computation is deterministic. Hence, although ℓ_{min} and ℓ_{max} depend on the instance P , their values do not vary upon the execution of **MGreedyMin**, unlike the solutions computed by **Greedy**, **MGreedy**, **Concat-Cycles**, and other approximation algorithms.

4 Implementation and experimental results

Here, we explain how each step of Algorithm **MGreedyMin** is implemented. First it builds the EHOg of P in memory: for this, we rely on the data structure named **COvI**, a compact implementation of the EHOg that can be used as an indexing and supports queries on overlaps [3]. The algorithm that builds **COvI**, first builds a compact version of the Aho-Corasick automaton of P [1], then prunes its set of states (or nodes in the tree) to keep only nodes that represent overlaps between words of P . When visiting a node of the EHOg, we

Algorithm 2: Computing the Superstring Graph.

```

1 Input:  $\text{EHOG}(P)$ ; Output:  $V_{SG}$ ,  $n(V_E)$ ,  $d(V_E)$  and  $Ov_{SG}(V_E)$ ;
2  $V_{SG} \leftarrow \emptyset$ ;
3  $\forall u \in V_E : Ov_{SG}'(u) \leftarrow 0$ ;  $n'(u) \leftarrow 0$ ;  $d'(u) \leftarrow 0$ ;
4  $Q \leftarrow$  a reverse depth order on the nodes of  $\text{EHOG}(P)$ ;
5 for  $u \in Q$  do
6    $(s, p) \leftarrow (p_{su}(u), p_{pr}(u))$ ;
7   if  $u$  is a leaf then
8      $(n'(u), d'(u)) \leftarrow (1, 1)$ ;
9   else
10     $Ov_{SG}'(u) \leftarrow \min(n'(u), d'(u))$ ;
11    if  $n'(u) > d'(u)$  then  $(n'(u), d'(u)) \leftarrow (n'(u) - d'(u), 0)$  ;
12    else  $(n'(u), d'(u)) \leftarrow (0, d'(u) - n'(u))$  ;
13     $n'(s) \leftarrow n'(s) + n'(u)$ ;
14     $d'(p) \leftarrow d'(p) + d'(u)$ ;
15    if  $d'(u) \neq 0$  or  $n'(u) \neq 0$  then  $V_{SG} \leftarrow V_{SG} \cup \{u\}$  ;
16 return  $V_{SG}$ ,  $n'$ ,  $d'$  and  $Ov_{SG}'$ 

```

need to know the length of the substring it represents. In COvI , for each node this length is accessible in constant time. For a node u of the EHOG , we can also access in constant time $p_{pr}(u)$ (resp. $p_{su}(u)$), which denotes the node of the EHOG corresponding to the longest prefix (resp. suffix) of u .

Then, computing the Superstring Graph from the EHOG is done with Algorithm 2.

► **Proposition 10.** *Algorithm 2 builds a superstring graph in time linear in $\|P\|$.*

Proof. *Complexity :* Finding a reverse depth order on the nodes of $\text{EHOG}(P)$ may be done in linear time. The **for** loop is executed once for each node of $\text{EHOG}(P)$, and there are at most $\|P\|$ nodes. All operations inside the loop are assignments or comparisons of integers. *Correctness :* Since when starting the **for** loop (line 5), we have $n'(u) = \sum_{(v,u) \in S_E} n(v)$ and $d'(u) = \sum_{(u,v) \in P_E} d(v)$, at the end of the loop (line 15), we get $n'(u) = n(u)$ and $d'(u) = d(u)$. ◀

Let Comp be the table of size $|V_E|$ that maps each node of the superstring graph to its connected component, and all other nodes to 0.

► **Proposition 11.** *Algorithm 3 computes Comp in time linear in $\|P\|$.*

Proof. The superstring graph being Eulerian, if there is a path q from a node u to a node v , there is another path from v to u that do not share any edge with q . Using this property, it is possible to recursively follow all paths in the superstring graph from a node to itself while marking all traversed nodes. Applying this process on every node of graph allows to discover all its connected components. The number of arcs of the superstring graph is linear in $\|P\|$, and during the whole process each arc of the superstring graph is visited once and only once, implying that Algorithm 4 takes linear time. ◀

► **Proposition 12.** *Let P be a set of strings. We can compute $\text{Cut}(P)$ in linear time in $\|P\|$.*

Algorithm 3: Algorithm to build the table `Comp`.

```

1 Input:  $\text{EHOG}(P)$ ,  $V_{SG}$ ,  $n(V_E)$  and  $d(V_E)$ ; Output: Comp;
2 Comp  $\leftarrow$  Table of size  $|V_E|$  initialised to 0;
3  $nb \leftarrow 1$ ;
4 for  $u \in V_E$  do
5   Update_Component_Table( $u$ , Comp,  $nb$ );
6    $nb \leftarrow nb + 1$ ;
7 return Comp

```

Algorithm 4: Algorithm `Update_Component_Table`.

```

1 Input:  $T$ : an integer table,  $u$ : element of  $T$ ,  $k$ : an integer; Output:  $T$  updated;
2 if  $T[u] = 0$  then
3   if  $n(u) \neq 0$  then
4      $T[u] \leftarrow k$ ;
5      $v \leftarrow$  Parent of  $u$  in  $(V_E, S_E)$ ;
6     Update_Component_Table( $v$ ,  $T$ ,  $k$ );
7   for all children  $v$  of  $u$  in  $(V_E, P_E)$  do
8     if  $d(v) \neq 0$  then
9        $T[u] \leftarrow k$ ;
10      Update_Component_Table( $v$ ,  $T$ ,  $k$ );

```

Proof. By Proposition 11, we can compute the table `Comp` in linear time. Using `Comp`, we can easily obtain the node with the least string depth of each connected component. \blacktriangleleft

► Proposition 13. Let P be a set of strings. We can compute ℓ_{\min} and ℓ_{\max} in linear time in $\|P\|$.

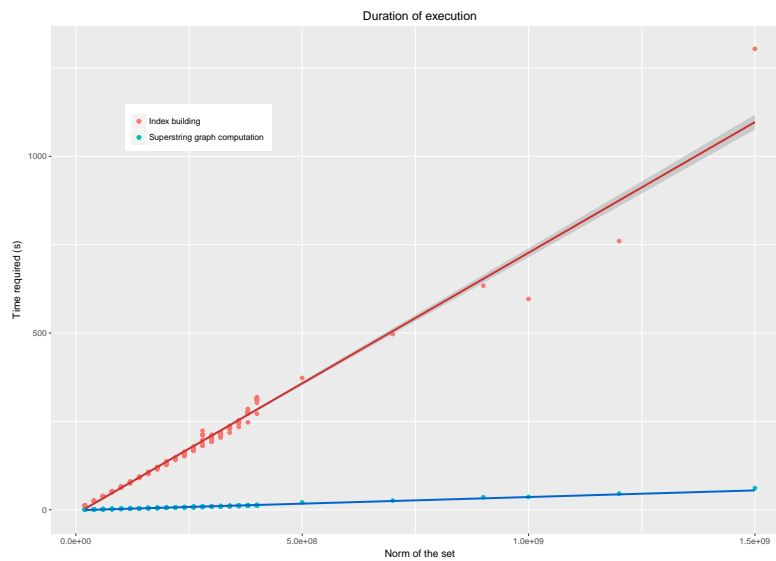
Proof. By Proposition 8, we have that $\ell_{\max} = \|P\| - \sum_{u \in V_E} \text{OvsG}(u) \times |u| + \text{Cut}(P)$. By Proposition 10, Algorithm 2 computes $\text{OvsG}(V_E)$ in linear time in $\|P\|$. By Proposition 12, we can compute $\text{Cut}(P)$ in linear time in $\|P\|$. Hence, it follows that we can compute ℓ_{\max} in linear time in $\|P\|$.

By Proposition 7, we have that $\ell_{\min} := \max\left(\frac{\ell_{\max}}{4}, \|P\| - \sum_{u \in V_E} \text{OvsG}(u) \times |u|\right)$ can be computed in linear time in $\|P\|$. \blacktriangleleft

4.1 Empirical results

We performed experimental tests to check how close the bounds ℓ_{\min} and ℓ_{\max} are from an optimal superstring length. We used one synthetic dataset and one real dataset from a genomic experiment on the *E. coli* genome (Strain K-12 substrain MG1655); the data is available at <https://github.com/PacificBiosciences/DevNet/wiki/EcoliK12MG1655HybridAssembly>.

Results on synthetic data. We randomly generated large sets of words of length 100 for a DNA alphabet (4 symbols). The model for random words is an unbiased Bernoulli model. The instances have an increasing number of words.



■ **Figure 2** Execution times in function of the norm of the input set for i/ building the index (red dots) and for ii/ computing the Superstring graph and the solution (blue dots).

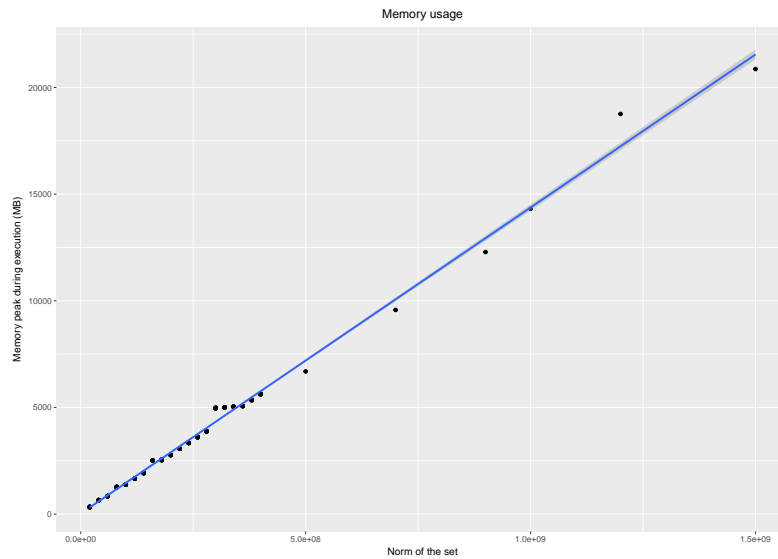
1. From 200,000 to 4,000,000 words with a step of 200,000 words. The norm of such instances goes from 20 to 400 million symbols. For each size of instances, we ran 10 generations and executions, and took the average times, and memory usages.
2. Then instances of 500, 700, 900, 1,000, 1,200 and 1,500 million words of length 100.

Test were run using a single core on a desktop machine (x86_64 processor) running Linux 4.13.0-26-generic with 32 gigabytes of RAM.

Running times are displayed in Figure 2, and for each run, the largest memory consumption over the entire execution is shown in Figure 3. Most of the time is spent, and most of the memory used, while building the EHOg with COvI. Comparatively, the computation of ℓ_{\min} and ℓ_{\max} becomes rapidly at least an order of magnitude faster than COvI construction. The peak of memory for the largest instance, with 1.5 billion words, reached 22 gigabytes. In turn, COvI construction spends most of its time and space while building the Aho-Corasick automaton [3]. Thus, it would be advantageous to build the EHOg from a compressible and more compact index than COvI. However, the linear increases of running time and memory usage with the norm of the instances suggest that **LCGreedyMin** is very efficient and scalable.

Our algorithm computes the length of an approximate superstring. However, with some modifications, it could also output the computed superstring rather than only its length. Surprisingly, for 67% of the instances ℓ_{\min} and ℓ_{\max} are equal. In the remaining instances their difference (*i.e.*, $\ell_{\max} - \ell_{\min}$) is at most 0.0001% of the norm of the instance. This shows that most instances are entirely or almost "solved" with **LCGreedyMin**. This is coherent with theoretical results [15].

Results on real data. We used a publicly available set of genomic reads obtained from an Illumina sequencing machine. The reads of length 100 make up a coverage on the *E. coli* genome of 50x, meaning that every position appears on average in 50 reads. Such data are designed for genome assembly purposes and thus contain a huge number of overlaps between the reads. The set contains 4,503,422 reads for a norm of 454,845,622 symbols.



■ **Figure 3** Memory peak (black dots) during execution in function of the norm of the input set.

Our algorithm ran on a simple core of a standard laptop equipped with 8 gigabytes of RAM; it took 272 seconds and used less than 5.5 gigabytes of memory. The EHO had 46,566,901 nodes. The Shortest Cyclic Cover had length 187,250,434, ℓ_{\min} was equal to 187,250,434, while ℓ_{\max} was 187,250,672 symbols long (41% of $\|P\|$), making a difference of 710 symbols. Hence, the results on real data confirm our observations on synthetic data.

5 Conclusions

Here, we provide an algorithm to compute practical lower and upper bounds on the length of an optimal superstring. Importantly, those bounds are computed in a deterministic way. They appear to be very tight in practice on synthetic and genomic data (although there is little to compare to due to the lack of published experiments on approximation of known algorithms). Theorem 2 gives an upper bound of 4 for the ratio between ℓ_{\max} and ℓ_{\min} . Empirically, this ratio is several orders of magnitude lower, meaning that the superstring is very close to the optimum. This result does not contradict the existence of a lower bound for SLS approximation ratio (see Figure 1 or [22]). For SLS, improving **MGreedy** into **TGreedy** algorithm led to an approximation ratio of 3. The same improvement is possible with our algorithm **MGreedyMin** and also would lead to the same ratio. This is left for future work.

Unfortunately, it is complex to understand why **MGreedyMin** yields an empirical ratio so close to the optimum. Several factors come into play. First, it turns out that the Shortest Cyclic Cover of P often contains a single cyclic word. In that case, this optimal cyclic cover also is an optimal cyclic superstring, which is necessarily shorter than the optimal linear superstring. Second, the cyclic superstring often corresponds to a path that uses the empty word as an overlap. In that case, the cyclic superstring can be cut between the two corresponding words and makes up a shortest linear superstring of exactly the same length, which is then optimal [6]. Another fact is important: if a cyclic string c_k of the cyclic cover merges at least two words of P , say s_i and s_j , then the difference between a shortest superstring of these words and c_k is smaller than the shortest word occurring in c_k .

The fact that **MGreedyMin** concatenates in an arbitrary order the linear strings to form a superstring makes no sense in DNA assembly or in genomics applications. The order of strings obtained by merging reads (which are called *contigs*) are determined *a posteriori* by a subsequent step of assembly pipelines named *scaffolding* using additional data like optical or genomic maps, long reads, or chromosomal capture data (Hi-C).

References

- 1 Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of ACM*, 18(6):333–340, 1975.
- 2 Avrim Blum, Tao Jiang, Ming Li, John Tromp, and Mihalis Yannakakis. Linear approximation of shortest superstrings. *Journal of the ACM*, 41(4):630–647, 1994. doi:10.1145/179812.179818.
- 3 Rodrigo Cánovas, Bastien Cazaux, and Eric Rivals. The Compressed Overlap Index. *ArXiv e-prints*, abs/1707.05613, 2017. arXiv:1707.05613.
- 4 B. Cazaux, R. Cánovas, and E. Rivals. Shortest DNA cyclic cover in compressed space. In *Data Compression Conference DCC*, pages 536–545. IEEE Computer Society Press, 2016.
- 5 B. Cazaux and E. Rivals. Hierarchical Overlap Graph. *ArXiv e-prints*, 1802.04632v2, 2018. URL: <https://arxiv.org/abs/1802.04632v2>.
- 6 Bastien Cazaux and Eric Rivals. A linear time algorithm for Shortest Cyclic Cover of Strings. *J. of Discrete Algorithms*, 37:56–67, 2016. <http://dx.doi.org/10.1016/j.jda.2016.05.001>.
- 7 Bastien Cazaux and Eric Rivals. Superstrings with multiplicities. In David Sankoff Gonzalo Navarro and Binhai Zhu, editors, *29th Annual Symposium on Combinatorial Pattern Matching, CPM 2018, July 2–4, 2018, Qingdao, China*, volume 105 of *LIPICs*, page in press, 2018. doi:10.4230/LIPICs.CPM.2017.5.
- 8 Maxime Crochemore, Marek Cygan, Costas S. Iliopoulos, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Algorithms for three versions of the shortest common superstring problem. In *Proc. of 21st Annual Symp. Combinatorial Pattern Matching (CPM)*, volume 6129 of *Lecture Notes in Computer Science*, pages 299–309. Springer, 2010. doi:10.1007/978-3-642-13509-5_27.
- 9 Gabriele Fici, Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. On the greedy algorithm for the Shortest Common Superstring problem with reversals. *Information Processing Letters*, 116(3):245–251, 2016.
- 10 Theodoros P. Gevezes and Leonidas S. Pitsoulis. *Optimization in Science and Engineering: In Honor of the 60th Birthday of Panos M. Pardalos*, chapter The Shortest Superstring Problem, pages 189–227. Springer New York, New York, NY, 2014. doi:10.1007/978-1-4939-0808-0_10.
- 11 A. Golovnev, A.S. Kulikov, and I Mihajlin. Approximating Shortest Superstring Problem Using de Bruijn Graphs. In *Combinatorial Pattern Matching*, volume 7922 of *Lecture Notes in Computer Science*, pages 120–129. Springer Verlag, 2013.
- 12 D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, 1997.
- 13 Tao Jiang, Ming Li, and Ding-Zhu Du. A note on shortest superstrings with flipping. *Information Processing Letters*, 44(4):195–199, 1992.
- 14 John D. Kececioğlu and Eugene W. Myers. Combinatorial algorithms for DNA sequence assembly. *Algorithmica*, 13(1/2):7–51, 1995.
- 15 Bin Ma. Why greed works for shortest common superstring problem. *Theoretical Computer Science*, 410(51):5374–5381, 2009.
- 16 Marcin Mucha. Lyndon words and short superstrings. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 958–972, 2013.

- 17 Katarzyna E. Paluch. Better Approximation Algorithms for Maximum Asymmetric Traveling Salesman and Shortest Superstring. *CoRR*, abs/1401.3670, 2014. URL: <http://arxiv.org/abs/1401.3670>.
- 18 Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization : algorithms and complexity*. Dover Publications, Inc., 2nd edition, 1998. 496 p.
- 19 Heidi J. Romero, Carlos A. Brizuela, and Andrei Tcherynykh. An Experimental Comparison of Approximation Algorithms for the Shortest Common Superstring Problem. In *5th Mexican International Conference on Computer Science*, pages 27–34, 2004.
- 20 Jorma Tarhio and Esko Ukkonen. A greedy approximation algorithm for constructing shortest common superstrings. *Theoretical Computer Science*, 57:131–145, 1988. doi:10.1016/0304-3975(88)90167-3.
- 21 Esko Ukkonen. A Linear-Time Algorithm for Finding Approximate Shortest Common Superstrings. *Algorithmica*, 5:313–323, 1990.
- 22 Virginia Vassilevska. Explicit inapproximability bounds for the shortest superstring problem. In *30th Int. Symp. on Mathematical Foundations of Computer Science (MFCS)*, volume 3618 of *Lecture Notes in Computer Science*, pages 793–800. Springer, 2005.
- 23 Maik Weinard and Georg Schnitger. On the greedy superstring conjecture. *SIAM Journal on Discrete Mathematics*, 20(2):502–522, 2006. doi:10.1137/040619144.

Experimental Study of Compressed Stack Algorithms in Limited Memory Environments

Jean-François Baffier¹

JSPS International Research Fellow — Department of Industrial Engineering and Economics,
School of Engineering, Tokyo Institute of Technology, Tokyo, Japan
baffier.j.aa@m.titech.ac.jp

Yago Diez²

Yamagata University, Yamagata, Japan
yago@sci.kj.yamagata-u.ac.jp

Matias Korman³

Tohoku University, Sendai, Japan
mati@dais.is.tohoku.ac.jp

Abstract

The *compressed stack* is a data structure designed by Barba *et al.* (Algorithmica 2015) that allows to reduce the amount of memory needed by a certain class of algorithms at the cost of increasing its runtime. In this paper we introduce the first implementation of this data structure and make its source code publicly available.

Together with the implementation we analyse the performance of the compressed stack. In our synthetic experiments, considering different test scenarios and using data sizes ranging up to 2^{30} elements, we compare it with the classic (uncompressed) stack, both in terms of runtime and memory used.

Our experiments show that the compressed stack needs significantly less memory than the usual stack (this difference is significant for inputs containing 2000 or more elements). Overall, with a proper choice of parameters, we can save a significant amount of space (from two to four orders of magnitude) with a small increase in the runtime (2.32 times slower on average than the classic stack). These results hold even in test scenarios specifically designed to be challenging for the compressed stack.

2012 ACM Subject Classification Theory of computation → Computational geometry

Keywords and phrases Stack algorithms, time-space trade-off, convex hull, implementation

Digital Object Identifier 10.4230/LIPIcs.SEA.2018.19

Related Version <https://arxiv.org/abs/1706.04708>, [8]

Supplement Material <https://github.com/Azzaare/CompressedStacks.cpp>

¹ Partially supported by JST ERATO Grant Number JPMJER1305, Japan.

² Partially supported by the Impact TRC project from Japan's Science and Technology agency.

³ Partially supported by MEXT KAKENHI No. 12H00855, 15H02665, and 17K12635.



© Jean-François Baffier, Yago Diez, and Matias Korman;
licensed under Creative Commons License CC-BY

17th International Symposium on Experimental Algorithms (SEA 2018).

Editor: Gianlorenzo D'Angelo; Article No. 19; pp. 19:1–19:13

Leibniz International Proceedings in Informatics



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

1 Introduction

Small computing devices have greatly increased their presence in our everyday lives in recent years. Most notably smartphones have become ubiquitous, but other devices such as sensors or security cameras have dramatically increased their number and everyday uses. These devices often have memory availability and computational power constraints due to price consideration, technology limitations or in order to discourage theft.

This growth tendency has coupled with the longstanding theoretical interest within the theoretical computer science community in the issue of memory usage in algorithmic design. This has yielded the creation of *time-space trade-off* algorithms [4]). These algorithms are designed so they work regardless of the amount of memory available but their performance improves with memory availability. We refer the interested reader to [15] for a survey on the different models that have been proposed to handle space constraints.

The major interest from a theoretical point of view is the relationship between time and space. In most cases, the dependency has been linear or almost linear [2, 3, 11, 16, 17]: that is, when we double the amount of available space we expect the runtime to more or less halve. However, several exceptions are known. For example, the best algorithm for the well-studied *minimum spanning tree* problem (given a set of points in the plane, embed a tree that spans all of them with the shortest possible length) needs cubic time to be solved in the presence of space constraints [5] (whereas it is solved in almost linear time when sufficient memory is available [13]).

On the positive side, for some problems we know that the dependency can be exponential. The focus of this paper is the *compressed stack* data structure introduced by Barba *et al.* [10]. This data structure applies to any deterministic incremental algorithm whose internal structure is a stack (more details below). When the space available is very small (say, s words of space for some s that is less than logarithmic in size when compared to the input), then the runtime is $O(n^2 \log n / 2^s)$. Notice that the dependency is exponential: increasing the memory by a small constant we can *halve* the runtime. When the amount of space grows the dependency quickly becomes logarithmic (we need to *double* the amount of space to see any difference in the runtimes).

Another interesting property of the structure is that, if properly implemented, the algorithm is unaware of which data structure it utilizes: we only need to replace the classic (uncompressed) stack data structure with a compressed stack to obtain an algorithm that uses less memory. This modular transparency makes it very easy for users to adopt. More interestingly, it provides the ideal setting for comparison purposes.

We believe that this (theoretical) exponential dependency shows potential for usability in practical applications. In this paper we take a more hands-on approach on the topic. Rather than studying theoretical dependency, we implement the compressed stack and thoroughly assess its behavior when executed using benchmark data of varying difficulty. Specifically, we are interested in seeing how close the theoretical bound is to real runtimes, and if so what would be a good balance between time and space (that is, how much can we reduce the amount of memory consumed while making sure that runtimes remain reasonable). In order to tell the full story, it is also important to state that the use of the compressed stack is circumscribed to situations of high memory consumption. Thus those are the situations that we will consider throughout the paper. In those situations where the number of elements present in the stack is low enough not to ever represent a memory problem one should just use the classic stack.

Thus, our study has two objectives. First, to verify that the theoretical dependency between time and space actually matches practice. Also, we want to provide some guidelines on how to find this breakpoint in the dependency so that the user can choose the right amount of memory to achieve the fastest algorithm that fits their memory constraints.

1.1 Results and Paper Organization

Our main contribution is the implementation of the compressed stack data structure of Barba *et al.* [10]. This implementation is freely available at [6]. With the use of this library, one can implement any algorithm that uses this data structure quickly and efficiently (see examples as Problem 1 and in [8, Subsection 3.6]).

In Section 2 we give a brief overview of stack algorithms and the compressed stack data structure. In Section 3 we give a brief overview of our library. Due to lack of space, further description of the library as well as discussion of the minor differences between our implementation and the theoretical formulation by Barba *et al.* is available in [8, Section 3].

In Section 4 we present a thorough study on the behaviour of the compressed stack. Our study is based on two scenarios (one favourable and one unfavourable for the compressed stack). In our experiments we decided to use synthetic data. The main reason for this is that synthetic data allows us to focus only on the time and amount of memory needed by the stack data structure. Stack algorithms perform other computations apart from handling of the stack itself. For example, when computing the convex hull of a set of points we need to do triple orientation checks to decide if a point is popped from the stack (see Subsection 2.1 for a description of the problem and overview of the algorithm). These checks compute the determinant of a small matrix and when all of them are put together, they have a significant impact on the runtime. Although it is technically feasible to measure the computation time spent only in handling the stack for any given algorithm, this introduces precision errors and complicates the discussion of results. Furthermore using artificial data allows us to fully control the parameters in our experiments (input size, number of pushes and pops...). This allows us to focus on the properties that we want to study and give a clear view of the strong and weak points of the compressed stack.

As expected, the compressed stack structure uses significantly less memory than the classic stack. From the theory we know that the running times must increase, but only the asymptotic trend can be also empirically observed. From the experiments done in this paper we can deduce more clear guidelines for prospective users so that they can drastically reduce the amount of memory consumed while we keep the runtimes relatively low. Further discussion about parameter settings is done in Section 5.

2 Preliminaries

The compressed stack data structure can only be used with a certain family of algorithms (called *stack algorithms*). This class includes widely used algorithms addressing problems such as computing the convex hull of a set of points, approximating a histogram by a unimodal function, or computing the visibility region of a point inside a polygonal domain. See [9, 10] for more examples of stack algorithms.

In full generality, we look at algorithms whose input is a list of elements $\mathcal{I} = \{a_1, \dots, a_n\}$, and the goal is to find a subset of \mathcal{I} that satisfies a previously defined property. In a nutshell, we are looking at deterministic incremental algorithms that use a stack, and possibly other small data structures $\mathcal{C} \uparrow \Downarrow$ (this additional structure is called the *context* and ideally only consists of a few integers).

A stack algorithm solves the problem in an incremental fashion, scanning the elements of \mathcal{I} one by one. At any point during the execution, the stack keeps the values that form the solution up to that point. For each new element a that is taken from \mathcal{I} , the algorithm pops all values of the stack that do not satisfy a predefined “pop condition” and if a meets some other “push condition”, it is pushed into the stack. The algorithm then proceeds to the next element in \mathcal{I} until all elements have been processed. The final result is normally contained in the stack, and at the end it is reported. This is done by simply reporting the top of the stack, popping the top vertex, and repeating until the stack is empty. Thus, an algorithm \mathcal{A} that follows this scheme is called a *stack* algorithm (see the pseudo-code in [8, Algorithm 1]).

2.1 Sample problem: convex hull computation

A typical example of a stack algorithm is the convex hull problem: given a list of points p_1, \dots, p_n in the plane sorted in increasing values of their x -coordinate, we want to compute their convex hull, i.e., the smallest convex set that encloses all of them. Among the many algorithms that solve this problem, [1]⁴, the one by Lee [18] falls in the class of stack algorithms.

The algorithm processes the points sequentially and stores the elements that are currently candidates for being in the convex hull. For simplicity, we discuss how to compute the upper hull (i.e., points that are in the convex hull and are above the line passing through the rightmost and leftmost point) of a set of points⁵.

When a new element is processed it may be witness to several points that were previously in the upper hull and should not be there any more (see [8, Figure 1]). The key property is that those points must be the last ones that were considered as candidates. Consequently, they are removed in reverse order of insertion and thus a stack is the perfect data structure to store the list of candidates.

We refer the interested reader to [14] for more details on the convex hull problem and its applications. As an illustration on the simplicity in implementing stack algorithms, we have implemented this algorithm as part of our library (details are given in [8, Subsection 3.6]).

2.2 Compressed Stack Overview

During the execution of a stack algorithm, one may have many elements of the input in the stack. In a classic stack these elements are stored explicitly, which may cause high memory requirements.

In the compressed stack structure we do not explicitly store all elements. First, the user chooses a parameter p to indicate the amount of space that the algorithm is allowed to use (the larger the p , the more memory it will use). Then the input is split into p blocks. Whenever a block has been fully processed (i.e., the incremental algorithm has scanned the last element of the block) we *compress* the block: rather than explicitly storing whatever part of it has been pushed into the stack, we store a small amount of information that can be used to afterwards determine exactly which elements were pushed.

⁴ The algorithms in this survey actually compute a slightly more general problem: computing the convex hull of a simple polygon, but both problems are almost identical. The simple polygon case has a few more difficult cases (such as when the polygon spirals around itself), but they have no impact on the way in which the stack is handled. Thus, for simplicity we only describe the simpler case.

⁵ The traditional algorithm scans once the input to compute the upper hull and a second time to compute the lower hull, but we note that the same algorithm can be modified to work in one pass by using two stacks. In any case, neither of these two options have a large impact on the overall working of the stack algorithm, so we ignore this and focus in the upper hull only.

Naturally, this saves a lot of memory, since a block could have many elements in the stack but only a fixed amount of information per block is stored instead. Since we scan the input in a monotone fashion only one block is partially processed at any instant of time. We store that block and its preceding block in *uncompressed format* (i.e., almost explicitly), while all other ones are compressed⁶.

Stack algorithms have access only to the top of the stack at any given time. Since the top element is part of an explicitly stored block, its information is always known and can be accessed as with a usual stack. Eventually, the algorithm may pop many elements, and then the information inside a block that was compressed will be needed. This information can be *reconstructed* by re-executing the same algorithm, but only restricted to a portion of the input. The key trick to keeping the runtime small is to make sure that few reconstructions are needed, and always restricted to small portions of the input.

We emphasize that the working of the compressed stack is *transparent* to the algorithm. The algorithm is running, does some push and pop operations as well as reading the top of the stack. The stack data structure handles compression of information independently. Sometime during the execution, a pop will trigger a reconstruction. In this moment, the algorithm is paused and we launch a copy of the same algorithm (with a smaller input). Once the small execution ends, the needed information is available in memory, and we can resume with the main execution.

From a theory standpoint, when the memory available is very small (i.e, we can only use s words of space for some $s \in o(\log n)$), stack algorithms run in $O(n^2 \log n / 2^s)$ time. When the space available becomes $\Theta(\log n)$ the dependency in the runtime changes; the algorithm runs in $O(n^{1+\frac{1}{\log p}})$ time using $O(p \log_p n)$ space for any parameter $p \in \{2, \dots, n\}$. In particular, when p is a relatively large number (say, $p = 500$) the algorithm runs in slightly superlinear time, and uses logarithmic space. On the other hand, when $p = n^\varepsilon$ the algorithm runs in linear time and uses $O(n^\varepsilon)$ space. For comparison purposes, the usual stack runs in linear time and uses linear space, so the latter case consumes more memory without reducing the runtime (this is of course from a theoretical point of view. Since additional reconstructions are needed the runtime will increase).

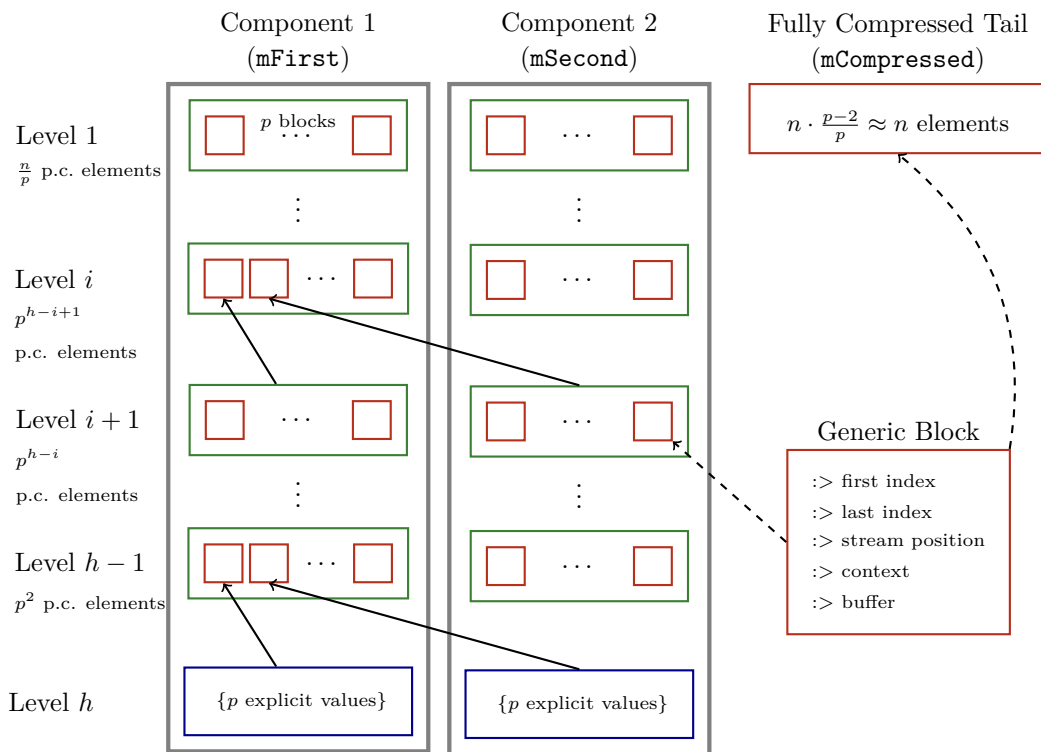
3 Implementation

In this section we give a brief overview of the `CompressedStacks.cpp` library. A complete description is given in [8, Section 3]. This library was implemented following the C++11 standard, and is available at [6] as an open source library under the MIT license.

Our stack algorithm library consists of three main classes: the `Data` class that handles the input, the `CompressedStack` class that instantiates transparently a space-optimized stack structure, and the `StackAlgorithm` class itself.

The `CompressedStack` implements the compressed stack data structure. As such, it handles push and pop operations (as well as all extra computations needed for compression). See Figure 1 for an overview of how data is stored in this class. The key trick of saving space is to partition the input into blocks, that are recursively partitioned into blocks, and so on. As introduced by Barba *et al.* [10], a block may be stored *explicitly* (if it stores all elements of the block that have been pushed into the stack), *partially compressed* (which allows reconstructing any portion of the block) or *fully compressed* (which allows fully reconstructing the block). This data structure compresses information that is further down

⁶ There are two ways in which the block can be compressed, but this is not relevant for the overview.



■ **Figure 1** General sketch of a Compressed Stack: red boxes are blocks, green boxes are levels (vector of blocks), blue boxes are explicit values, plain arrows shows the partial compression (p.c.) of a level $i + 1$ into a block at level i . Recall that p is a parameter set by the user (denoting how much to compress the data), and that $h \approx \log_p n$ will denote in how many levels we subdivide the input.

in the stack, thus unlikely to be accessed in the near future. Depending on the value of the parameter p (set by the user) we may compress more or fewer blocks. For comparison purposes we also provide an implementation of a classic stack that stores the input explicitly (and does not do any data compression).

The `StackAlgorithm` is a class used for solving the different stack algorithms. This is the class that the user needs to implement depending on his or her application. This class must be inherited and then a few operations like `pushCondition` or `popCondition` must be implemented (see more details in [8, Section 3]). The implementation pays special attention to modularity, so the stack algorithm is *transparent* to the kind of stack that is actually being used. That is, the algorithm sends push and pop requests and need not know if they are being handled by a regular or a compressed stack.

All modifications needed to deal with space constraints (if used) are handled by the compressed stack class. Our library contains two examples of compressed stack algorithms. One is the upper hull problem (described in section 2.1) and the other is a synthetic problem described below.

► **Problem 1 (Test Run).** *This is an artificial stack algorithm for experimental and debugging purposes. It reads the whole input and executes push and pop operations as requested. The data type D is a pair of positive integers separated by a comma. The first number indicates the value to be pushed into the stack whereas the second indicates the number of pops that should be done after processing the first value (in lines 4-13 of [8, Algorithm 2]).*

4 Experimental Results

4.1 Data and evaluation measures

As mentioned in the introduction our experiments use synthetic data. Keeping in mind that our aim is to measure the differences between using a regular and a compressed stack we have tried to keep additional overhead to a minimum. In general, stack algorithms spend a significant amount of time in computations related to the problem being solved, but not related to the stack data structure being used. For example, in the upper hull algorithm, the `popCondition` operation must compute the determinant of a 3×3 matrix in order to determine whether a point is a candidate to belong to the upper hull. These computations can affect the leading term for the running time, obscuring the difference in performance of both stack types. Consequently, we present experiments using the `testrun` problem described as Problem 1 in Section 3 (an artificial problem with the smallest possible overhead).

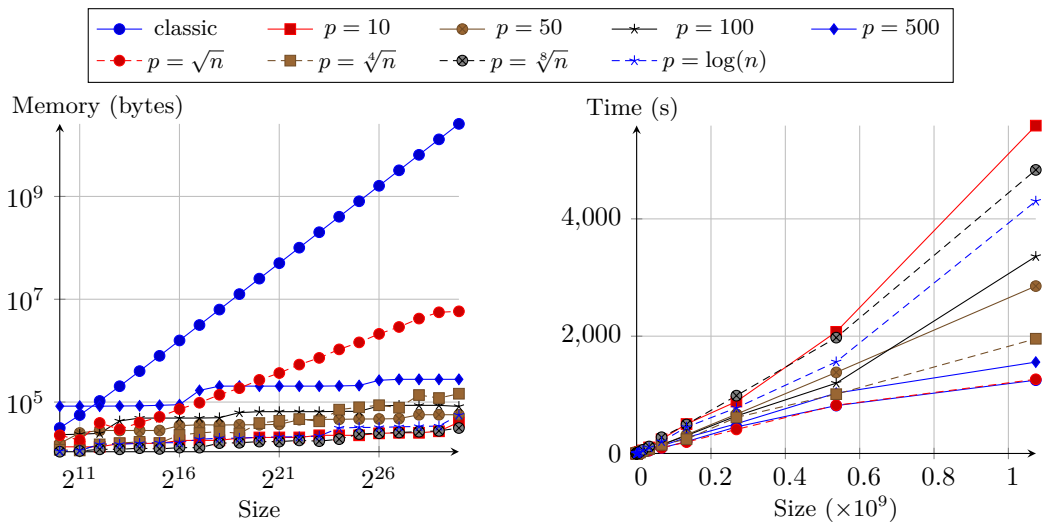
Another interesting property of synthetic data is that it gives us full control over the conditions in which both stacks have to work (for example, we can control the sparsity of the data and number of reconstructions that will be executed). We use this to our advantage and create two completely different scenarios in terms of the difficulty of data compression. The first experiment represents a very favourable situation for the compressed stack: a case in which data is accessed almost sequentially (and thus it is ideal for compression purposes). The goal in this case is to show the potential of memory saving that this data structure can achieve. The second test aims at setting a much more challenging scenario: continuous pops are set in a way such that scattered positions of the input file are accessed. This forces the compressed stack to repeatedly reconstruct portions of the input, and thus potentially lose a lot of time when compared to the classic stack.

For comparison purposes, all problem instances generated are solved with both the classic and compressed stack. In order to measure the performance we focused on two quantities: the maximum amount of memory used by the algorithm and the running time. To measure the first, we used a heap profiler called *massif* [19] belonging to the Valgrind software suite. This software keeps track of the memory allocated in the execution heap at intervals of predefined length and outputs detailed heap memory usage. While this software allowed us to measure maximum memory usage, its use made the running of the algorithms much slower.

The second quantity that we were interested in was run-time, we needed to make two separate runs for every test case. In the first execution, we run *massif* alongside our code, obtaining memory usage data. In the second execution we run the test code alone in order to obtain unadulterated run-time readings. Consequently, in this section we present memory usage readings as reported by *massif* (in bytes) as well as the times of the algorithms (in seconds) when run without *massif*. In order to be able to present values for widely different sizes, in each execution, we doubled the size of the input n (of size $n = 2^i$ for increasing values of i).

The behaviour of the compressed stack highly depends on the ‘ p ’ parameter introduced in Subsection 2.2. For the purposes of this experiment, it suffices to know that the larger p is, the more space is used by the compressed stack (and fewer reconstructions are needed).

In order to illustrate the effect of this parameter in the performance of the compressed stack, we present results for eight different values of p . Specifically, the first four values are fixed (10, 50, 100 and 500) while the other four change with the size of the input n : \sqrt{n} , $\sqrt[4]{n}$, $\sqrt[8]{n}$ and $\log n$. Fixed values allow us to illustrate how an imbalance between n and p may result in very high running times (or memory requirements). In order to obtain more



(a) Memory comparison classic vs compressed. (b) Time comparison classic vs compressed. No scaling in either axis is done for this figure. For ease of visualization, a logarithmic scale was used in both axes.

■ **Figure 2** As expected from theory, the classic stack uses a linear amount of space in a linear-space problem instance. In comparison, the compressed stack only uses a logarithmic space. Regarding runtime, the classic stack has, as expected, the best performance of all. However, for large values of p the running time is almost the same as of the classic stack (indeed, on average the compressed stack with $p = \sqrt{n}$ is only 1.031 times slower than the classic stack).

balanced values for all size ranges, we use varying values for p (as a function of n). In order to keep the section within reasonable length limits, we only present summary figures of memory usage and running times. Detailed tables can be downloaded at [6, wiki section].

4.2 Linear sized stack

In this first test we aim at creating a scenario that maximised the possibility of memory saving by the compressed stack with minor impact on the runtime. We consider the case in which the stack contains a linear fraction of the input. Specifically, fix a probability $\rho \in [0, 1]$; then every element of the input is pushed, and a pop will be executed with probability $1 - \rho$. In terms of the testrun problem defined, this stood for an input made up of a text file with a list of pairs of integers. The first integer is the actual number that will be pushed into the stack (whose exact value is irrelevant and thus was a random integer) and the second was chosen to be equal to one with probability $1 - \rho$, or zero otherwise (recall that the second number is the number of pops to be executed).

After processing the whole input, the expected number of elements in the stack is ρn , and thus the memory used by the classic stack will be linear. Instead, the compressed stack will only store a logarithmic amount of elements (the exact number will depend on the value of the p parameter). Figures 2b and 2a show the memory used and runtime of our experiments for the case in which $\rho = 1$ (and thus no pops are ever executed).

While we acknowledge that this is not a realistic situation, it does highlight the potential saving of space achieved by the compressed stack in cases where a large portion of it does not change. Moreover, in order to simulate more realistic situations we repeated the experiment

with different values of ρ . In all cases, the tendencies observed were similar to the case without pops: the larger the value of ρ the fewer pops are executed, thus the more memory is needed (for example, the compressed stack with $p = \sqrt[4]{n}$ the memory used when $\rho = 1$, is between 1.6 and 2 times that of $\rho = 0.1$).

Figure 2a depicts the maximum amount of memory needed in this test. A logarithmic scale (of base 2 for the x axis and base 10 for the y axis) is used for ease of visualization. The figure shows how in this test the classic stack needs much more memory than any of the compressed variants. Note that, as expected, the memory consumed by the classic stack grows linearly with the input size.

There are two exceptional cases in which the classic stack uses less memory than a compressed stack algorithm, but it only happens for extremely large values of p when compared to the input size (specifically, the compressed stack with $p = 500$ and input sizes of $2^{10}, 2^{11}$). This shows how the choice of parameter p is important to optimize the performance of the compressed stack. In both cases p is too close to n ($n = 2^{10} = 1024$) and the structure of the stack is wasted as most values are kept explicitly nonetheless.

The memory needed by the normal stack exceeds that of any compressed variant by two orders of magnitude already by size 2^{19} . This difference grows together with n (reaching four orders of magnitude for 2^{29}). The maximum memory needed by the classic stack in the test was over 25 Gigabytes for an input file of size 2^{30} (over 1000 million). Conversely, the compressed stack, in the worst case, only needed 5.8 Megabytes.

If we compare the different values of p for the compressed stack, we observe that, as expected, smaller values of p result in lower memory usage. Moreover, the results of algorithms with fixed values of p match the ones of variable p at expected values (for example, the algorithm with $p = 500$ should perform like the algorithm with $p = \sqrt{n}$ when $\sqrt{n} = 500 \leftarrow n = 250000 \approx 2^{19}$, and the two curves meet around that value). For fixed values of p it is easy to see that the memory grows logarithmically (specifically, we see a bump in the memory requirements when the value of $\lceil \log_p n \rceil$ changes), whereas the growth is smoother for variable values of p . In both cases the growth is very monotone, which matches the expectation from theory.

The downside of using a small value of p is that the running time will increase. The effect of the growth of p is only mildly visible in Figures 2a and 2b but will be more evident in Subsection 4.3. The reason for this is that the number of times that the reconstruct function is invoked is small, and the major time sink of the compressed stack is in the reconstruct operation. We defer a deeper analysis on runtime to the next experiment.

4.3 A challenging scenario for the compressed stack

We now consider a different test scenario that is tailored to be difficult for the compressed stack: we set the input to produce push-pop cycles in a way that forces many reconstructions. Moreover, the values that are pushed are at non-contiguous positions, making it difficult for the information to be compressed. We can also observe how the overall data that needs to be stored grows, but not at a linear rate. This is again a very artificial construction, but we believe that it shows that even under difficult conditions the compressed stack performs reasonably well. In order to create this setting, the instance forces the following operations into the stack:

- Push 8 elements, pop half of them (4).
- Repeat the previous step 8 times. At this point we have processed 64 elements and keep half of them (32) in the stack.
- Pop half of the stack, resulting in a stack of 16 elements.

19:10 Compressed Stack Algorithms

- Repeat this double loop 8 times, resulting in 128 elements in the stack after 512 elements have been processed.
- We again pop half of the stack, keeping only 64 elements in the stack.
- Repeat now the triple loop 8 times, and so on

This procedure keeps adding cycles of increasing length until the desired input size n is reached. The stack stores 4 elements that are consecutive in the input, but the spacing between numbers to store grows exponentially, creating a very difficult situation for the compressed stack (since reconstruction operations will have to scan large portions of the input for just a few elements that are stored explicitly in the regular stack).

We call this test the *Christmas tree* test (because the height of the stack forms a Christmas tree-like shape, see [8, Figure 9] for a graphical representation on the number of elements present in the stack as a function of the input size. As in the first scenario, we run Christmas tree instances whose total number of elements is a power of two. Note that the choice of making loops with 8 iterations (and popping half of the stack at each step) is arbitrary.

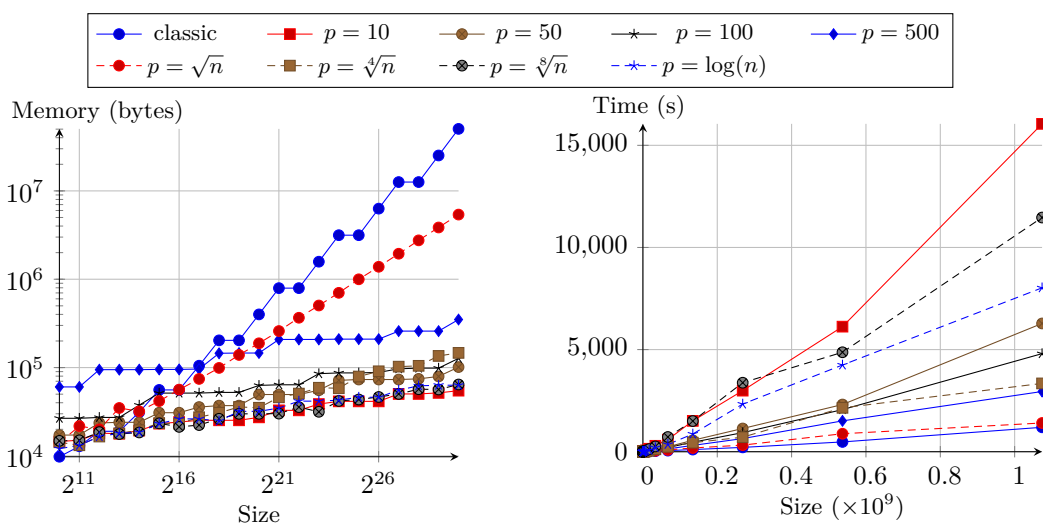
The main advantage of using a power of 2 for the number of iterations is that the memory usage patterns become easier to predict: imagine that for some value of n , the problem instance finished with a large pop removing half of the elements in its stack. When we double the instance size, the second half of the execution is spent in filling in the stack. Because the stack had been emptied, both executions are expected to use the same amount of memory.

Figure 3a shows the memory used in the Christmas tree test by all different stacks. Notice that the previously described stepwise pattern in memory usage is clearly visible. We also note that the amount of memory used by the classic stack is significantly lower than in the previous experiment. Again, this is because of significantly fewer elements are added into the stack: every factor of eight that the input grows, the space requirements only grow by a factor of 4, hence the stack has roughly $n^{2/3}$ elements.

We observe that at the amount of memory needed by the different parameters of the compressed stack is very similar to the one needed in the previous experiment. Indeed, for fixed values of n and p , the ratio of memory consumption in a linear-sized instance by the one needed by a Christmas-tree instance is very close to 1 (it depends on the exact value of n and p , but the average is 1.05 and variance is 0.1 or lower). This is consistent with the inner working of the compressed stack. Indeed, regardless of how many elements have been pushed, the compressed stack just stores a small amount of them explicitly (the exact amount will depend on n and p). The variation between the amount of memory needed by different instances happens because our implementation prioritizes saving as much memory as possible. Thus, if some block is empty we release that block of memory (and memory used by the smaller blocks nested inside).

The amount of memory saved by the compressed stack is significantly smaller than in the previous problem instance. This is to be expected, since fewer elements are pushed into the stack (and thus the classic stack needs less space). Indeed, for small values of n and extreme values of p the compressed stack even needs more memory than the classic stack. However, even in such an ill conceived example, when we use proper parameters of p is used (for example $p = \sqrt{n}$), the maximum memory required by a compressed stack is two orders of magnitude smaller than the classic stack.

The Christmas tree instance is specifically designed to force a lot of reconstructions. These reconstructions are computationally expensive, and as such we see a larger difference in runtimes between the classic stack and the compressed one (see Figure 3b). We observe that, although for small values of p the running times become infeasible, for larger values the runtime is comparable to the one of a regular stack. For example, for $p = \sqrt[4]{n}$, the runtime is in average 2.32 times slower than the classic stack (4.50 times slower in the worst case).



(a) Memory comparison classic vs compressed (logarithmic scale used in both axes), CT test. (b) Time comparison classic vs compressed, CT test. Linear scale was used.

■ **Figure 3** Christmas Tree experiments. In this test, designed to be challenging for the compressed stack, the memory saving in respect to the classic stack is much smaller (and in a few instances the classic stack even needs less memory than some compressed stacks). Concerning time, the constant calls to the reconstruct function make the compressed stack much slower (as exemplified, for example by the behavior of the compressed stack with $p = 10$). However, even in this tailored scenario, we can see how the compressed stack maintains a capped memory usage as well as relatively low running times if the value of p is chosen appropriately (as can be seen for example, in the values of $p = \sqrt[4]{n}$).

4.4 Choosing the right value of p

From a theoretical point of view, setting p to be equal to a large constant gives the best performance for the compressed stack: a fixed value of p increases runtime over the classic stack by a logarithmic factor whereas space constraints are exponentially decreased. Values of p that depend on n have a worse time-space product.

Although this may be true in theory, in our experiments we have seen that a fixed value does not always perform well (for too small instances it may consume even more memory than the classic stack, and for larger instances the runtime may increase too much). We believe that in practical applications the value of p should depend on n . In this section we use the previous experiments to give guidelines on how to choose the correct value of p .

Naturally, the compressed stack is only recommended for cases in which the stack contains many elements (since otherwise a classic stack would be good enough). Now, assuming that the stack is going to be majorly full, there are two settings in which the compressed stack shines. In the first one we have a hard cap on the amount of memory available, and we need to make sure that the algorithms does not exceed that amount. In this case, we naturally want the largest possible value of p that keeps the memory requirements below the threshold.

A simple way to make sure this happens is to combine the compressed stack with a heap profiler. If at any point the memory constraints are exceeded, we stop and restart the algorithm with a smaller value of p . Although it will certainly work, such a brute force approach is not needed. In our experiments we observed that the memory used by the compressed stack is almost constant regardless of the scenario. We can use this to predict the amount of memory used without having to resort to a heap profiler. In our experiments,

we have observed that the amount of memory used is slightly below $300p\lceil\log_p n\rceil$ bytes. Of course, this value is not static and will depend on many parameters (such as operating system, compiler used, and so on).

More importantly, the exact constant will also depend in the type of elements that are pushed into the stack; For example, in our experiments we push integers. Our implementation of the compressed stack is abstract and can handle any data type, but the exact data type chosen will have an impact in the memory needed by the algorithms. In any case, given the stability of the space constraints of a compressed stack, it is not hard to predict the memory requirements of the compressed stack from a small sample of experiments.

Another scenario for the compressed stack to shine is when we want to overall keep memory requirements low, but no hard caps in the space requirements are present. In such cases, we believe that a practical compromise is obtained by fixing $p = \sqrt[n]{n}$: even in unfavourable scenarios it has a drastic reduction in memory consumption (orders of magnitude in difference as early as $n \approx 2^{15} \approx 32.000$) while it only brings a small increase in the runtime (in average 2.32 times slower than the classic stack).

5 Conclusions

Other than our preliminary implementation [7], this is the very first implementation of the compressed stack data structure. The source code along with the data from the experiments presented in this paper is available for download as [6]. Parallel to our work, a similar experimental study for another time-space trade-off problem previously only studied from a theoretical standpoint was done by [12]. Specifically, they studied the time-space dependency for the problem of computing the shortest path between two points in a simple polygon. We believe that a trend of similar studies will soon follow for these or related problems.

The reduction of memory of this data structure is undeniable, even in the very unfavourable scenario of the Christmas tree. Specifically, Subsection 4.2 presented a (synthetic) situation where a normal stack needed 25 Gigabytes of memory while compressed stack implementations needed at most 5.8 Megabytes. This situation represents a challenge for current desktop computers and is infeasible in even the more advanced mobile phones (Apple's iPhone 7, for example has 2 Gigabytes of RAM memory). Although this was a tailor-made case, it still showcases how the compressed stack nicely limits memory usage.

The drawback of the compressed stack is the increase in runtime when compared to a classic stack as known from its theoretical design. In this paper we have quantified how much of a penalty to expect as a function of p . This has allowed us to find a balance between the amount of memory saved and the increase in runtime. Most interestingly, we have observed that the amount of memory needed by the compressed stack is very stable regardless of the actual scenario. This makes the compressed stack very robust at holding memory limitations, even for situations in which we do not know much about the structure of the input.

References

- 1 Greg Aloupis. A history of linear-time convex hull algorithms for simple polygons, 2005. URL: <http://cgm.cs.mcgill.ca/~athens/cs601/>.
- 2 Boris Aronov, Matias Korman, Simon Pratt, André van Renssen, and Marcel Roeloffzen. Time-space trade-offs for triangulating a simple polygon. *Journal of Computational Geometry*, 8(1):105–124, 2017.
- 3 Tetsuo Asano, Kevin Buchin, Maike Buchin, Matias Korman, Wolfgang Mulzer, Günter Rote, and André Schulz. Memory-constrained algorithms for simple polygons. *Computational Geometry: Theory and Applications*, 46(8):959–969, 2012. Special issue of

- selected papers from the 28th European Workshop on Computational Geometry. doi:10.1016/j.comgeo.2013.04.005.
- 4 Tetsuo Asano, Kevin Buchin, Maike Buchin, Matias Korman, Wolfgang Mulzer, Günter Rote, and André Schulz. Memory-constrained algorithms for simple polygons. *Computational Geometry: Theory and Applications*, 46(8):959–969, 2013.
 - 5 Tetsuo Asano, Wolfgang Mulzer, Günter Rote, and Yajun Wang. Constant-work-space algorithms for geometric problems. *Journal of Computational Geometry*, 2(1):46–68, 2011.
 - 6 Jean-François Baffier, Yago Diez, and Matias Korman. Compressed stack library (C++). <https://github.com/Azzaare/CompressedStacks.cpp.git>, 2016.
 - 7 Jean-François Baffier, Yago Diez, and Matias Korman. Compressed stack library (Julia). <https://github.com/Azzaare/CompressedStacks.jl.git>, 2016.
 - 8 Jean-François Baffier, Yago Diez, and Matias Korman. Experimental study of compressed stack algorithms in limited memory environments. *CoRR*, abs/1706.04708, 2017. arXiv:1706.04708.
 - 9 Niranka Banerjee, Sankardeep Chakraborty, Venkatesh Raman, Sasanka Roy, and Saket Saurabh. Time-space tradeoffs for dynamic programming algorithms in trees and bounded treewidth graphs. In *COCOON*, pages 349–360, 2015.
 - 10 Luis Barba, Matias Korman, Stefan Langerman, Kunihiko Sadakane, and Rodrigo I. Silveira. Space-time trade-offs for stack-based algorithms. *Algorithmica*, 72(4):1097–1129, 2014. doi:10.1007/s00453-014-9893-5.
 - 11 Luis Barba, Matias Korman, Stefan Langerman, and Rodrigo I. Silveira. Computing the visibility polygon using few variables. *Computational Geometry: Theory and Applications*, 47(9):918–926, 2013.
 - 12 Jonas Cleve and Wolfgang Mulzer. An experimental study of algorithms for geodesic shortest paths in the constant workspace model. In *EuroCG*, pages 165–168, 2017.
 - 13 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. URL: <http://mitpress.mit.edu/books/introduction-algorithms>.
 - 14 Mark de Berg, Mark van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 3rd edition, 2008.
 - 15 Matias Korman. Memory-constrained algorithms. In *Encyclopedia of Algorithms*, pages 1260–1264. Springer, 2016.
 - 16 Matias Korman, Wolfgang Mulzer, André van Renssen, Marcel Roeloffzen, Paul Seiferth, and Yannik Stein. Time-space trade-offs for triangulations and Voronoi diagrams. In *Algorithms and Data Structures Symposium (WADS)*, pages 482–494, 2015.
 - 17 Matias Korman, Wolfgang Mulzer, André van Renssen, Marcel Roeloffzen, Paul Seiferth, and Yannik Stein. Time-space trade-offs for triangulations and voronoi diagrams. *Computational Geometry: Theory and Applications*, 2017. Special issue of selected papers from the 31st European Workshop on Computational Geometry. In press.
 - 18 Der-Tsai Lee. On finding the convex hull of a simple polygon. *International Journal of Parallel Programming*, 12(2):87–98, 1983. doi:10.1007/BF00993195.
 - 19 Nicholas Nethercote, Robert Walsh, and Jeremy Fitzhardinge. Building workload characterization tools with valgrind. In *IISWC*, pages 2–2, Oct 2006. doi:10.1109/IISWC.2006.302723.

Restructuring Expression Dags for Efficient Parallelization

Martin Wilhelm

Institut für Simulation und Graphik, Otto-von-Guericke-Universität Magdeburg
Universitätsplatz 2, D-39106 Magdeburg, Germany
martin.wilhelm@ovgu.de

Abstract

In the field of robust geometric computation it is often necessary to make exact decisions based on inexact floating-point arithmetic. One common approach is to store the computation history in an arithmetic expression dag and to re-evaluate the expression with increasing precision until an exact decision can be made. We show that exact-decisions number types based on expression dags can be evaluated faster in practice through parallelization on multiple cores. We compare the impact of several restructuring methods for the expression dag on its running time in a parallel environment.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis, Theory of computation → Computational geometry, Computing methodologies → Parallel algorithms

Keywords and phrases exact computation, expression dag, parallel evaluation, restructuring

Digital Object Identifier 10.4230/LIPIcs.SEA.2018.20

1 Introduction

In Computational Geometry, many algorithms rely on the correctness of geometric predicates, such as orientation tests or incircle tests, and may fail or produce drastically wrong output if they do not return the correct result [9]. The Exact Geometric Computation Paradigm establishes a framework for guaranteeing exact decisions based on inexact arithmetic, as it is present in real processors [13]. In accordance with this paradigm, many exact-decisions number types have been developed, such as `leda::real` [4], `Core::Expr` [5, 14], `Real_algebraic` [7] and `LEA` [1]. All four named number types are based on arithmetic expression dags, i.e., they store the computation history in a directed acyclic graph and use this data structure to adaptively (re-)evaluate the expression when a decision has to be made.

While all of these number types are able to make exact decisions, they are also very slow compared to standard floating point arithmetic. Therefore continuous effort is taken to make these data types more efficient. However, none of these number types implements a strategy to take advantage of multiple cores yet. In this work, we show that multithreading can improve the performance of expression-dag-based number types by presenting the design of a multithreaded implementation for `Real_algebraic`, as well as experimental results comparing it to its single-threaded version.

To enable an efficient parallelization, we implement several restructuring methods for the underlying data structure and compare them with respect to their effect on multithreading. We aim for using these techniques in a general purpose number type, for which the user need not worry about implementation details. Therefore we look specifically at situations in which restructuring increases the running time. We propose a new approach to avoid some of these situations and thus to lower the risk of worsening the performance.



© Martin Wilhelm;
licensed under Creative Commons License CC-BY
17th International Symposium on Experimental Algorithms (SEA 2018).

Editor: Gianlorenzo D'Angelo; Article No. 20; pp. 20:1–20:13



Leibniz International Proceedings in Informatics

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

1.1 Preliminaries

An (arithmetic) expression dag is a rooted ordered directed acyclic graph that is either

1. A single node containing a number or
2. A node representing a unary operation ($\sqrt[n]{}, -$) with one, or a binary operation ($+, -, *, /$) with two, not necessarily disjoint, arithmetic expression dags as children.

The number type `Real_algebraic` is based on the concept of accuracy-driven computation¹ [13]. Applying operations leads to the creation of an expression dag instead of the calculation of an approximation. When a decision has to be made, the maximum accuracy needed for the decision to be exact is determined and each node is (re)computed with a precision that is sufficient to guarantee the desired final accuracy².

Let $\text{val}(E)$ be the value represented by an expression dag E . When an approximation for $\text{val}(E)$ is computed, then for each node v in E , approximations for the children of v , and consequently for all of its descendants, must be computed before v itself can be processed. Hence the computations we have to perform are highly dependent on each other. Whether the evaluation of an expression dag can be efficiently parallelized is therefore largely determined by its structure. Generally, a shallower, more balanced structure leads to less dependencies and can be expected to facilitate a more efficient parallel evaluation.

1.2 Related work

Few attempts have been made to restructure arithmetic expression dags. Richard Brent showed in 1974 how to restructure arithmetic expression trees to guarantee optimal parallel evaluation time [2]. In 1985, Miller and Reif improved this strategy by showing how the restructuring process itself can be done in optimal time [6]. In our previous work, arithmetic expression dags are restructured to improve single-threaded performance by replacing tree-like substructures, containing only additions or only multiplications, by equivalent balanced trees [11]. We call this method *AM-Balancing*.

We implement a variation of Brent’s approach for tree-like substructures in arithmetic expression dags and compare it with AM-Balancing. Furthermore, we refine the algorithm based on practical observations. We do not use the strategy by Miller and Reif, since restructuring the expression dag is very cheap compared to the evaluation of bigfloat operations and therefore only minor performance gain, if any³, is to be expected.

2 Design

In this section we briefly describe our implementation of parallel evaluation and restructuring for the dag-based number type `Real_algebraic`. A more detailed description of the parallelization can be found in the associated technical report [12].

2.1 Parallelization

The running time for the evaluation of expression dags is dominated by the execution of bigfloat operations. They usually sum up to around 95% of the total running time. Therefore

¹ Also known, less accurately, by the name “Precision-driven computation”

² Actually, this is an iterative process with increasing accuracy and checks for exactness after each iteration.

³ Considerable effort would be needed to even neutralize the overhead from creating and managing different threads.

we focus on parallelizing bigfloat operations and allow for serial preprocessing. Accuracy-driven evaluation is usually done recursively with possible re-evaluations of single nodes. For an efficient parallelization, it is necessary to eliminate recalculations to avoid expensive lock-usage. Therefore we assign the required precision to each node in a (serial) preprocessing step and afterwards evaluate the nodes in topological order as proposed by Mörig et al. [8].

In our approach, we assign one task to every bigfloat operation. Tasks that can be solved independently are then sorted into a task pool, where they may be executed in parallel. Each dependent task is assigned a dependency counter, which gets reduced as soon as the tasks on which they depend are finished. When the dependency counter reaches zero, the task gets sorted into the task pool. With this strategy we reduce the shared data to a minimum, such that atomic operations can be facilitated at critical points to eliminate race conditions.

The maximum number of threads working in the task pool can be adjusted, depending on the hardware available. Our tests are run with a maximum number of four threads simultaneously working on the tasks plus the main thread, which stays locked during the computation.

2.2 Restructuring

In AM-Balancing, so-called “operator trees”, i.e., connected subgraphs consisting of operator nodes with only one predecessor, are replaced by equivalent balanced trees if they represent a large sum or a large product. Since sums and products are associative, this can be done without increasing the total number of operations and therefore without a large risk of significantly increasing the running time, aside from some subtleties [11].

In this work we extend AM-Balancing, based on the tree restructuring by Brent [2]. In Section 2.2.2 we consider operator trees consisting of all basic arithmetic operations except roots and reduce their depth by continuously splitting long paths and moving the lower half to the top. In Section 2.2.3 we describe a modification to this algorithm, which avoids some steps that are particularly expensive.

2.2.1 Notation

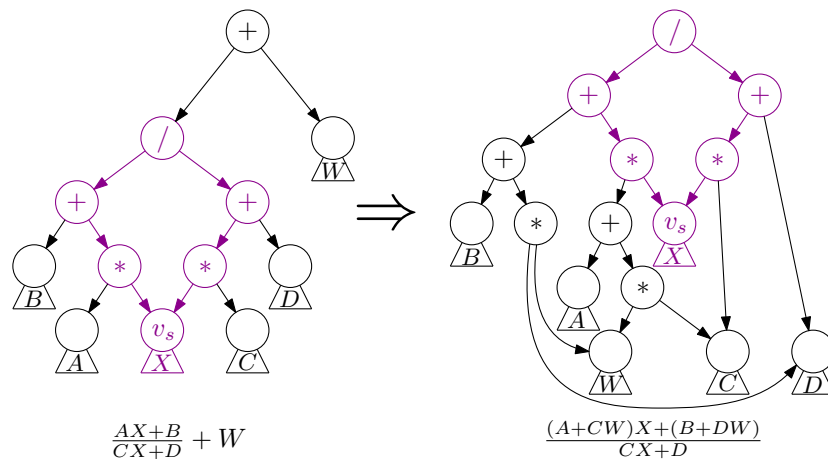
Let E be an expression dag. We call a subgraph T an *operator tree* if T is a maximal connected subgraph of E , consisting of nodes of type $\{+, -, *, /, - \text{ (unary)}\}$ such that no node except the root of T has more than one parent. We denote the set of operator nodes having one of the allowed operator types V_{op} . We call the children of the leaves of an operator tree T its *operands* and let $\phi(T)$ denote their number.

An operator tree is always a tree and all operator trees in an expression E are disjoint. Therefore each node $v \in V_{\text{op}}$ is part of exactly one operator tree $T(v)$. For each $v \in V_{\text{op}}$ we call the operands of $T(v)$ that are part of the subtree rooted at v the *operands of v* and denote their number by $\phi(v)$. We call a path from the root of an operator tree to a leaf *critical* if each node on the path has at least as many operands as its siblings.

2.2.2 Move-To-Root Restructuring

We briefly describe our variation of Brent’s algorithm, which we refer to as *Move-To-Root (MTR) Restructuring*. Each operator tree T in an expression dag E is restructured separately. In each node v in T we store $\phi(v)$, i.e., the number of operands of v . Then we search for a split node v_s on a critical path of T , such that $\phi(v_s) \leq \frac{1}{2}\phi(T) < \phi(\text{parent}(v_s))$.

Let X be the subexpression at v_s . We restructure T , such that it now represents an equivalent expression of the form $\frac{AX+B}{CX+D}$ with (maybe trivial) subexpressions A, B, C, D .



■ **Figure 1** Incorporating an addition into an expression of the form $(AX + B)/(CX + D)$. The main structure of the expression is highlighted before and after the restructuring. The expression dag grows by two additional multiplications and one additional addition due to the denominator.

Starting with the expression X at v_s , we raise v_s to the top of the tree while maintaining the form $\frac{AX+B}{CX+D}$. We say that we *incorporate* the operations along the way from v_s to the root into the expression. The restructuring needed to incorporate an addition is shown exemplarily in Figure 1.

After restructuring, the length of the path from the root to v_s is reduced to a constant. The same applies to the respective root nodes for the expressions A , B , C and D . The algorithm then recurses on these nodes, i.e., the operator trees representing X , A , B , C , D are restructured.

2.2.2.1 Comparison to Brent's algorithm

Like MTR Restructuring, Brent's original algorithm searches for a split node v_s on a critical path and raises it to the top of the expression tree. However, it does so in a more sophisticated manner by repeatedly splitting the path from the root to v_s in half and at the same time balancing the "upper half" of the tree, i.e., the part of the tree that is left when removing the subtree rooted at v_s . MTR Restructuring uses a simpler approach, which moves the split node to the top first and balances the remaining parts afterwards. It is easier to implement, but leads to an increased depth when subexpressions are reused, since nodes cannot be restructured as part of a larger operator tree if they have multiple parents. We chose this approach, since it is better suited to test possible improvements and can still be expected to behave similar to Brent's algorithm in many situations.

2.2.3 Parameterization of MTR Restructuring

Different operations on bigfloats have different costs⁴. In the algorithm introduced in the previous section, divisions in the upper half of a critical path are risen to the top. Since divisions are expensive, this is beneficial if two divisions fuse and one of them can be replaced by a multiplication. However, each addition or subtraction that is passed adds one or two expensive multiplications to the expression (cf. Figure 1).

If this affects a large number of additions and subtractions, the benefit of raising the division vanishes. If the number of cores and therefore the expected gain from an optimal

⁴ In our tests on `mpfr` bigfloats, multiplications take on average 10-20 times as long as subtractions or additions and divisions take 1.5-2 times as long as multiplications. Interestingly, this behavior appears

Algorithm 1: The main part of Parameterized MTR Restructuring. A counter for the number of additions and subtractions above the current node is maintained, which may lead to a split at division nodes. If the current node is a split node, an expression of the form $(AX + B)/(CX + D)$ gets initialized. Otherwise the recursion continues and the node gets incorporated into the expression of the child (cf. Figure 1). Finally the root node (i.e. the operator tree) gets replaced by the new expression dag and the subexpressions get restructured.

```

1 Function restructure(root):
2   if root can be restructured then
3     exp = raise(root, root, 0)
4     set root to exp
5     restructure(exp.A); restructure(exp.B); restructure(exp.C);
6     restructure(exp.D); restructure(exp.X)
7   end
8 Function raise(node, root, counter):
9   Create new Expression exp
10  if  $\phi(\textit{node}) \leq \phi(\textit{root})/2$  or node is division and counter > THRESHOLD then
11    | exp.init(node)
12  else
13    | if node is addition or subtraction then
14      | counter++
15    | end
16    | if node is division then
17      | counter = 0
18    | end
19    | if node has no right child or  $\phi(\textit{node.left}) \geq \phi(\textit{node.right})$  then
20      | exp =raise(node.left,root,counter)
21    | else
22      | exp =raise(node.right,root,counter)
23    | end
24    | exp.incorporate(node)
25  end
26  return exp

```

parallelization is small, it might be worthwhile to allow for an increased depth to save multiplications. Our modified algorithm works the same as the algorithm described in Section 2.2.2, except it counts the number of additions and subtractions along a critical path and splits at a division node if their number surpasses a certain threshold, even if the division node still contains more than half of the operands of the operator tree. If a division node is passed while the counter is still smaller than the threshold, the counter is reset to zero, since then the additions and subtractions above the division node cause additional multiplications anyway. We refer to this strategy as *Parameterized MTR Restructuring*. A sketch of the main method is shown in Algorithm 1.

to be almost independent of the size of the bigfloats.

In the worst case, we may split at a linear number of division nodes if between two division nodes are just enough additions or subtractions to pass the threshold. Then the height of the operator tree, and therefore the running time with arbitrarily many processors, grows by a linear amount. However, each split increases the height of the tree only by a constant. We expect a similar strategy to be applicable to Brent's original algorithm, where divisions are risen to the top as well. Since a more complex counting procedure would be necessary to adapt the parameterization, this hypothesis is not evaluated in more detail in this paper.

3 Experiments

We perform several experiments on a dual core machine (named `dual_core`) with an Intel Core i5 660, 8GB RAM and a quad core machine (named `quad_core`) with an Intel Core i7-4700MQ, 16GB RAM. For `Real_algebraic` we use Boost interval arithmetic as floating-point-filter and MPFR bigfloats for the bigfloat arithmetic. The code is compiled on Ubuntu 17.10 using g++ 7.2.0 with C++11 on optimization level `O3` and linked against Boost 1.62.0 and MPFR 3.1.0. Test results are averaged over 25 runs each. The variance for each data point is small (the total range is usually in $\pm 5\%$ of its value). In each experiment the real value of the respective expression is computed to $|q| = 50000$ binary places.

We analyze four different restructuring strategies: No restructuring (`def`), AM-Balancing (`amb`), MTR Restructuring (`mtr`) and Parameterized MTR Restructuring with a threshold of five (`mtr[5]`). In all of the four strategies the evaluation is done in topological order to avoid distortion of the results. For each strategy we compare the results with and without multithreading (`m`).

3.1 Binomial coefficient

The AM-Balancing method is particularly effective if the expression contains large sums or large products. It was conjectured that applying this restructuring method makes an expression dag more suitable for a parallel evaluation. We calculate the generalized binomial coefficient

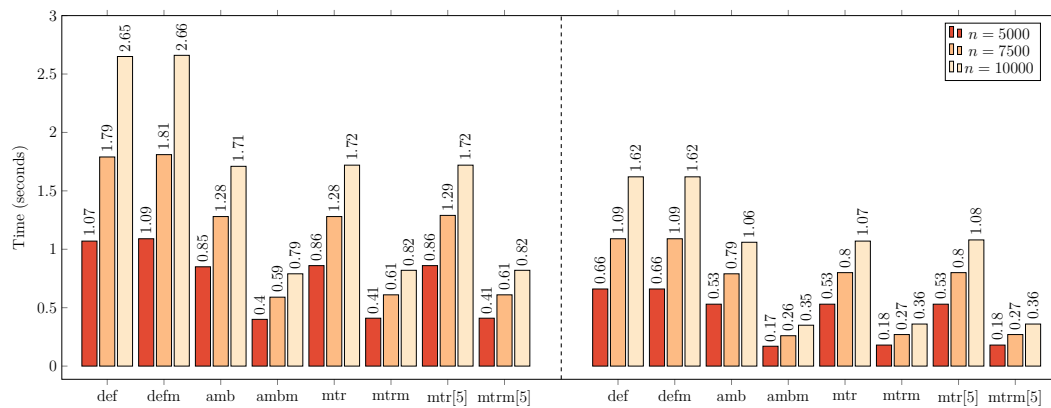
$$\binom{\sqrt{13}}{n} = \frac{\sqrt{13}(\sqrt{13}-1)\cdots(\sqrt{13}-n+1)}{n(n-1)\cdots 1}$$

iteratively as in the AM-Balancing paper (cf. [11]).

```
template <class NT> void bin_coeff(const int n, const long q){
    NT b = sqrt(NT(13)); NT num = NT(1); NT denom = NT(1);
    for (int i = 0; i < n; ++i) { num *= b - NT(i); denom *= NT(i+1); }
    NT bc = num/denom;
    bc.guarantee_absolute_error_two_to(q);
}
```

In this method, both the numerator and the denominator of `bc` are large, sequentially computed products. Both of them can be balanced without adding additional operations because of the associativity of the multiplication. We run `bin_coeff` in our test environment.

The results are shown in Figure 2. Switching to multithreading while retaining the structure of the expression dag does not have a positive effect on the performance, since the operator nodes are highly dependent on each other. Applying AM-Balancing does not only directly increase the performance, but also makes the structure much more favorable for parallel evaluation. On the dual core machine the maximal possible performance gain is achieved. With a quad core we still get an improvement, but only by a factor of about 2.8.



■ **Figure 2** The results of the binomial coefficient test for `dual_core` (left) and `quad_core` (right). The original structure of the expression dag is not suited for a parallel evaluation. AM-Balancing leads to a beneficial structure. MTR Restructuring and Parameterized MTR Restructuring have a similar effect as AM-Balancing.

For large additions and large multiplications, MTR Restructuring behaves similar to AM-Balancing in the sense that it builds an (almost) balanced tree. So, unsurprisingly, the running times for MTR Restructuring and Parameterized MTR Restructuring closely resemble the results for AM-Balancing.

3.2 Random operations

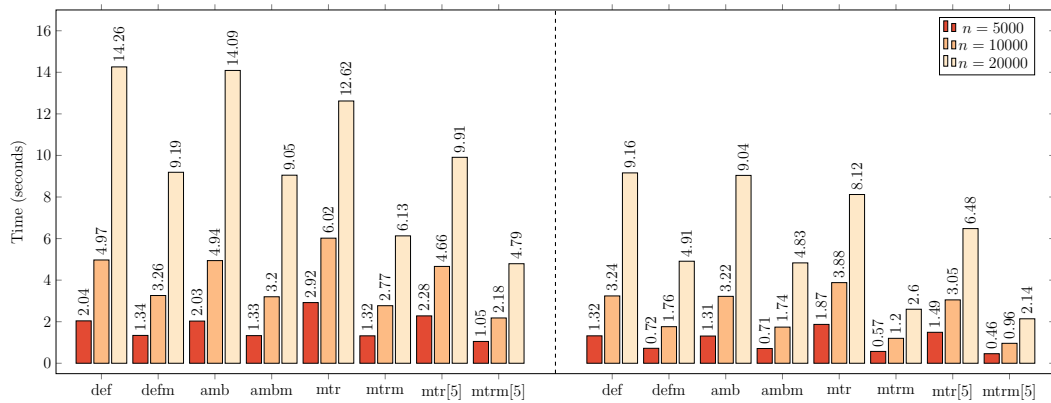
A different behavior of the restructuring methods is to be expected if algorithms use many different operators in varying order. We simulate this behavior by performing random operations on an expression.

```
template <class NT> void random_operations(const int n, const long q, ←
const int FADD = 1, const int FSUB = 1, const int FMUL = 1, const int ←
FDIV = 1){
    std::random_device rd; std::mt19937 gen(rd());
    std::uniform_int_distribution<> idis(0, FADD+FSUB+FMUL+FDIV-1);
    std::exponential_distribution<> rdis(1);

    double r; NT result = NT(1);
    for (int i = 0; i < n; ++i) {
        const int nbr = idis(gen);
        do { r = rdis(gen); } while (r == 0);

        if (nbr < FADD) result += sqrt(NT(r));
        else if (nbr < FADD+FSUB) result -= sqrt(NT(r));
        else if (nbr < FADD+FSUB+FMUL) result *= sqrt(NT(r));
        else result /= sqrt(NT(r));
    }
    result.guarantee_absolute_error_two_to(q);
}
```

We exploit two kinds of randomness in this test. First, we randomly choose one of the operators $\{+, -, *, /\}$. The parameters `FADD`, `FSUB`, `FMUL` and `FDIV` determine their respective fractions of the total number of operators. If all of them are set to one, the operations are equally distributed. Second, we randomly choose a real positive number on which to apply the operator. We use an exponential distribution with a mean of one instead of a uniform



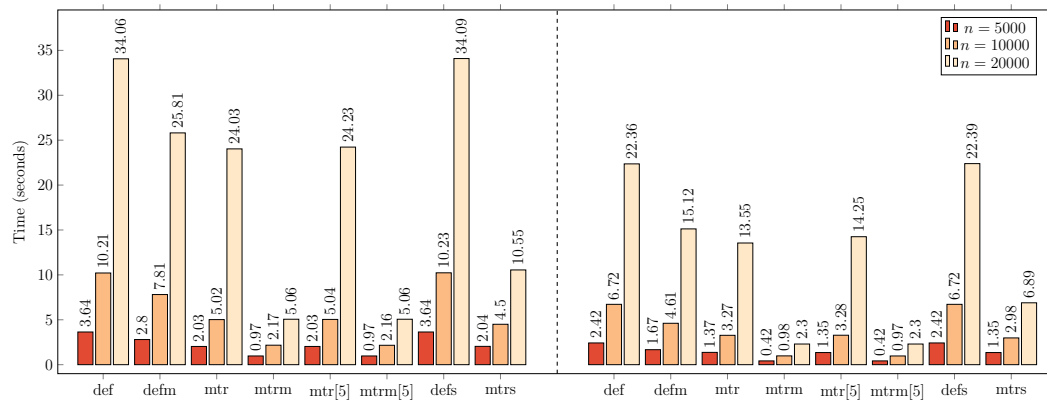
■ **Figure 3** Results for performing random operations uniformly distributed on `dual_core` (left) and `quad_core` (right). AM-Balancing has no significant effect. MTR Restructuring and Parameterized MTR Restructuring worsen the single-threaded performance for small numbers of operands, but improve the multithreaded performance. Parameterized MTR Restructuring performs better than MTR Restructuring in all cases.

distribution. `Real_algebraic` behaves differently for very large or very small numbers. In a uniform distribution most of the random numbers are larger than one. Therefore through repeated multiplication (division) numbers get very large (very small). Since we want to be able to compare the actual costs of multiplications and divisions, we have to avoid distorting the results through side effects of the experiment.

In our test we use the square roots of the random floating point numbers we get to generate numbers with an infinite floating point representation. When performing a division, `Real_algebraic` must check whether the denominator is zero. So it must decide at which point the established accuracy is sufficient to guarantee that the value of an expression is zero. This decision is made by comparing the current error bound to a separation bound, which is a number $\text{sep}(X)$ for an expression X , for which $|\text{value}(X)| > 0 \Rightarrow |\text{value}(X)| > \text{sep}(X)$. The separation bound we use is a variation of the separation bound by Burnikel et al. [3, 10]. It relies on having a meaningful bound for the algebraic degree of an expression, which we cannot provide in an expression with a lot of square roots. However, since we know that none of the denominators we get during our experiments can become zero, we can safely set the separation bound to zero and stop computing as soon as we can separate our denominator from zero with an error bound.

The results for a uniform distribution of operators are shown in Figure 3. Although the structure of the dag is largely unsuited for parallelization, we can see a significant difference between the single-threaded and the multi-threaded variant even without restructuring due to the parallel evaluation of the square root operations. AM-Balancing shows no effect at all on the overall performance.

For smaller inputs, MTR Restructuring has a negative effect on the single-threaded performance. The predominant cause of this is the propagation of divisions as described in Section 2.2, which results on average in about nine multiplications per processed division. Parameterized MTR Restructuring reduces this ratio to about 5.5 multiplications per processed division by leaving about one tenth of them unprocessed. For large inputs, MTR Restructuring has a positive effect even without multithreading due to the decrease in depth of the main operator tree and, consequently, a decrease in the maximum accuracy needed for the square root operations in its leaves (cf. [11]).



■ **Figure 4** Test results for random operations with mostly divisions on `dual_core` (left) and `quad_core` (right). MTR Restructuring causes an enormous increase in performance. The large jump between single- and multithreading can be partially ascribed to a slight change in implementation that shows a positive effect on the restructured expression dag. Applying this change without multithreading leads to the results shown in `defs` and `mtrs`. Parameterized MTR Restructuring behaves similar to MTR Restructuring.

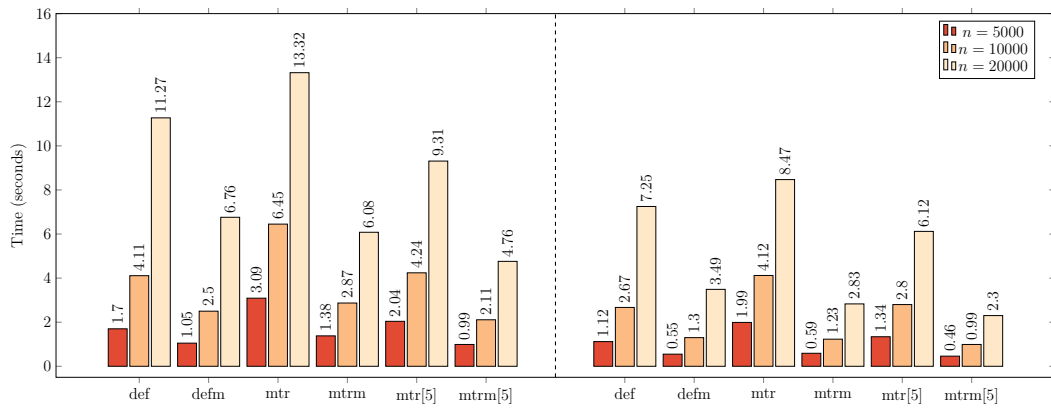
In the case of multithreading, both approaches based on Brent’s algorithm show the desired effect, increasing the speedup for the dual core from around 1.7 to an optimal 2 and for the quad core from around 1.8 to about 2.8. As a consequence, they are able to beat direct parallelization in every test case. Parameterized MTR Restructuring performs slightly better than MTR Restructuring due to the increase in single-threaded performance while maintaining the same speedup.

3.3 Random operations with mostly divisions

Raising divisions to the top is bad only if many additions and subtractions are passed during the procedure. If instead divisions can be combined and therefore replaced by multiplications, the overall effect is positive. We test this by shifting the operator distribution such that nine out of ten operators are divisions, i.e., by setting $FADD = 1$, $FSUB = 1$, $FMUL = 1$, $FDIV = 27$. Since AM-Balancing shows no differences compared to the default number type, we exclude it from further experiments.

Considering the results shown in Figure 4 it becomes evident that restructuring alone reduces the running time significantly in this situation. When switching to multithreading we get an even bigger improvement, which, however, can not be explained by parallelizing alone. Instead about half of the improvement stems from an implementation detail when computing the separation bound for separating denominators from zero in the multithreaded version. The change usually leads to a slight overhead, but also happens to prevent a slow-down if many checks have to be made. It is explained in detail in the associated technical report [12]. After restructuring we have many big denominators, for which a separation bound must be computed and therefore we get an improvement. The default number type, on the other hand, does not benefit from the new separation bound computation strategy. The data points for `defs` and `mtrs` represent the test results for single-threaded evaluation with the change in separation bound computation applied.

In contrast to our previous test, Parameterized MTR Restructuring does not perform better than MTR Restructuring, since there are few to none situations in which the condition for the improvement gets triggered. However, more importantly, the modified algorithm also does not perform worse than the original one.



■ **Figure 5** Test results for random operations with few divisions on `dual_core` (left) and `quad_core` (right). MTR Restructuring worsens the single-threaded performance and is not able to outperform direct multithreading for smaller inputs. Parameterized MTR Restructuring performs better in all tests, although having a worse speedup factor on the quad core.

3.4 Random operations with few divisions

With few divisions, compared to the number of additions and subtractions, we should expect the opposite effect from the previous experiment. MTR Restructuring should lead to an decrease in singlethreaded performance and Parameterized MTR Restructuring should perform better than MTR Restructuring. We set our input parameters to `FADD = 3`, `FSUB = 3`, `FMUL = 3`, `FDIV = 1`, such that only one out of ten operations is a division.

Our test results confirm these expectations (cf. Figure 5). For small inputs, MTR Restructuring performs worse than no restructuring even in a parallel environment. Parameterized MTR Restructuring on the other hand performs better than the default in all parallel tests except for the test with the smallest number of operands on a dual core. This effect strengthens when the number of divisions further decreases.

However, it should be noted that while on a dual core the speedup through parallelization is optimal for both variants, on a quad core MTR Restructuring allows for a speedup of about 3, whereas Parameterized MTR Restructuring only reaches a speedup of around 2.7. The modified algorithm leaves about forty percent of the divisions untouched for this operator distribution, which manifests in a significant effect on the expression dag’s degree of independence.

3.5 Parameter-dependence of Parameterized MTR Restructuring

In the experiments in the previous sections we set the threshold k for Parameterized MTR Restructuring to five without an explanation. In this section we make evident that, while there is always an optimal choice for this parameter, most choices are not actually bad compared to standard MTR Restructuring.

The threshold indicates the number of additions and subtractions that are affected by incorporating a division node into the new structure and therefore sets the benefit from raising a division to the top in relation to the number of additional multiplication nodes it causes (cf. Section 2.2.3). If $k = 0$, restructuring never incorporates divisions in its expressions, therefore all expressions are of the form $AX + B$. For $k = 1$ divisions are only incorporated if they are followed exclusively by divisions, multiplications or negations. With increasing k it becomes less frequent that a division cannot be passed during restructuring, leading to Parameterized MTR Restructuring behaving more and more similar to MTR Restructuring.

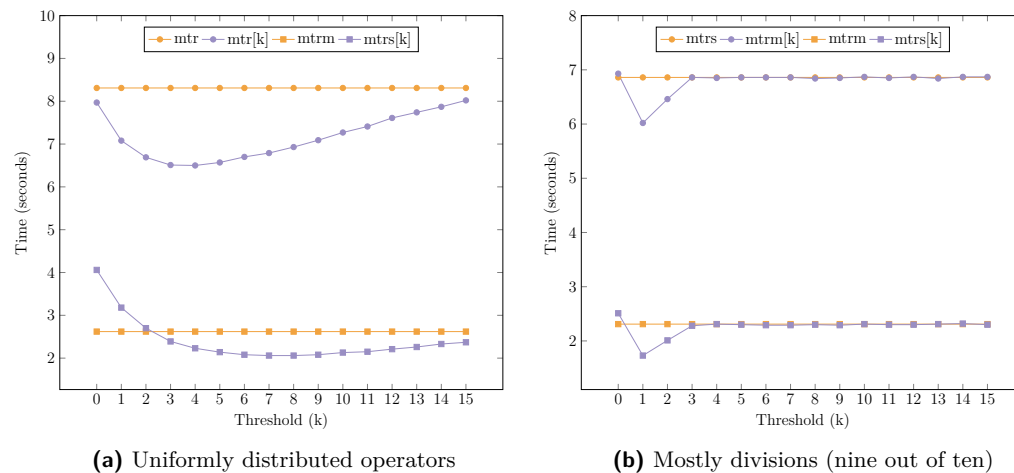


Figure 6 Comparison of different parameter choices for Parameterized MTR Restructuring with $n = 20000$ random operations on `quad_core`. For large values of k , the parameterized approach becomes increasingly similar to MTR Restructuring. For small values of k the original approach is slightly better in the multithreaded version. The optimal choice for k depends on the division ratio.

In Figure 6 the running time for the experiments from Section 3.2 and Section 3.3 for different values of k are shown. For the second experiment, we use the modified separation bound computation strategy to ensure comparability (cf. Section 3.3). The results demonstrate that most of the choices for k improve the performance of the algorithm. Also, they confirm that for high k the parameterized approach is almost identical to the original algorithm.

For $k = 0$ the parameterized approach performs worse in both experiments in the parallel version, although slightly increasing the single-threaded performance for uniformly distributed operators. Surprisingly, Parameterized MTR Restructuring is faster than MTR Restructuring for $k = 1$ when nine out of ten operators are divisions, despite in this case restructuring tends to replace divisions by multiplications, which then can be balanced due to their associativity. However, since only one out of fifteen operations is an addition or subtraction, the loss of independence is small compared to the gain from avoiding additional multiplications.

The optimal choice for k depends on the ratio between divisions and additions/subtractions. If this ratio gets smaller, the optimal k increases. At the same time, the difference between different choices for k decreases. Therefore for a small ratio and smaller values of k the parameterized approach still performs better than the original version.

4 Summary

Multithreading can be an effective tool to speed up the performance of expression-dag-based number types. Applying MTR Restructuring to expression dags allows us to benefit from multithreading even when faced with a structure with a high number of dependencies, although bearing the risk of lowering the performance. AM-Balancing can create favorable structures with low risk of worsening the end result, but is not widely applicable. The parameterized version of MTR Restructuring lowers the risk of significant performance loss, while at the same time maintaining most of the benefits of the original algorithm. In a general purpose number type, we therefore suggest using Parameterized MTR Restructuring, with a sensible choice of k , if the evaluation should be done in parallel. For k , a small number greater than zero is advisable.

5 Future Work

This work addresses parallelization on multiple CPUs. While it seems unlikely that complex expression dags can be efficiently parallelized on a GPU, it may be possible to do so for the underlying bigfloats. Since bigfloat operations still constitute the most expensive part of exact-decisions number types, this may lead to a significant speedup. Furthermore in this work we only restructure tree-like subgraphs to avoid (possibly exponential) blow-up of our structure. However, with a larger number of cores it might be worthwhile to split up some nodes with multiple parents to eliminate or shorten critical paths or at least weigh such nodes accordingly in the higher-level operator trees. Finally, the performance gain due to the parameterization cannot be fully attributed to interactions between divisions and additions. It may prove useful to extend the new strategy to consider multiplications over large sums.

References

- 1 Mohand Ourabah Benouamer, P. Jaillon, Dominique Michelucci, and Jean-Michel Moreau. A lazy exact arithmetic. In *11th Symposium on Computer Arithmetic, 29 June - 2 July 1993, Windsor, Canada, Proceedings.*, pages 242–249, 1993. doi:10.1109/ARITH.1993.378086.
- 2 Richard P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, 1974.
- 3 Christoph Burnikel, Stefan Funke, Kurt Mehlhorn, Stefan Schirra, and Susanne Schmitt. A separation bound for real algebraic expressions. *Algorithmica*, 55(1):14–28, 2009. doi:10.1007/s00453-007-9132-4.
- 4 Christoph Burnikel, Kurt Mehlhorn, and Stefan Schirra. The leda class real number. Report MPI-I-1996-1-001, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 1996.
- 5 Vijay Karamcheti, C. Li, Igor Pechtchanski, and Chee-Keng Yap. A core library for robust numeric and geometric computation. In *Proceedings of the Fifteenth Annual Symposium on Computational Geometry, Miami Beach, Florida, USA, June 13-16, 1999*, pages 351–359, 1999. doi:10.1145/304893.304989.
- 6 Gary L. Miller and John H. Reif. Parallel tree contraction and its application. In *26th Annual Symposium on Foundations of Computer Science, Portland, Oregon, USA, 21-23 October 1985*, pages 478–489, 1985.
- 7 Marc Mörig, Ivo Rössling, and Stefan Schirra. On design and implementation of a generic number type for real algebraic number computations based on expression dags. *Mathematics in Computer Science*, 4(4):539–556, 2010. doi:10.1007/s11786-011-0086-1.
- 8 Marc Mörig and Stefan Schirra. Precision-driven computation in the evaluation of expression-dags with common subexpressions: Problems and solutions. In *6th International Conference on Mathematical Aspects of Computer and Information Sciences, MACIS*, pages 451–465, 2015. doi:10.1007/978-3-319-32859-1_39.
- 9 Stefan Schirra. Robustness and precision issues in geometric computation. In *Handbook of Computational Geometry*, pages 597–632. Elsevier, 2000.
- 10 Susanne Schmitt. Improved separation bounds for the diamond operator. Report ECG-TR-363108-01, Effective Computational Geometry for Curves and Surfaces, Sophia Antipolis, France, 2004.
- 11 Martin Wilhelm. Balancing expression dags for more efficient lazy adaptive evaluation. In *Mathematical Aspects of Computer and Information Sciences - 7th International Conference, MACIS 2017, Vienna, Austria, November 15-17, 2017, Proceedings*, pages 19–33, 2017. doi:10.1007/978-3-319-72453-9_2.
- 12 Martin Wilhelm. Multithreading for the expression-dag-based number type Real_algebraic. Technical Report FIN-001-2018, Otto-von-Guericke-Universität Magdeburg, 2018.

- 13 Chee-Keng Yap. Towards exact geometric computation. *Comput. Geom.*, 7:3–23, 1997. doi:10.1016/0925-7721(95)00040-2.
- 14 Jihun Yu, Chee-Keng Yap, Zilin Du, Sylvain Pion, and Hervé Brönnimann. The design of core 2: A library for exact numeric computation in geometry and algebra. In *Proceedings of the Third International Congress on Mathematical Software, ICMS*, pages 121–141, 2010. doi:10.1007/978-3-642-15582-6_24.

Enumerating Graph Partitions Without Too Small Connected Components Using Zero-suppressed Binary and Ternary Decision Diagrams

Yu Nakahata

Nara Institute of Science and Technology, Ikoma, Japan
nakahata.yu.nm2@is.naist.jp

Jun Kawahara

Nara Institute of Science and Technology, Ikoma, Japan
jkawahara@is.naist.jp

Shoji Kasahara

Nara Institute of Science and Technology, Ikoma, Japan
kasahara@is.naist.jp

Abstract

Partitioning a graph into balanced components is important for several applications. For multi-objective problems, it is useful not only to find one solution but also to enumerate all the solutions with good values of objectives. However, there are a vast number of graph partitions in a graph, and thus it is difficult to enumerate desired graph partitions efficiently. In this paper, an algorithm to enumerate all the graph partitions such that all the weights of the connected components are at least a specified value is proposed. To deal with a large search space, we use zero-suppressed binary decision diagrams (ZDDs) to represent sets of graph partitions and we design a new algorithm based on frontier-based search, which is a framework to directly construct a ZDD. Our algorithm utilizes not only ZDDs but also ternary decision diagrams (TDDs) and realizes an operation which seems difficult to be designed only by ZDDs. Experimental results show that the proposed algorithm runs up to tens of times faster than an existing state-of-the-art algorithm.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms, Mathematics of computing → Graph enumeration, Mathematics of computing → Decision diagrams

Keywords and phrases Graph algorithm, Graph partitioning, Decision diagram, Frontier-based search, Enumeration problem

Digital Object Identifier 10.4230/LIPIcs.SEA.2018.21

Related Version A full version of the paper is available at <https://arxiv.org/pdf/1804.02160.pdf>.

1 Introduction

Partitioning a graph is a fundamental problem in computer science and has several important applications such as evacuation planning, political redistricting, VLSI design, and so on. In some applications among them, it is often required to balance the weights of connected components in a partition. For example, the task of the evacuation planning is to design which evacuation shelter inhabitants escape to. This problem is formulated as a graph partitioning problem, and it is important to obtain a graph partition consisting of balanced connected components (each of which contains a shelter and satisfies some conditions).



© Yu Nakahata, Jun Kawahara, and Shoji Kasahara;
licensed under Creative Commons License CC-BY

17th International Symposium on Experimental Algorithms (SEA 2018).

Editor: Gianlorenzo D'Angelo; Article No. 21; pp. 21:1–21:13

Leibniz International Proceedings in Informatics



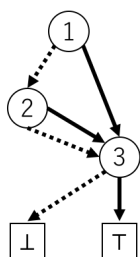
LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Another example is political redistricting, the purpose of which is to divide a region (such as a prefecture) into several balanced political districts for fairness.

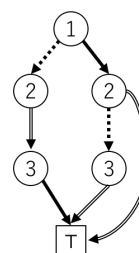
There are a vast number of studies for graph optimization problems. An approach is to use a *zero-suppressed binary decision diagram* (ZDD) [12], which has originally been proposed as a compressed representation of a family of sets. A distinguished characteristic of the approach is not only to compute the single optimal solution but also to *enumerate* all the feasible solutions in the form of a ZDD. In addition, using several queries for a family of sets provided by ZDDs, we can impose various constraint conditions on solutions represented by a ZDD. Using this approach, Inoue et al. [4] designed an algorithm that constructs the ZDD representing the set of rooted spanning forests and utilized it to minimize the loss of electricity in an electrical distribution network under complex conditions, e.g., voltage, electric current and phase. There are other applications such as solving a variant of the longest path problem [8], reliability evaluation [2, 3], some puzzle problems [16], and exact calculation of impact diffusion in Web [11].

For balanced graph partitioning, Kawahara et al. [6] proposed an algorithm to construct a ZDD representing the set of balanced graph partitions by frontier-based search [7, 10, 14], which is a framework to directly construct a ZDD, and applied it to political redistricting. However, their method stores the weights of connected components, represented as integers, into the ZDD, which generates a not compressed ZDD. As a result, the computation is tractable only for graphs only with less than 100 vertices. Nakahata et al. [13] proposed an algorithm to construct the ZDD representing the set of partitions such that all the weights of connected components are bounded by a given upper threshold (and applied it to evacuation planning). Their approach enumerates connected components with weight more than the upper threshold as a ZDD, say *forbidden components*, and constructs a ZDD representing partitions not containing any forbidden component *as a subgraph* by set operations, which are performed by so-called apply-like methods [1]. However, it seems difficult to directly use their method to obtain balanced partitions by letting connected components with weight less than a lower threshold be forbidden components because partitions not containing any forbidden component *as a connected component* (i.e., one of parts in a partition coincides a forbidden component) cannot be obtained by apply-like methods.

In this paper, for a ZDD Z_A and an integer L , we propose a novel algorithm to construct the ZDD representing the set of graph partitions such that the partitions are represented by Z_A and all the weights of the connected components in the partitions are at least L . The input ZDD Z_A can be the sets of spanning forests used for evacuation planning (e.g., [13]), rooted spanning forests used for power distribution networks (e.g., [4]), and simply connected components representing regions (e.g., [6]), all of which satisfy complex conditions according to problems. We generically call these structures “partitions.” Roughly speaking, our algorithm excludes partitions containing any forbidden component as a connected component from Z_A . We first construct the ZDD, say Z_S , representing the set of forbidden components, each of which has weight less than L . Then, for a component in Z_S , we consider the cutset that separates the input graph into the component and the rest. We represent the set of pairs of every component in Z_S and its cutset as a *ternary decision diagram* (TDD) [15], say T_{S^\pm} . We propose a method to construct the TDD T_{S^\pm} from Z_S by frontier-based search. By using the TDD T_{S^\pm} , we show how to obtain partitions each of which belongs to Z_A , contains all the edges in a component of a pair in T_{S^\pm} and contains no edge in the cutset of the pair. Finally, we exclude such partitions from Z_A and obtain the desired partitions. By numerical experiments, we show that the proposed algorithm runs up to tens of times faster than an existing state-of-the-art algorithm.



■ **Figure 1** The ZDD representing the family $\{\{1, 3\}, \{2, 3\}, \{3\}\}$. A square represents a terminal node. A circle is a non-terminal node and the number in it is a label. A solid arc is a 1-arc and a dashed arc is a 0-arc.



■ **Figure 2** The TDD representing the signed family $\{\{+1, -2\}, \{+1, -3\}, \{-2, +3\}\}$. A dashed arc is a ZERO-arc, a solid single arc is a POS-arc and a solid double arc is NEG-arc. For simplicity, \perp and the arcs pointing at it are omitted.

This paper is organized as follows. In Sec. 2, we give some preliminaries and explain ZDDs, TDDs, and frontier-based search. We describe an overview of our algorithm in Sec. 3.1, and the detail in the rest of Sec. 3. Section 4 gives experimental results. The conclusion is described in Sec. 5.

2 Preliminaries

2.1 Notation

Let \mathbb{Z}^+ be the set of positive integers. For $k \in \mathbb{Z}^+$, we define $[k] = \{1, 2, \dots, k\}$. In this paper, we deal with a vertex-weighted undirected graph $G = (V, E, p)$, where $V = [n]$ is the vertex set and $E = \{e_1, e_2, \dots, e_m\} \subseteq \{\{u, v\} \mid u, v \in V\}$ is the edge set. The function $p: V \rightarrow \mathbb{Z}^+$ gives the weights of the vertices. We often drop p from (V, E, p) when there is no ambiguity. For an edge set $E' \subseteq E$, we call the subgraph (V, E') a *graph partition*. We often identify the edge set E' with the partition (V, E') by fixing the graph G . For edge sets E', E'' with $E'' \subseteq E' \subseteq E$ and a vertex set $V'' \subseteq V$, we say that (V'', E'') is included in the partition (V, E') as a *subgraph*. The subgraph (V'', E'') is called a *connected component* in the partition (V, E') if $V'' = \text{dom}(E'')$ holds, there is no edge in $E' \setminus E''$ incident with a vertex in V'' , and for any two distinct vertices $u, v \in V''$, there is a u - v path on (V'', E'') , where $\text{dom}(E'')$ is the set of vertices which are endpoints of at least one edge in E'' . In this case, we say that (V'', E'') is included in the partition (V, E') as a *connected component*. We denote the neighborhood of a vertex v in a partition $E' \subseteq E$ by $N(E', v) = \{u \mid \{u, v\} \in E'\}$. For $i \in [m]$, $E^{\leq i}$ denotes the set of edges whose indices are at most i . We define $E^{< i}$, $E^{\geq i}$ and $E^{> i}$ in the same way.

For a set U , let $U^+ = \{+e \mid e \in U\}$, $U^- = \{-e \mid e \in U\}$ and $U^\pm = U^+ \cup U^-$. A *signed set* is a subset of U^\pm such that, for all $e \in U$, the set contains at most one of $+e$ and $-e$. For example, when $U = [3]$, both $\{+1, -2\}$ and $\{-3\}$ are signed sets but $\{+1, -1, +3\}$ is not. A *signed family* is a family of signed sets. In particular, when $U = E$, we sometimes call a signed set a *signed subgraph* and call a signed family a *set of signed subgraphs*. For a signed set S^\pm , we define $\text{abs}(S^\pm) = \{e \mid (+e \in S^\pm) \vee (-e \in S^\pm)\}$.

2.2 Zero-suppressed binary decision diagram

A *zero-suppressed binary decision diagram* (ZDD) [12] is a directed acyclic graph $Z = (N_Z, A_Z)$ representing a family of sets. Here N_Z is the set of *nodes* and A_Z is the set of *arcs*.¹ N_Z contains two *terminal nodes* \top and \perp . The other nodes than the terminal nodes are called *non-terminal nodes*. Each non-terminal node α has the *0-arc*, the *1-arc*, and the *label* corresponding to an item in the universe set. For $x \in \{0, 1\}$, we call the destination of the x -arc of a non-terminal node α the x -*child* of α . We denote the label of α by $l(\alpha)$ and in this paper, assume that $l(\alpha) \in \mathbb{Z}^+ \cup \{\infty\}$ for any $\alpha \in N_Z$. For convenience, we let $l(\top) = l(\perp) = \infty$. For each directed arc $(\alpha, \beta) \in A_Z$, the inequality $l(\alpha) < l(\beta)$ holds, which ensures that Z is acyclic. There is exactly one node whose in-degree is zero, called the *root node* and denoted by r_Z . The number of the non-terminal nodes of Z is called the *size* of Z and denoted by $|Z|$.

Z represents the family of sets in the following way. Let \mathcal{P}_Z be the set of all the directed paths from r_Z to \top . For a directed path $p = (n_1, a_1, n_2, a_2, \dots, n_k, a_k, \top) \in \mathcal{P}_Z$ with $n_i \in N_Z$, $a_i \in A_Z$ and $n_1 = r_Z$, we define $S_p = \{l(n_i) \mid a_i \in A_{Z,1}, i \in [k]\}$, where $A_{Z,1}$ is the set of the 1-arcs of Z . We interpret that Z represents the family $\{S_p \mid p \in \mathcal{P}_Z\}$. In other words, a directed path from r_Z to \top corresponds to a set in the family represented by Z . As an example, we illustrate the ZDD representing the family $\{\{1, 3\}, \{2, 3\}, \{3\}\}$ in Fig. 1. In the figure, a dashed arc ($--\rightarrow$) and a solid arc (\rightarrow) are a 0-arc and a 1-arc, respectively. On the ZDD in Fig. 1, there are three directed paths from the root node to \top : $1 \rightarrow 3 \rightarrow \top$, $1 --\rightarrow 2 \rightarrow 3 \rightarrow \top$, and $1 --\rightarrow 2 --\rightarrow 3 \rightarrow \top$, which correspond to $\{1, 3\}$, $\{2, 3\}$, and $\{3\}$, respectively. We denote a ZDD representing a family \mathcal{F} by $Z_{\mathcal{F}}$.

2.3 Ternary decision diagram

A *ternary decision diagram* (TDD) [15] is a directed acyclic graph $T = (N_T, A_T)$ representing a signed family. A TDD shares many concepts with a ZDD, and thus we use the same notation as a ZDD for a TDD. The difference between a ZDD and a TDD is that, while a node of the former has two arcs, that of the latter has three, which are called the *ZERO-arc*, the *POS-arc*, and the *NEG-arc*.

T represents the signed family in the following way. For a directed path $p = (n_1, a_1, n_2, a_2, \dots, n_k, a_k, \top) \in \mathcal{P}_T$ with $n_i \in N_T$, $a_i \in A_T$ and $n_1 = r_T$, we define $S_p^{\pm} = \{+l(n_i) \mid a_i \in A_{T,+}, i \in [k]\} \cup \{-l(n_i) \mid a_i \in A_{T,-}, i \in [k]\}$, where $A_{T,+}$ and $A_{T,-}$ are the set of the POS-arcs of T and the set of the NEG-arcs of T , respectively. We interpret that T represents the signed family $\{S_p^{\pm} \mid p \in \mathcal{P}_T\}$. We illustrate the TDD representing the signed family $\{\{+1, -2\}, \{+1, -3\}, \{-2, +3\}\}$ in Fig. 2 for example. In the figure, a dashed arc ($--\rightarrow$), a solid single arc (\rightarrow), and a solid double arc (\Rightarrow) are a ZERO-arc, a POS-arc, and a NEG-arc, respectively. The TDD in the figure has three directed paths from the root node to \top : $1 \rightarrow 2 \Rightarrow \top$, $1 \rightarrow 2 --\rightarrow 3 \Rightarrow \top$, and $1 --\rightarrow 2 \Rightarrow 3 \rightarrow \top$, which correspond to $\{+1, -2\}$, $\{+1, -3\}$, and $\{-2, +3\}$, respectively.

2.4 Frontier-based search

Frontier-based search [7, 10, 14] is a framework of algorithms that efficiently construct a decision diagram representing the set of subgraphs satisfying given constraints of an input graph. We explain the general framework of frontier-based search. Given a graph G , let

¹ To avoid confusion, we use the words “vertex” and “edge” for input graphs and “nodes” and “arcs” for decision diagrams.

\mathcal{M} be a class of subgraphs we would like to enumerate (for example, \mathcal{M} is the set of all the s - t paths on G). Frontier-based search constructs the ZDD representing the family \mathcal{M} of subgraphs. By fixing G , a subgraph is identified with the edge set the subgraph has, and thus the ZDD represents the family of edge sets actually. Non-terminal nodes of ZDDs constructed by frontier-based search have labels e_1, \dots, e_m . We identify e_i with the integer i . We assume that it is determined in advance which edge in G has which index i of e_i .

We directly construct the ZDD in a breadth-first manner. We first create the root node of the ZDD, make it have label e_1 , and then we carry out the following procedure for $i = 1, \dots, m$. For each node n_i with label e_i , we create two nodes, each of which is either a terminal node or a non-terminal node whose label is e_{i+1} (if $i = m$, the candidate is only a terminal node), as the 0-child and the 1-child of n_i .

Which node the x -arc of a node n_i with label e_i points at is determined by a function, called MAKENEWNODE, of which we design the detail according to \mathcal{M} , i.e., what subgraphs we want to enumerate. Here we describe the generalized nature that MAKENEWNODE must possess. The node n_i represents the set of the subgraphs, denoted by $\mathcal{G}(n_i)$, corresponding to the set of the directed paths from the root node to n_i . Each subgraph in $\mathcal{G}(n_i)$ contains only edges in $E^{<i}$. Note that $\mathcal{G}(\top)$ is the desired set of subgraphs represented by the ZDD after the construction finishes. To decide which node the x -arc of n_i points at without traversing the ZDD (under construction), we make each node n_i have the information $n_i.\text{conf}$, which is shared by all the subgraphs in $\mathcal{G}(n_i)$. The content of $n_i.\text{conf}$ also depends on \mathcal{M} (for example, in the case of s - t paths, we store degrees and components of the subgraphs in $\mathcal{G}(n_i)$ into $n_i.\text{conf}$). MAKENEWNODE creates a new node, say n_{new} , with label e_{i+1} and must behave in the following manner.

1. For all edge sets $S^{<i} \in \mathcal{G}(n_{\text{new}})$, if there is no edge set $S^{>i} \subseteq E^{>i}$ such that $S^{<i} \cup S^{>i} \in \mathcal{M}$, the function discards n_{new} and returns \perp to avoid redundant expansion of nodes. (*pruning*)
2. Otherwise, if $i = m$, the function returns \top .
3. Otherwise, the function calculates $n_{\text{new}}.\text{conf}$ from $n_i.\text{conf}$. If there is a node n_{i+1} such that whose label is e_{i+1} and $n_{\text{new}}.\text{conf} = n_{i+1}.\text{conf}$, the function abandons n_{new} and returns n_{i+1} . (*node merging*) If not, the function returns n_{new} .

We make the x -arc of n_i point at the node returned by MAKENEWNODE.

As for $n_i.\text{conf}$, in the case of several kinds of subgraphs such as paths and cycles, it is known that we only have to store states relating to the vertices to which both an edge in $E^{<i}$ and an edge in $E^{>i}$ are incident into each node [10] (in the case of s - t paths, we store degrees and components of such vertices into each node). The set of the vertices are called the *frontier*. More precisely, the i -th *frontier* is defined as $F_i = (\bigcup_{j=1}^{i-1} e_j) \cap (\bigcup_{k=i}^m e_k)$. For convenience, we define $F_0 = F_m = \emptyset$. States of vertices in F_{i-1} are stored into $n_i.\text{conf}$. By limiting the domain of the information to the frontier, we can reduce memory consumption and share more nodes, which leads to a more efficient algorithm.

The efficiency of an algorithm based on frontier-based search is often evaluated by the *width of a ZDD* constructed by the algorithm. The width W_Z of a ZDD Z is defined as $W_Z = \max\{|\mathcal{N}_i| \mid i \in [m]\}$, where \mathcal{N}_i denotes the set of nodes whose labels are e_i . Using W_Z , the number of nodes in Z can be written as $|Z| = \mathcal{O}(mW_Z)$ and the time complexity of the algorithm is $\mathcal{O}(\tau|Z|)$, where τ denotes the time complexity of MAKENEWNODE for one node.

3 Algorithms

3.1 Overview of the proposed algorithms

In this section, for a ZDD $Z_{\mathcal{A}}$ and $L \in \mathbb{Z}^+$, we propose a novel algorithm to construct the ZDD representing the set of graph partitions such that the partitions are represented by $Z_{\mathcal{A}}$ and each connected component in the partitions has weight at least L . In general, there are two techniques to obtain ZDDs having desired conditions. One is frontier-based search, described in the previous section. The method proposed by Kawahara et al. [6] directly stores the weight of each component into ZDD nodes (as `conf`) and prunes a node when it is determined that the weight of a component is less than L . However, for two nodes, if the weight of a single component on the one node differs from that on the other node, the two nodes cannot be merged. Consequently, node merging rarely occurs in Kawahara et al.'s method and thus the size of the resulting ZDD is too large to construct it if the input graph has more than 100 vertices.

The other technique is the usage of the recursive structure of a ZDD. Methods based on the recursive structure are called *apply-like* methods [1]. For each node α of a ZDD, the nodes and arcs reachable from α compose another ZDD, whose root is α . For a ZDD Z and $x \in \{0, 1\}$, let $c_x(Z)$ be the ZDD composed by the nodes and arcs reachable from the x -child of the root. For (one or more) ZDDs F (and G), an apply-like method constructs a target ZDD by recursively calling itself against $c_0(F)$ and $c_1(F)$ (and $c_0(G)$ and $c_1(G)$). For example, the ZDD representing $F \cap G$ can be computed from $c_0(F) \cap c_0(G)$ and $c_1(F) \cap c_1(G)$. Apply-like methods support various set operations [1, 10].

Nakahata et al. [13] developed an algorithm to upperbound the weights of connected components in each partition, i.e., to construct the ZDD representing the set \mathcal{A} of partitions included in a given ZDD and the weights of all the components in the partitions are at most $H \in \mathbb{Z}^+$. Their algorithm first constructs the ZDD $Z_{\mathcal{S}}$ representing the set of forbidden components (described in the introduction) with weight more than H by frontier-based search. Then, the algorithm constructs the ZDD representing $\{A \in \mathcal{A} \mid \exists S \in \mathcal{S}, A \supseteq S\}$, written as $Z_{\mathcal{A}}.\text{restrict}(Z_{\mathcal{S}})$, which means the set of all the partitions each of which includes a component in \mathcal{S} as a subgraph, in a way of apply-like methods. Finally, we extract subgraphs not in $Z_{\mathcal{A}}.\text{restrict}(Z_{\mathcal{S}})$ from $Z_{\mathcal{A}}$ by the set difference operation $Z_{\mathcal{A}} \setminus (Z_{\mathcal{A}}.\text{restrict}(Z_{\mathcal{S}}))$ [12], which is also an apply-like method.

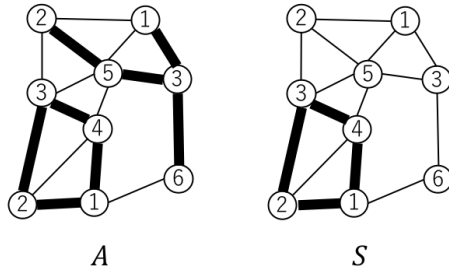
In our case, lowerbounding the weights of components, it is difficult to compute desired partitions by the above approach because a partition including a forbidden component (i.e., weight less than L) *as a subgraph* can be a feasible solution. We want to obtain a partition including a forbidden component *as a connected component*. Although we can perform various set operations by designing apply-like methods, it seems difficult to obtain such partitions by direct set operations.

Our idea in this paper is to employ the family of signed sets to represent the set of pairs of every forbidden component and its cutset. We use the following observation.

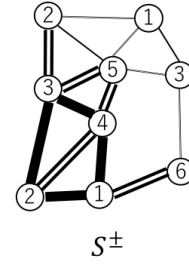
► **Observation 1.** *Let A be a graph partition of $G = (V, E)$ and $S \subseteq E$ be an edge set such that $(\text{dom}(S), S)$ is connected. The partition A contains $(\text{dom}(S), S)$ as a connected component if and only if both of the following hold.*

1. A contains all the edges in S .
2. A does not contain any edge e in $E \setminus S$ such that e has at least one vertex in $\text{dom}(S)$.

Based on Observation 1, we associate a signed subgraph S^{\pm} with a connected subgraph $(\text{dom}(S), S)$:



■ **Figure 3** A graph partition A and a connected subgraph S . Bold lines are edges contained in the partition or the subgraph. Values in vertices are its weights. A contains S as a connected component. The weight of S is $1 + 3 = 4 < 5$, and thus, when $L = 5$, A does not satisfy the lower bound constraint.



■ **Figure 4** A signed subgraph S^\pm with minimal cutset corresponding to S in Fig. 3. Thin single lines, bold single lines, and doubled lines are zero edges, positive edges, and negative edges, respectively.

$$S^\pm = S^+ \cup S^-, \quad (1)$$

$$S^+ = \{+e \mid e \in S\}, \quad (2)$$

$$S^- = \{-e \mid (e \in E \setminus S) \wedge (e \cap \text{dom}(S) \neq \emptyset)\}. \quad (3)$$

S^\pm is a signed subgraph such that $\text{abs}(S^+)$ and $\text{abs}(S^-)$ are sets of edges satisfying Conditions 1 and 2 in Observation 1, respectively. Note that $\text{abs}(S^-)$ is a cutset of G , that is, removing the edges in $\text{abs}(S^-)$ separates G into the connected component $(\text{dom}(\text{abs}(S^+)), \text{abs}(S^+))$ and the rest. In addition, $\text{abs}(S^-)$ is minimal among such cutsets. In this sense, we say that S^\pm is a *signed subgraph with minimal cutset for S* .

Hereinafter, we call edges in $\text{abs}(S^+)$ *positive edges*, $\text{abs}(S^-)$ *negative edges* and the other edges *zero edges*. Figure 4 shows S^\pm associated with S in Fig. 3. The partition A in Fig. 3 indeed contains all the edges in $\text{abs}(S^+)$ and does not contain any edges in $\text{abs}(S^-)$. For a graph partition $E' \subseteq E$, when the weights of all the connected components of E' is at least L , we say that E' *satisfies the lower bound constraint*. To extract partitions not satisfying the lower bound constraint from an input ZDD, we compute the set of partitions each of which has all the edges in $\text{abs}(S^+)$ and no edge in $\text{abs}(S^-)$ for some $S \in \mathcal{S}$.

The overview of the proposed method is as follows. In the following, let \mathcal{A} be the set of graph partitions represented by the input ZDD and \mathcal{B} be the set of graph partitions each of which belongs to \mathcal{A} and satisfies the lower bound constraint.

1. We construct the ZDD $Z_{\mathcal{S}}$ representing the set \mathcal{S} of forbidden components, where \mathcal{S} is the set of the connected components of G whose weights are less than L .
2. Using $Z_{\mathcal{S}}$, we construct the TDD $T_{\mathcal{S}^\pm}$, where \mathcal{S}^\pm is a set of signed subgraphs with minimal cutset corresponding to \mathcal{S} by a way of frontier-based search.
3. Using $T_{\mathcal{S}^\pm}$, we construct the ZDD $Z_{\mathcal{S}^\dagger}$, where \mathcal{S}^\dagger is the set of partitions each of which contains at least one forbidden component in \mathcal{S} as a connected component.
4. We obtain the ZDD $Z_{\mathcal{B}}$ by the set difference operation $Z_{\mathcal{A}} \setminus Z_{\mathcal{S}^\dagger}$ [12].

In the rest of this section, we describe each step from 1 to 3.

3.2 Constructing $Z_{\mathcal{S}}$

We describe how to construct $Z_{\mathcal{S}}$, which represents the set \mathcal{S} of forbidden subgraphs whose weights are less than L . In this subsection, we consider only forbidden components with at least one edge. Note that a component with only one vertex cannot be distinguished by sets of edges because all such subgraphs are represented by the empty edge set. We show how to deal with components having only one vertex in Sec. 3.4.

We can construct $Z_{\mathcal{S}}$ using frontier-based search. Due to the page restriction, we describe a brief overview. To construct $Z_{\mathcal{S}}$, in the frontier-based search, it suffices to ensure that every enumerated subgraph has only one connected component and its weight is less than L . The former can be dealt by storing the connectivity of the vertices in the frontier as `comp` [7]. The latter can be checked by managing the total weight of vertices such that at least one edge is incident to as `weight`.

Let us analyze the width of $Z_{\mathcal{S}}$. For nodes with the same label, there are $\mathcal{O}(B_f)$ different states for `comp` [6], where, for $k \in \mathbb{Z}^+$, B_k is the k -th Bell number and $f = \max\{|F_i| \mid i \in [m]\}$. As for `weight`, when `weight` exceeds L , we can immediately conclude that the subgraphs whose weights are less than L are generated no more. If we prune such cases, there are $\mathcal{O}(L)$ different states for `weight`. As a result, we can obtain the following lemma on the width of $Z_{\mathcal{S}}$.

► **Lemma 2.** *The width of $Z_{\mathcal{S}}$ is $\mathcal{O}(B_f L)$, where $f = \max\{|F_i| \mid i \in [m]\}$.*

3.3 Constructing $T_{\mathcal{S}^{\pm}}$

In this subsection, we propose an algorithm to construct $T_{\mathcal{S}^{\pm}}$. First, we show how to construct the TDD representing the set of all the signed subgraphs with minimal cutset, including a disconnected one. Next, we describe the method to construct $T_{\mathcal{S}^{\pm}}$ using $Z_{\mathcal{S}}$.

Let $S^{\pm} = S^+ \cup S^-$ be a signed subgraph. Our algorithm uses the following observation on signed subgraphs with minimal cutset.

► **Observation 3.** *A signed subgraph S^{\pm} is a signed subgraph with minimal cutset if and only if the following two conditions hold:*

1. *For all $v \in V$, at most one of a zero edge or a positive edge is incident to v .*
 2. *For all the negative edges $\{u, v\}$, a positive edge is incident to at least one of u and v .*
- Conditions 1 and 2 in Observation 3 ensure that $\text{abs}(S^-)$ is a cutset such that removing it leaves the connected component whose edge set is $\text{abs}(S^+)$ and the minimality of $\text{abs}(S^-)$. This shows the correctness of the observation. We design an algorithm based on frontier-based search to construct a TDD representing the set of all the signed subgraphs satisfying Conditions 1 and 2 in Observation 3.

First, we consider Condition 1. To ensure Condition 1, we store an array `colors` : $V \rightarrow 2^{\{0,+, -\}}$ into each TDD node. For all $v \in F_{i-1}$, we manage $n_i.\text{colors}[v]$ so that it is equal to the set of types of edges incident to v . For example, if a zero edge and a positive edge are incident to v and no negative edges are, `colors`[v] must be $\{0, +\}$. We can prune the case such that Condition 1 is violated using `colors`, which ensures Condition 1.

Next, we consider Condition 2. Let $\{u, v\}$ be a negative edge. When u and v leave the frontier at the same time, we check if Condition 2 is satisfied from `colors`[u] and `colors`[v] and, if not, we prune the case. When one of u or v leaves the frontier (without loss of generality, we assume the vertex is u), if no positive edges are incident to u , at least one positive edge must be incident to v later. To deal with this situation, we store an array `reserved` : $V \rightarrow \{0, 1\}$ into each TDD node. For all $v \in F_{i-1}$, we manage `reserved`[v] so

that $\text{reserved}[v] = 1$ if and only if at least one positive edge must be incident to v later. We can prune the cases such that $v \in V$ is leaving the frontier and both $\text{reserved}[v] = 1$ and $+ \notin \text{colors}[v]$ hold, which violate Condition 2. We show pseudocode of `MAKE_NEWNODE` function in Appendix of the full version.

We give the following lemma on the width of a ZDD constructed by the algorithm. We show a proof of the lemma in the full version.

► **Lemma 4.** *The width W_T of a ZDD constructed by the above algorithm is $W_T = \mathcal{O}(6^f)$.*

Next, we show how to construct $T_{\mathcal{S}^\pm}$ using $Z_{\mathcal{S}}$. We can achieve this goal using *subsetting* technique [5] with the above algorithm. Subsetting technique is a framework to construct a decision diagram corresponding to another decision diagram. We ensure that, for all $S^\pm = S^+ \cup S^- \in \mathcal{S}^\pm$, there exists $S \in \mathcal{S}$ such that $\text{abs}(S^+) = S$ in the construction of $T_{\mathcal{S}^\pm}$ using subsetting technique.

3.4 Constructing $Z_{\mathcal{S}^\uparrow}$

In this section, we show how to construct $Z_{\mathcal{S}^\uparrow}$ and how to deal with forbidden components consisting only of one vertex whose weight is less than L , which was left as a problem in Sec. 3.2. From Observation 1 and Eqs. (1)–(3), \mathcal{S}^\uparrow can be written as

$$\mathcal{S}^\uparrow = \{E' \subseteq E \mid \exists S^\pm \in \mathcal{S}^\pm, (\forall +e \in S^\pm, e \in E') \wedge (\forall -e \in S^\pm, e \notin E')\}. \quad (4)$$

Using $T_{\mathcal{S}^\pm}$, we can construct $Z_{\mathcal{S}}$ by the existing algorithm [9].

Finally, we show how to deal with a graph partition containing a single vertex v such that $p(v) < L$ as a connected component, i.e., a partition has an isolated vertex with small weight. Let \mathcal{F}_v be the set of graph partitions containing $(\{v\}, \emptyset)$ as a connected component. A graph partition $E' \subseteq E$ belongs to \mathcal{F}_v if and only if E' does not contain any edge incident to v . Using this, we can construct the ZDD Z_v representing \mathcal{F}_v in $\mathcal{O}(m)$ time. For each $v \in V$ such that $p(v) < L$, we construct Z_v and update $Z_{\mathcal{S}^\uparrow} \leftarrow Z_{\mathcal{S}^\uparrow} \cup Z_v$. In this way, we can deal with all the graph partitions containing a connected component whose weight is less than L . We show an example of execution of the whole algorithm in Appendix of the full version.

4 Experimental results

We conducted computational experiments to evaluate the proposed algorithm and to compare it with the existing state-of-the-art algorithm of Kawahara et al [6]. We used a machine with an Intel Xeon Processor E5-2690v2 (3.00 GHz) CPU and a 64 GB memory (Oracle Linux 6) for the experiments. We have implemented the algorithms in C++ and compiled them by g++ with the `-O3` optimization option. In the implementation, we used the `TdZdd` library [5] and the `SAPPORO_BDD` library.² The timeout is set to be an hour.

We used graphs representing some prefectures in Japan for the input graphs. The vertices represent cities and there is an edge between two cities if and only if they have the common border. The weight of a vertex represents the number of residents living in the city represented by the vertex. As for the input ZDD $Z_{\mathcal{A}}$, we adopted three types of graph partitions: graph partitions such that each connected component is an induced subgraph [6], which we call *induced partition*, forests, and rooted forests. There is a one-to-one correspondence between

² Although the `SAPPORO_BDD` library is not released officially, you can see the code in <https://github.com/takemaru/graphillion/tree/master/src/SAPPOROBDD>.

■ **Table 1** Summary of input graphs and input graph partitions.

Name	n	m	k	Induced partition		Forest		Rooted forest	
				$ Z_{\mathcal{A}} $	$ \mathcal{A} $	$ Z_{\mathcal{A}} $	$ \mathcal{A} $	$ Z_{\mathcal{A}} $	$ \mathcal{A} $
G_1 (Gumma)	37	80	4	10236	1.25×10^8	26361	1.01×10^{19}	8957	1.66×10^{16}
G_2 (Ibaraki)	44	95	7	17107	6.38×10^{13}	15553	6.14×10^{23}	3238	1.94×10^{19}
G_3 (Chiba)	60	134	14	301946	6.69×10^{22}	213773	4.86×10^{33}	15741	5.04×10^{25}
G_4 (Aichi)	69	173	17	1598213	9.26×10^{29}	879361	1.78×10^{42}	43465	3.10×10^{30}
G_5 (Nagano)	77	185	5	13203	2.77×10^{17}	44804	2.95×10^{43}	26476	7.66×10^{39}

induced partitions and partitions of the vertex set. A rooted forest is a forest such that each tree in the forest has exactly one specified vertex. We chose special vertices for each graph randomly. A summary of input graphs and input graph partitions is in Tab. 1. In the table, we show graph names and the prefecture represented by the graph, the number of vertices (n), edges (m) and connected components (k) in graph partitions. The groups of columns “Induced partition”, “Forest”, and “Rooted forest” indicate the types of input graph partitions. Inside each of them, we show the size (the number of non-terminal nodes) of $Z_{\mathcal{A}}$ and the cardinality of \mathcal{A} .

The lower bounds of weights are determined as follows. Let k be the number of connected components in a graph partition and r be the maximum ratio of the weights of two connected components in the graph partition. From k and r , we can derive the necessary condition that the weight of every connected component must be at least $L(k, r) = P/(r(k-1) + 1)$, where $P = \sum_{v \in V} p(v)$ [6]. We used $L(k, r)$ as the lower bound of weights in the experiment. For each graph, we run the algorithms in $r = 1.1, 1.2, 1.3, 1.4$, and 1.5 .

We show the experimental results in Tab. 2. In the table, we show the graph name, the value of r and $L(k, r)$, and the execution time of *Alg. N*, the proposed algorithm, and *Alg. K*, the algorithm of Kawahara et al. The size of $Z_{\mathcal{B}}$ and the cardinality of \mathcal{B} are also shown. “OOM” means *out of memory* and “-” means both algorithms failed to construct the ZDD (due to timeout or out of memory). We marked the values of the time of the algorithm which finished faster as bold.

First, we analyze the results for induced partitions. For the input graphs from G_1 to G_4 , both Alg. N and Alg. K succeeded in constructing $Z_{\mathcal{B}}$, except when $r = 1.1$ in G_4 for Alg. K. In cases where both algorithms succeeded in constructing $Z_{\mathcal{B}}$, the time for Alg. N to construct the ZDD is 2–32 times shorter than that for Alg. K. In addition, Alg. N succeeded in constructing the ZDD when $r = 1.1$ in G_4 , where Alg. K failed to construct the ZDD because of out of memory. These results show the efficiency of our algorithm. In contrast, for G_5 , although both algorithms failed to construct the ZDD when $r = 1.1, 1.2, 1.3$ and 1.4 , only Alg. K succeeded when $r = 1.5$. In this case, the size of the ZDD constructed by Alg. N did stay in the limitation of memory while, in our algorithm, the size of $Z_{\mathcal{S}^\uparrow}$ exceeded the limitation of memory.

Second, we investigate the results for forests. Both Alg. N and Alg. K succeeded in constructing $Z_{\mathcal{B}}$ for the input graph from G_1 to G_4 . In all those cases, Alg. N was faster than Alg. K. Comparing the results with those of induced partitions, we found that the execution time of Alg. K depends on the input partitions more than Alg. N does. For example, for G_1 , while the execution time of Alg. N is almost irrelevant to the types of input ZDDs, that of Alg. K differ up to about five times. This is because the efficiency of Alg. K strongly depends on the sizes of input ZDDs. This makes the sizes of output ZDDs constructed by Alg. K large, which implies the increase in the execution time of Alg. K. In contrast, the execution time of Alg. N does not depend on the sizes of input ZDDs in many cases because Alg. N

Table 2 Experimental results for three types of input graph partitions.

	r	$L(r, k)$	Induced partition			Forest			Rooted forest					
			Alg. N	Alg. K	$ B $	Alg. N	Alg. K	$ B $	Alg. N	Alg. K	$ B $			
G_1	1.1	458947	4.22	12.07	4912	1.74×10^4	4.03	50.84	29502	8.24×10^{12}	3.95	14.96	17920	3.52×10^{11}
	1.2	429016	2.06	10.50	3500	5.40×10^4	2.04	47.30	21364	3.10×10^{13}	2.02	13.34	6331	1.68×10^{12}
	1.3	402750	1.15	7.49	2986	9.02×10^4	1.18	36.10	18113	7.42×10^{13}	1.17	10.54	4655	4.44×10^{12}
	1.4	379514	0.99	5.72	3115	2.52×10^5	1.03	24.41	20605	3.84×10^{14}	1.03	6.97	7677	3.18×10^{13}
	1.5	358813	0.90	5.12	3562	2.99×10^5	0.89	23.29	20367	7.19×10^{14}	0.88	6.52	6719	6.17×10^{13}
G_2	1.1	383928	3.70	29.48	27927	1.91×10^6	3.60	35.28	47461	2.56×10^{13}	3.53	2.19	391	4.32×10^6
	1.2	355836	3.03	23.03	83053	1.25×10^8	2.92	25.59	143455	2.11×10^{15}	2.95	1.81	3103	3.72×10^9
	1.3	331574	1.73	16.25	92334	1.02×10^9	1.70	18.09	154449	1.41×10^{16}	1.60	1.74	5861	1.36×10^{11}
	1.4	310410	1.21	12.45	105507	4.54×10^9	1.30	14.03	179186	1.02×10^{17}	1.28	1.55	5710	1.54×10^{12}
	1.5	291785	0.73	8.88	98231	1.25×10^{10}	0.74	9.38	149403	3.06×10^{17}	0.70	1.21	5855	6.74×10^{12}
G_3	1.1	377742	83.76	1008.11	0	0	77.19	811.03	0	0	78.68	66.96	0	0
	1.2	348159	32.87	852.47	6641	2.32×10^5	27.12	657.89	17252	1.34×10^{13}	27.27	89.75	0	0
	1.3	322874	23.33	626.94	261978	3.12×10^{10}	20.87	452.10	768876	1.53×10^{19}	36.20	36.30	0	0
	1.4	301013	12.08	386.91	328581	4.92×10^{11}	10.88	266.19	917102	3.23×10^{20}	9.70	22.14	0	0
	1.5	281924	10.81	315.40	405816	3.02×10^{12}	9.29	205.90	1062331	9.94×10^{20}	7.64	19.44	606	2.88×10^{10}
G_4	1.1	402370	155.05	OOM	190520	1.54×10^{10}	64.12	1032.53	374111	5.43×10^{18}	51.95	0.65	0	0
	1.2	370499	86.91	628.93	739356	1.98×10^{14}	24.09	317.44	1374522	1.41×10^{23}	20.82	0.96	0	0
	1.3	343307	125.06	408.97	1148330	1.98×10^{16}	14.83	190.25	2005760	7.27×10^{24}	11.69	1.48	0	0
	1.4	319833	108.25	281.81	1465722	6.32×10^{17}	12.18	134.15	2495000	1.87×10^{26}	8.31	3.09	5645	2.19×10^{11}
	1.5	299363	29.13	190.59	1761682	1.65×10^{19}	9.60	85.84	2434632	4.02×10^{27}	5.55	3.46	15587	9.56×10^{14}
G_5	1.1	388844	> 1 h	OOM	-	-	> 1 h	OOM	-	-	> 1 h	< 0.01	0	0
	1.2	362027	> 1 h	OOM	-	-	> 1 h	OOM	-	-	> 1 h	< 0.01	0	0
	1.3	338670	OOM	OOM	-	-	> 1 h	OOM	-	-	> 1 h	< 0.01	0	0
	1.4	318145	OOM	OOM	-	-	OOM	OOM	-	-	OOM	< 0.01	0	0
	1.5	299965	OOM	1960.28	393178	9.20×10^{13}	OOM	OOM	-	-	OOM	< 0.01	0	0

■ **Table 3** Detailed experimental results of the proposed algorithm for G_3 (Chiba) and G_4 (Aichi) when the input graph partitions are induced partitions.

	r	Z_S			T_{S^\pm}		Z_{S^\dagger}			$Z_A \setminus Z_{S^\dagger}$
		time	node	card	time	node	time	node	card	time
G_3	1.1	1.90	54745	4.24×10^8	0.93	99057	75.88	2117874	2.17532×10^{40}	5.05
	1.2	1.01	39845	1.67×10^8	0.69	75581	27.94	977840	2.17528×10^{40}	3.23
	1.3	0.58	31030	6.62×10^7	0.51	60034	18.83	814538	2.17498×10^{40}	3.41
	1.4	0.34	24066	3.30×10^7	0.38	48818	8.49	490753	2.17490×10^{40}	2.87
	1.5	0.25	19877	1.42×10^7	0.34	40340	7.23	410152	2.17486×10^{40}	2.99
G_4	1.1	0.02	2376	2.09×10^4	0.32	11109	80.03	3074734	1.19200×10^{52}	74.68
	1.2	0.01	1686	1.03×10^4	0.20	8511	22.24	1205320	1.19174×10^{52}	64.46
	1.3	0.01	1235	6.11×10^3	0.17	6935	11.51	692798	1.19170×10^{52}	113.37
	1.4	< 0.01	961	3.67×10^3	0.14	5808	8.30	529214	1.19164×10^{52}	99.81
	1.5	< 0.01	756	2.67×10^3	0.13	4930	5.30	348832	1.19153×10^{52}	23.70

uses the input ZDD only in the set difference operation, which is executed in the last of the algorithm (by the existing apply-like method). As we show later, the bottleneck of Alg. N is the construction of Z_{S^\dagger} . Therefore, in many cases, the sizes of input ZDDs do not change the execution time of Alg. N.

Third, we examine the results when the input graph partitions are rooted forests. There are 13 cases such that Alg. K was faster than Alg. N. In the cases, the sizes of input ZDDs and output ZDDs are small, that is, thousands, or even zero. These results show that Alg. K tends to be faster when the sizes of input ZDDs and output ZDDs are small.

In order to assess the efficiency of our algorithm in each step, we show detailed experimental results for G_3 and G_4 when the input graph partitions are induced partitions in Tab. 3. In the table, we show the time to construct decision diagrams, the size of decision diagrams, and the cardinality of the family represented by ZDDs. The cardinality of S^\pm is omitted because it is equal to that of \mathcal{S} . The size and cardinality for $Z_A \setminus Z_{S^\dagger}$ are also omitted because they are the same as $|Z_B|$ and $|\mathcal{B}|$, which are shown in Tab. 2. For both G_3 and G_4 , the time to construct Z_S and T_{S^\pm} are within one or two seconds. The most time-consuming parts are the construction of Z_{S^\dagger} in G_3 and Z_{S^\dagger} or $Z_A \setminus Z_{S^\dagger}$ in G_4 . The set difference operation in G_4 took a lot of time because the sizes of Z_A and Z_{S^\dagger} are large, that is, more than a hundred. The reason why the construction of Z_{S^\dagger} takes a lot of time is the increase in the sizes of decision diagrams. While the size of T_{S^\pm} is only 2–7 times larger than that of Z_S , that of Z_{S^\dagger} is about 10–276 times larger than that of T_{S^\pm} . This also made the execution of the algorithm in G_5 impossible.

5 Conclusion

In this paper, we have proposed an algorithm to construct a ZDD representing all the graph partitions such that all the weights of its connected components are at least a given value. As shown in the experimental results, the proposed algorithm has succeeded in constructing a ZDD representing a set of more than 10^{12} graph partitions in ten seconds, which is 30 times faster than the existing state-of-the-art algorithm. Future work is devising a more memory efficient algorithm that enables us to deal with larger graphs, that is, graphs with hundreds of vertices. It is also important to seek for efficient algorithms to deal with other constraints on weights such that the ratio of the maximum and the minimum of weights is at most a specified value.

References


- 1 Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 100(8):677–691, 1986.
- 2 Gary Hardy, Corinne Lucet, and Nikolaos Limnios. K-terminal network reliability measures with binary decision diagrams. *IEEE Transactions on Reliability*, 56(3):506–515, 2007.
- 3 Hiroshi Imai, Kyoko Sekine, and Keiko Imai. Computational investigations of all-terminal network reliability via BDDs. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E82-A:714–721, 1999.
- 4 Takeru Inoue, Keiji Takano, Takayuki Watanabe, Jun Kawahara, Ryo Yoshinaka, Akihiro Kishimoto, Koji Tsuda, Shin-ichi Minato, and Yasuhiro Hayashi. Distribution loss minimization with guaranteed error bound. *IEEE Transactions on Smart Grid*, 5(1):102–111, 2014.
- 5 Hiroaki Iwashita and Shin-ichi Minato. Efficient top-down ZDD construction techniques using recursive specifications. *TCS Technical Reports*, TCS-TR-A-13-69, 2013.
- 6 Jun Kawahara, Takashi Horiyama, Keisuke Hotta, and Shin-ichi Minato. Generating all patterns of graph partitions within a disparity bound. In *Proc. of the 11th International Conference and Workshops on Algorithms and Computation (WALCOM)*, pages 119–131, 2017.
- 7 Jun Kawahara, Takeru Inoue, Hiroaki Iwashita, and Shin-ichi Minato. Frontier-based search for enumerating all constrained subgraphs with compressed representation. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 100(9):1773–1784, 2017.
- 8 Jun Kawahara, Toshiki Saitoh, Hirofumi Suzuki, and Ryo Yoshinaka. Solving the longest oneway-ticket problem and enumerating letter graphs by augmenting the two representative approaches with ZDDs. In *Computational Intelligence in Information Systems*, pages 294–305, 2017.
- 9 Jun Kawahara, Toshiki Saitoh, Hirofumi Suzuki, and Ryo Yoshinaka. Enumerating all subgraphs without forbidden induced subgraphs via multivalued decision diagrams. *CoRR*, 2018. [arXiv:1804.03822](https://arxiv.org/abs/1804.03822).
- 10 Donald E. Knuth. *The art of computer programming, Vol. 4A, Combinatorial algorithms, Part 1*. Addison-Wesley, 2011.
- 11 Takanori Maehara, Hirofumi Suzuki, and Masakazu Ishihata. Exact computation of influence spread by binary decision diagrams. In *Proc. of the 26th International World Wide Conference (WWW)*, pages 947–956, 2017.
- 12 Shin-ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proc. of the 30th ACM/IEEE design automation conference*, pages 272–277, 1993.
- 13 Yu Nakahata, Jun Kawahara, Takashi Horiyama, and Shoji Kasahara. Enumerating all spanning shortest path forests with distance and capacity constraints. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* (to appear).
- 14 Kyoko Sekine, Hiroshi Imai, and Seiichiro Tani. Computing the Tutte polynomial of a graph of moderate size. In *Proc. of the 6th International Symposium on Algorithms and Computation (ISAAC)*, pages 224–233, 1995.
- 15 Koichi Yasuoka. A new method to represent sets of products: ternary decision diagrams. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 78(12):1722–1728, 1995.
- 16 Ryo Yoshinaka, Toshiki Saitoh, Jun Kawahara, Koji Tsuruma, Hiroaki Iwashita, and Shin-ichi Minato. Finding all solutions and instances of numberlink and slitherlink by ZDDs. *Algorithms*, 5(2):176–213, 2012.

Exact Algorithms for the Maximum Planar Subgraph Problem: New Models and Experiments

Markus Chimani¹

Theoretical Computer Science, Osnabrück University, Germany


markus.chimani@uni-osnabrueck.de

 <https://orcid.org/0000-0002-4681-5550>

Ivo Hedtke

Data Strategy & Analytics, Schenker AG, Essen, Germany


ivo.hedtke@dbschenker.com

 <https://orcid.org/0000-0003-0335-7825>

Tilo Wiedera¹

Theoretical Computer Science, Osnabrück University, Germany

tilo.wiedera@uni-osnabrueck.de

 <https://orcid.org/0000-0002-5923-4114>

Abstract

Given a graph G , the NP-hard *Maximum Planar Subgraph* problem asks for a planar subgraph of G with the maximum number of edges. The only known non-trivial exact algorithm utilizes Kuratowski's famous planarity criterion and can be formulated as an integer linear program (ILP) or a pseudo-boolean satisfiability problem (PBS). We examine three alternative characterizations of planarity regarding their applicability to model maximum planar subgraphs. For each, we consider both ILP and PBS variants, investigate diverse formulation aspects, and evaluate their practical performance.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms

Keywords and phrases maximum planar subgraph, integer linear programming, pseudo boolean satisfiability, graph drawing, algorithm engineering

Digital Object Identifier 10.4230/LIPIcs.SEA.2018.22

1 Introduction

The NP-hard *Maximum Planar Subgraph Problem* (MPS) is a long known problem in graph theory, already discussed in the classical textbook by Garey and Johnson [16, 24]. Given a graph $G = (V, E)$, we ask for a largest edge subset $F \subseteq E$ such that the graph induced by F is planar. The closely related *maximal* planar subgraph problem asks for a set of edges that we cannot extend without violating planarity and is trivially solvable in polynomial time. Sometimes, the inverse measure *skewness* $skew(G)$ is considered, where we ask for the minimum number of edges to delete until obtaining planarity. MPS has received significant attention for diverse reasons. Firstly, skewness is considered a very natural measure of non-planarity and resides among the most common ones (such as crossing number and genus). Secondly, determining a large planar subgraph is the foundation of the planarization method [1, 8] that is heavily employed in graph drawing: during planarization, one draws a large – favorably maximum – planar subgraph and re-inserts the deleted edges, usually to

¹ Supported by the German Research Foundation (DFG) project CH 897/2-1.



obtain a low number of overall crossings. In fact, this gives an approximation algorithm with ratio roughly $\mathcal{O}(\Delta \cdot skew(G))$ [10], where Δ denotes the maximum node degree. Thirdly, there are graph problems that become easier when the skewness of the input is small or constant. E.g., we can compute a maximum flow in time $\mathcal{O}(skew(G)^3 \cdot |V| \log |V|)$ [19], i.e., for constant skewness we obtain the same runtime complexity as on planar graphs.

On the positive side, we know that a spanning tree already approximates MPS by $1/3$. The best known approximation algorithm is due to Călinescu et al., achieving an approximation ratio of $4/9$ [4]. On the downside, Călinescu et al. also show that the problem is MaxSNP-hard, i.e., there is an upper bound < 1 on the obtainable approximation ratio unless $P = NP$. Just recently, a new algorithm with approximation ratio $13/33$ [6] was discovered. The only non-trivial algorithm in literature for *exactly* computing a maximum planar subgraph is based on integer linear programming and Kuratowski's characterization of planarity [26]. Since its inception over two decades ago, *no* other exact algorithm has been proposed, and only few related algorithmic advances improved its performance, see [11].

Besides this famous K_5 - $K_{3,3}$ -subdivision criteria by Kuratowski [22] (see Section 2) there is an abundance of planarity criteria. A (non-complete) list can be found in [23, 31]. In this paper, we aim at evaluating planarity criteria regarding their usefulness in ILP/PBS formulations to obtain new, alternative exact MPS algorithms. Naturally, we restrict ourselves to a subset of criteria that we deem promising for this investigation. We hope to pinpoint new possible ways of considering the problem, to gain new insight into the structure of the MPS, and to lay the groundwork for developing faster exact algorithms. We present our three new models in Sections 3–5. For each of the possible formulations, there are several options and parameter choices. We report on algorithmic and experimental decisions thereto directly after their description, based on pilot studies². In Section 6 we present a full comparison of the best parameterization for each formulation.

2 Preliminaries

In *Linear Programming* (LP), one is given a vector $c \in \mathbb{R}^d$, a set of linear inequalities that define a polyhedron P in \mathbb{R}^d , and asked to find an element $x \in P$ that maximizes $c^\top x$. *Integer Linear Programming* (ILP) additionally requires the components of x to be integral. Closely related is the concept of *Pseudo Boolean Satisfiability* (PBS), sometimes referred to as 0-1-integer linear programming (a special form of ILP: the given polyhedron is a subset of $[0, 1]^d$), but typically described as a generalization of SAT: its describing constraints are called *clauses* and usually have the form of first order Boolean formulae. Modern solvers directly support clauses that require a certain number of literals (instead of just one) to be true. The main difference between PBS and 0-1-ILP is the solution strategy: the first uses fast enumeration and clause learning, whereas the latter employs LP-relaxations. We use these concepts to design models for MPS that can be solved by arbitrary ILP/PBS solvers.

It often is beneficial to not add all constraints to a program but instead identify a relevant subset of constraints in the solving process. This is usually referred to as the *Cutting-Plane Method*. We utilize it in branch-and-cut-based ILP solving either on fractional or integral solutions. In PBS solvers, one has to rely on a less sophisticated approach that iteratively solves the PBS formula, adds new constraints as appropriate, and re-solves the extended formula while maintaining some information from the previous runs. We refer to clauses that are added iteratively to a PBS formula as *lazy constraints*.

² The experimental setting is the same as discussed in Section 6.

2.1 Notation

Throughout this paper, our input graph $G = (V, E)$ is undirected and simple with $n := |V|$ and $m := |E|$. For general graphs H , we refer to its nodes as $V(H)$ and its edges as $E(H)$. For a directed graph H' we denote the arcs by $A(H')$ and may write $E(H')$ whenever considering H' undirected. For any $k \in \mathbb{N}$, we denote the set $\{0, 1, \dots, k-1\}$ by $[k]$ and operations on the members are to be understood modulo k . For any edge e of G , let $V^e := V \setminus e$ denote the nodes that are not incident with e . Given a node v , its neighbors are denoted by $N(v)$. In a directed graph, we refer to the outgoing (incoming) arcs of a node v as $\delta^+(v)$ ($\delta^-(v)$, respectively). For any two nodes u and v , we denote an arc from u to v by uv . If unambiguous, we might also refer to an undirected edge $\{u, v\}$ as uv . We denote the undirected counterpart of an arc a as $e(a)$. Given an arc $a = uv$, we define its reversal $\text{rev}(a) := vu$. Given a set X , the set of all ordered k -tuples (all k -cardinality subsets) consisting of elements from X is referred to as $X^{(k)}$ ($X^{\{k\}}$, respectively). We abbreviate *pairwise different* by *p.d.*

2.2 Common Foundation of Models

We assume our input graph G to be biconnected non-planar, with edge weights $w: E(G) \rightarrow \mathbb{N}$ and minimum node degree 3. This can be achieved in linear time using the *Non-Planar Core* reduction [7] as a preprocessing, without changing the graph's skewness.

All models are presented as ILPs. Since the PBS counterparts directly map to the ILPs where clauses naturally correspond to constraints, we do not explicitly list the PBS formulations. We highlight optional constraints that we include in the hope to help quickly finding strong dual bounds with the symbol \star . We use solution variables $s_e \in \{0, 1\}$ (for all $e \in E(G)$) that are 1 if and only if edge e is in the planar subgraph. The objective is given by

$$\max \sum_{e \in E(G)} w(e) \cdot s_e.$$

We always use Euler's bound on the number of edges in planar graphs:

$$\sum_{e \in E(G)} s_e \leq 3n - 6.$$

2.3 Known Formulation: Kuratowski Subdivisions

► **Theorem 1** (Kuratowski's Theorem [22]). *A graph is planar if and only if it neither contains a subdivision of a K_5 nor that of a $K_{3,3}$.*

Hence, it suffices to ask for any member of the (exponentially sized) set $\mathcal{K}(G)$ of all Kuratowski subdivisions that at least one of its edges is deleted:

$$\sum_{e \in E(K)} s_e \leq |E(K)| - 1 \quad \forall K \in \mathcal{K}(G).$$

This formulation is due to Mutzel [26]. Later, Jünger and Mutzel showed that these constraints form facets of the planar subgraph polytope [20]. Clearly, we cannot solve the model by writing down every constraint explicitly. Instead, a sufficiently large but in many practical cases small subset of constraints is identified by a (heuristic) separation procedure. Over the years, the performance of this approach was improved by strong preprocessing [7], finding *multiple* violated constraints in linear time [12], and good heuristics [11].

Algorithm engineering and preliminary benchmarks. Using an ILP solver, we separate on LP-solutions by rounding the computed fractional values, thus obtaining a graph $H \subset G$ and extracting Kuratowski subdivisions from H . Our experiments indicate that rounding down values that are smaller than 0.99 (and 0.9 in a second round), yields locally optimal (w.r.t. the algorithm's parameter space) results. We use a heap to collect 50 most violated constraints per LP-solution while maintaining linear runtime for the extraction of up to 250 Kuratowski subdivisions. For the PBS solver, we iteratively search for satisfying variable assignments and check each for planarity, adding up to 50 lazy Kuratowski constraints each.

3 Facial Walks

For any connected planar graph, there is an *embedding* Π , i.e., a cyclic order of edges around the nodes while the graph is drawn planarly. The regions bounded by the edges are the *faces* of Π . The facial walk model is based on an idea developed in [2] for computing the genus of a graph; it constitutes the only known model for the latter problem. It simulates the face tracing algorithm that visits each face, traversing their borders in clockwise order. Let \bar{f} be an upper bound on the number of attainable faces. Let A denote the bidirected counterpart of the undirected edges of G . We add the following binary variables:

- $x_i \quad \forall i \in [\bar{f}]$ Has value 1 iff face i exists.
- $c_a^i \quad \forall a \in A, i \in [\bar{f}]$ Has value 1 iff arc a bounds face i : traversing i in clockwise order visits $e(a)$ in the orientation of a .
- $p_{u,w}^v \quad \forall v \in V, u, w \in N(v)$ Has value 1 iff w is the direct successor of u in the cyclic order around v .

We define the following short-hand notations:

$$\begin{aligned} p^v(U \times W) &:= \sum_{u \in U} \sum_{w \in W} p_{u,w}^v, & x(I) &:= \sum_{i \in I} x_i, \\ s_v(W) &:= \sum_{w \in W} s_{vw}, & c^I(J) &:= \sum_{i \in I} \sum_{j \in J} c_j^i. \end{aligned}$$

We then complete our model with the constraints below:

$$n + x([\bar{f}]) = 2 + \sum_{e \in E} s_e \tag{1a}$$

$$x_i = 1 \quad \forall i \in [3] \quad \star \tag{1b}$$

$$x_i \geq x_{i+1} \quad \forall i \in [\bar{f} - 1] \quad \star \tag{1c}$$

$$x_i \leq c^{\{i\}}(A)/3 \quad \forall i \in [\bar{f}] \tag{1d}$$

$$c_a^i \leq x_i \quad \forall a \in A, i \in [\bar{f}] \tag{1e}$$

$$c^{[\bar{f}]}(a) = s_{e(a)} \quad \forall a \in A \tag{1f}$$

$$c^{\{i\}}(\delta^-(v)) = c^{\{i\}}(\delta^+(v)) \quad \forall i \in [\bar{f}], v \in V \tag{1g}$$

$$c_{vw}^i \geq c_{uv}^i + p_{u,w}^v - 1 \quad \forall i \in [\bar{f}], v \in V, u, w \in N(v) \tag{1h}$$

$$c_{uv}^i \geq c_{vw}^i + p_{u,w}^v - 1 \quad \forall i \in [\bar{f}], v \in V, u, w \in N(v) \tag{1i}$$

$$p^v(u \times N(v)) = s_{vu} \quad \forall v \in V \tag{1j}$$

$$p^v(N(v) \times w) = s_{vw} \quad \forall v \in V, w \in A \tag{1k}$$

$$p^v(U \times N(v) \setminus U) \geq s_v(\{u, \tilde{u}\}) - 1 \quad \forall v \in V, \emptyset \neq U \subsetneq N(v), u \in U, \tilde{u} \in N(v) \setminus U \tag{1l}$$

Inequality (1a) ensures that the number of nodes, faces, and edges satisfy Euler's polyhedron formula. Constraints (1d) account for the fact that each face needs at least three arcs. Conversely, for any arc to be assigned to a face, the face needs to exist ($\rightarrow 1e$). For any arc whose edge is in the planar subgraph there must exist exactly one face that contains the

arc ($\rightarrow 1f$). Constraints (1g) ensure that the number of inbound arcs equals the number of outbound arcs at a fixed node in a fixed face. By adding constraints (1h,1i), we make sure to respect the successor-variables. Constraints (1j,1k) ensure there are successor variables selected for any edge that is in the solution. The exponentially large set of cut constraints (1l) prohibits multiple cycles in the successor relation. Optionally, we can force the use of at least the first 3 faces ($\rightarrow 1b$), otherwise the solution is outerplanar and thus not maximal; and we can use faces in order of their indices ($\rightarrow 1c$) to break symmetries.

Special variables/constraints for degree-3 nodes. Consider any degree-3 node v with neighbors u_0^v, u_1^v, u_2^v . If all its incident edges are in the solution, we have two possible cyclic orders. Otherwise, the cyclic order is even unique. Thus, instead of introducing six successor-variables p^v and constraints (1h–1l), we can use a single binary variable p^v , and straight-forwardly simplified constraints, for all $i \in [\bar{f}]$, $j \in [3]$, and all degree-3 nodes v :

$$\begin{aligned} c_{vu_{j+1}^v}^i &\geq c_{u_j^v}^i + (p^v - 1) + (s_{vu_{j+1}^v} - 1) & c_{vu_{j+2}^v}^i &\geq c_{u_j^v}^i + (p^v - 1) + (s_{vu_{j+2}^v} - 1) - s_{vu_{j+1}^v} \\ c_{u_j^v}^i &\geq c_{vu_{j+1}^v}^i + (p^v - 1) + (s_{u_j^v} - 1) & c_{u_j^v}^i &\geq c_{vu_{j+2}^v}^i + (p^v - 1) + (s_{u_j^v} - 1) - s_{vu_{j+1}^v} \\ c_{vu_j^v}^i &\geq c_{u_{j+1}^v}^i - p^v + (s_{vu_j^v} - 1) & c_{vu_j^v}^i &\geq c_{u_{j+2}^v}^i - p^v + (s_{vu_j^v} - 1) - s_{vu_{j+1}^v} \\ c_{u_{j+1}^v}^i &\geq c_{vu_j^v}^i - p^v + (s_{u_{j+1}^v} - 1) & c_{u_{j+2}^v}^i &\geq c_{vu_j^v}^i - p^v + (s_{u_{j+2}^v} - 1) - s_{vu_{j+1}^v} \end{aligned}$$

It can be easily verified by a case analysis that the above inequalities cover every possible configuration of neighbors, where we might assume that there is at least one neighbor since every maximal solution must be connected.

Algorithm engineering and preliminary benchmarks. In our experiments, the special degree-3 node model did not solve more instances but resulted in a marginal reduction (0.8%) of runtime; so we use it. The PBS variant on the other hand suffers from the special degree-3 model, solving 9.38% less instances. An ILP variant where we eliminate the solution variables s_e (directly using the containment variables c_a^i instead) solved 3.29% less instances. We refrain from testing polynomially sized models (betweenness- and index-based instead of constraints (1h–1l)) as our exact genus experiments suggest this does not pay off [2].

4 Schnyder Orders

A *partially ordered set* (*poset*) is a pair $P = (S, \prec)$ where \prec is a strict partial order (transitive, irreflexive, binary relation) over the elements of S . Every poset has a *realizer*, i.e., a set \mathcal{R} of total orders (transitive, antisymmetric, total, binary relation) on S whose intersection is \prec [30]. This means that $x \prec y$ if and only if $x <_i y$ for all $<_i \in \mathcal{R}$. The *Dushnik-Miller dimension* $\dim P$ of P is the minimum cardinality over all realizers of P [15]. We associate a poset $P_G = (V \cup E, \prec_G)$ to G such that $x \prec_G y$ if and only if $y = \{v, w\} \in E$ and $x \in y$. The dimension of G is defined as the Dushnik-Miller dimension of P_G . We have

► **Theorem 2** (Schnyder's Theorem, 4.1 and 6.2 of [28]). *A graph is planar if and only if its dimension is at most three.*

In fact, a graph with dimension 1 (2) is an isolated node (path, respectively). Therefore, we propose a model to check for dimension three. While we could directly use the above criterion for an ILP, Schnyder provides another, related and favorable, characterization:

► **Lemma 3** (Lemma 2.1 of [28]). *A graph $G = (V, E)$ has dimension at most d if and only if there exists a set of total orders $<_1, \dots, <_d$ on V such that*

1. the intersection of $\langle_1, \dots, \langle_d$ is empty; and
2. for each edge $\{x, y\} \in E$ and each node $z \notin \{x, y\}$ of G , there is at least one order \langle_i such that $x \langle_i z$ and $y \langle_i z$.

To use this criterion, we add (additionally to $s_e, \forall e \in E$) the following binary variables:

- $t_{u,v}^i \quad \forall i \in [3], \forall u, v \in V: u \neq v$ Has value 1 iff $u \langle_i v$.
- $a_{e,v}^i \quad \forall i \in [3], e \in E, v \in V^e$ Can have value 1 only if $u \langle_i v \quad \forall u \in e$.

We are now able to complete the Schnyder orders ILP by adding:

$$s_e \leq \sum_{i=0}^2 a_{e,v}^i \quad \forall e \in E, v \in V^e \quad (2a)$$

$$a_{e,v}^i \leq t_{u,v}^i \quad \forall i \in [3], e \in E, u \in e, v \in V^e \quad (2b)$$

$$\sum_{i=0}^2 t_{u,v}^i \leq 2 \quad \forall u, v \in V: u \neq v \quad \star (2c)$$

$$t_{u,v}^i + t_{v,w}^i - 1 \leq t_{u,w}^i \quad \forall i \in [3], \text{p.d. } u, v, w \in V \quad (2d)$$

$$t_{u,v}^i + t_{v,u}^i = 1 \quad \forall i \in [3], u, v \in V: u \neq v \quad (2e)$$

Constraints (2a) ensure that for any edge in the solution the Schnyder-property for any non-incident node is satisfied by at least one of the three orders. By inequalities (2b), we make sure that the second requirement of the Schnyder-property is respected. Transitivity of the total orders is obtained by (2d). Finally, we require totality by adding (2e).

As Schnyder states [28], we may omit the intersection criterion (2c) as this is satisfied by any non-trivial solution. Note that for any two adjacent edges uv, vw in the solution and any $i \in [3]$, we cannot have $a_{uv,w}^i = a_{vw,u}^i = 1$, since the orders induced by the a -variables are conflicting. Hence, we might pick a single triangle $T = \{e_1, e_2, e_3\}$ in the input graph and assign realizing orders to each edge; thereby v_i denotes the node incident to both of $T \setminus \{e_i\}$:

$$\sum_{j \in [3] \setminus \{i\}} a_{e_i, v_i}^j = 0 \quad \forall i \in [3] \quad \star (2f)$$

Analogously, we might apply the same symmetry breaking constraint to two adjacent edges if the graph is triangle-free. (Then $e_3 \notin E$, we let $i \in [2]$ but retain the subscript at the sum.)

Algorithm engineering and preliminary benchmarks. We tested omitting the symmetry breaking constraints (2f) (9.12% less solved instances), omitting intersection constraints (2c) (0.85% less), manually separating the transitivity constraints (which does not change the overall number of solved instances but increases runtime by 4.00%), and using Theorem 2 – the partial order on $V \cup E$ – instead (leading to a related but different model that we do not describe here), where we solve 39.89% fewer instances (each when using an ILP solver). Employing the PBS solver, we obtain similar results for omitting symmetry breaking constraints (9.37% less) and for omitting intersection constraints (0.79% less). In contrast to above, using lazy transitivity constraints leads to 5.24% fewer solved instances. We did not investigate a PBS variant based on Theorem 2 as the ILP performance was already strikingly underwhelming. We did consider a variant where we use betweenness variables [5] to describe each of the three total orders. This allows us to omit the a -variables, but it did not yield satisfactory runtime already on rather trivial instances.

5 Left-Right Edge Coloring

A *Trémaux tree* T is a rooted tree in a graph H such that for any *cotree* edge $\{u, v\} \in E_H^T := E(H) \setminus E(T)$, we can traverse the nodes of the tree-path between u and v , such that the levels of the nodes (i.e., their distances in T to the root) are strictly increasing. Any *DFS-tree*

(depth-first-search-tree), rooted at the start node, is a Trémaux tree. For any edge e we refer to the node closer to the root of T as \dot{e} and the other one as \ddot{e} (this is unique by the Trémaux property). Any Trémaux tree T defines a partial order on the nodes: for each edge $e \in E(T)$ we set $\dot{e} \prec \ddot{e}$, the partial order is obtained by extending this relationship to its transitive hull.

► **Definition 4** (*T*-alike and *T*-opposite relations). We denote the *meet* (closest common ancestor) of two nodes u, v in \prec by $u \wedge v$. De Fraysseix and Rosenstiehl [13] define binary relations between cotree edges as follows:

- P1.** For any $\alpha, \beta, \gamma \in E_H^T$ such that $\dot{\gamma} \prec \dot{\alpha} \leq \dot{\beta} \prec \dot{\alpha} \wedge \dot{\beta} \wedge \dot{\gamma} \prec \dot{\alpha} \wedge \dot{\beta}$, α and β are *T*-alike.
- P2.** For any $\alpha, \beta, \gamma \in E_H^T$ such that $\dot{\gamma} \prec \dot{\alpha} \prec \dot{\beta} \prec \dot{\alpha} \wedge \dot{\beta} \wedge \dot{\gamma} \prec \dot{\beta} \wedge \dot{\gamma}$, α and β are *T*-opposite.
- P3.** For any $\alpha, \beta, \gamma, \delta \in E_H^T$ such that $\dot{\gamma} = \dot{\delta} \prec \dot{\alpha} = \dot{\beta} \prec \dot{\alpha} \wedge \dot{\beta} \prec \dot{\alpha} \wedge \dot{\gamma}$, and $\dot{\alpha} \wedge \dot{\beta} \prec \dot{\beta} \wedge \dot{\gamma}$, α and β are *T*-opposite.

► **Theorem 5** (Section 2 of [13]). *A connected graph H with a Trémaux tree T is planar if and only if there exists a partition of E_H^T into two classes, such that any two edges which are T-alike (T-opposite) belong to the same class (different classes, respectively).*

Using this characterization, we design a model that describes a Trémaux tree with a feasible bicoloring of cotree edges for any connected, planar subgraph. We introduce the following set of binary variables, additionally to s_e for all $e \in E$:

- $t_d \quad \forall d \in A$ Has value 1 iff arc d is in the Trémaux tree T .
- $\ell_{uv} \quad \forall u, v \in V$ Has value 1 iff node u lies on the path from the root to node v in T .
Always true for $u = v$ and whenever u is the root of T .
Models the partial Trémaux ordering $u \prec v \iff \ell_{uv} = 1$.
- $r_e \quad \forall e \in E$ Has value 1 iff edge e is colored red (otherwise colored blue).

First, we establish a Trémaux tree. It has $n - 1$ edges ($\rightarrow 3a$), chosen from the planar subgraph ($\rightarrow 3b$). Its edges seed the partial order on the nodes ($\rightarrow 3c$). To make sure the order described by the ℓ -variables is exactly the transitive hull of the tree, we need that nodes with the same parent in the tree are not comparable ($\rightarrow 3d$). Whenever two nodes u, v are smaller than a third one, u must be comparable to v ($\rightarrow 3e$). Constraints (3f), (3h), and (3g) model transitivity, reflexivity, and antisymmetry, respectively. Finally, the Trémaux tree property – any edge of the planar solution being incident with two comparable nodes – is enforced by constraints (3i). Note that the t -variables will always describe a tree, i.e., there are no cycles as this would conflict with the induced partial order by (3c,3f,3g).

$$\sum_{d \in A} t_d = |V| - 1 \quad (3a)$$

$$t_d \leq s_{e(d)} \quad \forall d \in A \quad (3b)$$

$$t_d \leq \ell_d \quad \forall d \in A \quad (3c)$$

$$\ell_{vw} + \ell_wv + t_{uv} + t_{uw} \leq 2 \quad \forall u \in V, \{uv, uw\} \in A^{\{2\}} \quad (3d)$$

$$\ell_{uw} + \ell_{vw} \leq 1 + \ell_{uv} + \ell_{vu} \quad \forall (u, v, w) \in V^{\{3\}} \quad (3e)$$

$$\ell_{uv} + \ell_{vw} \leq \ell_{uw} + 1 \quad \forall (u, v, w) \in V^{\{3\}} \quad (3f)$$

$$\ell_{uv} + \ell_{vu} \leq 1 \quad \forall \{u, v\} \in V^{\{2\}} \quad (3g)$$

$$\ell_{vv} = 1 \quad \forall v \in V \quad (3h)$$

$$s_e \leq \ell_{\dot{e}} + \ell_{\ddot{e}} \quad \forall e \in E \quad (3i)$$

Aiming at cutting off some symmetrical solutions, we may demand that tree edges and deleted edges are colored blue:

$$t_{e\circ} + t_{e\bullet} + r_e \leq 1 \quad \forall e \in E \quad \star \quad (3j)$$

$$r_e \leq s_e \quad \forall e \in E \quad \star \quad (3k)$$

We may also enforce a unique Trémaux tree for each given assignment of s -variables: pick an arbitrary root node $r \in V$, set its incoming arcs to 0 and those of every other node to 1 ($\rightarrow 3l, 3m$). Let $<_\pi$ denote a fixed non-cyclic order on the adjacency entries for each node. We may demand that the first feasible edge in this order is always picked for the tree, thus obtaining a distinct feasible DFS-tree for each assignment of s -variables ($\rightarrow 3n$).

$$\sum_{wr \in A} t_{wr} = 0 \quad \star \quad (3l)$$

$$\sum_{wv \in A} t_{wv} = 1 \quad \forall v \in V \setminus \{r\} \quad \star \quad (3m)$$

$$t_{uw} + \ell_{uw} + s_{uw} \leq 2 \quad \forall uv <_\pi uw \in A \quad \star \quad (3n)$$

We now establish a feasible bicoloring of the cotree edges. We define $R_{\alpha, \beta, \gamma}^{u, v} := C_{\{\alpha, \beta, \gamma\}} + \ell_{\dot{\gamma}\dot{\alpha}} + \ell_{uv} - 2$, where $C_F := \sum_{d \in F} (\ell_d + s_{e(d)} - t_d - t_{\text{rev}(d)} - 2)$ for any $F \subseteq A$.

$$P_{\alpha, \beta}^1(\gamma, u, v) := R_{\alpha, \beta, \gamma}^{u, v} + \ell_{\dot{\alpha}\dot{\beta}} + \ell_{\dot{\beta}u} + \ell_{u\dot{\gamma}} - \ell_{v\dot{\gamma}} + \ell_{v\dot{\alpha}} + \ell_{v\dot{\beta}} - 5,$$

$$P_{\alpha, \beta}^2(\gamma, u, v) := R_{\alpha, \beta, \gamma}^{u, v} + \ell_{\dot{\alpha}\dot{\beta}} + \ell_{\dot{\beta}u} + \ell_{u\dot{\alpha}} - \ell_{v\dot{\alpha}} + \ell_{v\dot{\beta}} + \ell_{v\dot{\gamma}} - 5,$$

$$P_{\alpha, \beta}^3(\gamma, \delta, u, v, w) := R_{\alpha, \beta, \gamma}^{u, v} + C_{\{\delta\}} + \ell_{\dot{\alpha}u} + \ell_{uv} + \ell_{uw} + \ell_{u\dot{\alpha}} + \ell_{u\dot{\beta}} \\ + \ell_{v\dot{\alpha}} - \ell_{v\dot{\beta}} + \ell_{v\dot{\gamma}} - \ell_{v\dot{\delta}} - \ell_{w\dot{\alpha}} + \ell_{w\dot{\beta}} - \ell_{w\dot{\gamma}} + \ell_{w\dot{\delta}} - 9.$$

We model coloring restrictions of type P1 (T -alike), P2 (T -opposite by one other cotree edge), and P3 (T -opposite by two other cotree edges) by constraints (3o–3q), respectively:

$$r_{e(\alpha)} - r_{e(\beta)} \geq P_{\alpha, \beta}^1(\gamma, u, v) \quad \forall \text{ arcs } \alpha, \beta, \gamma \in A \text{ of p.d. edges,} \quad (3o)$$

$$r_{e(\beta)} - r_{e(\alpha)} \geq P_{\alpha, \beta}^1(\gamma, u, v) \quad u \neq v \in V: \dot{\gamma} \neq \dot{\alpha} \wedge \dot{\beta} \neq u$$

$$r_{e(\alpha)} + r_{e(\beta)} \geq 1 + P_{\alpha, \beta}^2(\gamma, u, v) \quad \forall \text{ arcs } \alpha, \beta, \gamma \in A \text{ of p.d. edges,} \quad (3p)$$

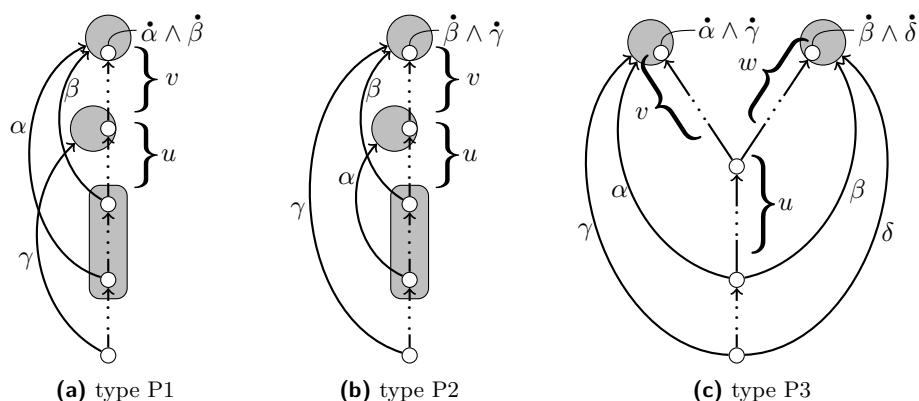
$$r_{e(\alpha)} + r_{e(\beta)} \leq 1 - P_{\alpha, \beta}^2(\gamma, u, v) \quad u \neq v \in V: \dot{\gamma} \neq \dot{\alpha} \neq \dot{\beta} \neq u$$

$$r_{e(\alpha)} + r_{e(\beta)} \geq 1 + P_{\alpha, \beta}^3(\gamma, \delta, u, v, w) \quad \forall \text{ arcs } \alpha, \beta, \gamma, \delta \in A \text{ of p.d. edges,} \quad (3q)$$

$$r_{e(\alpha)} + r_{e(\beta)} \leq 1 - P_{\alpha, \beta}^3(\gamma, \delta, u, v, w) \quad u, v, w \in V: v \neq w \text{ and} \\ \dot{\alpha} = \dot{\beta} \wedge \dot{\gamma} = \dot{\delta} \wedge \dot{\gamma} \neq \dot{\alpha} \neq u$$

To comprehend the latter three constraint classes (3o–3q), one first needs to understand that for any $F \subseteq A$: $-C_F \in \mathbb{N}$ by definition (for any feasible variable assignment) and $C_F = 0$ if and only if each arc of F is a cotree edge of the subgraph induced by the s -variables and directed from the smaller to the larger node. Following this pattern, we define the terms $P_{\alpha, \beta}^1(\gamma, u, v)$, $P_{\alpha, \beta}^2(\gamma, u, v)$, $P_{\alpha, \beta}^3(\gamma, \delta, u, v, w)$ each equal to 0 if and only if we have a configuration of type P1, P2, or P3, respectively, and smaller than or equal to -1 otherwise. Using these terms we can enforce T -alike- and T -oppositeness for any pair α, β as given by constraints (3o–3q); see Figure 1 for the selection of nodes u, v , and w .

DFS-based branching rule. Apart from a traditional automatic selection of branching variables by the ILP solver, we consider a more specialized scheme. Given a vertex in the branch-and-bound (B&B) tree, we traverse the locally non-deleted edges of G (i.e., the edges



■ **Figure 1** Schematics of configurations inducing T -alike and T -opposite with ranges to pick nodes u, v, w from, such that constraints (3o–3q) are tight. Nodes at the bottom are (closest to) the root. Tree paths are straight (cotree edges are bent), partially dotted lines. Subgraphs of arbitrary structure (possibly just a single node) are shaded in gray.

that have a local upper bound of 1) in their unique DFS order until we find an edge e that is not yet chosen to be in the DFS-tree (i.e., the lower bound of the respective arc variable is not 1). We spawn two new B&B subproblems, where e either is deleted or in the DFS-tree, respectively. While we lose the potential benefit of always branching on a strongly fractional value, we can fix two instead of just one free variable in both new B&B branches.

Algorithm engineering and preliminary benchmarks. Since we cannot hope to explicitly write down all coloring constraints (3o–3q), we separate on integral ILP solutions and use lazy constraints in the PBS variant. We use a simple $\mathcal{O}(n^4)$ routine that identifies all violated bicoloring constraints for a given non-planar subgraph. We can terminate this routine prematurely if we consider the set of identified constraints to be locally sufficient.

We evaluated ILP variants where we omitted the symmetry breaking constraints (3l–3n) (49.91% less solved instances), use our custom branch rule while limiting its application to B&B-depth at most 6 (13.22% more) as well as without this limit (32.40% more), and increased the limit of added constraints per LP run from the default of 100 to 1000 (0.93% more). Using the PBS solver, we obtain similar results when omitting symmetry breaking constraints (65.74% less). Furthermore, we investigated a separation routine based on directed cuts³ to cut off infeasible t -variable assignments; this does not seem to be beneficial.

6 Experimental Evaluation

Setup. All our programs are implemented in C++, compiled with GCC 6.3.0, and use the OGDF (version based on snapshot 2017-07-23) [9]. We use SCIP 4.0.1 for solving ILPs with CPLEX 12.7.1 as the underlying LP solver [25]. For PBS-based algorithms, we utilize Clasp 3.3.3 [17]. Each MPS-computation uses a single physical core of a Xeon Gold 6134 CPU (3.2 GHz) with a memory speed of 2666 MHz. We apply a time limit of 20 minutes and a memory limit of 8 GB per computation. Our instances and results, giving runtime and skewness (if solved), are available for download at <http://tcs.uos.de/research/mps>.

³ Directed cut constraints of the form $\sum_{w \in W, v \in V \setminus W} t_{wv} \geq 1$ for all W with $\{r\} \subseteq W \subsetneq V$.

■ **Table 1** Ratios of solved instances. The Kuratowski ILP dominates all other algorithms.

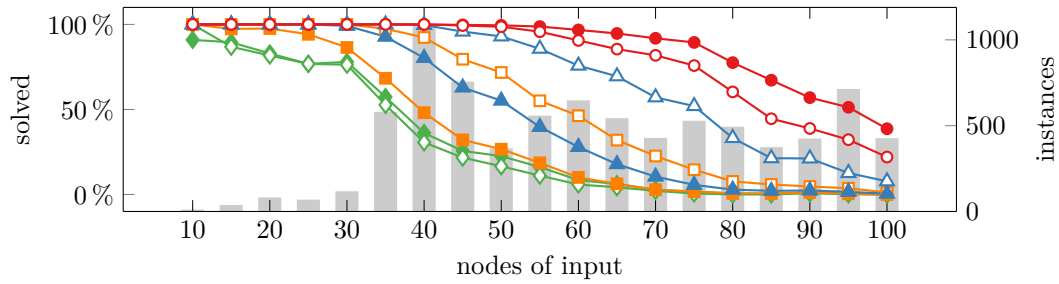
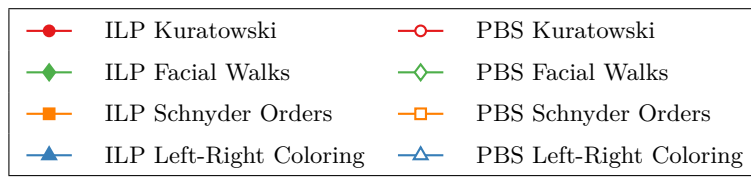
	Rome	North	Expanders	SteinLib
# instances	8249	423	480	105
ILP Kuratowski	85.70 %	73.75 %	22.75 %	9.52 %
ILP Facial Walks	17.82 %	29.78 %	4.31 %	2.85 %
ILP Schnyder Orders	21.69 %	48.22 %	8.96 %	3.80 %
ILP Left-Right Coloring	36.64 %	60.75 %	12.93 %	3.80 %
PBS Kuratowski	77.43 %	69.73 %	10.34 %	9.52 %
PBS Facial Walks	15.21 %	30.02 %	0.68 %	0.95 %
PBS Schnyder Orders	46.24 %	61.93 %	6.89 %	5.71 %
PBS Left-Right Coloring	65.07 %	66.43 %	10.00 %	7.61 %

Instances and configurations. We use the non-planar graphs of the established benchmark sets North [27], Rome [14, Section 3.2], and a subset of the SteinLib [21] all of which include real-world instances. In addition, we generated a set of random regular [29] graphs that are *expander graphs* with high probability. In [11] it was observed that such graphs seem to be especially hard at least for the Kuratowski formulation. For formulations that allow multiple configurations, we determined the most promising one in a preliminary benchmark on a set of 1224 Rome and North graphs, as reported in the previous sections. This *fixed* subset of instances was sampled by partitioning the instances into buckets based on the number of nodes and choosing a fixed number of graphs from each bucket with uniform probability.

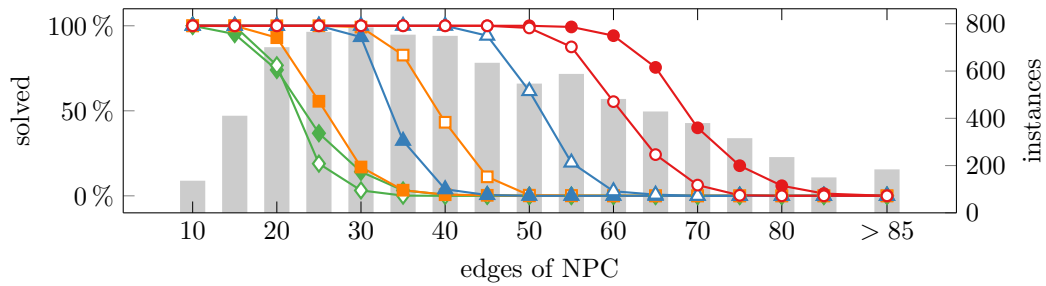
For parameters where we had a non-binary choice (e.g., heap size in ILP separation) we rely mostly on the values identified in [18].

Our algorithms use strong primal heuristics, whose common foundation is a maximal planar subgraph algorithm based on the simpler cactus algorithm by Călinescu et al., with approximation ratio $7/18$, that was identified in [11, denoted by $\mathbf{C+}$] to be among the practically best heuristics.

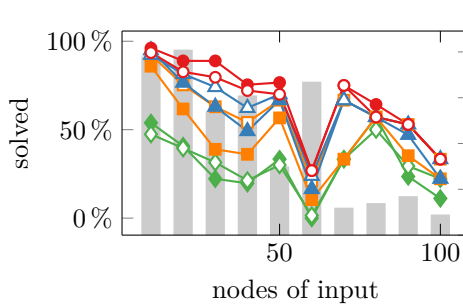
Results. Table 1 summarizes the ratios of solved instances. Evidently, the Kuratowski ILP dominates all implementations. To our surprise, the rather intricate left-right edge coloring model constitutes the most successful one among the new variants. The facial walk model falls behind all other formulations. A similar picture is obtained from a more detailed look at the success rates. In Figure 2a (2b), we show the relative number of solved instances among the Rome graphs over the nodes in the input (resp. number of edges in the non-planar core), clustered to the nearest multiple of five. As expected, the more edges there are in the core, the harder the instance is in practice. This is particularly clear on the Rome graphs and becomes a little distorted on the North graphs, see Figures 2c and 2d, that include some instances where we have to delete very few edges to obtain a (near) triangulation with an (almost) trivial upper bound. Figures 2e and 2f show the number of solved instances over our total runtime. The runtime is represented logarithmically. Again, the Kuratowski ILP is the clear winner and solves more instances than any other variant at any point in time. While the number of solved instances for all algorithms skyrockets in the first milliseconds and only very slowly increases over the course of 20 minutes, we can see that some algorithms gain more than others from an increase in runtime. Surprisingly, the Schnyder orders ILP seems to benefit only on the considerably harder North graphs from increasing the runtime. In most cases, particularly on the Rome and North instances, the PBS variant is stronger



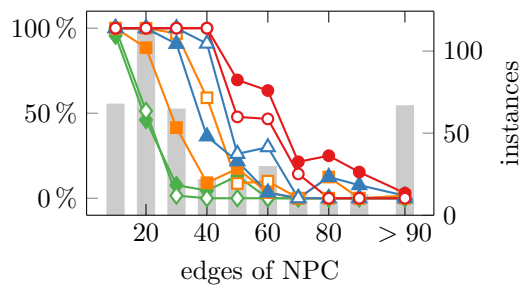
(a) solved Rome graphs by input nodes



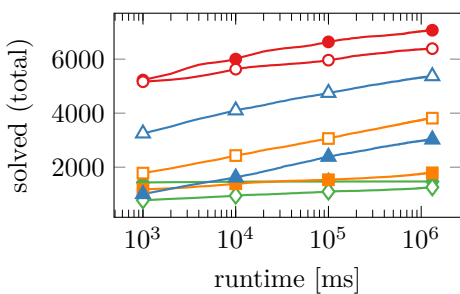
(b) solved Rome graphs by NPC edges



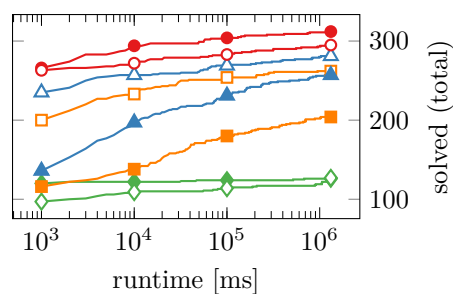
(c) solved North graphs by input nodes



(d) solved North graphs by NPC edges

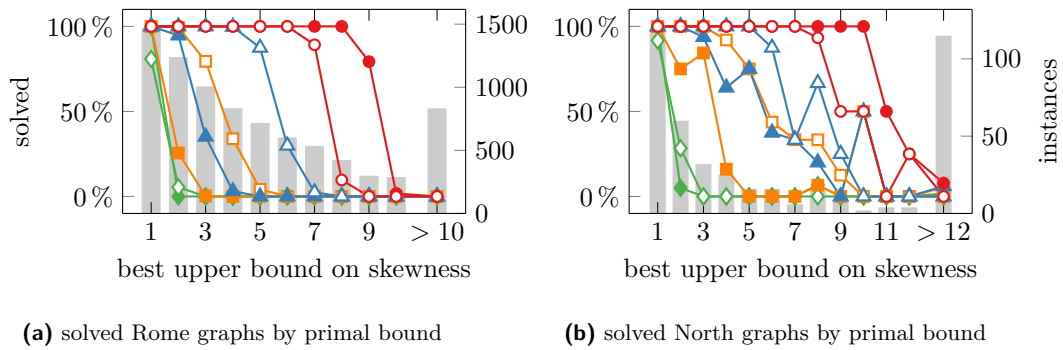


(e) solved Rome graphs by time



(f) solved North graphs by time

■ **Figure 2** Success rate and runtime.



■ **Figure 3** Relation of skewness (bounds) and success rate on Rome graphs. The same legend as in Figure 2 applies.

than its ILP counterpart, with a clear exception for the Kuratowski model. Finally, Figure 3 relates upper bounds on the skewness with the number of solved instances. We can see that there is little success on graphs with a skewness larger than 12, on both the Rome and North set. The same holds, although not as clear cut, for the other instance sets.

7 Findings and Conclusion

The main goal of this paper was to investigate novel ways of approaching the MPS problem, after over two decades of no progress w.r.t. exact models. We succeeded in the sense that we showed that there are indeed viable alternatives. However, we also showed experimentally that a modern implementation of the old Kuratowski formulation remains the strongest option to solve MPS in practice. Although negative, this is an interesting observation.

We should keep in mind that the thereby required efficient separation builds upon years of algorithmic development [3, 12], and it is the only ILP where we currently know how to (heuristically) separate on *fractional* solutions. Equipped with similar tools, i.e., a sensible rounding scheme and a linear time separation routine (a modified left-right planarity test), the left-right edge coloring formulation might yield very competitive performance. This, in fact, may be a reasonable target for future research.

For the genus problem, a facial walk model similar to our MPS formulation is the only known feasible approach. However, we clearly see that it is not favorable for MPS as we have stronger and more direct options at our disposal. The facial walk model optimizes over all possible embeddings (there are exponentially many already for a fixed subgraph) of all planar subgraphs, which might help explain its underwhelming performance. The Schnyder orders model does not perform very well in practice despite its very elegant characterization. This might be due to the fact that in contrast to the left-right edge coloring, we search for three feasible orders on the planar subgraph instead of just one (the partial order corresponding to the Trémaux tree). To solve the Schnyder orders model efficiently, a fast solver for linear ordering problems seems to be required. The Schnyder and left-right edge coloring PBS formulations usually beat their ILP counterparts, indicating that their LP relaxations are rather weak. As expected, the expander graphs constitute a particularly hard class of instances and may be a good starting point for tuning and extending our algorithms.

Finally, the strong performance of the Kuratowski model (in particular the ILP variant) is a clear indication that it deserves more attention in the future. The fact that no additional strong constraint classes have been identified for more than two decades is provocative.

References

- 1 Carlo Batini, Maurizio Talamo, and Roberto Tamassia. Computer aided layout of entity relationship diagrams. *Journal of Systems and Software*, 4(2-3):163–173, 1984. doi:10.1016/0164-1212(84)90006-2.
- 2 Stephan Beyer, Markus Chimani, Ivo Hedtke, and Michal Kotrbčik. A Practical Method for the Minimum Genus of a Graph: Models and Experiments. In Andrew V. Goldberg and Alexander S. Kulikov, editors, *Experimental Algorithms - 15th International Symposium, SEA 2016, St. Petersburg, Russia, June 5-8, 2016, Proceedings*, volume 9685 of *Lecture Notes in Computer Science*, pages 75–88. Springer, 2016. doi:10.1007/978-3-319-38851-9_6.
- 3 John M. Boyer and Wendy J. Myrvold. On the Cutting Edge: Simplified $O(n)$ Planarity by Edge Addition. *Journal of Graph Algorithms and Applications*, 8(3):241–273, 2004. doi:10.7155/jgaa.00091.
- 4 Gruia Călinescu, Cristina Gomes Fernandes, Ulrich Finkler, and Howard Karloff. A Better Approximation Algorithm for Finding Planar Subgraphs. *Journal of Algorithms. Cognition, Informatics and Logic*, 27(2):269–302, 1998. 7th Annual ACM-SIAM Symposium on Discrete Algorithms (Atlanta, GA, 1996). doi:10.1006/jagm.1997.0920.
- 5 Alberto Caprara, Marcus Oswald, Gerhard Reinelt, Robert Schwarz, and Emiliano Traversi. Optimal linear arrangements using betweenness variables. *Mathematical Programming Computation*, 3(3):261–280, 2011. doi:10.1007/s12532-011-0027-7.
- 6 Parinya Chalermsook and Andreas Schmid. Finding triangles for maximum planar subgraphs. In Sheung-Hung Poon, Md. Saidur Rahman, and Hsu-Chun Yen, editors, *WALCOM: Algorithms and Computation, 11th International Conference and Workshops, WALCOM 2017, Hsinchu, Taiwan, March 29-31, 2017, Proceedings.*, volume 10167 of *Lecture Notes in Computer Science*, pages 373–384. Springer, 2017. doi:10.1007/978-3-319-53925-6_29.
- 7 Markus Chimani and Carsten Gutwenger. Non-planar core reduction of graphs. *Discrete Mathematics*, 309(7):1838–1855, 2009. doi:10.1016/j.disc.2007.12.078.
- 8 Markus Chimani and Carsten Gutwenger. Advances in the planarization method: effective mutiple edge insertions. *Journal of Graph Algorithms and Applications*, 16(3):729–757, 2012. doi:10.7155/jgaa.00264.
- 9 Markus Chimani, Carsten Gutwenger, Michael Jünger, Gunnar W. Klau, Karsten Klein, and Petra Mutzel. The Open Graph Drawing Framework (OGDF). In Roberto Tamassia, editor, *Handbook on Graph Drawing and Visualization*, pages 543–569. Chapman and Hall/CRC, 2013. URL: crcpress.com/Handbook-of-Graph-Drawing-and-Visualization/Tamassia/9781584884125.
- 10 Markus Chimani and Petr Hlinený. A tighter insertion-based approximation of the crossing number. *Journal of Combinatorial Optimization*, 33(4):1183–1225, 2017. doi:10.1007/s10878-016-0030-z.
- 11 Markus Chimani, Karsten Klein, and Tilo Wiedera. A Note on the Practicality of Maximal Planar Subgraph Algorithms. In Yifan Hu and Martin Nöllenburg, editors, *Proceedings of the 24th International Symposium on Graph Drawing and Network Visualization (GD 2016)*, volume abs/1609.02443. CoRR, 2016.
- 12 Markus Chimani, Petra Mutzel, and Jens M. Schmidt. Efficient extraction of multiple kuratowski subdivisions. In Seok-Hee Hong, Takao Nishizeki, and Wu Quan, editors, *Graph Drawing, 15th International Symposium, GD 2007, Sydney, Australia, September 24-26, 2007. Revised Papers*, volume 4875 of *Lecture Notes in Computer Science*, pages 159–170. Springer, 2007. doi:10.1007/978-3-540-77537-9_17.

- 13 H. de Fraysseix and P. Rosenstiehl. A characterization of planar graphs by Trémaux orders. *Combinatorica. An International Journal of the János Bolyai Mathematical Society*, 5(2):127–135, 1985. doi:10.1007/BF02579375.
- 14 Giuseppe Di Battista, Ashim Garg, Giuseppe Liotta, Roberto Tamassia, Emanuele Tassinari, and Francesco Vargiu. An experimental comparison of four graph drawing algorithms. *Computational Geometry. Theory and Applications*, 7(5-6):303–325, 1997. 11th ACM Symposium on Computational Geometry (Vancouver, BC, 1995). doi:10.1016/S0925-7721(96)00005-3.
- 15 Ben Dushnik and E. W. Miller. Partially ordered sets. *American Journal of Mathematics*, 63:600–610, 1941. doi:10.2307/2371374.
- 16 Michael R. Garey and David S. Johnson. *Computers and intractability. A guide to the theory of NP-completeness*. W. H. Freeman and Co., San Francisco, Calif., 1979.
- 17 Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Thomas Schneider. Potassco: The potsdam answer set solving collection. *AI Communications*, 24(2):107–124, 2011. doi:10.3233/AIC-2011-0491.
- 18 Ivo Hedtke. *Minimum Genus and Maximum Planar Subgraph: Exact Algorithms and General Limits of Approximation Algorithms*. PhD thesis, Osnabrück University, 2017. URL: repositorium.uos.de/handle/urn:nbn:de:gbv:700-2017082416212.
- 19 Jan M. Hochstein and Karsten Weihe. Maximum s - t -flow with k crossings in $O(kn \log n)$ time. In Nikhil Bansal, Kirk Pruhs, and Clifford Stein, editors, *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007*, pages 843–847. SIAM, 2007. URL: <http://dl.acm.org/citation.cfm?id=1283383.1283473>.
- 20 Michael Jünger and Petra Mutzel. Maximum Planar Subgraphs and Nice Embeddings: Practical Layout Tools. *Algorithmica. An International Journal in Computer Science*, 16(1):33–59, 1996. doi:10.1007/s004539900036.
- 21 T. Koch, A. Martin, and S. Voß. SteinLib: An updated library on steiner tree problems in graphs. Technical Report ZIB-Report 00-37, Konrad-Zuse-Zentrum für Informationstechnik Berlin, Takustr. 7, Berlin, 2000. URL: <http://elib.zib.de/steinlib>.
- 22 Kazimierz Kuratowski. Sur le problème des courbes gauches en topologie. *Fundamenta Mathematicae*, 15:271–283, 1930.
- 23 Charles H. C. Little and G. Sanjith. Another characterisation of planar graphs. *Electronic Journal of Combinatorics*, 17(1):Note 15, 7, 2010.
- 24 P. C. Liu and R. C. Geldmacher. On the deletion of nonplanar edges of a graph. In *Proceedings of the Tenth Southeastern Conference on Combinatorics, Graph Theory and Computing (Florida Atlantic Univ., Boca Raton, Fla., 1979)*, Congress. Numer., XXIII–XXIV, pages 727–738. Utilitas Math., Winnipeg, Man., 1979.
- 25 Stephen J. Maher, Tobias Fischer, Tristan Gally, Gerald Gamrath, Ambros Gleixner, Robert Lion Gottwald, Gregor Hendel, Thorsten Koch, Marco E. Lübbecke, Matthias Miltenberger, Benjamin Müller, Marc E. Pfetsch, Christian Puchert, Daniel Rehfeldt, Sebastian Schenker, Robert Schwarz, Felipe Serrano, Yuji Shinano, Dieter Weninger, Jonas T. Witt, and Jakob Witzig. The SCIP Optimization Suite 4.0. Technical Report 17-12, ZIB, Takustr. 7, 14195 Berlin, 2017.
- 26 Petra Mutzel. *The maximum planar subgraph problem*. PhD thesis, Köln University, 1994.
- 27 Stephen C. North. 5114 directed graphs, 1995. Manuscript.
- 28 Walter Schnyder. Planar Graphs and Poset Dimension. *Order. A Journal on the Theory of Ordered Sets and its Applications*, 5(4):323–343, 1989. doi:10.1007/BF00353652.
- 29 A. Steger and N. C. Wormald. Generating random regular graphs quickly. *Combinatorics, Probability and Computing*, 8(4):377–396, 1999. Random graphs and combinatorial structures (Oberwolfach, 1997). doi:10.1017/S0963548399003867.

- 30 Edward Szpilrajn. Sur l'extension de l'ordre partiel. *Fundamenta Mathematicae*, 16(1):386–389, 1930. URL: <http://eudml.org/doc/212499>.
- 31 Carsten Thomassen. Planarity and Duality of Finite and Infinite Graphs. *Journal of Combinatorial Theory. Series B*, 29(2):244–271, 1980. doi:10.1016/0095-8956(80)90083-0.

A Linear-Time Algorithm for Finding Induced Planar Subgraphs

Shixun Huang

RMIT University, Melbourne, Australia

Zhifeng Bao

RMIT University, Melbourne, Australia

J. Shane Culpepper

RMIT University, Melbourne, Australia

Ping Zhang

Wuhan University, Wuhan, China

Bang Zhang

DATA61 — CSIRO, Australia

Abstract

In this paper we study the problem of efficiently and effectively extracting induced planar subgraphs. Edwards and Farr proposed an algorithm with $O(mn)$ time complexity to find an induced planar subgraph of at least $3n/(d+1)$ vertices in a graph of maximum degree d . They also proposed an alternative algorithm with $O(mn)$ time complexity to find an induced planar subgraph graph of at least $3n/(\bar{d}+1)$ vertices, where \bar{d} is the average degree of the graph. These two methods appear to be best known when d and \bar{d} are small. Unfortunately, they sacrifice accuracy for lower time complexity by using indirect indicators of planarity. A limitation of those approaches is that the algorithms do not implicitly test for planarity, and the additional costs of this test can be significant in large graphs. In contrast, we propose a linear-time algorithm that finds an induced planar subgraph of $n - \nu$ vertices in a graph of n vertices, where ν denotes the total number of vertices shared by the detected Kuratowski subdivisions. An added benefit of our approach is that we are able to detect when a graph is planar, and terminate the reduction. The resulting planar subgraphs also do not have any rigid constraints on the maximum degree of the induced subgraph. The experiment results show that our method achieves better performance than current methods on graphs with small skewness.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms

Keywords and phrases induced planar subgraphs, experimental analysis

Digital Object Identifier 10.4230/LIPIcs.SEA.2018.23

Funding This work was supported by ARC (DP170102726, DP170102231, DP180102050), NSF of China (61728204, 91646204). Zhifeng Bao is a recipient of Google Faculty Award.

1 Introduction

A graph is *planar* if it admits a *planar drawing* which means that the graph can be drawn on the plane such that its edges only intersect at their endpoints. The goal of the graph planarization problem is to find a planar subgraph by removing edges or vertices from an input graph. It can be applied in many areas, such as facility layout design [8], circuit design [18], graph drawing [15], and automated graphical display systems [28]. One popular reformulation of the graph planarization problem, called the Maximum Induced Planar Subgraph (MIPS)



© Shixun Huang, Zhifeng Bao, J. Shane Culpepper, Ping Zhang, and Bang Zhang; licensed under Creative Commons License CC-BY

17th International Symposium on Experimental Algorithms (SEA 2018).

Editor: Gianlorenzo D'Angelo; Article No. 23; pp. 23:1–23:15

Leibniz International Proceedings in Informatics



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

problem, aims to find the largest number of vertices which induce a planar subgraph. This problem is known to be NP-hard, and also surprisingly hard to approximate [19, 23, 26]. The MIPS problem can also be used to compute the coefficient of fragmentability of a class of graphs, which is the proportion of vertices necessary to produce subgraphs of a bounded size [11].

Related Work. In this paper, we study the MIPS problem and we assume that the reader is familiar with basic graph theory (see for example [13, 30]). No graphs being considered contain edge loops, n denotes the number of vertices, m denotes the number of edges, d denotes the maximum degree, and \bar{d} denotes the average degree in a graph.

Halldórsson and Lau [14] proposed a linear-time algorithm (denoted as HL) for the MIPS problem with a performance ratio of $1/\lceil(d+1)/3\rceil$ in a graph G . They presented several practical algorithms for partitioning graphs into a fixed number of vertex-disjoint subgraphs with degree constraints. In order to solve the problem, they capitalize on the Lovász [25] Theorem: Let a_1, a_2, \dots, a_k be non-negative integers such that $\sum_{i=1}^k (a_i + 1) - 1 = d$. Then G can be partitioned into k induced subgraphs G_1, G_2, \dots, G_k such that the maximum degree of G_i is not greater than a_i . With this theorem, a graph can be partitioned into at most $\lceil(d+1)/3\rceil$ induced subgraphs of degree at most 2, and the largest subgraph is the planarized result. The approach of Halldórsson and Lau can induce such a partition in linear time. However, the maximum degree of the planarization result is restricted to be at most 2.

Edwards and Farr [10] proposed an algorithm (denoted as Vertex Addition) to find an induced planar subgraph of at least $3n/(d+1)$ vertices in $O(mn)$ time, which has a performance ratio of at least $3/(d+1)$. Compared to the algorithm of Halldórsson and Lau, the performance ratio is improved when $d \not\equiv 2 \pmod{3}$. The induced planar subgraphs found by this algorithm is also not constrained to have maximum degree of 2. The algorithm works as follows. Suppose that P is an initially empty set and $R = V(G) \setminus P$, this algorithm works by adding vertices from R one by one into P while maintaining the planarity of $\langle P \rangle$. In some instances, a vertex from R is swapped with one from P . The restrictions on the swapping operations are stricter than that on maintaining planarity. By doing this, some properties in the graph are maintained, which allows the performance of the algorithm to be analyzed. For further information, please see [11, 10]. This leads to a fact that sometimes it still swaps some vertices even if planarity could be maintained when all vertices involved in the swapping operations are included in the planarization result.

Edwards and Farr [11] propose another algorithm (denoted as Vertex Removal) with time complexity $O(mn)$ for the MIPS problem in a graph of average degree \bar{d} , which achieves a performance ratio of at least $3/(\bar{d}+1)$ when $\bar{d} \geq 4$ or a graph is connected and $\bar{d} \geq 2$. This algorithm begins by removing any isolated vertex, any vertex of degree 1, and any vertex of degree 2. For a vertex of degree 2, if its neighbours are not adjacent, they are joined by an extra edge. Then a *reduced graph* is obtained by repeating the operations above until no further changes are possible. Then, it proceeds to remove the vertex of the highest degree in the reduced graph iteratively. In order to reduce complexity and improve efficiency, this algorithm avoids the planarity test in each iteration. Instead, a loose upper bound of the number of vertices to be removed is computed, which can result in the algorithm continuing to remove vertices until the upper bound is reached, regardless of whether the current result is already planar or not. Morgan and Farr [27] later proposed a modified algorithm (denoted as Vertex Subset Removal) which instead iteratively removes a vertex v with the largest number of neighbors with degree less than the degree of v in the reduced graph. There is no known investigation of the impact of the different vertex removal strategies on the planarization results [24].

There is a simple example made by Morgan and Farr [27], which roughly summarizes a limitation shared by all methods mentioned above — all previous methods fail to leave K_4 minors in the induced planar subgraph even though K_4 is already planar. For a K_5 graph, they can only find an induced planar subgraph of size at most degree 3.

Preliminaries. We now review key definitions before introducing our contributions. According to Kuratowski [21], a graph is planar if and only if it does not contain a *Kuratowski subdivision* which can be any subdivision of K_5 (a complete graph of size 5) or of $K_{3,3}$ (a complete bipartite graph of size 6). A *subdivision* of a graph G is a graph resulting from the subdivision of edges in G . The subdivision of an edge e with endpoints (u, v) yields a graph containing one new vertex w , with an edge set replacing e by two new edges, (u, w) and (w, v) . The *skewness* of a graph is the minimum number of edges whose removal results in a planar graph [6].

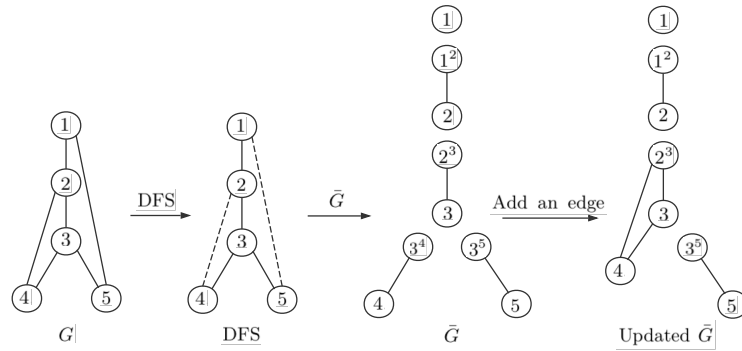
A graph is a *nearly planar graph* if it is a k -graph (contains most k edge crossings) or a k -*skewness graph* when k is small. Several previous studies have studied graphs with similar properties in the context of straight-line drawing [17], visualization [12, 9, 7], and edge intersection [5]. Applications may also require non-planar graphs to be drawn on a plane even if edge crossings cannot be avoided [1, 8]. So it is naturally desirable to draw graphs as close to planar as possible. We therefore focus on these cases in our experiments.

Contributions and Outline. We present an algorithm including planarity test to solve the MIPS problem that does not remove any additional vertices once the graph becomes planar. An additional benefit of our approach is that the maximum degree of the planarization result is not constrained, which overcomes some of the limitations of previous work in this area. The algorithm runs in $O(n + m + E(S))$ time, with S being the set of detected Kuratowski subdivisions, and $E(S)$ being the sum of the number of edges in the subdivisions. The time complexity is linear w.r.t. $E(S)$, and graph size. The induced planar subgraph produced by our algorithm is of size $n - \nu$, with ν being the total number of vertices shared by the Kuratowski subdivisions detected. We conduct several experiments to show that our method outperforms all other methods for graphs with small skewness.

In Section 2, we first introduce a planarity test algorithm which is the basis of our approach. Next, we describe our planarization algorithm and proofs of correctness. In Section 3, we conduct intensive experiments on real-world graphs. Finally, we conclude this paper in Section 4.

2 Motivation and Approach

Planarity Testing. Our work is based on one of the most efficient planarity test algorithms, originally presented by Boyer and Myrvold [3]. For more detailed information, please refer to the original work [3]. The algorithm, denoted as DETECT, works by checking if a graph produces a *planar drawing*. DETECT begins by creating a depth first search tree (DFS tree) of the graph. Each vertex is assigned to a depth first index (DFI), and edges are divided into tree edges forming the DFS tree and backedges (the remaining edges). In this paper, let v be the vertex currently being processed and \tilde{G} be the plane for embedding the graph. Initially, the DFS tree is embedded in \tilde{G} . DETECT processes vertices in descending DFI order. In each iteration of v , DETECT attempts to embed each backedge (u, v) , where u has a larger DFI than v , while maintaining planarity.



■ **Figure 1** An example illustrating the DETECT algorithm.

Initially, each tree edge (z, v) is represented as an independent biconnected component (z, v^z) formed by the vertex with a largest DFI and a *virtual vertex*. The vertex z is a DFS child of v , and we denote this virtual vertex as v^z to distinguish it from other copies of v . We use v' to denote a virtual vertex whose child is not specified. We say that the vertex v^z is the *root* of this biconnected component. Biconnected components are merged to form a larger subgraph as backedges are embedded.

Each iteration involves two processes: a WALKUP and a WALKDOWN. A WALKUP identifies relevant biconnected components for a backedge embedding, and classifies vertices as follows. A vertex w is *pertinent* if there is a backedge (w, v) to be embedded, or it has a child biconnected component in \tilde{G} which contains a pertinent vertex. A backedge (w, v) is *pertinent* if w is pertinent and w is marked with an `EdgeFlag`. A biconnected component is *pertinent* if it contains a pertinent vertex. A vertex w is *external* if there is a backedge (w, u) to be embedded later, where u has a smaller DFI than v , or it has a child biconnected component in \tilde{G} which contains an external vertex. Each vertex is equipped with a `PertinentRoots` list that stores the roots of its pertinent child components. For every backedge (w, u) , WALKUP traverses from w to u along the paths on the external face of the biconnected components.

A WALKDOWN then embeds pertinent backedges and merges relevant biconnected components traversed by the WALKUP. The process is initiated with two traversals for each biconnected child component rooted by a virtual point v' : one in the clockwise direction along the external faces of the biconnected child component, and a second one in the opposite direction. When WALKDOWN reaches a pertinent vertex u with an `EdgeFlag`, the relevant components are merged, and the backedge (u, v) is embedded. The process continues until reaching an external but non-pertinent vertex (denoted as *stopping vertex*), or v' is found again. This is the *halting condition* for the algorithm, and is the only possible indicator of non-embeddability of backedges. If a pertinent backedge cannot be embedded, the graph is not planar, as DETECT has identified a Kuratowski subdivision.

A Linear Time Solution for The MIPS Problem. DETECT terminates when a pertinent backedge exists that is not embedded due to a stopping vertex s – meaning the graph is non-planar. The reason is that if the algorithm embedded an edge after passing s , then s cannot remain on the outer face of the graph. The embedding of a backedge (s, u) in a later iteration would result in intersecting edges, which cannot admit a planar drawing.

Let s be a stopping vertex, and the *influenced region* of s be the collection of paths which can be visited by a WALKDOWN only after it has visited s . The vertex v being processed is an *obstruction vertex* if there exists at least one pertinent unembedded backedge. We observe

Algorithm 1: PLANARIZATIONBYREGIONSKIP(G).

Input : A graph G
Output : An induced planar subgraph P

- 1 Construct a DFS tree of G ;
- 2 Initialize the embedding structure \bar{G} ;
- 3 Initialize **ObstructionsList** ;
- 4 **for** each vertex v in descending DFI order **do**
- 5 **foreach** backedge (w, v) of G where $w > v$ **do**
- 6 **if** w is not an obstruction **then**
- 7 $\text{WALKUP}(\bar{G}, v, w)$;
- 8 **foreach** DFS child c of v in G **do**
- 9 $\text{WALKDOWNWITHSKIPS}(\bar{G}, v^c)$;
- 10 **foreach** back edge (w, v) of G where $w > v$ **do**
- 11 **if** (w, v) not in \bar{G} **then**
- 12 **ObstructionsList**[$v.index$] $\leftarrow v.DFI$;
- 13 **break**;
- 14 graph $P \leftarrow \text{REMOVEOBSTRUCTIONS}(\text{ObstructionsList}, G)$;
- 15 **return** P ;

that a stopping vertex s only influences the embedding of a backedge (w, v) when w is in the influenced region of s . We therefore propose the algorithm PLANARIZATIONBYREGIONSKIP which embeds all possible pertinent backedges in each iteration by skipping influenced regions of stopping vertices encountered during the WALKDOWN. When the embedding process is completed, all obstruction vertices are removed from the input graph, which produces an induced planar subgraph. This observation produces an algorithm which can test for planarity and produce a solution for the MIPS problem simultaneously.

Solution Overview. Algorithm 1 presents our solution for finding an induced planar subgraph. It begins by building a DFS tree, and initializing the embedding structure \bar{G} (line 1 to 2). Then we use an **ObstructionsList** to store the indexes of obstruction vertices in an adjacency list. Each element is initialized to -1 (line 3). Next, the embedding loop is initiated (line 4 to 12). Obstruction vertices identified in each iteration are excluded from the WALKUP process (line 7). Details of WALKUP are described in previous work [3].

In the WALKDOWNWITHSKIPS, traversals for each biconnected child component rooted by the virtual point v^c are initiated. This process embeds all of the pertinent backedges which are not influenced by stopping vertices with skipping operations over the influenced regions (line 9). Then, v is added into the **ObstructionsList** if there exists an unembedded pertinent backedge. When the main loop finishes, all obstruction vertices are removed from the graph.

The WalkdownWithSkips algorithm. As previously discussed, in order to embed backedges that are not influenced by stopping vertices, we need to perform skipping operations. The process WALKDOWNWITHSKIPS terminates when it reaches v' or a stopping vertex on the component whose root is v' . We denote such a component as a *root component*. If the stopping vertex encountered is not on a root component, a skipping operation needs to be performed. When a traversal descends from vertex r to root vertex r' of a non-root component, it needs to choose a direction to proceed. Boyer and Myrvold [3] proposed **short circuit edges** which enable r' to be directly connected to neighbors such that they are either pertinent or a

Algorithm 2: WALKDOWNWITHSKIPS(\bar{G}, v^c).

Input : The embedding structure \bar{G} and a virtual vertex v^c .

```

1 foreach traversal from  $v^c$  do
2    $w \leftarrow$  The successor along the external face;
3   while  $w$  is not  $v^c$  do
4     if  $w$  has an EdgeFlag then
5       Merge involved components;
6       Embed the backedge  $(w, v^c)$  and clear  $w$ 's EdgeFlag;
7        $w \leftarrow$  The successor along the external faces;
8     if  $w$ .PertinentRoots is not empty then
9        $r \leftarrow w$ .PertinentRoots[0];
10      Traverse down to the component rooted by  $r$ ;
11       $w \leftarrow$  The successor along the external faces;
12     if  $w$  is a stopping vertex then
13       if  $w$  is on the root component then
14         Embed the Short Circuit Edge  $(w, v^c)$ ;
15         break;
16       else
17          $x \leftarrow$  Another neighbor of  $r$ , the root of the current component;
18         if  $x$  is a stopping vertex then
19           Skip the components rooted by  $r$ ;
20         else  $w \leftarrow x$ ;
21     else  $w \leftarrow$  The successor along the external faces;
  
```

stopping vertex. Each **short circuit edge** is embedded in a previous iteration p between p' and the stopping vertex. This forms a new face such that interceding inactive vertices are removed from the external face. For more detailed information, please refer to Boyer and Myrvold [3]. For our purposes, when the WALKDOWNWITHSKIPS encounters a stopping vertex, it checks if another neighbor of r' is not a stopping vertex. If so, it skips to this neighbor. Otherwise, it skips the components rooted by r' which is then deleted from the *PertinentRoots* of r , and returns to the parent component. The algorithm terminates on the stopping vertex on the root component since there does not exist a parent component for the process to ascend to.

Algorithm 2 describes the rationale of the WALKDOWNWITHSKIPS. The algorithm begins a single traversal in a clockwise or counterclockwise direction (line 2). Let w be the next successor along the external face. If w has an **EdgeFlag**, the backedge (w, v^c) is embedded after the relevant components are merged (line 4 to 7). Then the traversal proceeds to the successor. When it encounters a pertinent vertex whose *PertinentRoots* list is not empty, it descends to the component rooted by the first element r in the list, and visits one of its neighbors (line 8 to 11). If w is a stopping vertex in the root component, a **short circuit edge** is embedded, and the traversal stops, after which another traversal is initiated from v^c in the opposite direction (line 13 to 15). Otherwise, the traversal performs a skip based on whether another neighbour of r is a stopping vertex (line 17 to 20).

Example of the Embedding Process. It is instructive to see an example of the embedding process on the pertinent subgraph in an iteration of c . The WALKUP process is invoked for each vertex with an **EdgeFlags**. Two parallel traversals are started from each vertex

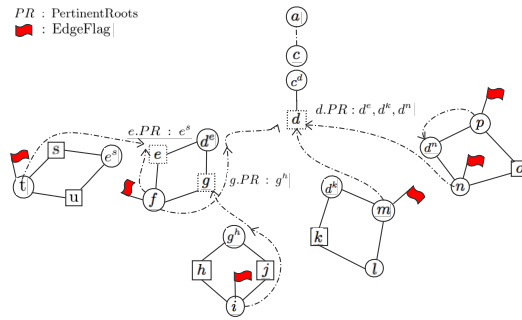


Figure 2 An example of the Walkup.

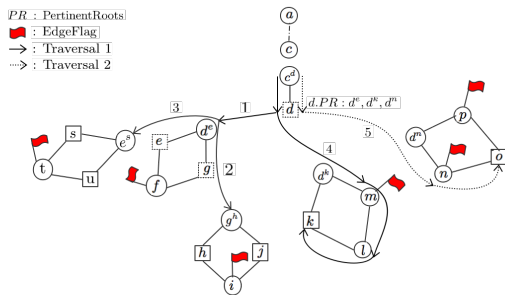


Figure 3 The WalkdownWithSkips in the iteration of c .

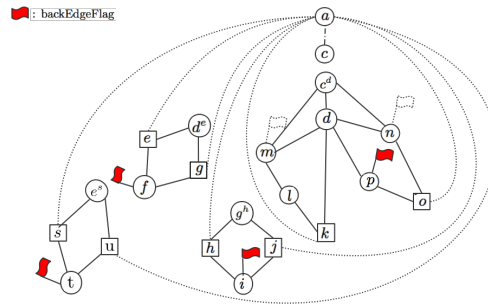


Figure 4 The status after the embedding.

and stops when either of them reaches the root of the current biconnected component. Afterwards, WALKUP starts another two parallel traversals on the parent components. The process terminates when a traversal reaches c or a vertex that has been visited before is encountered.

Figure 2 shows the process of the WALKUP of an example set of biconnected components (diamonds), external and pertinent vertices (dashed squares), stopping vertices (solid squares) and pertinent vertices with **EdgeFlags**. Only the traversals that reach root vertices first are shown. WALKUP begins at f . When it reaches d^e , d^e is then added to the **PertinentRoots** of d , and it starts traversals at d until reaching c . Then the WALKUP traversals of i are initiated. The vertex g^h is added to the **PertinentRoots** of g after being visited. When it reaches g , the traversal terminates since g has been visited before. The main purpose of WALKUP is to determine which components are involved in the embedding. Hence the traversals initiated from i do not have to continue. This process is repeated until all vertices with **EdgeFlags** have all done a WALKUP.

The main purpose of the WALKDOWNWITHSKIPS is to embed as many pertinent backedges as possible by skipping the influenced regions of the stopping vertices, and identify if the vertex being processed is an obstruction. Figure 3 describes WALKDOWNWITHSKIPS on the same example set of biconnected components as WALKUP. A traversal starts in one direction of the child component of c . Then it descends to the component rooted by d^e which is the first element in the **PertinentRoots** of d . Which direction to go from d^e depends on the types of the neighbors: a neighbor can be pertinent but not external, pertinent and external, or a stopping vertex (external but non-pertinent). The direction towards the neighbor of the first type is preferred, and the direction towards a stopping vertex will be chosen if no neighbors of the first two types exist. Since both e and g are of the same type, a neighbor g is randomly selected.

Afterwards, the algorithm descends to the component rooted by g^h which is deleted from the `PertinentRoots` of g , and then returns to g because it cannot reach i without passing a stopping vertex. Next a skip to another neighbor e of d^e is initiated, and the process is repeated. Since external and pertinent vertices are stopping vertices once their `PertinentRoots` lists become empty, the traversal cannot reach f . So the components rooted by d^e are skipped and deleted from `PertinentRoots` of d , and processing returns to d . Then the algorithm descends to the component rooted by the next element d^k in the `PertinentRoots` of d . The backedge (m, c^d) is embedded and the relevant components rooted by c^d and d^k respectively are merged. The traversal terminates at k after embedding the `short circuit edge` (k, c^d) since k is on the root component after merging operations. Another traversal begins from c^d , and the process is repeated until a termination condition is reached.

Figure 4 shows the embedding for this example. The dashed edges refer to backedges which will be embedded by the algorithm. The vertices which still have `EdgeFlags` correspond to unembedded backedges in the last iteration. As we can see, the component rooted by c^d is larger after merging, and backedges (m, c^d) and (n, c^d) have been embedded. The embeddings of any other backedges in this figure would result in an intersection with the dashed edges. Thus, they cannot be embedded. Since there exist unembedded backedges, the vertex c is added to the `ObstructionsList`.

Removing Obstruction Vertices. After the main loop of the embedding process, the obstruction vertices are collected, which need to be removed from the graph to induce the planar subgraph (line 13 in Algorithm 1). The input graph is represented as an adjacency list, which is a collection of vertex lists. The first vertex in each vertex list is adjacent to the rest of the vertices. We denote a vertex list E as a list of e_1 if the vertex e_1 is the first element in E .

As discussed in Section 2, each index of the `ObstructionsList` refers to the index of a vertex in the adjacency list, and its content is initialized to -1 . Since we assume that all index values are non-negative, after the main loop, we can identify which vertex is an obstruction based on whether the content of the corresponding vertex is non-negative. For each vertex list of e_1 where `ObstructionsList`[e_1] ≥ 0 , we just remove them directly from the adjacency list. For each vertex list of e_1 where `ObstructionsList`[e_1] ≤ 0 , we process each of the rest elements e_i in the vertex list of e_1 by checking `ObstructionsList`[e_i]. If `ObstructionsList`[e_i] ≤ 0 , we leave this element and process the next one. Otherwise, this element is deleted from this vertex list and we continue processing. After all vertex lists have been processed, the adjacency list is an induced graph where each obstruction vertex o (`ObstructionsList`[o] ≥ 0) has been removed. The overall cost includes the $O(n)$ vertex lists, and the total number of elements in vertex lists are $O(m)$. Since each element is processed in $O(1)$, the total time complexity is $O(n + m)$.

Proof of Correctness. In this section, we prove that the induced subgraph found by our algorithm is planar and has a linear time complexity.

► **Lemma 1.** *Given a graph G , the main embedding loop finds a planar subgraph of G .*

Proof. Boyer and Myrvold [3] have proved that, in the iteration of v , Kuratowski subdivisions will occur if and only if the `WALKDOWN` passes stopping vertices to embed backedges. Since the embedding process works by skipping the influenced regions of stopping vertices in each iteration, any Kuratowski subdivision cannot exist in the graph. Kuratowski [21] proved that a graph is not planar if and only if it contains a Kuratowski subdivision. The embedding loop preserves planarity since no Kuratowski subdivisions exist in the graph. ◀

► **Theorem 2.** *Given a graph G , removal of obstruction vertices leads to an induced planar subgraph of G .*

Proof. Although the graph is already planar after the embedding process, it is not an induced graph since we only remove certain edges. In order to have an induced planar subgraph, we need to remove one of two endpoints of each removed edge. Since all removed edges were connected to obstruction vertices, the removal of such vertices leads to an induced planar subgraph. ◀

► **Theorem 3.** *Given a graph G with n vertices and m edges, our algorithm is bounded by $O(n + m + E(S))$ and therefore linear.*

Proof. The construction of the DFS tree can be accomplished in linear time with a well-known algorithm [29]. The initialization of the embedding structure \bar{G} and the `ObstructionsList` is also a linear time process. During the main backedge embedding loop, embedding edges runs in linear time since the cost of embedding each edge is $O(1)$. If the input graph is planar, the cost of `WALKUP` is bounded by the faces formed by the embedded edges. The faces formed by the embedded backedges and short circuit edges bound the cost of the `WALKDOWNWITHSKIPS`. Thus the cost of `WALKUP` and `WALKDOWN` is linear since the number of faces is at most twice the number of edges in the graph. So, each edge can only be traversed at most two times throughout the entire embedding loop. However, if the input graph is not planar, the cost of `WALKUP` and `WALKDOWN` cannot be bounded by the faces formed by backedges since some of the backedges are unembedded in order to preserve planarity. This means that some edges along the external faces of the graph are traversed multiple times before new external faces are formed, which then includes these edges in the internal faces. Such an edge is traversed at most k times where k denotes the number of Kuratowski Subdivisions which contain this edge. If S is a collection of all Kuratowski Subdivisions detected in the entire embedding process, and $E(S)$ denotes the size of subdivisions in S , then our algorithm runs in $O(n + m + E(S))$ time, which is output sensitive, and linear w.r.t. $E(S)$ and the graph size. ◀

3 Experimental Evaluation

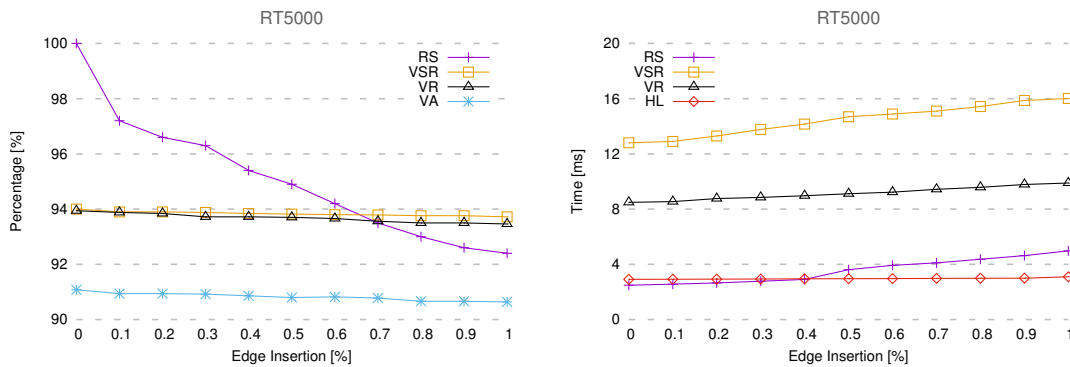
In this section, we compare our `PLANARIZATIONBYREGIONSKIP` algorithm (RS) with the baselines described in Section 1: HL [14], Vertex Addition (VA) [10], Vertex Removal (VR) [11], and Vertex Subset Removal (VSR) [27]. All baselines were implemented by Morgan and Farr [27], and are publicly available. Morgan and Farr [27] also proposed additional algorithms for the MIPS problem. We have selected the subset of algorithms listed above for the following reasons: 1. VR is best known for average degree d , and it achieved second best accuracy in the original work [27]. 2. VSR, as a modified algorithm of VR, has the same approximate ratio as VR, and achieved the best accuracy previously. 3. VA is best known for maximum degree \bar{d} . 4. HL has linear-time complexity and was the most efficient. Note that we do not include the EPS algorithm as it is a post-processing enhancement [27]. This operation can be applied to the planarization result of any of the algorithms explored in this work to improve the approximation ratio further.

All programs are implemented in C, compiled using GCC 4.2.1, and are available online¹. All experiments are performed on a machine with two Intel Core i5 (2.6 GHz) and 8 GB RAM.

¹ <https://github.com/rmitbgroup/GraphPlanarization>

■ **Table 1** Basic properties of the test collections.

Dataset	n	m	Description
<i>RT</i>	1,379,917	1,921,660	The planar road network of Texas.
<i>RD</i>	1,088,092	1,541,898	The planar road network of Pennsylvania.
<i>IN</i>	26,475	53,581	Non-planar network of autonomous systems in the CAIDA project.
<i>PG</i>	10,680	24,316	A non-planar social network of the Pretty Good Privacy algorithm.
<i>UG</i>	4,941	6,594	The non-planar power grid network of the Western States in US.
<i>MP</i>	212	244	A non-planar network of protein-protein interactions from PDZBase.

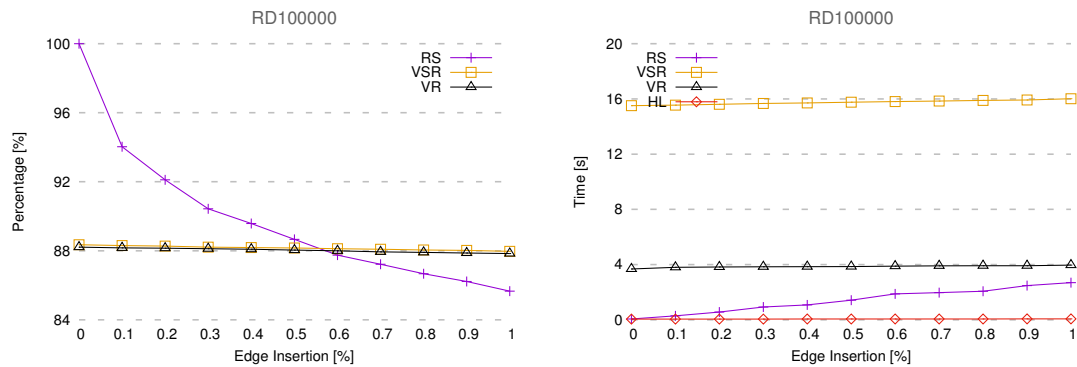


■ **Figure 5** Experiments on RT5000 with edge insertion. The left figure shows the percentage achieved by different algorithms (HL achieves 35% on average, and is not shown on the graph). The right figure shows the running time of different algorithms (VA requires 24,888 ms on average, and is not shown).

In this section, we use the term **Percentage** to describe how many vertices from the input graph are retained in the planarization result. We conduct experiments on real-world graphs collected from KONECT [20] and SNAP [22]. Table 1 summarizes the basic properties of the datasets. For detailed information about the chosen datasets, please refer to KONECT².

Experiments on graphs with small skewness. In this section, we conduct experiments on two datasets: *RT5000* which contains 5,000 vertices from *RT*, and *RD100000* which contains 100,000 vertices from *RD*. We construct the graphs of increased skewness by randomly inserting edges between existing vertices. We insert edges up to 0.1% of the input graph size. Figure 5 shows the experimental results on *RT5000*. As we can see, even if the graph is already planar (no edge insertions), only RS achieves a percentage of 100%. All other methods remove vertices based on the requirements of their corresponding indirect indicators of planarity. With incremental edge insertions, the performance of RS can vary significantly since each inserted edge may introduce multiple Kuratowski subdivisions. This behavior also indicates that the performance of RS is related to the direct indicator of planarity. On the other hand, the performance of other methods do not change much since a small number of edge insertions do not change the size of graph in any meaningful way. VSR only achieves around 0.2% percentage more than VR on average. In term of efficiency, the running time of RS grows linearly, which indicates that its performance is linearly associated with the Kuratowski subdivisions detected in the graph since the graph sizes are similar. Figure 6

² <http://konect.uni-koblenz.de/>



■ **Figure 6** Experiments on RD100000 with edge insertion excluding VA. The left figure shows the percentage achieved by the algorithms (HL achieves 34% on average, and is not shown). The right figure shows the running time achieved by all of the algorithms.

■ **Table 2** Experimental results on almost planar graphs using the RD dataset.

Vertex Increase (%)	Percentage (%)				Time (s)			
	RS	VSR	VR	HL	RS	VSR	VR	HL
10	99.99	88.29	88.13	34.28	0.07	17.61	4.51	0.06
20	99.99	89.13	88.97	34.31	0.11	67.03	16.64	0.12
30	99.99	89.28	89.14	34.33	0.20	148.58	34.63	0.18
40	99.99	89.95	89.75	34.35	0.28	272.61	62.98	0.22

shows the experiment results on *RD100000*, and produces similar observations. As the graph size increases, the superiority of the efficiency of HL becomes more pronounced.

We also perform experiments on almost planar graphs which belong to a special class of graphs with skewness equal to one [16]. Previous studies working on almost planar graphs have taken a similar approach [16, 2, 4]. We construct almost planar graphs based on the *RD* dataset. The number of vertices of those graphs range from 10% to 40% of *RD*. As we can see in Table 2, RS always achieves a percentage of 99.99%, which corresponds with the definition of almost planar graphs. The percentage achieved by other methods are all below 90%, and are sensitive to graph size, which reflects the over-reliance on the indirect indicators of planarity used by these methods. The average running time of HL and RS are 0.15 s and 0.17 s respectively. Performance of RS varies little since Kuratowski subdivisions are rarely introduced, and this is the main property which affects its performance. On average, RS is 150 times faster than VR and 640 times faster than VSR.

Experiments on non-planar graphs. In this section, we explore the performance on real-world non-planar graphs: *MP*, *UG*, *PG* and *IN*. Based on each graph, we construct graphs by removing a certain percentage of vertices from the original graphs in descending order of the maximum degree. When the skewness of the input graph is not small, VR and VSR tend to perform well since they iteratively remove a vertex with the maximum degree, and this has the same effect as removing multiple Kuratowski subdivisions at once. On the other hand, RS consistently achieves a local optima by removing the obstruction vertex shared by Kuratowski subdivisions detected in each iteration. Due to limited space, we use only *MP* and *IN* to demonstrate this effect. Additional results are in the Appendix.

Figure 7 shows the results on the dataset *MP*. RS always achieves the best percentage, and reaches 100% when the vertex removal rate is 3.5%. Other methods cannot achieve 100% even though the graph is already planar. The performance of VR and VSR are almost

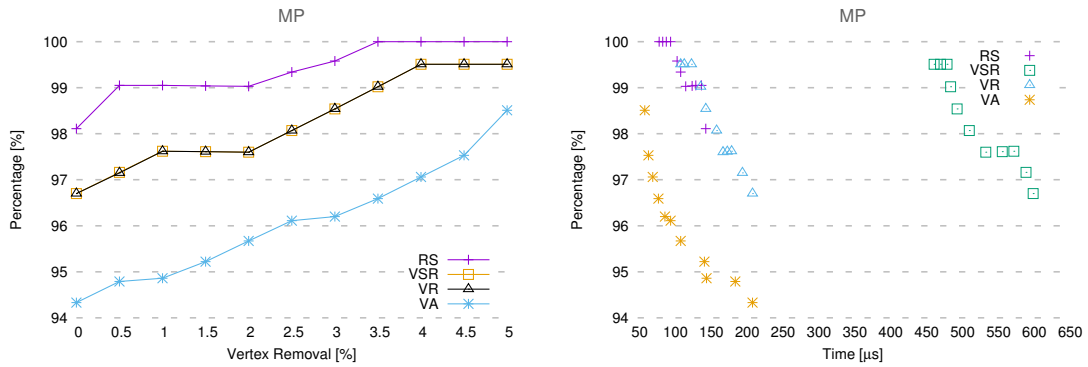


Figure 7 Experiments on *MP* with vertex removal. The left figure shows the percentages achieved by different algorithms (HL achieves only 25% on average and is not shown). The right figure shows the Efficiency / Effectiveness relationship (the running time of HL is 356 ms on average and not shown).

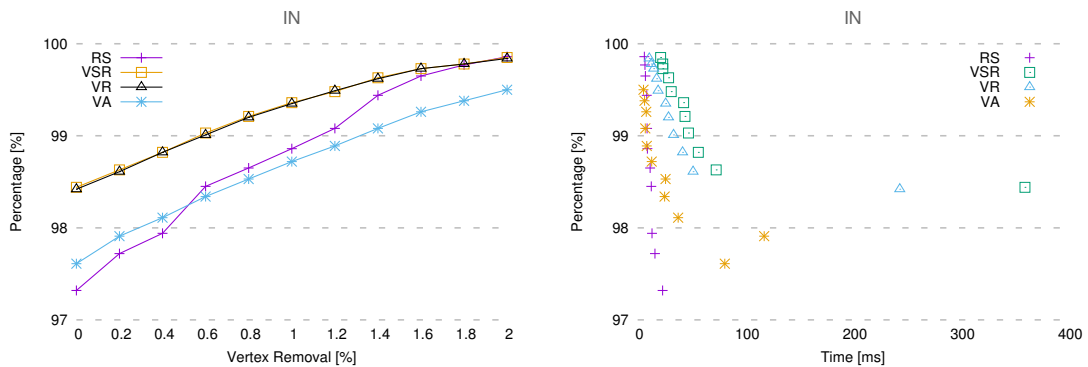


Figure 8 Experiments on *IN* with vertex removal. The left figure shows the percentages achieved by different algorithms (HL achieves 9% on average and is not shown). The right figure shows the Efficiency / Effectiveness relationship (the running time of HL is 10 ms on average and not shown).

the same. VSR achieves at most 0.001% more than VR. Methods such as VA and VR exhibit higher efficiency than HL, and VA is more efficient than RS in many cases. The reason is that the graph size is so small that these methods converge very quickly. For example, the reduced graph mentioned in Section 1 is so small that VR only has to remove a few vertices from the graph. On this dataset, RS and VSR outperforms all other approaches if both accuracy and efficiency are considered.

Figure 8 shows experimental results on *IN*. Initially, RS achieves 1.2% less than VSR and 0.3% less than VA. As the percentage of vertex removals increases, the gap between RS and VSR is narrowed and RS outperforms all other methods when 2% of vertices are removed. A higher percentage indicates a higher vertex removal rate, which also indicates a smaller graph size. From the right figure, it is worth noting that there is a rapid change of efficiency of VR and VSR when the vertex removal rate increases to 0.2%. The increased cost in VR and VSR are caused by the iterative removal of maximum degree vertices. Since, we have already removed vertices of the maximum degree before the algorithms are initialized, their costs are therefore greatly reduced. Another behavior needs worth noting is that VA runs slower even though the graph size is smaller when the vertex removal rate increases to 0.2%. The performance of VA cannot be predicted based on the graph size since it depends on finding paths between vertices, which can vary significantly based on the connectivity in the graph.

Small changes in the overall structure of the graph can lead to large changes in efficiency. In summary, RS outperforms all other methods on nearly planar graphs. When the graph is not ‘close to’ planar, RS provides a good option when a tradeoff between efficiency and accuracy needs to be made since RS is more efficient than all previous methods, and its accuracy is still competitive.

4 Conclusion

In this paper, we studied the Maximum Induced Planar Subgraph (MIPS) problem which aims to find the largest size of vertices which induce a planar subgraph. As in many related problems, there is a trade-off between the quality of the approximation and the efficiency of the algorithm. By observing that both planarity testing and planarization can be accomplished simultaneously, we were able to produce a linear time algorithm for the MIPS problem, and the new approach is competitive in both efficiency and effectiveness.

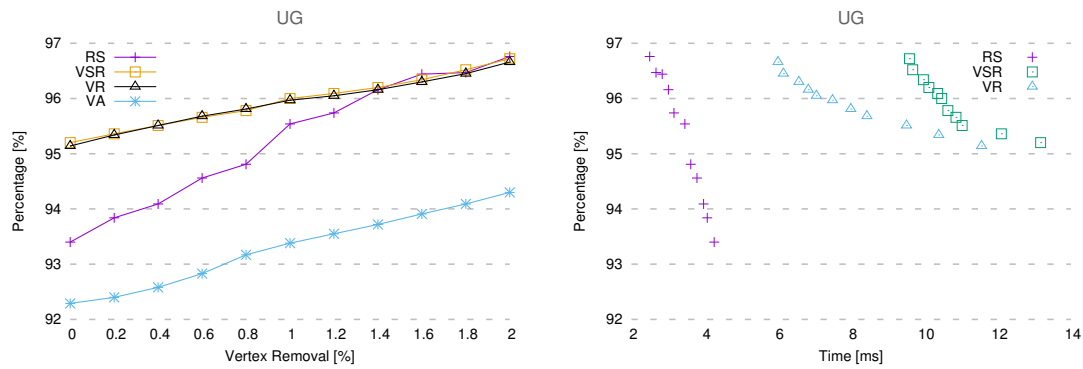
References

- 1 Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G Tollis. *Graph drawing: Algorithms for the visualization of graphs*. Prentice Hall PTR, 1998.
- 2 Itai Benjamini and Oded Schramm. Harmonic functions on planar and almost planar graphs and manifolds, via circle packings. *Inventiones Mathematicae*, 126(3):565–587, 1996.
- 3 John M Boyer and Wendy J Myrvold. On the cutting edge: Simplified $o(n)$ planarity by edge addition. *J. Graph Algorithms Appl.*, 8(2):241–273, 2004.
- 4 Sergio Cabello and Bojan Mohar. Crossing and weighted crossing number of near-planar graphs. In *Proc. GD*, pages 38–49. Springer, 2008.
- 5 Gek L Chia and Chan L Lee. Regular raphs with small skewness and crossing numbers. *Bulletin of the Malaysian Mathematical Sciences Society*, 39(4):1687–1693, 2016.
- 6 Robert J Cimikowski. Graph planarization and skewness. *Congressus Numerantium*, pages 21–21, 1992.
- 7 Alice M Dean, William S Evans, Ellen Gethner, Joshua D Laison, Mohammad Ali Safari, and William T Trotter. Bar k-visibility graphs. *J. Graph Algorithms Appl.*, 11(1):45–59, 2007.
- 8 Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G Tollis. Algorithms for drawing graphs: an annotated bibliography. *Computational Geometry*, 4(5):235–282, 1994.
- 9 Walter Didimo and Giuseppe Liotta. The crossing-angle resolution in graph drawing. In *Thirty Essays on Geometric Graph Theory*, pages 167–184. Springer, 2013.
- 10 Keith Edwards and Graham Farr. An algorithm for finding large induced planar subgraphs. In *Proc. GD*, pages 75–83. Springer, 2001.
- 11 Keith Edwards and Graham Farr. Planarization and fragmentability of some classes of graphs. *Discrete Mathematics*, 308(12):2396–2406, 2008.
- 12 William Evans, Michael Kaufmann, William Lenhart, Tamara Mchedlidze, and Stephen Wismath. Bar 1-visibility graphs and their relation to other nearly planar graphs. *J. Graph Algorithms Appl.*, 18(5):721–739, 2014.
- 13 Alan Gibbons. *Algorithmic graph theory*. Cambridge university press, 1985.
- 14 Magnús M Halldórsson and Hoong Chuin Lau. Low-degree graph partitioning via local search with applications to constraint satisfaction, max cut, and coloring. In *Graph Algorithms And Applications I*, pages 45–57. World Scientific, 2002.
- 15 Mohsen MD Hassan and Gary L Hogg. A review of graph theory application to the facilities layout problem. *Omega*, 15(4):291–300, 1987.

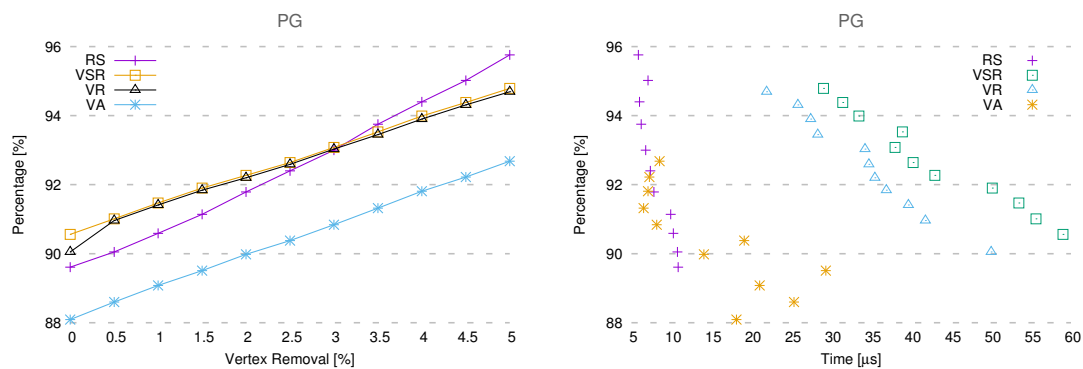
- 16 Petr Hliněný and Gelasio Salazar. On the crossing number of almost planar graphs. In *Proc. GD*, pages 162–173. Springer, 2006.
- 17 Seok-Hee Hong, Peter Eades, Giuseppe Liotta, and Sheung-Hung Poon. Fáry’s theorem for 1-planar graphs. In *Proc. COCOON*, pages 335–346, 2012.
- 18 Michael Jünger and Petra Mutzel. Maximum planar subgraphs and nice embeddings: Practical layout tools. *Algorithmica*, 16(1):33–59, 1996.
- 19 Mukkai S Krishnamoorthy and Narsingh Deo. Node-deletion np-complete problems. *SIAM Journal on Computing*, 8(4):619–625, 1979.
- 20 Jérôme Kunegis. KONECT: The koblenz network collection. In *Proc. WWW*, pages 1343–1350, 2013.
- 21 Casimir Kuratowski. Sur le probleme des courbes gauches en topologie. *Fundamenta Mathematicae*, 15(1):271–283, 1930.
- 22 Jure Leskovec and Andrej Krevl. Snap datasets: Stanford large network dataset collection, 2015. URL: <http://snap.stanford.edu/data>.
- 23 John M Lewis and Mihalis Yannakakis. The node-deletion problem for hereditary properties is np-complete. *Journal of Computer and System Sciences*, 20(2):219–230, 1980.
- 24 Annegret Liebers. Planarizing graphs—a survey and annotated bibliography. In *Graph Algorithms And Applications 2*, pages 257–330. World Scientific, 2004.
- 25 László Lovász. On decomposition of graphs. *Studia Sci. Math. Hungar*, 1(273):238, 1966.
- 26 Carsten Lund and Mihalis Yannakakis. The approximation of maximum subgraph problems. In *Proc. ICALP*, pages 40–51. Springer, 1993.
- 27 Kerri Morgan and Graham Farr. Approximation algorithms for the maximum induced planar and outerplanar subgraph problems. *J. Graph Algorithms Appl.*, 11(1):165–193, 2007.
- 28 Mauricio GC Resende and Celso C Ribeiro. A grasp for graph planarization. *Networks*, 29(3):173–189, 1997.
- 29 Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.
- 30 Douglas Brent West et al. *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, 2001.

A Experiments on non-planar graphs

Figure 9 and Figure 10 show experimental results for UG and PG respectively. As in the experiments on MP and IN , when the vertex removal rate increases, the gaps between the percentages achieved by VA and VSR are reduced, and VA outperforms the other methods once the vertex removal rate is high. Even though the graph size of UG is smaller than IN , VA runs around five times slower on UG than on IN . The efficiency of VA is remarkably unstable on PG as the vertex removal rate increases.



■ **Figure 9** Vertex Removal experiments on the *UG* dataset. The left figure shows the percentage achieved by different algorithms (HL achieves 28% on average and is not shown). The right figure shows the Effectiveness / Efficiency trade-off (the running time of HL is 4 ms, and VA is 1,526 ms on average – neither are shown to maintain the graph scale).



■ **Figure 10** Vertex removal experiments on the *PG* dataset. The left figure shows the percentage achieved by the algorithms (HL achieves 12% on average and is not shown). The right figure shows the Effectiveness / Efficiency trade-off (the running time of HL is 10 ms on average and not shown).

Fast Spherical Drawing of Triangulations: An Experimental Study of Graph Drawing Tools

Luca Castelli Aleardi

LIX - École Polytechnique, Palaiseau, France
amturing@lix.polytechnique.fr

Gaspard Denis

LIX - École Polytechnique, Palaiseau, France
gaspard.denis@hotmail.fr

Éric Fusy

LIX - École Polytechnique, Palaiseau, France
fusy@lix.polytechnique.fr

Abstract

We consider the problem of computing a spherical crossing-free geodesic drawing of a planar graph: this problem, as well as the closely related spherical parameterization problem, has attracted a lot of attention in the last two decades both in theory and in practice, motivated by a number of applications ranging from texture mapping to mesh remeshing and morphing. Our main concern is to design and implement a linear time algorithm for the computation of spherical drawings provided with theoretical guarantees. While not being aesthetically pleasing, our method is extremely fast and can be used as initial placer for spherical iterative methods and spring embedders. We provide experimental comparison with initial placers based on planar Tutte parameterization. Finally we explore the use of spherical drawings as initial layouts for (Euclidean) spring embedders: experimental evidence shows that this greatly helps to untangle the layout and to reach better local minima.

2012 ACM Subject Classification Mathematics of computing → Graph theory

Keywords and phrases Graph drawing, planar triangulations, spherical parameterizations

Digital Object Identifier 10.4230/LIPIcs.SEA.2018.24

Funding This work was partially supported by the French ANR GATO (ANR-16-CE40-0009-01).

1 Introduction

In this work we consider the problem of computing in a fast and robust way a spherical layout (crossing-free geodesic spherical drawing) of a genus 0 simple triangulation. Several solutions have been developed in the computer graphics and geometry processing communities [1, 2, 3, 15, 18, 26, 29, 32] for this problem, and a few recent works [6, 7, 8, 11, 22] attempted to extend standard tools from graph drawing to deal with graphs on surfaces. On one hand, force-directed methods and iterative solvers are successful to obtain very nice layouts achieving several desirable aesthetic criteria, such as uniform edge lengths, low angle distortion or even the preservation of symmetries. Their main drawbacks rely on the lack of rigorous theoretical guarantees and on their expensive runtime costs, since their implementation requires linear solvers (for large sparse matrices) or sometimes non-linear optimization methods, making these approaches slower and less robust than combinatorial graph drawing tools. On the other hand, some well known combinatorial drawing tools (e.g. linear-time grid embeddings [10, 27]) are provided with worst-case theoretical guarantees allowing us to



© Luca C. Aleardi, Gaspard Denis, and Éric Fusy;
licensed under Creative Commons License CC-BY

17th International Symposium on Experimental Algorithms (SEA 2018).

Editor: Gianlorenzo D'Angelo; Article No. 24; pp. 24:1–24:14

Leibniz International Proceedings in Informatics



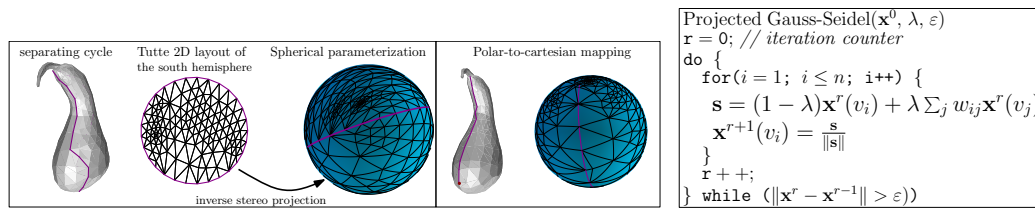
LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

compute in a fast and robust way a crossing-free layout with bounded resolution: just observe that their practical performances allow processing several millions of vertices per second on a standard (single-core) CPU. Unfortunately, the resulting layouts are rather unpleasing and fail to achieve some basic aesthetic criteria that help readability (they often have long edges and large clusters of tiny triangles).

Motivation. It is commonly assumed that starting from a good initial layout (called *initial guess* in [26]) is crucial for both iterative methods and spring embedders (we refer to [23] for a more comprehensive discussion). A nice initial configuration, closer to the final result, should help to obtain nicer layouts: this was explored in [13] for the planar case. This could be even more relevant for the spherical case, where an initial layout with many edge-crossings can be difficult to unfold in order to obtain a valid spherical drawing. Moreover, the absence of boundary constraints on the sphere prevents in some cases from eliminating all crossings before the layouts collapse to a degenerate configuration. One of the motivations of this work is to get benefit of a prior knowledge of the graph structure: if its combinatorics is known in advance, then one can make use of fast graph drawing tools and compute a crossing-free layout to be used as starting point for running more expensive force-directed tools.

Related works. A first approach for computing a spherical drawing consists in projecting a (convex) polyhedral representation of the input graph on the unit sphere: one of the first works [28] provided a constructive version of Steinitz theorem (unfortunately its time complexity was quadratic). Another very simple approach consists in planarizing the graph and to apply well known tools from mesh parameterizations (see Section 2.1 for more details): the main drawback is that, after spherical projection, the layout does not always remain crossing-free. Along another line of research, several works proposed generalizations of the barycentric Tutte parameterization to the sphere. Unlike the planar case, where boundary constraints guarantee the existence of crossing-free layouts, in the spherical case both the theoretical analysis and the practical implementations are much more challenging. Several works in the geometry processing community [3, 15, 26, 32] expressed the layout problem as an energy minimization problem (with non-linear constraints) and proposed a variety of iterative or optimization methods to solve the spherical Tutte equations: while achieving nice results on the tested 3D meshes, these methods lack rigorous theoretical guarantees on the quality of the layout in the worst case (for a discussion on the existence of non degenerate solutions of the spherical Tutte equations we refer to [18]). A very recent work [1] proposed an adaptation of the approach based on the Euclidean orbifold Tutte parameterization [2] to the spherical case: the experimental results are very promising and come with some theoretical guarantees (a couple of weak assumptions are still necessary to guarantee the validity of the drawing). However the layout computation becomes much more expensive since it involves solving non-linear problems, as reported in [1]. A few papers in the graph drawing domain also considered the spherical drawing problem. Fowler and Kobourov proposed a framework to adapt force-directed methods [16] to spherical geometry, and a few recent works [6, 7, 8, 11] extend some combinatorial tools to produce planar layouts of non-planar graphs: some of these tools can be combined to deal with the spherical case, as we will show in this work (as far as we know, there are not existing implementations of these algorithms).

Our contributions. Our first main contribution is to design and implement a fast algorithm for the computation of spherical drawings. We make use of several ingredients [6, 7, 11] involving the well-known *canonical orderings* and exploit an adaptation of the *shift* paradigm



■ **Figure 1** (left) Two spherical parameterizations of the **gourd** graph obtained via Tutte’s planar parameterization. (right) The pseudo-code of the *Projected Gauss-Seidel* method.

proposed by de Fraysseix, Pach and Pollack [10]. As illustrated by our experiments, our procedure is extremely fast, with theoretical guarantees on both the runtime complexity and the layout resolution.

While not being aesthetically pleasing (as in the planar case), our layouts can be used as initial vertex placement for iterative parameterization methods [3, 26] or spherical spring embedders [22]. Following the approach suggested by Fowler and Kobourov [13], we compare our combinatorial algorithm with two standard initial placers used in previous existing works [26, 32] relying on Tutte planar parameterizations: our experimental evaluations involve runtime performances and statistics concerning edge lengths.

As an application, we show in Section 5 how spherical drawings can be used as initial layouts for (Euclidean) spring embedders: as illustrated by our tests, starting from a spherical drawing greatly helps to untangle the layout and to escape from bad local minima.

2 Preliminaries

Planar graphs and spherical drawings. In this work we deal with *planar maps* (graphs endowed with a combinatorial planar embedding), and we consider in particular *planar triangulations* which are simple genus 0 maps where all faces are triangles (they correspond to the combinatorics underlying genus 0 3D triangle meshes). Given a graph $G = (V, E)$ we denote by $n = |V|$ (resp. by $|F(G)|$) the number of its vertices (resp. faces) and by $N(v_i)$ the set of neighbors of vertex v_i ; $\mathbf{x}(v_i)$ will denote the Euclidean coordinates of vertex v_i .

The notion of planar drawings can be naturally generalized to the spherical case: the main difference is that edges are mapped to *geodesic arcs* on the unit sphere \mathbb{S}^2 , which are minor arcs of great circles (obtained as intersection of \mathbb{S}^2 with a hyperplane passing through the origin). A *geodesic drawing* of a map should preserve the cyclic order of neighbors around each vertex (such an embedding is unique for triangulations, up to reflexions of the sphere). As in the planar case, we would aim to obtain *crossing-free* geodesic drawings, where geodesic arcs do not intersect (except at their extremities). In the rest of this work we will make use of the term *spherical drawings* when referring to drawings satisfying the requirements above. Sometimes, the weaker notion of *spherical parameterization* (an homeomorphism between an input mesh and \mathbb{S}^2) is considered for dealing with applications in the geometry processing domain (such as mesh morphing): while the bijectivity between the mesh and \mathbb{S}^2 is guaranteed, there are no guarantees that the triangle faces are mapped to spherical triangles with no overlaps (obviously a spherical drawing leads to a spherical parameterization).

2.1 Initial Layouts

Part of this work will be devoted to comparing our drawing algorithm (Section 3) to two spherical parameterization methods involving Tutte planar parameterization: both methods have been used as *initial placers* for more sophisticated iterative spherical layout algorithms.

Inverse Stereo Projection layout (ISP). For the first initial placer, we follow the approach suggested in [26] (see Fig. 1). The faces of the input graph G are partitioned into two components homeomorphic to a disk: this is achieved by computing a vertex separator defining a simple cycle of small size (having $O(\sqrt{n})$ vertices) whose removal produces a balanced partition (G^S, G^N) of the faces of G . The two graphs G^S and G^N are then drawn in the plane using Tutte’s barycentric method: boundary vertices lying on the separator are mapped on the unit disk. Combining a Moebius inversion with the inverse of a stereographic projection we obtain a spherical parameterization of the input graph: while preserving some of the aesthetic appeal of Tutte’s planar drawings, this map is bijective but cannot produce in general a crossing-free spherical drawing (straight-line segments in the plane are not mapped to geodesics by inverse stereographic projection). In our experiments we adopt a growing-region heuristic to compute a simple separating cycle: while not having theoretical guarantees, our approach is simple to implement and very fast, achieving balanced partitions in practice (separators are of size roughly $\Theta(\sqrt{n})$ and the balance ratio $\varrho = \frac{\min(|F(G^S)|, |F(G^N)|)}{|F(G)|}$ is always between 0.39 and 0.49 for the tested data)¹.

Polar-to-Cartesian layout (PC). The approach adopted in [32] consists in planarizing the graph by cutting the edges along a simple path from a south pole v^S to a north pole v^N . A planar rectangular layout is computed by applying Tutte parameterization with respect to the azimuthal angle $\theta \in (0, 2\pi)$ and to the polar angle $\phi \in [0, \pi]$: the spherical layout, obtained by the polar-to-cartesian projection, is bijective but not guaranteed to be crossing-free.

2.2 Spherical drawings and parameterizations

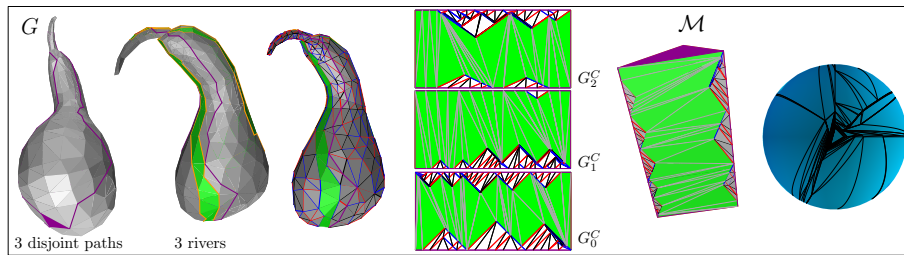
The spherical layouts described above can be used as initial guess for more sophisticated iterative schemes and force-directed methods for computing spherical drawings. For the sake of completeness we provide an overview of the algorithms that will be tested in Section 4.

Iterative relaxation: projected Gauss-Seidel. The first method can be viewed as an adaptation of the iterative scheme solving Tutte equations (see Fig. 1). This scheme consists in moving points on the sphere in tangential direction in order to minimize the spring energy

$$\mathcal{E} = \frac{1}{2} \sum_{i=1}^n \sum_{j \in N(i)} w_{ij} \|\mathbf{x}(v_i) - \mathbf{x}(v_j)\|^2 \quad (1)$$

with the only constraint $\|\mathbf{x}(v_i)\| = 1$ for $i = 1 \dots n$ (in this work we consider uniform weights w_{ij} , as in Tutte’s work). As opposed to the planar case, there are no boundary constraints on the sphere, which makes the resulting layouts collapse in many cases to degenerate solutions. As observed in [18, 26] this method does not always converge to a valid spherical drawing, and its practical performance strongly depends on the geometry of the starting initial layout \mathbf{x}^0 . While not having theoretical guarantees, this method is quite fast allowing to quickly decrease the residual error: it thus can be used in a first phase and combined with more stable iterative schemes leading in practice to better convergence results [26] (still lacking of rigorous theoretical guarantees).

¹ The computation of small cycle separators for planar triangulations is a very challenging task. This work does not focus on this problem: we refer to recent results [14] providing the first practical implementations with theoretical guarantees.



■ **Figure 2** Computation of a spherical drawing based on a prism layout of the gourd graph (326 vertices). Three vertex-disjoint chord-free paths lead to the partition of the faces of G into three regions which are each separated by one river (green faces). Our variant of the FPP algorithm allows to produce three rectangular layouts, where boundary vertex locations do match on identified (horizontal) sides. One can thus glue the planar layouts to obtain a 3D prism: its central projection on the sphere produces a spherical drawing. Edge colors (blue, red and black) are assigned during the incremental computation of a *canonical labeling* [11], according to the Schnyder wood local rule.

Alexa's method. In order to avoid the collapse of the layout, without using artificial constraints, Alexa [3] modified the iterative relaxation above by penalizing long edges (tending to move vertices in a same hemisphere). More precisely, the vertex v_i is moved according to a displacement $\Delta_i = c \frac{1}{deg(v_i)} \sum_j (\mathbf{x}(v_i) - \mathbf{x}(v_j)) \|\mathbf{x}(v_i) - \mathbf{x}(v_j)\|$ and then reprojected on the sphere. The parameter c regulates the step length, and can be chosen to be proportional to the inverse of the longest edge incident to a vertex, improving the convergence speed.

(Spherical) Spring Embedders. While spring embedders are originally designed to produce 2D or 3D layouts, one can adapt them to non euclidean geometries. We have implemented the standard *spring-electrical model* introduced in [16] (referred to as **FR**), and the spherical version following the framework described by Kobourov and Wampler [22] (called **Spherical FR**). As in [16] we compute attractive forces (between adjacent vertices) and repulsive forces (for any pair of vertices) acting on vertex u , defined by:

$$F_a(u) = \sum_{(u,v) \in E} \frac{\|\mathbf{x}(u) - \mathbf{x}(v)\|}{K} (\mathbf{x}(u) - \mathbf{x}(v)), \quad F_r(u) = \sum_{v \in V, v \neq u} \frac{-CK^2(\mathbf{x}(v) - \mathbf{x}(u))}{\|\mathbf{x}(u) - \mathbf{x}(v)\|^2}$$

where the values C (the strength of the forces) and K (the optimal distance) are scale parameters. In the spherical case, we shift the repulsive forces by a constant term, making the force acting on pairs of antipodal vertices zero.

3 Fast spherical embedding with theoretical guarantees: SFPP layout

We now provide an overview of our algorithm for computing a spherical drawing of a planar triangulation G in linear time, called **SFPP layout** (see Fig. 2 for an illustration). We make use of an adaptation of the *shift* method used in the incremental algorithm of de Fraysseix, Pach and Pollack [10] (referred to as **FPP layout**): our solution relies on the combination of several ideas developed in [11, 6, 7]. A more detailed presentation can be found in [5].

Mesh segmentation. Assuming that there are two non-adjacent faces f^N and f^S , one can find 3 disjoint and chord-free paths P_0, P_1 and P_2 from f^S to f^N (planar triangulations are 3-connected). Denote by u_0^N, u_1^N and u_2^N the three vertices of f^N on P_0, P_1 and P_2 (define similarly the three neighbors u_0^S, u_1^S, u_2^S of the face f^S). We first compute a partition of the

faces of G into 3 regions, cutting G along the paths above and removing f^S and f^N . We thus have three quasi-triangulations G_0^C , G_1^C and G_2^C that are planar maps whose inner faces are triangles, and where the edges on the outer boundary are partitioned into four sides. The first pair of opposite sides only consist of an edge (drawn as vertical segment in Fig. 2), while the remaining pair of opposite sides contains vertices lying on P_i and P_{i+1} respectively (indices being modulo 3): according to these definitions, G_i^C and G_{i+1}^C share the vertices lying on P_{i+1} (drawn as a path of horizontal segments in Fig. 2).

Grid drawing of rectangular frames. We apply the algorithm described in [11] to obtain three rectangular layouts of G_0^C , G_1^C and G_2^C : this algorithm first separates each G_i^C into two sub-graphs by removing a so-called *river*: an outer-planar graph consisting of a face-connected set of triangles which corresponds to a simple path in the dual graph, starting at f^S and going toward f^N . The two-subgraphs are then processed making use of the *canonical labeling* defined in [11]: the resulting layouts are stretched and then merged with the set of edges in the river, in order to fit into a rectangular frame. Just observe that in our case a pair of opposite sides only consists of two edges, which leads to an algorithm considerably simpler to implement in practice. Finally, we apply the two-phases adaptation of the shift algorithm described in [6] to obtain a planar grid drawing of each map G_i^C , such that the positions of vertices on the path P_i in G_i^C do match the positions of corresponding vertices on P_i in G_{i+1}^C . The grid size of drawing of G_i^C is $O(n) \times O(n)$ (using the fact that the two opposite sides (u_i^N, \dots, u_i^S) and $(u_{i+1}^N, \dots, u_{i+1}^S)$ of G_i^C are at distance 1).

Spherical layout. To conclude, we glue together the drawings of G_0^C , G_1^C and G_2^C computed above in order to obtain a drawing of G on a triangular prism. By a translation within the 3D ambient space we can make the origin coincides with the center of mass of the prism (upon seeing it as a solid polyhedron). Then a central projection from the origin maps each vertex on \mathcal{M} to a point on the sphere: each edge (u, v) is mapped to a geodesic arc, obtained by intersecting the sphere with the plane passing through the origin and the segment relying u and v on the prism (crossings are forbidden since the map is bijective).

► **Theorem 1.** *Let G be a planar triangulation of size n , having two non-adjacent faces f^S and f^N . Then one can compute in $O(n)$ time a spherical drawing of G , where edges are drawn as (non-crossing) geodesic arcs of length $\Omega(\frac{1}{n})$.*

Some heuristics. We use as last initial placer our combinatorial algorithm of Section 3. For the computation of the three disjoint paths P_0 , P_1 and P_2 , we adopt again a heuristic based on a growing-region approach: while not having theoretical guarantees on the quality of the partition and the length of the paths, our results suggest that well balanced partitions are achieved for most tested graphs. A crucial point to obtain a nice layout resides in the choice of the *canonical labeling* (its computation is performed with an incremental approach based on vertex removal). A bad canonical labeling could lead to unpleasant configurations, where a large number of vertices on the boundaries of the bottom and top sub-regions of each graph G_i are drawn along the same direction: as side effects, a few triangles use a lot of area, and the set of interior chordal edges in the river can be highly stretched, especially those close to the south and north poles. To partially address this problem, we design a few heuristics during the computation of the canonical labeling, in order to obtain more balanced layouts. Firstly, we delay the conquest of the vertices which are close to the south and north poles: this way these extremal vertices are assigned low labels (in the canonical labeling), leading to smaller and thicker triangles close to the poles. Moreover the selection

of the vertices is done so as to keep the height of the *triangle caps* more balanced in the final layout. Finally, we adjust the horizontal stretch of the edges, to get more equally spaced vertices on the paths P_0 , P_1 and P_2 .

4 Experimental results and comparison

Experimental settings and datasets. In order to obtain a fair comparison of runtime performances, we have written pure Java implementations of all algorithms and drawing methods presented in this work². Our tests involve two dozen of graphs, including the 1-skeleton of 3D models (made available by the *AIM@SHAPE* repository) as well as random planar triangulations obtained with a uniform random sampler [25]. In our tests we take as an input the combinatorial structure of a planar map encoded in OFF format: nevertheless we do not make any assumption on the geometric realization of the input triangulation in 2D or 3D space. Observe that the fact of knowing the combinatorial embedding of a graph G (the set of its faces) is a weak assumption, since such an embedding is essentially unique for planar triangulations and it can be retrieved from the graph connectivity in linear time [24]. We run our experiments on a HP EliteBook, equipped with an Intel Core i7 2.60GHz (with Ubuntu 16.04, Java 1.8 64-bit, using a single core, and 4GB of RAM for the JVM).

4.1 Quantitative evaluation of aesthetic criteria

In order to obtain a quantitative evaluation of the layout quality we compute the spring energy \mathcal{E} defined by Eq. 1 and two metrics measuring the edge lengths and the triangle areas. As suggested in [13] we compute the average percent deviation of edge lengths, according to

$$\epsilon l := 1 - \left(\frac{1}{|E|} \sum_{e \in E} \frac{|l_g(e) - l_{avg}|}{\max(l_{avg}, l_{max} - l_{avg})} \right)$$

where $l_g(e)$ denotes the geodesic distance of the edge e , and l_{avg} (resp. l_{max}) is the average geodesic edge length (resp. maximal geodesic edge length) in the layout. In a similar manner we compute the average percent deviation of triangle areas, denoted by \mathbf{a} . The metrics ϵl and \mathbf{a} take values in $[0 \dots 1]$, and higher values indicate more uniform edge lengths and triangle areas³.

4.2 Timing performances: comparison

The runtime performances reported in Table 1 clearly show that our SFPP algorithm has an asymptotic linear-time behavior and in practice is much faster than ISP and PC. For instance the ISP layout adopted in [26] requires to solve large linear systems: among the tested Java libraries (MTJ, Colt, PColt, Jama), we found that the linear solvers of the MTJ have the best runtime performances for the solution of large sparse linear systems (in our tests we run the *conjugate gradient* solver, setting a numeric tolerance of 10^{-6}). Observe that a slightly better performance can be achieved with more sophisticated schemes or tools (e.g. Matlab solvers) as done in [2, 26]. Nevertheless the timing cost still remains much larger than ours: as reported in [2] the orbifold parameterization of the dragon graph requires 19 seconds (for solving the linear systems, on a 3.5GHz Intel i7 CPU).

² Datasets, source codes and runnable Java applications are available <http://www.lix.polytechnique.fr/~amturing/software.html>

³ Observe that one common metric considered in the geometric processing community is the (angle) distortion: in our case this metric cannot be taken into account since our input is a combinatorial structure (without any geometric embedding).

■ **Table 1** This table reports the runtime performance of all steps involved in the computation of the SFPP layout obtained with the algorithm of Section 3. The overall cost (red chart) includes the preprocessing phase (computing the three rivers and the canonical labeling) and the layout computation (running the two-phases shift algorithm, constructing and projecting the prism). The last two columns report the timing cost for solving the linear systems for the ISP and PC layouts (see blue/green charts), using the MTJ conjugate gradient solver. All results are expressed in seconds.

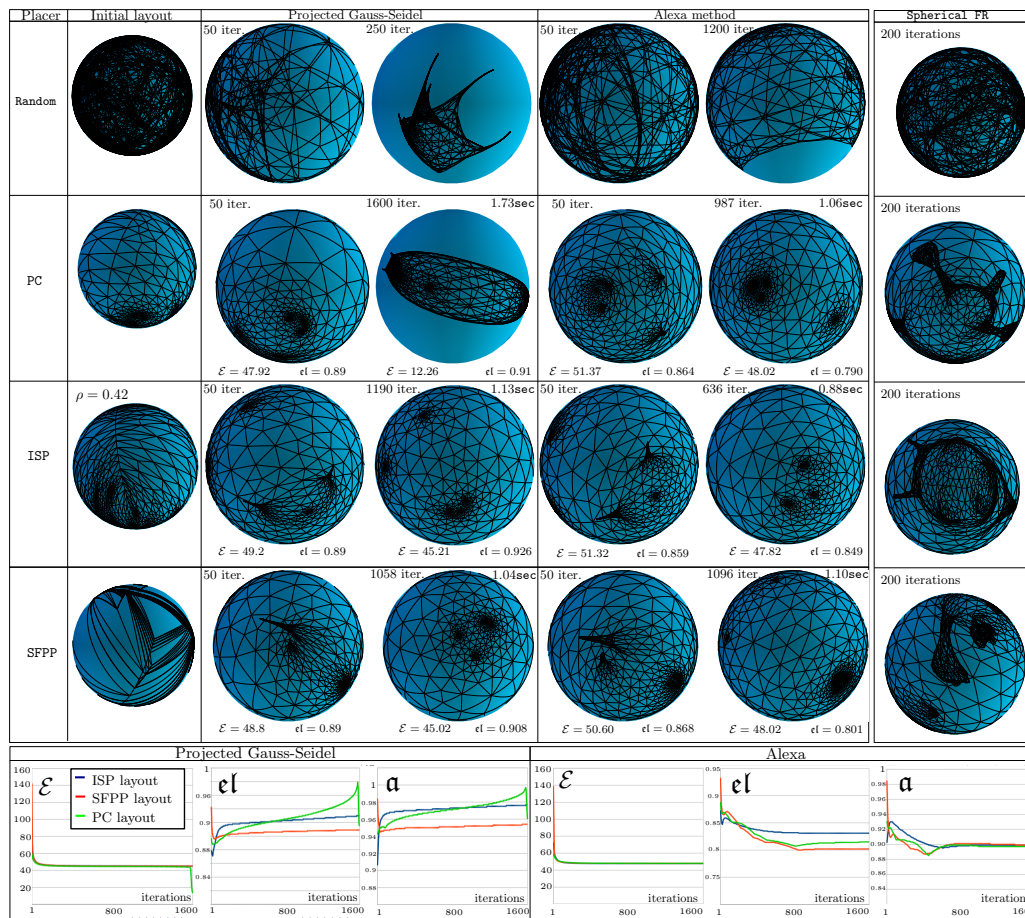
mesh	vertices	faces	preprocessing		Layout computation		PC	ISP
			rivers comput.	canonical labeling	shift algorithm	prism projection	linear solver	linear solver
Egea	8268	16K	0.015	0.017	0.005	0.017	0.24	0.16
Gargoyle	10002	20K	0.016	0.018	0.007	0.025	0.26	0.22
Bunny	26002	52K	0.017	0.031	0.019	0.036	1.14	0.75
Iphigenia	49922	99K	0.023	0.049	0.025	0.046	2.38	1.44
Camille's hand	195557	391K	0.076	0.121	0.073	0.125	17.02	7.92
Eros	476596	950K	0.162	0.260	0.132	0.255	50.54	29.99
Chinese dragon	655980	1.3M	0.174	0.314	0.157	0.433	89.64	53.12

4.3 Evaluation of the layout quality: interpretation and comparisons

All our tests confirm that starting with random vertex locations is almost always a bad choice, since iterative methods lead in most cases to a collapse before reaching a valid spherical drawing (spherical spring embedders do not have this problem, but cannot always eliminate edge crossings, see Fig. 4). Our experiments (see Fig. 3 and 4) also confirm a well known fact: Alexa’s method is more robust compared to the projected Gauss-Seidel relaxation, leading almost always to a valid configuration without collapsing (more tests and statistics can be found in the longer version [5]).

Layout of mesh-like graphs. For the case of mesh-like structures, the ISP and PC methods always provide nicer initial layouts (Fig. 3 show the layout of the dog mesh). The drawings are rather pleasing, capturing the structure of the input graph and being not too far from the final spherical Tutte layout: we mention that the results obtained in our experiments strongly depend on the quality of the separator cycle (or cutting path). Our SFPP initial layout clearly fails to achieve similar aesthetic criteria: nevertheless, even not being pleasing in the first few iterations, it is possible to reach very often a valid final configuration (crossing-free) without collapsing, and whose quality is very close, in terms of energy and edge lengths and area statistics, to the ones obtained starting from the ISP or PC layouts (this is illustrated by the charts in Fig. 3). As we observed for many of the tested graphs, when starting from the SFPP layout the number of iterations required to reach a spherical drawing with good aesthetics is larger than starting from an ISP or PC layout. But the convergence speed can be slightly better in a few cases: Fig. 3 shows a valid spherical layout computed after 1058 iterations of the Gauss-Seidel relaxation (1190 iterations are required when starting from the ISP layout). We also observed that when starting from a PC layout it is sometimes impossible to eliminate all edge-crossings before collapsing (with Gauss-Seidel iteration): the layouts collapse more seldom in the case of ISP and SFPP, as the vertices are likely to be distributed in a more balanced way on the sphere.

The charts in Fig. 3 show that our SFPP has higher values of the edge lengths and area statistics in the first iterations: this reflects the fact that our layout has a polynomial resolution and thus triangles have a bounded aspect ratio and side lengths. When applying methods based on planar parameterization (ISP or PC) there could be a large number of tiny triangles clustered in some small regions (the size of coordinates could be exponentially small as n grows).

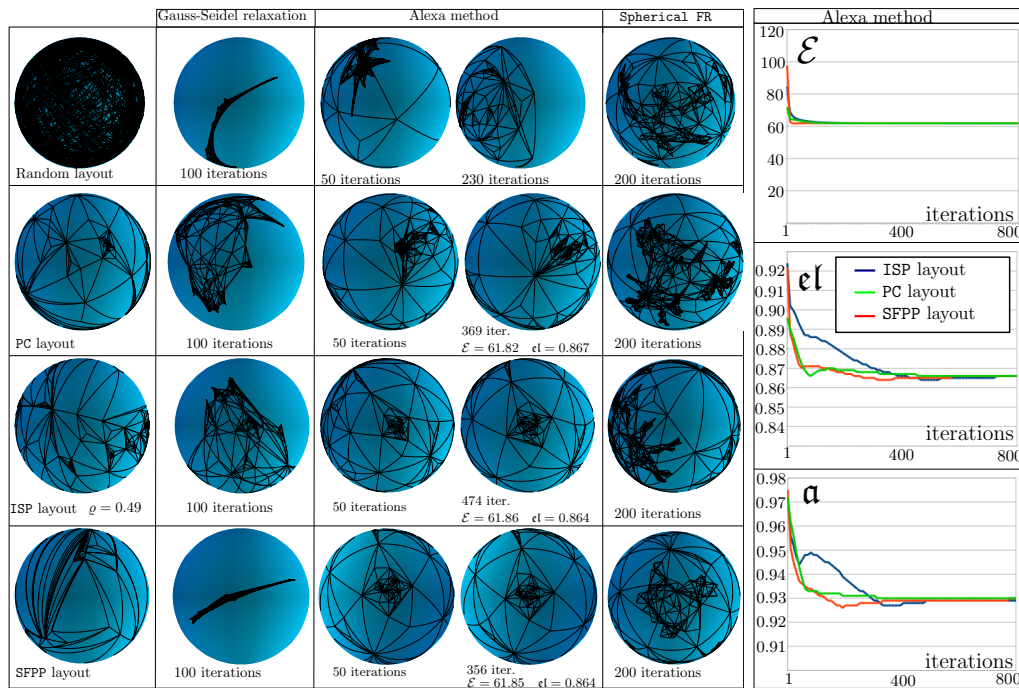


■ **Figure 3** These pictures illustrate the use of different initial placers as starting layouts for two iterative schemes on the *dog* graph (1480 vertices). For each initial layout, we first run 50 iterations of the projected Gauss-Seidel and Alexa method, and then we run the two methods until a valid spherical drawing (crossing free) is reached. The charts below show the energy, area and edge length statistics obtained running 1600 iterations of the projected Gauss-Seidel and Alexa methods.

Layout of random triangulations. When drawing random triangulations the behavior is somehow different: the performances obtained starting from our *SFPP* layout are often better than the ones achieved using the *ISP* layout (and similar to the ones of the *PC* layout). As illustrated by the pictures in Fig. 4 and 6, Alexa’s method is able to reach a non-crossing configuration requiring less iterations when using our *SFPP* layout instead of *ISP* layout: this is observed in most of our experiments, and clearly confirmed by the plots of the energy and statistics ϵl and α that converge faster to the values of the final layout (see charts in Fig. 4).

5 Spherical preprocessing for Euclidean spring embedders

In this section we investigate the use of spherical drawings as initial placers for spring embedders in 3D space. The fact of recovering the original topological shape of the graph, at least in the case of graphs that have a clear underlying geometric structure, is an important and well known ability of spring embedders. This occurs for the case of regular graphs used in Geometry Processing (the pictures in Fig. 5 show a few force-directed layouts of the *cow*

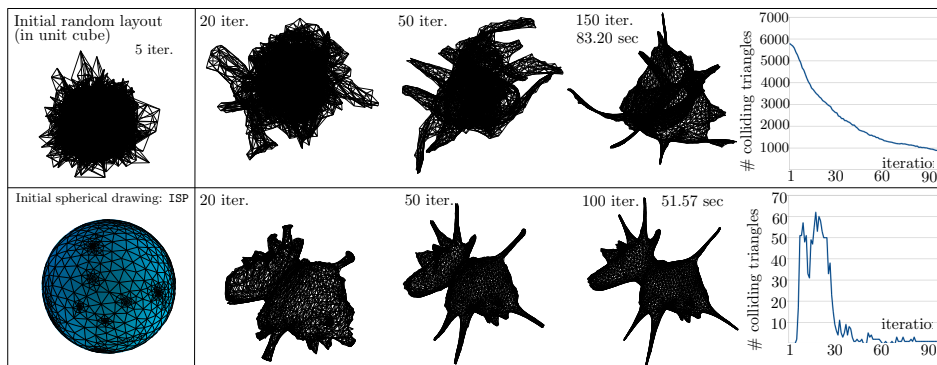


■ **Figure 4** Spherical layouts of a random triangulation with $1K$ faces. While the projected Gauss-Seidel relaxation always collapses, Alexa method is more robust, but also fails when starting from a random initial layout. When using the ISP, PC or our SFPP layouts Alexa method converges toward a crossing-free layout: starting from the SFPP layout allows getting the same aesthetic criteria as the ISP or the PC layouts (with even less iterations). Spring embedders [13] (Spherical FR) prevent from reaching a degenerate configuration, but have some difficulties to unfold the layout. The charts on the right show the plot of the energy, edge lengths and areas statistics computed when running 800 iterations of Alexa method (we compute these statistics every 10 iterations).

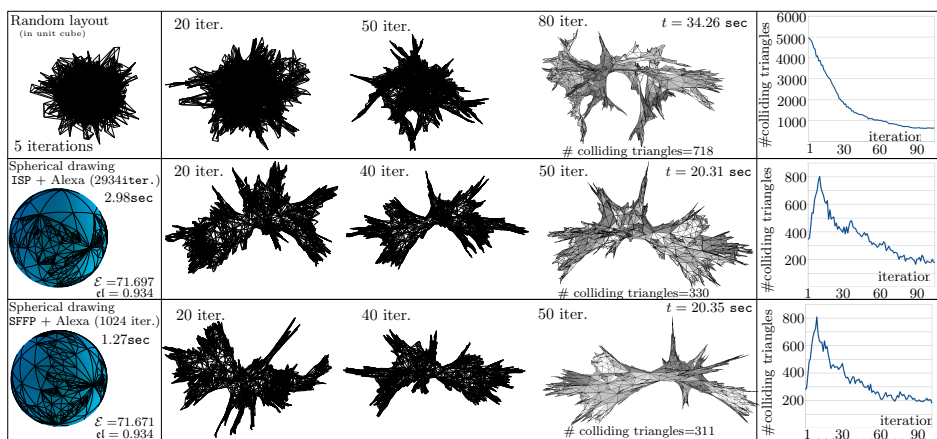
graph), and also for many mesh-like complex networks involved in physical and real-world applications (such as the networks made available by the **Sparse Matrix Collection** [9]). In the case of uniformly random embedded graphs (called maps) of a large size n on a fixed surface \mathcal{S} , the spring embedding algorithms (applied in the 3D ambient space) yield graph layouts that greatly capture the topological and structural features of the map (the genus of the surface is visible, the "central charge" of the model is reflected by the presence of spikes, etc.), a great variety of such representations can be seen at the very nice simulation gallery of Jérémie Bettinelli (<http://www.normalesup.org/~bettinelli/simul.html>). While common software and libraries (e.g. **GraphViz** [12], **Gephi** [4], **GraphStream**) for graph visualization provide implementations of many force-directed models, as far as we know they never try to exploit the strong combinatorial structure of surface-like graphs.

Discussion of experimental results. Our main goal is to show empirically that starting from a nice initial shape that captures the topological structure of the input graph greatly improves the convergence speed and layout quality.

In our first experiments (see Figures 5 and 6) we run our 3D implementation of the spring electrical model FR [16], where we make use of exact force computation and we adopt the cooling system proposed in [31] (with repulsive strength $C = 0.1$). We also perform some tests with the **Gephi** implementation of the Yifan Hu layout algorithm [20], which is a



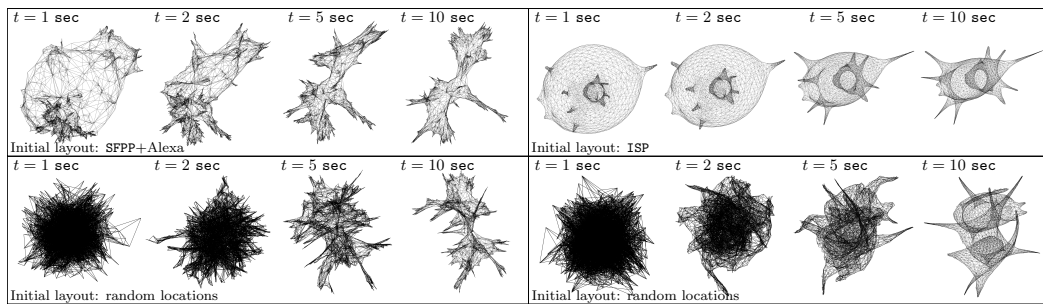
■ **Figure 5** These pictures illustrate the use of spherical drawings as initial placers for force-directed methods: we compute the layouts of the cow graph (2904 vertices, 5804 faces) using our 3D implementation of the FR spring embedder [16]. In the charts on the right we plot the number of colliding 3D triangles, over 100 iterations of the algorithm.



■ **Figure 6** These pictures illustrate the use of spherical drawings as initial placers for the 3D version of the FR spring embedder [16], for a random planar triangulation with $5K$ faces.

more sophisticated spring-embedder with fast approximate calculation of repulsive forces (see the layouts of Fig. 7). In order to quantify the layout quality, we evaluate the number of self-intersections of the resulting 3D shape during the iterative computation process⁴. To be more precise, we plot (over the first 100 iterations) the number of triangle faces that have a collision with a non adjacent triangle in 3D space. The charts of Fig. 5 and 6 clearly confirm the visual intuition suggested by pictures: when starting from a good initial shape the force-directed layouts seem to evolve according to an inflating process, which leads to better and faster untangle the graph layout. This phenomenon is observed in all our tests (on several mesh-like graphs and synthetic data): experimental evidence shows that an initial spherical drawing is a good starting point helping the spring embedder to reach nicer layout aesthetics and also to improve the runtime performances. Finally observe that from the computational point of view the computation of a spherical drawing has a negligible cost: iterative schemes (e.g. Alexa method) require $O(n)$ time per iteration, which must

⁴ We compute the intersections between all pairs of non adjacent triangles running a brute-force algorithm: the runtimes reported in Fig. 5 and 6 do not count the cost of computing the triangle collisions.



■ **Figure 7** The spherical drawings of the graphs in Fig. 5 and 6 are used as initial placers for the Yifan Hu algorithm [20]: we test the implementation provided by Gephi (after rescaling the layout by a factor 1000, we set an optimal distance of 10.0 and a parameter $\vartheta = 2.0$).

be compared to the complexity cost of force-directed methods, requiring between $O(n^2)$ or $O(n \log n)$ time per iteration (depending on the repulsive force calculation scheme). This is also confirmed in practice, according to the timing costs reported in Fig 5, 6 and 7.

6 Concluding remarks and future work

Our SFPP method is guaranteed to compute a crossing-free layout: unfortunately edge crossings can appear during the beautification process, when running iterative algorithms. It could be interesting to adapt to the spherical case existing methods [30] (which are designed for the Euclidean case) whose goal is to dissuade edge-crossings: one could obtain a sequence of layouts that converge to the final spherical drawing while always preserving the map. The results of Section 5 would suggest that starting from an initial nice layout could lead to faster algorithms and better results for mesh-like structures. It could be interesting to investigate whether this phenomenon arises for other classes of graphs, such as quadrangulated or 3-connected planar graphs, or non planar (e.g. toroidal) graphs, for which fast drawing methods also exist [6, 17]. We also plan to perform further tests in order to compare the 3D layouts of Section 5 to the results of more sophisticated multi-scale algorithms [21, 19] that are able to draw large graphs without requiring an initial vertex placement.

References

- 1 Noam Aigerman, Shahar Z. Kovalsky, and Yaron Lipman. Spherical orbifold tutte embeddings. *ACM Trans. Graph.*, 36(4):90:1–90:13, 2017.
- 2 Noam Aigerman and Yaron Lipman. Orbifold tutte embeddings. *ACM Trans. Graph.*, 34(6):190:1–190:12, 2015.
- 3 Marc Alexa. Merging polyhedral shapes with scattered features. *The Visual Computer*, 16(1):26–37, 2000.
- 4 Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. Gephi: An open source software for exploring and manipulating networks. In *Proc. of the Third Int. Conf. on Weblogs and Social Media, ICWSM 2009, 2009*, 2009.
- 5 Luca Castelli Aleardi, Gaspard Denis, and Eric Fusy. Fast spherical drawing of triangulations: an experimental study of graph drawing tools, 2018. URL: <https://hal.archives-ouvertes.fr/hal-01761754>.
- 6 Luca Castelli-Aleardi, Olivier Devillers, and Éric Fusy. Canonical ordering for triangulations on the cylinder, with applications to periodic straight-line drawings. In *Graph Drawing - 20th International Symposium*, pages 376–387, 2012.

- 7 Luca Castelli-Aleardi, Éric Fusy, and Anatolii Kostygin. Periodic planar straight-frame drawings with polynomial resolution. In *LATIN 2014: Theoretical Informatics - 11th Latin American Symposium*, pages 168–179, 2014.
- 8 Erin W. Chambers, David Eppstein, Michael T. Goodrich, and Maarten Löffler. Drawing graphs in the plane with a prescribed outer face and polynomial area. *J. Graph Algorithms Appl.*, 16(2):243–259, 2012.
- 9 Timothy A. Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, 2011.
- 10 Hubert de Fraysseix, János Pach, and Richard Pollack. How to draw a planar graph on a grid. *Combinatorica*, 10(1):41–51, 1990.
- 11 Christian A. Duncan, Michael T. Goodrich, and Stephen G. Kobourov. Planar drawings of higher-genus graphs. *J. Graph Algorithms Appl.*, 15(1):7–32, 2011.
- 12 John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz - open source graph drawing tools. In *Proc. of Graph Drawing*, pages 483–484, 2001.
- 13 J. Joseph Fowler and Stephen G. Kobourov. Planar preprocessing for spring embedders. In *Graph Drawing - 20th International Symposium*, pages 388–399, 2012.
- 14 Eli Fox-Epstein, Shay Mozes, Phitchaya Mangpo Phothilimthana, and Christian Sommer. Short and simple cycle separators in planar graphs. *ACM Journal of Experimental Algorithmics*, 21(1):2.2:1–2.2:24, 2016.
- 15 Ilja Friedel, Peter Schröder, and Mathieu Desbrun. Unconstrained spherical parameterization. *J. Graphics Tools*, 12(1):17–26, 2007.
- 16 Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Softw., Pract. Exper.*, 21(11):1129–1164, 1991.
- 17 Daniel Gonçalves and Benjamin Lévêque. Toroidal maps: Schnyder woods, orthogonal surfaces and straight-line representations. *Discrete & Computational Geometry*, 51(1):67–131, 2014. doi:10.1007/s00454-013-9552-7.
- 18 Craig Gotsman, Xianfeng Gu, and Alla Sheffer. Fundamentals of spherical parameterization for 3d meshes. *ACM Trans. Graph.*, 22(3):358–363, 2003.
- 19 Stefan Hachul and Michael Jünger. Large-graph layout algorithms at work: An experimental study. *J. Graph Algorithms Appl.*, 11(2):345–369, 2007.
- 20 Yifan Hu. Efficient, high-quality force-directed graph drawing. *The Mathematica Journal*, 10(1), 2006. URL: http://yifanhu.net/PUB/graph_draw_small.pdf.
- 21 Stephen G. Kobourov. Force-directed drawing algorithms. In *Handbook on Graph Drawing and Visualization*, pages 383–408. Chapman and Hall/CRC, 2013.
- 22 Stephen G. Kobourov and Kevin Wampler. Non-euclidean spring embedders. *IEEE Trans. Vis. Comput. Graph.*, 11(6):757–767, 2005.
- 23 Chris Muelder and Kwan-Liu Ma. A treemap based method for rapid layout of large graphs. In *IEEE VGTC Pacific Visualization Symposium 2008, PacificVis 2008*, pages 231–238, 2008.
- 24 Hiroshi Nagamochi, Takahisa Suzuki, and Toshimasa Ishii. A simple recognition of maximal planar graphs. *Inf. Process. Lett.*, 89(5):223–226, 2004.
- 25 Dominique Poulalhon and Gilles Schaeffer. Optimal coding and sampling of triangulations. *Algorithmica*, 46(3-4):505–527, 2006. doi:10.1007/s00453-006-0114-8.
- 26 S. Saba, I. Yavneh, C. Gotsman, and A. Sheffer. Practical spherical embedding of manifold triangle meshes. In *(SMI2005)*, pages 258–267, 2005.
- 27 Walter Schnyder. Embedding planar graphs on the grid. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, volume 90, pages 138–148, 1990. URL: <http://departamento.us.es/dma/euita/PAIX/Referencias/schnyder.pdf>.

24:14 Fast Spherical Drawing of Triangulations

- 28 Avner Shapiro and Ayellet Tal. Polyhedron realization for shape transformation. *The Visual Computer*, 14(8/9):429–444, 1998.
- 29 Alla Sheffer, Craig Gotsman, and Nira Dyn. Robust spherical parameterization of triangular meshes. *Computing*, 72(1-2):185–193, 2004.
- 30 Paolo Simonetto, Daniel W. Archambault, David Auber, and Romain Bourqui. Impred: An improved force-directed algorithm that prevents nodes from crossing edges. *Comput. Graph. Forum*, 30(3):1071–1080, 2011.
- 31 Chris Walshaw. A multilevel algorithm for force-directed graph-drawing. *J. Graph Algorithms Appl.*, 7(3):253–285, 2003.
- 32 Rhaleb Zayer, Christian Rössl, and Hans-Peter Seidel. Curvilinear spherical parameterization. In *Int. Conf. on Shape Modeling and Applications (SMI 2006)*, page 11, 2006.

Fleet Management for Autonomous Vehicles Using Multicommodity Coupled Flows in Time-Expanded Networks

Sahar Bsaybes

Université Grenoble Alpes
Institute of Engineering (Grenoble INP), G-SCOP F-38000 Grenoble, France
sahar.bsaybes@grenoble-inp.fr

Alain Quilliot

Université Clermont Auvergne
Laboratoire d'Informatique, de Modélisation et d'Optimisation des Systèmes (LIMOS)
BP 10125, 63173 Aubière Cedex, France
alain.quilliot@uca.fr

Annegret K. Wagler

Université Clermont Auvergne
Laboratoire d'Informatique, de Modélisation et d'Optimisation des Systèmes (LIMOS)
BP 10125, 63173 Aubière Cedex, France
annegret.wagler@uca.fr

Abstract

VIPAFLEET is a framework to develop models and algorithms for managing a fleet of Individual Public Autonomous Vehicles (VIPA). We consider a homogeneous fleet of such vehicles distributed at specified stations in a closed site to supply internal transportation, where the vehicles can be used in different modes of circulation (tram mode, elevator mode, taxi mode). We treat in this paper a variant of the Online Pickup-and-Delivery Problem related to the taxi mode by means of multicommodity coupled flows in a time-expanded network and propose a corresponding integer linear programming formulation. This enables us to compute optimal offline solutions. However, to apply the well-known meta-strategy Replan to the online situation by solving a sequence of offline subproblems, the computation times turned out to be too long, so that we devise a heuristic approach h-Replan based on the flow formulation. Finally, we evaluate the performance of h-Replan in comparison with the optimal offline solution, both in terms of competitive analysis and computational experiments, showing that h-Replan computes reasonable solutions, so that it suits for the online situation.

2012 ACM Subject Classification Mathematics of computing → Discrete mathematics

Keywords and phrases fleet management, offline and online pickup and delivery problem, multicommodity flows

Digital Object Identifier 10.4230/LIPIcs.SEA.2018.25

Funding This work was funded by the French National Research Agency, the European Commission (Feder funds) and the Région Auvergne in the Framework of the LabEx IMobS3.



© Sahar Bsaybes, Alain Quilliot, and Annegret K. Wagler;
licensed under Creative Commons License CC-BY

17th International Symposium on Experimental Algorithms (SEA 2018).

Editor: Gianlorenzo D'Angelo; Article No. 25; pp. 25:1–25:14

Leibniz International Proceedings in Informatics



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

1 Introduction

The project VIPAFLEET aims at contributing to sustainable mobility through the development of innovative urban mobility solutions by means of fleets of Individual Public Autonomous Vehicles (VIPA) allowing passenger transport in closed sites like industrial areas, medical complexes, campuses, or airports. This innovative project involves different partners in order to ensure the reliability of the transportation system [3]. A VIPA is an autonomous vehicle that does not require a driver nor an infrastructure to operate. It is developed by Easymile and Ligier [1, 2] thanks to innovative computer vision guidance technologies [21, 22], whereas the fleet management aspect is studied in [9].

A fleet of VIPAs shall be used in a closed site to transport employees, customers and visitors e.g. between parkings, buildings and from or to a restaurant at lunch breaks. To supply internal transportation, a VIPA can operate in three different transportation modes:

- *Tram mode*: VIPAs continuously run on predefined cycles in a predefined direction and stop at a station if requested to let users enter or leave.
- *Elevator mode*: VIPAs run on predefined lines and move to stations to let users enter or leave, thereby changing their driving direction if needed.
- *Taxi mode*: VIPAs run on a connected network to serve transport requests (from any start to any destination station within given time windows).

This leads to a Pickup-and-Delivery Problem (PDP) in a metric space encoding the considered closed site, where a fleet of servers shall transport goods or persons from a certain origin to a certain destination. If persons have to be transported, we usually speak about a Dial-a-Ride Problem. Many variants are studied in the literature, including the Dial-a-Ride Problem with time windows [14, 15]. In our case, we are confronted with an online situation, where transport requests are released over time [5, 8, 13]. Problems of this type are known to be \mathcal{NP} -hard, see e.g. [20], which also applies to the problem variant considered here, see Section 2.

In [11], we focus on the economic aspect of the problem where the objective is to minimize costs; several algorithms are presented and evaluated w.r.t. minimizing the total tour length for tram and elevator mode.

The taxi mode is the most advanced circulation mode for VIPAs in the dynamic fleet management system. The transport requests are released over time (from any start to any destination station within a network G) and need to be served within a specified time window. Due to the time windows, it is not always possible to serve all transport requests (e.g., if more requests are specified for a same time window than VIPAs are available in the fleet). Hence, the studied PDP is an admission problem as it includes firstly to accept/reject requests and secondly to generate tours for the VIPAs to serve the accepted requests. We are confronted with both the quality-of-service aspect of the problem (with the goal to accept as many requests as possible) and the economic aspect (with the goal to serve the accepted requests at minimum costs, expressed in terms of minimizing the total tour length of the constructed tours), see Section 2.

In [10, 12], a variant of the PDP is studied where the tours are supposed to be nonpreemptive and at each time, (at most) one customer can be transported by a VIPA (note: one customer can be a group of people less than the capacity of the VIPA), and a VIPA cannot serve other requests until the current one is delivered. This leads to a load nonpreemptive DARP with time windows and server capacity 1, where the goal is to accept as many requests as possible and to find tours of minimal length to serve all accepted requests.

In order to solve this problem, three approaches are considered in [12]:

- a simple Earliest Pickup Heuristic that incrementally constructs tours by always choosing from the subsequence $\sigma(t')$ of currently waiting requests this request with smallest possible start time and appending it to the tour with shortest distance from its current end to the requested origin;
- the two well-known meta-strategies Replan and Ignore (which have been analysed in [4, 6, 7] for the Online Traveling Salesman Problem and can be applied to any online problem in time-stamp model ¹, see e.g. [4, 6, 18, 23]) that determine which requests from $\sigma(t')$ can be accepted, and compute optimal (partial) tours to serve them, where Replan performs these tours until new requests are released, but Ignore completely performs these tours before it checks for newly released requests.

It turned out that Ignore is not suitable for the studied admission problem since the decision to accept/reject a request may be taken late, even after the time window to serve the request which does not comply to the quality-of-service aspect of the fleet management. Computational results from [12] show that Replan beats the Earliest Pickup Heuristic in terms of the number of accepted requests, but can only accept 64% of requests compared to the optimal offline solution.

This motivates to study another variant of the PDP related to the taxi mode without the requirement of constructing nonpreemptive tours and transporting, at each time, at most one customer in a VIPA on a direct way along a shortest path from its origin to its destination in the network G .

This problem variant is subject of the present paper. It leads to a more complex model and also computing solutions is more involved, but the expectation is to achieve a higher rate of accepted requests and, therefore, a better quality-of-service level for the fleet management.

It is natural to interpret the studied PDP by means of flows in a time-expanded version G_T of the original network G as, e.g., proposed by [17, 16, 19] for other variants of PDPs. In Section 3, we formulate the offline version of the problem as multicommodity coupled flows in the time-expanded network G_T , using one commodity per request coupled to the flow of VIPAs.

In order to solve the online version of the problem, we apply in Section 4 a Replan-like strategy that solves the online problem by computing a sequence of offline subproblems on certain subsequences of requests. (Recall that Ignore turned out to be not suitable for the studied admission problem, hence we focus here on Replan only.)

Computational experiments revealed that computing optimal offline solutions for the subproblems requires already long computation times, too long and thus not suitable for the online situation. However, we observed that only a small percentage of arcs in the time-expanded network G_T is used in the optimal solutions, so the idea is to reduce G_T to a network containing only arcs which are taken in the optimal solution with high probability, and then to compute the multicommodity coupled flows in the reduced network only. This leads to the flow-based heuristic h-Replan for the offline version of the studied problem.

¹ There are two common online paradigms, the sequence model and the time-stamp model, which differ in the way how information becomes available to the online algorithm: in the sequence model, the requests are given one by one and need to be served immediately and in this order, whereas in the time-stamp model, the requests become known over time at their release dates which allows the online algorithm to postpone and revoke decisions.

In Section 5, we evaluate the performance of h-Replan in comparison with the optimal offline solution both in theory (with the help of competitive analysis) and in practice (with the help of some computational results). We close with some concluding remarks on our approaches.

The results presented here were studied in [9].

2 Problem description and model

As proposed in [9, 11], we embed the VIPAFLEET management problem in the framework of a metric task system.

We encode the closed site where the VIPAFLEET system is running as a *metric space* $M = (V, d)$ induced by a connected network $G = (V, E)$, where the nodes correspond to stations, edges to their physical links in the closed site, and the distance d between two nodes $v_i, v_j \in V$ to the length of a shortest path from v_i to v_j in G . In V , we have a distinguished origin $v_o \in V$, the depot of the system, where all VIPAs are parked when the system is not running, i.e., outside a certain time horizon $[0, T]$.

An operator manages a fleet of k VIPAs each with a capacity for Cap passengers. The fleet management shall allow the operator to decide when and how to move the VIPAs in the network, and to assign requests to VIPAs. Hereby, any request r_j is defined as a 6-tuple $r_j = (t_j, x_j, y_j, p_j, q_j, z_j)$ where

- $t_j \in [0, T]$ is the release date (i.e., the time when r_j becomes known),
- $x_j \in V$ is the origin node,
- $y_j \in V$ is the destination node,
- $p_j \in [0, T]$ is the earliest possible start time,
- $q_j \in [0, T]$ is the latest possible arrival time,
- z_j specifies the number of passengers,

where t_j, p_j , and q_j are certain discrete time points within $[0, T]$ that satisfy $t_j \leq p_j$, $p_j + d(x_j, y_j) \leq q_j$ and where $z_j \leq \text{Cap}$ needs to hold². The operator monitors the evolution of the requests over time and

- decides which requests can be accepted (recall that some requests may have to be rejected if, e.g., more requests are specified for a same time window than VIPAs are available in the fleet), and
- creates tasks to serve accepted requests by moving the VIPAs to go to some station and to pickup, transport and deliver users.

More precisely, a *task* is defined by $\tau_j = (t_j, x_j, t_j^{\text{pick}}, y_j, t_j^{\text{drop}}, z_j)$. It is created by the operator in order to serve request $r_j = (t_j, x_j, y_j, p_j, q_j, z_j)$ and is sent at time t_j to a VIPA indicating that z_j passengers have to be picked up at station x_j at time t_j^{pick} and delivered at station y_j at time t_j^{drop} , where $p_j \leq t_j^{\text{pick}} \leq q_j - d(x_j, y_j)$ and $p_j + d(x_j, y_j) \leq t_j^{\text{drop}} \leq q_j$ must hold.

In order to fulfill the tasks, the operator creates *tours* for the VIPAs. Each tour consists of *moves* from one station in G to another station in G and of *actions* to pickup and deliver passengers. Hereby, we require only that each move carries at most Cap many passengers. That means, we allow

- to serve several requests simultaneously by the same VIPA (as long as the capacity is respected),

² Note that a request r_j with $z_j > \text{Cap}$ can be replaced by $\lceil \frac{z_j}{\text{Cap}} \rceil$ many requests r'_j respecting the constraint $z'_j \leq \text{Cap}$.

- detours to stations not lying on a shortest path from the origin of one request to its destination in order to pickup or deliver passengers from other requests,
- vehicle preemption (i.e. that passengers have to change VIPAs on the way to their destination).

A *transportation schedule* for (M, \mathcal{T}) consists of a collection of tours $\{\Gamma^1, \dots, \Gamma^k\}$ and is *feasible* when

- each of the k VIPAs has exactly one tour,
- each accepted request r_j is served within time window $[p_j, q_j]$,
- each tour starts and ends in the depot.

Our goal is to construct transportation schedules for the VIPAs operating in taxi mode respecting all the above constraints:

► **Problem 1** (Taxi Mode Problem $(M, \sigma, p, T, k, \text{Cap})$ (TMP)). *Given a metric space $M = (V, d)$ induced by a connected network $G = (V, E)$, a sequence of requests σ , profits p for accepted requests, a time horizon $[0, T]$ and k VIPAs of capacity Cap , determine a maximum subset σ_A of accepted requests and find a feasible transportation schedule $\{\Gamma^1, \dots, \Gamma^k\}$ of minimum total tour length to serve all requests in σ_A .*

Hereby, choosing sufficiently high profits and sufficiently small costs guarantees that indeed as many requests as possible are accepted, while small but positive costs ensure that unnecessary movements of VIPAs are avoided.

In order to solve the Offline TMP (Section 3), we propose to construct a time-expanded network G_T and compute multicommodity coupled flows in G_T .

In order to solve the Online TMP (Section 4), we propose the strategy h-Replan that considers at each moment in time t' the subsequence $\sigma(t')$ of currently waiting requests (i.e., already released but not yet served requests), determines which requests from $\sigma(t')$ can be accepted, and computes (partial) tours to serve them by multicommodity coupled flows in the reduced network related to $\sigma(t')$, performs these tours until new requests are released and recomputes $\sigma(t')$ and the tours (keeping already accepted requests).

3 Solving the Offline TMP

In order to solve the Offline TMP, we build a time-expanded network $G_T = (V_T, A_T)$ based on σ and the original network G . The node set V_T contains, for each station $v \in V$ and each discrete time point $t \in [0, T]$, a node $(v, t) \in V_T$ which represents station v at time t as a station where VIPAs can simply pass, pickup or deliver customers. The arc set $A_T = A_W \cup A_M$ is composed of

- wait arcs, from $(v, t) \in V_T$ to $(v, t + 1)$ with $t \in \{0, 1, \dots, T - 1\}$ in A_W ,
- transport arcs, from $(v, t) \in V_T$ to $(v', t + d(v, v'))$ for each edge (v, v') of G and each time point $t \in T$ with $t + d(v, v') \leq T$, in A_M .

On G_T , we define a VIPA flow F to encode the tour of the VIPAs through G_T . To correctly initialize the system, we use the nodes $(v_0, 0), (v_0, T) \in V_T$ as source and sink for the flow F and set the balance of the source accordingly to the number k of available vehicles, see (1b). For all internal nodes $(v, t) \in V_T \setminus \{(v_0, 0), (v_0, T)\}$, we use normal flow conservation constraints, see (1c), which also automatically ensure that a flow of value k is entering the sink (v_0, T) .

In order to encode the routing of each request $r_j \in \sigma$ we consider $|\sigma|$ commodities $f_1 \dots f_{|\sigma|}$. Each commodity f_j has a single source (x_j, p_j) where x_j is origin and p_j earliest

pickup time of the request r_j , also referred to as the commodity's origin, a single sink (y_j, q_j) where y_j is destination and q_j latest possible delivery time of r_j , also referred to as the commodity's destination, and a quantity z_j which is the load of the request r_j that must be routed along a single path from its source to its sink. In order to avoid that a request is partially served by a vehicle, we require that the quantity to be routed by each commodity f_j is equal to z_j but $f_j \in \{0, 1\}$.

To ensure that a request can be rejected and is not served more than once, we require that for each f_j at most one outgoing arc from the commodity's origin is chosen, see (1d). We use normal flow conservation constraints, see (1e), which also automatically ensure that for each commodity f_j the flow leaving the commodity's origin equals the flow entering its destination.

To ensure that the capacity of the VIPA is respected on all arcs $a \in A_M$, we couple the flows by

$$\sum_{r_j \in \sigma} f_j(a) \cdot z_j \leq \text{Cap} \cdot F(a) \quad \forall a \in A_M$$

such that the capacities for f_j on the transportation arcs are not given by constants but by a function. Note that due to these flow coupling constraints, the constraint matrix of the network is not totally unimodular (as in the case of uncoupled flows) and therefore integrality constraints for all flows are required (1h) and (1i), reflecting that solving the problem is \mathcal{NP} -hard.

Our objective function (1a) considers profits $p(j)$ on arcs $a \in \delta^-(x_j, p_j)$ for each commodity f_j to serve a request r_j , whereas all other arcs have zero profits. The costs correspond to the traveled distances $c(a) := d(u, v)$ on all arcs. The corresponding integer linear program is as follows:

$$\max \sum_{r_j \in \sigma} \sum_{a \in \delta^-(x_j, p_j)} p(j) f_j(a) - \sum_{a \in A_T} c(a) F(a) \quad (1a)$$

$$\text{s.t.} \quad \sum_{a \in \delta^+(v_0, 0)} F(a) = k \quad (1b)$$

$$\sum_{a \in \delta^-(v, t)} F(a) = \sum_{a \in \delta^+(v, t)} F(a) \quad \forall (v, t) \neq (v_0, 0), (v_0, T) \quad (1c)$$

$$\sum_{a \in \delta^-(x_j, p_j)} f_j(a) \leq 1 \quad \forall r_j \in \sigma \quad (1d)$$

$$\sum_{a \in \delta^-(v, t)} f_j(a) = \sum_{a \in \delta^+(v, t)} f_j(a) \quad \forall r_j \in \sigma \forall (v, t) \neq (x_j, p_j), (y_j, q_j) \quad (1e)$$

$$\sum_{r_j \in \sigma} f_j(a) \cdot z_j \leq \text{Cap} F(a) \quad \forall a \in A_M \quad (1f)$$

$$F(a) \geq 0 \quad \forall a \in A_T \quad (1g)$$

$$F(a) \in \mathbf{Z} \quad \forall a \in A_T \quad (1h)$$

$$f_j(a) \in \{0, 1\} \quad \forall a \in A_T, \forall r_j \in \sigma \quad (1i)$$

where $\delta^-(v, t)$ denotes the set of outgoing arcs of (v, t) , and $\delta^+(v, t)$ denotes the set of incoming arcs of (v, t) .

The integer linear program (1) solves the Offline TMP (where the whole sequence σ of requests is known at time $t = 0$) to optimality:

► **Theorem 2.** *The integer linear program (1) provides an optimal solution of the Offline TMP.*

4 Solving the Online TMP

To handle the online situation, where the requests in σ are released over time during a time horizon $[0, T]$, we propose a heuristic to solve a sequence of offline subproblems for certain time intervals $[t', T']$ within $[0, T]$ on accordingly modified time-expanded networks. A usual replan strategy is based on computing the optimal solution on the subsequence $\sigma(t')$ of requests released in each replanning step. Computing an optimal solution by multicommodity coupled flows is generally very slow and, thus, not applicable in online situations. In the proposed algorithm h-Replan, we thus use a heuristic to compute offline solutions on $\sigma(t')$. As experiments have shown that only a small percentage of arcs in G_T is used in the optimal solution while solving the Offline TMP, the idea is to reduce G_T to a network $G_R(t')$ containing only arcs which are taken in the optimal solution with high probability. Afterwards, we solve the flow problem on this reduced network $G_R(t')$. This does not lead to a globally optimal solution, but provides reasonable solutions in short time.

► **Algorithm 3** (h-Replan).

Input: $(M, \sigma, p, T, k, Cap)$

Output: σ_A , and tours $\Gamma^1, \dots, \Gamma^k$

1: initialize $t' = 0$, $\sigma_A = \emptyset$, $\sigma(t') = \{r_j \in \sigma : t_j = 0\}$, $\Gamma^i = (v_0, 0)$ for $1 \leq i \leq k$

2: WHILE $t' < T$ DO:

compute offline solution for σ_A , $\sigma(t')$, and $\Gamma^1, \dots, \Gamma^k$

perform the (modified) tours until new requests become known

update t' and $\sigma(t')$

3: return σ_A and $\Gamma^1, \dots, \Gamma^k$

To compute those offline solutions for the subsequences $\sigma(t')$, we build a reduced time-expanded network $G_R(t')$ based on $\sigma_A, \sigma(t')$ and the original network G that has the possible start positions of the VIPAs as source nodes in V_+ , internal nodes (v, t) for time points $t \in [t', T']$ relevant for the requests in $\sigma_A \cup \sigma(t')$, but far less arcs than G_T :

- To determine the possible source nodes in V_+ for the VIPAs from the current tours $\Gamma^1, \dots, \Gamma^k$, we proceed as follows. At the beginning, i.e. at time $t = 0$, we clearly have $(v_0, 0)$ as source for each VIPA. At any later time point t' , we have: If a VIPA is currently serving a request r_j , then (y_j, t_j^{drop}) is its source; if a VIPA is currently idle and situated at v , then (v, t') is its source.
- To determine the internal nodes and arcs in $A'_M \subseteq A_M$ and $A'_W \subseteq A_W$ which are taken in the optimal solution with high probability, we compute classic multicommodity flows with adjusted profits and costs taking only the request commodities into account, but not coupled to a VIPA flow. The reason is that we intend to construct “interesting” paths for the request commodities, starting from the commodity’s origin and ending at the commodity’s destination, without taking the route of the VIPAs into consideration:
 - a min cost multicommodity flow in G_T to determine a shortest path for the commodity of each request r_j from (x_j, p_j) to (y_j, q_j) ,
 - a max profit multicommodity flow in G_T to determine for each request r_j a path from (x_j, p_j) to (y_j, q_j) that has the potential to partially share paths of other commodities.
 In both cases, the constraint matrices are totally unimodular such that the computations can be done in short time, see [9] for details. Besides using the arcs with positive flow

from these two problems, we add further transport arcs from the destination of requests to reachable origins of other requests to ensure that requests can be served sequentially in one tour.

Thus, compared to the original time-expanded network $G_T = (V_T, A_T)$, we reduce in $G_R = (V'_T, A'_T)$ both the total number of nodes as well as of wait and transport arcs. We compute a transportation schedule by solving the max profit flow problem in $G_R(t')$ detailed in (2). Hereby, to keep previously accepted requests, we partition $\sigma(t')$ into the subsequences

- $\sigma_A(t')$ of previously accepted but until time t' not yet served requests and
- $\sigma_N(t') = \{r_j \in \sigma : t_j = t'\}$ of requests that are newly released at time t' .

$$\max \sum_{r_j \in \sigma(t')} \sum_{a \in \delta^-(x_j, p_j)} p(a) f'_j(a) - \sum_{a \in A'_T} c(a) F'(a) \quad (2a)$$

$$\text{s.t.} \quad \sum_{a \in \delta^+(v, t)} F'(a) = k(v) \quad \forall (v, t) \in V_+ \quad (2b)$$

$$\sum_{a \in \delta^-(v, t)} F'(a) = \sum_{a \in \delta^+(v, t)} F'(a) \quad \forall (v, t) \neq V_+, t < T' \quad (2c)$$

$$\sum_{a \in \delta^-(x_j, p_j)} f'_j(a) \leq 1 \quad \forall r_j \in \sigma_N(t') \quad (2d)$$

$$\sum_{a \in \delta^-(x_j, p_j)} f'_j(a) = 1 \quad \forall r_j \in \sigma_A(t') \quad (2e)$$

$$\sum_{a \in \delta^-(v, t)} f'_j(a) = \sum_{a \in \delta^+(v, t)} f'_j(a) \quad \forall r_j \in \sigma, \forall (v, t) \neq (x_j, p_j), (y_j, q_j) \quad (2f)$$

$$\sum_{r_j \in \sigma(t')} f'_j(a) \cdot z_j \leq \text{Cap} F'(a) \quad \forall a \in A'_M \quad (2g)$$

$$F'(a) \geq 0 \quad \forall a \in A'_T \quad (2h)$$

$$F'(a) \in \mathbf{Z} \quad \forall a \in A'_T \quad (2i)$$

$$f'_j(a) \in \{0, 1\} \quad \forall a \in A'_T, \forall r_j \in \sigma(t') \quad (2j)$$

where $A'_T = A'_W \cup A'_M$ and $k(v)$ denotes the number of VIPAs initially situated in v . Constraints (2e) ensure that previously accepted requests are served whereas constraints (2d) allow to reject newly released requests.

From the computed flows F' and f'_j in the reduced network $G_R(t')$, it is again straightforward to determine newly accepted requests and to construct (partial) tours $\Gamma^1, \dots, \Gamma^k$ for the VIPAs in the same way as for the offline situation.

5 Evaluation of online algorithms for the Online TMP

5.1 Competitive Analysis

It is standard to evaluate the quality of online algorithms with the help of competitive analysis. This can be viewed as a game between an online algorithm ALG and a malicious adversary who tries to generate a worst-case request sequence σ which maximizes the ratio between the online cost $\text{ALG}(\sigma)$ and the optimal offline cost $\text{OPT}(\sigma)$ knowing the entire request sequence σ in advance. ALG is called c -competitive for an online maximization problem if ALG produces for any request sequence σ a feasible solution with $\text{OPT}(\sigma) \leq c \text{ALG}(\sigma)$ for some given $c \leq 1$. The competitive ratio of ALG is the infimum over all c such that ALG is c -competitive.

In [12], we consider an *oblivious adversary* who knows the complete behavior of a (deterministic) online algorithm ALG and chooses a worst-case sequence for ALG. Hereby, an oblivious adversary is allowed to move VIPAs towards the origins x_j of not yet released requests r_j (but also has to respect the time windows $[p_j, q_j]$ to serve accepted requests r_j).

In [12], we showed that an oblivious adversary can force any (deterministic) online algorithm ALG for the Online TMP to reject all requests of a sequence while the adversary can accept and serve all requests, implying that ALG is not competitive.

Here, we consider a weaker adversary, called *non-abusive adversary*, who also knows the complete behavior of ALG and chooses a worst-case sequence for ALG, but is only allowed to move VIPAs towards origins (or destinations) of already released requests (and has also to respect the time windows).

We show that no (deterministic or non-deterministic) online algorithm ALG for the Online TMP is competitive against a non-abusive adversary, since the adversary can force ALG to accept at most one request and to reject all other requests of a sequence while the adversary can accept and serve all requests but one of the sequence.

► **Theorem 4.** *There is no competitive online algorithm for the Online TMP against a non-abusive adversary.*

Since the worst-case request sequence used to show the non-competitiveness result is only based on the reachability of requests, but not on a particular strategy of an online algorithm, we conclude:

► **Corollary 5.** *The online algorithm h-Replan is not competitive for the Online TMP against an oblivious or non-abusive adversary.*

5.2 Computational Results

This section deals with computational experiments for the optimal offline solutions of the (non-preemptive and preemptive) TMP and the two replan strategies, Replan studied for the non-preemptive case in [10, 12] and h-Replan proposed here for the preemptive case of the Online TMP. In fact, due to the very special request structures of the worst-case instances to prove the non-competitiveness of any online algorithm for the Online TMP, we can expect a better behavior of the proposed replan strategies for the Online TMP in average.

The computational results presented in this section support this expectation. They compare the total number of accepted (and thus served) requests by Replan and h-Replan with the optimal offline solutions OPT-NP for the non-preemptive case and OPT-P for the preemptive case. The computations use randomly generated instances with 20 stations, 5 to 10 VIPAs, time-horizons between 180 and 240 time units, and between 90 and 300 customer requests. These instances are based on the network from the industrial site of Michelin at Clermont-Ferrand and randomly generated request sequences resembling typical instances that occurred during an experimentation in Clermont-Ferrand performed from October 2015 until February 2016 [21].

The operating system for all tests is Linux CentOS with kernel version 2.6.32 clocked at 2.40 GHz, with 1 TB RAM. The approaches are implemented in Python and Gurobi 8.21 is used for solving the ILPs.

In the first resp. second set of 180 instances each, the requests have a random load between

- 4 and 10 (with 72% of the requests with a load above 5),
- 1 and 10 (with only 21% of the requests with a load above 5),

■ **Table 1** This table shows the percentage of improvement of the average number of accepted requests between the non-preemptive and preemptive optimal solutions and between Replan and h-Replan for the first set of instances.

req	T	k	OPT-NP	UB(OPT-P)	Imp (%)	Replan	h-Replan	Imp (%)
94	180	10	77	82,8	7,53	39	41,8	7,18
188	180	10	112	121,08	8,11	55	59,12	7,49
295	180	10	146,86	160,54	9,31	75,85	90,8	19,71
97	240	5	62,04	66,48	7,16	25,19	29,7	17,90
194	240	5	93,76	104,34	11,28	45,84	51,2	11,69
290	240	5	115,94	129,22	11,45	47,64	54,4	14,19

■ **Table 2** This table shows the percentage of improvement of the average number of accepted requests between OPT-NP and OPT-P and between Replan and h-Replan for the second set of instances.

req	T	k	OPT-NP	UB(OPT-P)	Imp (%)	Replan	h-Replan	Imp (%)
94	180	10	65,31	86,70	32,75	36,54	47,54	30,10
180	180	10	107,48	158,65	47,61	47,16	77,80	64,97
295	180	10	153,20	283,50	85,05	79,14	124,10	56,81
97	240	5	61,76	84,40	36,66	24,10	32,50	34,85
194	240	5	100,32	154,23	53,74	45,38	72,67	60,14
290	240	5	123,67	275,47	122,74	46,21	88,50	91,52

and in both cases VIPAs of capacity 10. The two replan strategies compute solutions within a reasonably short time (even for hReplan in less than 60 seconds in average for each replanning step).

As already reported in [10, 12], Replan achieves in average an acceptance rate of about 64% compared with OPT-NP for the first set of instances, and about 45% for the second. Our interest is whether or not allowing preemptive tours can significantly improve this acceptance rate.

Unfortunately, due to the long computation time for OPT-P, only an upper bound UB can be presented for most cases, obtained by computing an uncapacitated preemptive TMP with a time limit of four hours. Thus, we can mainly compare the improvements of the acceptance rate for OPT-NP and h-Replan only with this upper bound.

In the first set of instances, we observe that the percentage of improvement between OPT-NP and the upper bound of OPT-P is high (in average around 41% compared to UB), but it is not the case for the percentage of improvement between Replan and h-Replan (in average around 13%), see Table 1. The reason why there is no remarkable improvement in the acceptance rate is that in 72% of the requests, the load is greater than $Cap/2$ such that, in most of the times, the requests cannot be accumulated together to be served in one VIPA (recall that we allow vehicle preemption but not load preemption).

This changes in the second set of instances with in general smaller loads that allow us to serve more than one request simultaneously in one VIPA. Accordingly, we observe that the percentage of improvement between OPT-NP and OPT-P/UB increases to in average around 43% and between Replan and h-Replan to in average around 57%, (see Table 2). Detailed computational results are summarized in Table 3 and Table 4.

Note that computational results presented in Table 3 and Table 4 show only an upper bound for OPT-P. Therefore, this upper bound is sometimes far from the optimal solution

■ **Table 3** This table shows the computational results for the first set of 180 test instances of Replan respectively h-Replan in comparison to OPT-NP respectively to the upper bound of the optimal preemptive offline solution UB(OPT-P). The instances are grouped by the number of requests (1st column), the time horizon (2nd column) and the number of VIPAs (3rd column) with 30 instances per parameter set. Average values are shown for the total number $|\sigma_A|$ of accepted requests and for the total tour length TTL needed to serve the accepted requests. Finally, we provide the average runtime of Replan respectively h-Replan per recomputation step and the maximum runtime of the recomputation steps of Replan respectively h-Replan.

non-preemptive TMP									
			$ \sigma_A $			TTL		runtime (s)	
req	T	k	OPT-NP	Replan	ratio %	OPT-NP	Replan	AVG	MAX
94	180	10	77	52,13	67,7	667,5	424	0,49	1,6
188	180	10	112	70,45	62,9	831	580	3	10,83
295	180	10	146,86	97,2	66,19	1005	750,57	13,56	45,54
97	240	5	62,04	39,19	63,17	527,16	298,82	0,29	1,23
194	240	5	93,76	55,84	59,56	680,44	490	1,8	7,85
290	240	5	115,94	80,64	69,55	759,94	500,6	7,18	29,8
preemptive TMP									
			$ \sigma_A $			TTL		runtime (s)	
req	T	k	UB(OPT-P)	h-Replan	ratio %	OPT-P	h-Replan	AVG	MAX
94	180	10	83,72	62,21	74,31	678,1	456,54	2,18	9,76
188	180	10	150,08	76,17	50,75	875,43	600,62	6,29	38,77
295	180	10	232	110,54	47,65	1167,54	748,8	21,82	68,54
97	240	5	73,34	42,4	57,81	574,65	322,75	1,34	13,06
194	240	5	134,74	63,83	47,37	885,48	515,5	3,91	24,54
290	240	5	210,63	90,23	42,84	998,75	496,6	19,62	58,39

especially in the first set of instances, where the improvement between OPT-NP and the upper bound of OPT-P is high 41% due to the upper bound calculated by computing an uncapacitated preemptive TMP. Thus, the requests can be accumulated together, without considering their loads. This cannot be the case in the optimal solution. Note that in the first set of instance, the average acceptance ratio between Replan and OPT-NP is 65% while the average acceptance ratio between h-Replan and OPT-P/UB is 54% while in the second set of instance the average acceptance ratio between Replan and OPT-NP is 46% while the average acceptance ratio between h-Replan and OPT-P/UB is 43%. While Replan is not competitive in not competitive in theory, in practice it achieves a ratio about 2 compared to the optimal offline solution or the upper bound which is an acceptable ratio from a business point of view.

The computations in Table 5 use randomly generated instances with 10 stations, 2 to 3 VIPAs with capacity 10, time-horizon of 60 time units, and between 20 and 30 customer requests. In this set of 120 instances, the requests have a random load between

- 1 and 10 (with only 28% of the requests with a load above 5),

The two replan strategies compute solutions within a short time (less than 5 seconds in average) for each replanning step, therefore the average runtime is not shown in Table 5.

■ **Table 4** This table shows the computational results for the second set of instances.

non-preemptive TMP										
req	T	k	$ \sigma_A $			TTL		runtime (s)		
			OPT-NP	Replan	ratio %	OPT-NP	Replan	OPT-NP	AVG	MAX
94	180	10	65,31	36,54	55,95	667,5	416,7	12,6	0,68	5,32
188	180	10	107,48	47,16	43,888	831	596,35	181,23	2,69	12,45
295	180	10	153,2	79,14	51,66	1005	726,86	73456,5	12,67	27,42
97	240	5	61,76	24,1	39,02	527,16	279,15	697,75	0,86	6,45
194	240	5	100,32	45,38	45,24	680,44	504,7	846,5	3,7	14,6
290	240	5	123,67	46,21	37,37	759,94	527,45	116875,85	14,1	22,46
preemptive TMP										
req	T	k	$ \sigma_A $			TTL		runtime (s)		
			UB	h-Replan	ratio %	UB	h-Replan	UB	AVG	MAX
94	180	10	86,70	47,54	54,83	727,56	460,16	43824,50	2,58	11,58
188	180	10	158,65	77,80	49,04	930,75	649,67	125849,75	7,42	45,45
295	180	10	283(UB)	124,10	43,77	1175,25	878,25	97849(UB)	26,45	82,42
97	240	5	84,40	32,50	38,51	558,45	358,75	90470,67	1,36	14,36
194	240	5	154,23	72,67	47,12	825,74	609,40	156752,58	4,26	27,42
290	240	5	275(UB)	88,50	32,13	957,52	630,86	128417(UB)	21,78	65,80

In this set of instances, Replan achieves in average an acceptance rate of about 54% compared with OPT-NP, and h-Replan achieves in average an acceptance rate of about 70% compared with OPT-P (see Table 5).

6 Conclusion

Regarding the quality of the solutions obtained by the here proposed h-Replan strategy for the Online TMP, we summarize that

- in theory, h-Replan is (as any other online algorithm for the problem) not competitive since there is no finite c s.t. for all instances σ we have that $\text{OPT}(\sigma) \leq c \text{h-Replan}(\sigma)$, but
- in practice, h-Replan leads to a higher rate of accepted requests and, therefore, to a higher quality-of-service level for the fleet management than Replan constructing non-preemptive tours.

However, sometimes the transportation schedule returned by h-Replan contains preemptive tours which causes inconveniences for the users. Therefore, in order to handle the Online Taxi Mode Problem in the studied VIPAFLEET management system it is up to the operator to decide whether it is worth to have preemptive tours in order to increase the number of accepted requests, taking the ratio of request loads and VIPA capacities and, thus, the expected increase of the acceptance rate into account.

■ **Table 5** This table shows the computational results for the first set of 120 test instances of Replan respectively h-Replan in comparison to OPT-NP respectively to the optimal preemptive offline solution OPT-P. The instances are grouped by the number of requests (1st column), the time horizon (2nd column) and the number of VIPAs (3rd column) with 30 instances per parameter set. Average values are shown for the total number $|\sigma_A|$ of accepted requests and for the total tour length TTL needed to serve the accepted requests.

			NP-TaxiMP				
			$ \sigma_A $			TTL	
req	T	k	OPT-NP	Replan-NP	ratio %	OPT-NP	R-NP
18	60	2	11,23	6,58	58,59	95,7	52,85
26	60	2	16,42	8,43	51,34	118,46	65,17
17	60	3	13,57	8,1	59,69	124,78	71,63
28	60	3	20,32	9,74	47,93	138,6	76,25
			P-TaxiMP				
			$ \sigma_A $			TTL	
req	T	k	OPT-P	hReplan-P	ratio %	OPT-P	hReplan-P
18	60	2	14,43	9,64	66,81	90,37	85,83
26	60	2	22,73	16,61	73,08	103,54	91,37
17	60	3	15,72	10,95	69,66	111,35	92,61
28	60	3	25,43	18,13	71,29	123,74	105,75

References

- 1 Easymile, 2015. URL: <http://www.easymile.com>.
- 2 Ligier group, 2015. URL: <http://www.ligier.fr>.
- 3 Viaméca, 2015. URL: <http://www.viameca.fr/>.
- 4 Norbert Ascheuer, Sven O Krumke, and Jörg Rambau. *The online transportation problem: competitive scheduling of elevators*. ZIB, 1998.
- 5 Norbert Ascheuer, Sven O Krumke, and Jörg Rambau. Online dial-a-ride problems: Minimizing the completion time. In *STACS 2000*, pages 639–650. Springer, 2000.
- 6 Giorgio Ausiello, Esteban Feuerstein, Stefano Leonardi, Leen Stougie, and Maurizio Talamo. Competitive algorithms for the on-line traveling salesman. In *Workshop on Algorithms and Data Structures*, pages 206–217. Springer, 1995.
- 7 Giorgio Ausiello, Esteban Feuerstein, Stefano Leonardi, Leen Stougie, and Maurizio Talamo. Algorithms for the on-line travelling salesman 1. *Algorithmica*, 29(4):560–581, 2001.
- 8 Gerardo Berbeglia, Jean-François Cordeau, and Gilbert Laporte. Dynamic pickup and delivery problems. *European journal of operational research*, 202(1):8–15, 2010.
- 9 Sahar Bsaybes. *Modèles et algorithmes de gestion de flottes de véhicules VIPA*. PhD thesis, Université Clermont Auvergne, 2017.
- 10 Sahar Bsaybes, Alain Quilliot, and Annegret K Wagler. Fleet management for autonomous vehicles using flows in time-expanded networks. *Electronic Notes in Discrete Mathematics*, 62:255–260, 2017.
- 11 Sahar Bsaybes, Alain Quilliot, and Annegret K Wagler. Fleet management for autonomous vehicles: Online PDP under special constraints. *to appear on RAIRO - Operations Research*, 2018.

- 12 Sahar Bsaybes, Alain Quilliot, and Annegret K Wagler. Fleet management for autonomous vehicles using flows in time-expanded networks. *to appear on Journal of Advanced Transportation*, 2018.
- 13 Jean-François Cordeau and Gilbert Laporte. The dial-a-ride problem: models and algorithms. *Annals of Operations Research*, 153(1):29–46, 2007.
- 14 Samuel Deleplanque and Alain Quilliot. Transfers in the on-demand transportation: the DARPT Dial-a-Ride Problem with transfers allowed. In *Multidisciplinary International Scheduling Conference: Theory and Applications (MISTA)*, pages 185–205, 2013.
- 15 Anke Fabri and Peter Recht. Online dial-a-ride problem with time windows: an exact algorithm using status vectors. In *Operations Research Proceedings 2006*, pages 445–450. Springer, 2007.
- 16 Lester R Ford Jr and Delbert Ray Fulkerson. Constructing maximal dynamic flows from static flows. *Operations research*, 6(3):419–433, 1958.
- 17 Martin Groß and Martin Skutella. Generalized maximum flows over time. In *International Workshop on Approximation and Online Algorithms*, pages 247–260. Springer, 2011.
- 18 Martin Grötschel, Sven O Krumke, Jörg Rambau, Thomas Winter, and Uwe T Zimmermann. Combinatorial online optimization in real time. In *Online optimization of large scale systems*, pages 679–704. Springer, 2001.
- 19 Ronald Koch, Ebrahim Nasrabadi, and Martin Skutella. Continuous and discrete flows over time. *Mathematical Methods of Operations Research*, 73(3):301, 2011.
- 20 Jan Karel Lenstra and AHG Kan. Complexity of vehicle routing and scheduling problems. *Networks*, 11(2):221–227, 1981.
- 21 E Royer, F Marmoiton, S Alizon, D Ramadasan, M Slade, A Nizard, M Dhome, B Thuilot, and F Bonjean. Retour d’expérience après plus de 1000 km en navette sans conducteur guidée par vision.
- 22 Eric Royer, Jonathan Bom, Michel Dhome, Benoit Thuilot, Maxime Lhuillier, and François Marmoiton. Outdoor autonomous navigation using monocular vision. In *Intelligent Robots and Systems, 2005. (IROS 2005). 2005 IEEE/RSJ International Conference on*, pages 1253–1258. IEEE, 2005.
- 23 Jian Yang, Patrick Jaillet, and Hani Mahmassani. Real-time multivehicle truckload pickup and delivery problems. *Transportation Science*, 38(2):135–148, 2004.

The Steiner Multi Cycle Problem with Applications to a Collaborative Truckload Problem

Vinicius N. G. Pereira

Institute of Computing, University of Campinas (Unicamp)
Campinas - SP, Brazil
vinicius.pereira@ic.unicamp.br

Mário César San Felice

Department of Computing, Federal University of São Carlos (UFSCar)
São Carlos - SP, Brazil
felice@ufscar.br

Pedro Henrique D. B. Hokama

Production Engineering Department, Federal University of São Carlos (UFSCar)
São Carlos - SP, Brazil
hokama@ic.unicamp.br

Eduardo C. Xavier

Institute of Computing, University of Campinas (Unicamp)
Campinas - SP, Brazil
eduardo@ic.unicamp.br

Abstract

We introduce a new problem called Steiner Multi Cycle Problem that extends the Steiner Cycle problem in the same way the Steiner Forest extends the Steiner Tree problem. In this problem we are given a complete weighted graph $G = (V, E)$, which respects the triangle inequality, a collection of terminal sets $\{T_1, \dots, T_k\}$, where for each a in $[k]$ we have a subset T_a of V and these terminal sets are pairwise disjoint. The problem is to find a set of disjoint cycles of minimum cost such that for each a in $[k]$, all vertices of T_a belong to a same cycle. Our main interest is in a restricted case where $|T_a| = 2$, for each a in $[k]$, which models a collaborative less-than-truckload problem with pickup and delivery. In this problem, we have a set of agents where each agent is associated with a set T_a containing a pair of pickup and delivery vertices. This problem arises in the scenario where a company has to periodically exchange goods between two different locations, and different companies can collaborate to create a route that visits all its pairs of locations sharing the total cost of the route. We show that even the restricted problem is NP-Hard, and present some heuristics to solve it. In particular, a constructive heuristic called Refinement Search, which uses geometric properties to determine if agents are close to each other. We performed computational experiments to compare this heuristic to a GRASP based heuristic. The Refinement Search obtained the best solutions in little computational time.

2012 ACM Subject Classification Theory of computation \rightarrow Graph algorithms analysis, Theory of computation \rightarrow Routing and network design problems, Applied computing \rightarrow Transportation

Keywords and phrases Steiner Cycle, Routing, Pickup-and-Delivery, Less-than-Truckload

Digital Object Identifier 10.4230/LIPIcs.SEA.2018.26

Funding This work was supported by CNPq (proc. 154505/2015-3, 425340/2016-3, 304856/2017-7) and FAPESP (proc. 2017/11382-2, 2016/11082-6, 2015/11937-9, 2016/23552-7).



© Vinicius N. G. Pereira, Mário C. San Felice, Pedro H. D. B. Hokama, and Eduardo C. Xavier; licensed under Creative Commons License CC-BY

17th International Symposium on Experimental Algorithms (SEA 2018).

Editor: Gianlorenzo D'Angelo; Article No. 26; pp. 26:1–26:13

Leibniz International Proceedings in Informatics



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

1 Introduction

In this paper we present the Steiner Multi Cycle Problem (SMCP) and a restricted version of it, called Restricted Steiner Multi Cycle Problem (R-SMCP), which models a collaborative less-than-truckload problem with pickup and delivery. In the SMCP one is given a complete weighted metric graph $G = (V, E)$ and a collection of pairwise disjoint terminal sets $\{T_1, \dots, T_k\}$. The problem is to find a set of vertex disjoint cycles of minimum cost such that, for each $a \in [k]$, all vertices of T_a belong to the same cycle in a solution. In the R-SMCP we assume each T_a contains exactly two vertices, a pickup and a delivery, and that $\{T_1, \dots, T_k\}$ forms a partition of V . This problem models a collaborative less-than-truckload problem where companies have pickup and delivery locations, and are willing to collaborate in order to reduce their individual transportation costs, given by a solution where each company creates a separate route containing only their own pickup and delivery vertices. Notice that we do not assume an order between the pickup and delivery vertices, since we consider a recurrent route which a truck will use several times. Throughout the text we refer to each T_a as an agent that has associated with it the pickup and delivery vertices in T_a .

The SMCP is a generalization of the Steiner Cycle problem (SCP) in the same way as the Steiner Forest generalizes the Steiner Tree problem. The SCP was introduced by Salazar-González [18], where he analysed the polyhedral structure associated with the problem, introducing two lifting procedures to extend facet-defining inequalities from the travelling salesman polytope. Steinová [19] studied the approximability of the SCP, showing that the general problem in directed graphs does not admit an approximation algorithm with polynomial ratio in the input size, unless $P=NP$. Moreover, Steinová presents a $\frac{3}{2}$ -approximation algorithm for the case in which the input graph is undirected and metric.

As mentioned, the R-SMCP is related to vehicle routing problems, in particular to a collaborative less-than-truckload problem. Literature in vehicle routing problems is vast and considers very different constraints, as can be seen at [20] and [1]. The R-SMCP is similar to the traditional vehicle routing problem with pickup and delivery, for which Parragh et al. [16] presents a survey. The R-SMCP was also inspired by the Lane Covering Problem with less-than-truckload shipments (see Ergun et al. [4, 5]). The objective of this problem is to find a minimum cost set of directed cycles which covers a given set of arcs. Another related problem is the location-routing problem with simultaneous pickup and delivery, presented by Karaoglan et al. [13]. In this problem, one is given a graph with customer vertices and possible depot vertices. Each customer has a pickup and delivery demand to be served by a route that starts at an opened depot. Pickup demands have to be transported to the depot and delivery demands have to be delivered from it.

In the less-than-truckload Problem modeled by R-SMCP, we consider that the demand of each agent is periodically delivered and much smaller than the vehicle capacity, e.g., letters or small deliveries among mail companies, cash or promissory notes among banks.

Our contributions. We introduce the R-SMCP, prove that it is NP-hard, and present some algorithms to solve it. In particular, we present an effective heuristic that uses geometric properties to cluster agents, called Refinement Search, and compare it with a proposed GRASP (Greedy Adaptive Randomized Search Procedure) heuristic. Also, we show a 4-approximation to the general problem, SMCP.

The paper is organized as follows. Section 2 presents the formal description of the problems, an ILP formulation and a 4-approximation algorithm for the SMCP. Section 3 presents heuristics we used to solve the Travelling Salesman Problem. Sections 4 and 5 describe the proposed methods to solve R-SMCP. Finally, Section 6 presents the computational results.

2 The Steiner Multi Cycle Problem

In this section we present a formal definition for the SMCP and prove that even its restricted version is NP-Hard. We also show an integer linear programming formulation for the problem.

The input for the SMCP is a complete graph $G = (V, E)$, with a metric cost function $c : E \rightarrow \mathbb{R}^+$, and a collection of terminals sets $\mathcal{T} = \{T_1, \dots, T_k\}$, where $T_a \subseteq V$ and $T_a \cap T_b = \emptyset$ for a and b in $[k]$ with $a \neq b$. The problem is to find a set \mathcal{C} of vertex disjoint cycles such that, for each $a \in [k]$, all terminals in T_a belong to a same cycle. The cost of a solution \mathcal{C} is the sum of the edges' costs used in its cycles, i.e., $\sum_{C \in \mathcal{C}} \sum_{e \in C} c_e$. The goal is to find a minimum cost solution.

In the restricted version of the problem, R-SMCP, the terminals collection $\{T_1, \dots, T_k\}$ forms a partition of V and each T_a contains exactly two vertices, a pickup and a delivery. In this case, we assume that the vertex set contains $2n$ vertices, $V = \{1, 2, \dots, 2n\}$ where each $T_a = \{a, a + n\}$ for $a \in [n]$. We define a set of agents $A = \{1, 2, \dots, n\}$, such that each agent $a \in A$ has a pickup point p_a on vertex a and a delivery point d_a on vertex $a + n$.

Note that, as G is a metric graph, there is always an optimum solution which does not use Steiner vertices. To see this, suppose an optimum solution with a cycle $C = (u_1, u_2, \dots)$, where some u_j is a Steiner vertex. We can connect vertices u_{j-1} and u_{j+1} directly and remove u_j from C , resulting in a solution with cost at most the cost of C , due to the triangle inequality. Moreover, this holds to SMCP and R-SMCP.

We can show that the SMCP is NP-Hard since it generalizes the minimum Steiner Cycle problem. In fact, we proved that even R-SMCP is NP-Hard.

► **Theorem 1.** *The Restricted Steiner Multi Cycle Problem is NP-Hard.*¹

2.1 An ILP Formulation and its Linear Relaxation

Given an instance (G, c, T) of the SMCP, consider a function $f : 2^V \rightarrow \{0, 1\}$ such that for each non-empty set $S \subset V$ we have $f(S) = 1$ if, and only if, for some $T_a \in \mathcal{T}$ we have that $S \cap T_a \neq \emptyset$ and $T_a \not\subseteq S$, i.e, S is a cut that separates terminals in T_a . With abuse of notation we use $i \in T$ to denote a vertex that is a terminal. The SMCP can be formulated with the following integer linear program:

$$\min \quad \sum_{e \in E} c_e x_e \quad (1)$$

$$\text{s.t.} \quad \sum_{e \in \delta(i)} x_e = 2 \quad \forall i \in \mathcal{T} \quad (2)$$

$$\sum_{e \in \delta(S)} x_e \geq 2f(S) \quad \emptyset \neq S \subset V \quad (3)$$

$$x_e \in \{0, 1, 2\} \quad e \in E, \quad (4)$$

where $\delta(S)$ denotes the set of edges having exactly one end point in S . The variable x_e indicates if an edge is used in the solution, constraint (2) assures that exactly one cycle covers each terminal, constraint (3) assures that vertices belonging to a terminal set $T_a \in \mathcal{T}$ are connected, and constraint (4) allows each edge to be used at most twice, since in the case where T_a has just two vertices, a single cycle between them is a valid solution.

Relaxing the integrality constraints (4) we obtain a linear program useful to find lower bounds for the SMCP and for its restricted version, R-SMCP. Note that the number of constraints (3) is exponential. However, it is possible to solve the relaxed LP in polynomial time by solving the separation problem in polynomial time [9]. We find violated constraints in polynomial time by solving maximum flow problems between each pair of vertices in a same T_a , through the use of Gomory-Hu trees [8, 11].

¹ Proof omitted due to space constraints.

2.2 A 4-approximation Algorithm for the Steiner Multi Cycle Problem

A 2-approximation algorithm for the Steiner forest problem was presented by Goemans and Williamson [7]. We use this algorithm to obtain a 4-approximation algorithm for the SMCP.

Given an instance (G, c, \mathcal{T}) of the SMCP, we use the 2-approximation algorithm of [7] to obtain a Steiner forest F with cost $c(F) \leq 2\text{OPT}(G)$, where $\text{OPT}(G)$ is the cost of an optimal solution for the SMCP. Duplicate each edge of F , and then find an Eulerian circuit for each component of F . The cost of all Eulerian circuits is limited by $2c(F) \leq 4\text{OPT}(G)$. Finally, for each component, perform shortcuts in order to transform each Eulerian Circuit into a cycle, obtaining a valid solution \mathcal{C} for the SMCP with cost at most $4\text{OPT}(G)$.

► **Proposition 2.** *There is a 4-approximation algorithm for the Steiner Multi Cycle Problem.*

In the remaining of the text we present heuristics for the R-SMCP, since our main interest is in solving the related collaborative less-than-truckload problem with pickup and delivery.

3 TSP Heuristics

In this section we present the TSP heuristics that are used by our algorithms. All of them build a cycle cover by first choosing a promising agent partition, and then transforming each part into a cycle, by using a TSP heuristic. A partition of a set A is a collection $\{C_1, \dots, C_k\}$, where each $C_i \subseteq A$, $\cup_{i=1}^k C_i = A$, and for each pair of different parts C_i and C_j we have $C_i \cap C_j = \emptyset$. We call each set C_i a cluster or a part of A , for $i \in [k]$.

Given a cluster $C \subseteq V$, we use two heuristics to find a minimum cost TSP of C . One is the Nearest Insertion algorithm and the other is the Christofides algorithm. We apply the 2-OPT local search over the solution found by both heuristics. A detailed description of these heuristics can be found in [2]. The Nearest Insertion has a time complexity of $O(n^2)$, while Christofides has time complexity $O(n^3 \lg(n))$, where n is the number of vertices. The 2-OPT local search can have an exponential time complexity in the worst case. However, it is usually fast when the initial solution is good.

4 Refinement Search Heuristic

In this section we describe a deterministic heuristic, called Refinement Search, for the Restricted Steiner Multi Cycle Problem. We first define a measure of proximity, based on geometric regions, between an agent and any vertex of the graph G . This proximity measure is used to establish a neighborhood relation between each pair of agents, which determines if the agents are close to each other. From this, we obtain a partition of the agents and, for each part, we use a TSP heuristic to obtain a cycle that covers all its agents.

4.1 Region types

Given an agent a in A and a non-negative real number r , we define three types of regions.

Circular Intersection: denoted as $R_I(a, r)$, is defined by the intersection of two circles with radius r , one with center in p_a and other with center in d_a . More precisely, a vertex v is in the circular intersection region $R_I(a, r)$ if, and only if, $c(v, p_a) \leq r$ **and** $c(v, d_a) \leq r$.

Circular Union: denoted as $R_U(a, r)$, is defined by the union of two circles with radius r , one with center in p_a and other with center in d_a . More precisely, a vertex v is in the circular union region $R_U(a, r)$ if, and only if, $c(v, p_a) \leq r$ **or** $c(v, d_a) \leq r$.

Elliptical: denoted as $R_E(a, r)$, is defined by the ellipse with focal points p_a and d_a , and eccentricity $c(p_a, d_a)/2r$. More precisely, a vertex v is in the elliptical region $R_E(a, r)$ if, and only if, $c(v, p_a) + c(v, d_a) \leq 2r$.

In the algorithm, we use a factor $\alpha \in \mathbb{R}^+$ to define the radius used by all agents. Given an α value, an agent a will have its regions defined with radius $r_a = \alpha \cdot c(p_a, d_a)$, where $c(p_a, d_a)$ is the edge cost connecting the agent's vertices in T_a .

It is not hard to see that, for any type of region, if we increase the radius which defines it, the region can only increase with new vertices being added to it.

► **Lemma 3.** *Given an agent $a \in A$ and two radius $r, r' \in \mathbb{R}^+$, with $r \leq r'$, we have that $R_x(a, r) \subseteq R_x(a, r')$, for any region type $R_x \in \{R_I, R_E, R_U\}$.*

Given an agent, its circular intersection region is contained in its elliptical region which is contained in its circular union region, as long as these are defined with the same radius.

► **Lemma 4.** *Given $a \in A$ and $r \in \mathbb{R}^+$, we have that $R_I(a, r) \subseteq R_E(a, r) \subseteq R_U(a, r)$.*

While the concept of region allows us to relate an agent with a vertex, in order to establish a pairwise relationship among agents, we introduce the concept of neighborhood.

4.2 Neighborhood types

Given agents a and b in A , with regions R_a and R_b , we define two types of neighborhood.

Total: agents a and b are total neighbors if p_b and d_b are in R_a , or if p_a and d_a are in R_b .

Partial: agents a and b are partial neighbors if p_b or d_b are in R_a , or if p_a or d_a are in R_b .

Intuitively, two agents are total neighbors if both vertices of an agent are inside the other's region. Similarly, two agents are partial neighbors if at least one vertex from an agent is inside the other's region. Thus, every total neighbors are partial neighbors. Also, note that the neighborhood relation is symmetric.

Now that the pairwise relationship among agents is defined, we introduce the concept of component in an auxiliary graph to establish a partition of agents.

4.3 Auxiliary Graph and Components

Take a set of agents A , a collection of regions \mathcal{R} , such that each agent $a \in A$ has a region $R_a \in \mathcal{R}$, and a neighborhood type $N_y \in \{N_T, N_P\}$, where N_T stands for the Total neighborhood and N_P for the Partial neighborhood. Consider an auxiliary graph in which there is a vertex for each agent, and there is an edge between two vertices if, and only if, the corresponding agents are neighbors. Notice that the connected components of this auxiliary graph form a partition \mathcal{P} on the set of agents. Thus, from now on we use the words component and part (of the partition) interchangeably to refer to the agents of a same component of this auxiliary graph.

When considering partitions, there is an interesting property called refinement. We say that a partition \mathcal{P} is finer than \mathcal{P}' (or that \mathcal{P}' is coarser than \mathcal{P}) if, and only if, every set in \mathcal{P} is uniquely contained by some set of \mathcal{P}' . Note that the refinement property is transitive.

The following lemma shows a relationship between regions and refinement.

► **Lemma 5 (Region Refinement).** *Consider a set of agents A , two collections of regions \mathcal{R} and \mathcal{R}' , each containing one region R_a for each agent $a \in A$, and a neighborhood type $N_y \in \{N_T, N_P\}$. Suppose that, for each agent $a \in A$, $R_a \subseteq R'_a$, where $R_a \in \mathcal{R}$ and $R'_a \in \mathcal{R}'$. The components obtained using \mathcal{R} and \mathcal{R}' , with the same neighborhood type N_y , induce partitions \mathcal{P} and \mathcal{P}' such that \mathcal{P} is finer than \mathcal{P}' .*

Due to Lemmas 3 and 4, the previous lemma implies that, when all other parameters are constant, if the radius factor α increases, or the region type R_I changes to R_E , or R_E changes to R_U , then we have a new partition which is coarser than the previous one.

The following lemma shows a relationship between neighborhood types and refinement.

► **Lemma 6 (Neighborhood Refinement).** *Consider a set of agents A , a collection of regions \mathcal{R} , and neighborhood types N_T and N_P . The components obtained using \mathcal{R} , with neighborhood types N_T and N_P , induce partitions \mathcal{P}_T and \mathcal{P}_P such that \mathcal{P}_T is finer than \mathcal{P}_P .*

4.4 Dynamic Programming Algorithm for the Refinement Search

Given an input (G, c, T) of the R-SMCP, consider a partition $\mathcal{P}_{(\alpha, R_x, N_y)}$ induced by radius factor $\alpha \in \mathbb{R}^+$, region type R_x and neighborhood type N_y . We denote by $S_{(\alpha, R_x, N_y)}$ the solution for R-SMCP obtained by using a TSP heuristic to construct a cycle on each component of $\mathcal{P}_{(\alpha, R_x, N_y)}$. Moreover, given a component $C \in \mathcal{P}_{(\alpha, R_x, N_y)}$, we denote by $S_{(\alpha, R_x, N_y)}(C)$ the solution of $S_{(\alpha, R_x, N_y)}$ restricted to the agents in C .

The idea of the algorithm is to construct partitions of the agents using different types of regions, different values of α , and different types of neighborhood. First, let's describe how we choose the different values of α . We fix a region type R_x and neighborhood type N_y . For each pair of agents a and b , we find the smallest factor α_{ab} such that there is an edge between a and b in the auxiliary graph. Notice that, if we use a radius factor $\alpha_{min} = \min_{a, b \in A} \alpha_{ab} - \epsilon$ then $|A|$ components exist in the auxiliary graph. We construct a list of size $O(|A|^2)$ of radius factors containing α_{min} and α_{ab} for each pair of agents $a, b \in A$. A shorter list of radius factors is constructed as follows. Consider the list of factors sorted in increasing order, where the first value α_{min} is inserted in the beginning of the list. For the remaining values of factors, if its use decreases the number of components in the current auxiliary graph, then this α value is included in the list. Thus, the shorter list of radius factors contains $|A|$ values, since we start with $|A|$ components and finish with just a single component.

For each region type R_x and neighborhood type N_y we have a list of radius factors. So, there are at most $6|A|$ radius factors which we save in the list $\alpha = [\alpha_1, \alpha_2, \dots]$ arranged in increasing order. A basic idea for a heuristic to solve the R-SMCP is to generate a partition $\mathcal{P}_{(\alpha, R_x, N_y)}$ for each region type R_x , neighborhood type N_y , and radius factor α in the final list of factors. Then, for each component in $\mathcal{P}_{(\alpha, R_x, N_y)}$ construct a cycle using a TSP heuristic obtaining a solution $S_{(\alpha, R_x, N_y)}$. Thus, we could return the best solution found considering all these $6|A|$ solutions.

However, we can find better solutions using the refinement property, since it allows us to mix solutions of different region types, neighborhood types and radius factors. For example, consider an instance with 10 agents where, if we use $\alpha = 0.5$, region R_U , and neighborhood N_P , we obtain the partition $\mathcal{P}(0.5, R_U, N_P) = \{\{a_1, a_5, a_6\}, \{a_2, a_3, a_4, a_7\}, \{a_8, a_9, a_{10}\}\}$. From Lemmas 3, 5 and 6, we can obtain finer solutions by changing either the region type, or the neighborhood type, or the value of α . Thus, we can obtain the following finer solutions $\mathcal{P}(0.5, R_I, N_P) = \{\{a_1, a_5\}, \{a_6\}, \{a_2, a_3\}, \{a_4, a_7\}, \{a_8, a_9, a_{10}\}\}$, and $\mathcal{P}(0.5, R_U, N_T) = \{\{a_1, a_5, a_6\}, \{a_2, a_3, a_4\}, \{a_7\}, \{a_8, a_9\}, \{a_{10}\}\}$, and $\mathcal{P}(0.4, R_U, N_P) = \{\{a_1\}, \{a_5, a_6\}, \{a_2, a_3\}, \{a_4, a_7\}, \{a_8, a_9, a_{10}\}\}$. Then, when finding the best solution for $\mathcal{P}(0.5, R_U, N_P)$ we could use $\{a_2, a_3\}, \{a_4, a_7\}$ from $\mathcal{P}(0.4, R_U, N_P)$ if the two cycles have a smaller cost than the single cycle for $\{a_2, a_3, a_4, a_7\}$ obtained by $\mathcal{P}(0.5, R_U, N_P)$. This way the best solution could be $\mathcal{P}(0.5, R_U, N_P)^* = \{\{a_1, a_5, a_6\}, \{a_2, a_3\}, \{a_4, a_7\}, \{a_8, a_9, a_{10}\}\}$.

Since the refinement property is transitive, we can recurse further until the finest possible level, checking which parts of a solution should be replaced by smaller parts of finer solutions.

Algorithm 1: Refinement Search.

Input : (G, c, T) and a vector $\alpha = [\alpha_1, \alpha_2, \dots, \alpha_{6|A|}]$ radius factors in increasing order.
Output : A cycle cover \mathcal{C} .

```

1.1 foreach  $\alpha_i, i \in [1, 2, \dots, 6|A|]$  do
1.2   foreach neighbor type  $N_y \in (N_T, N_P)$  do
1.3     foreach region type  $R_x \in (R_I, R_E, R_U)$  do
1.4       foreach  $C \in S_{(\alpha_i, R_x, N_y)}$  do
1.5          $C_1 \leftarrow S_{(\alpha_{i-1}, R_x, N_y)}^*(C)$ 
1.6          $C_2 \leftarrow S_{(\alpha_i, R_{x-1}, N_y)}^*(C)$ 
1.7          $C_3 \leftarrow S_{(\alpha_i, R_x, N_{y-1})}^*(C)$ 
1.8          $S_{(\alpha_i, R_x, N_y)}^* \leftarrow (S_{(\alpha_i, R_x, N_y)} \setminus C) \cup \min\_cost\{C, C_1, C_2, C_3\}$ 
1.9 return  $S_{(\alpha_{6|A|}, R_P, R_U)}^*$ 

```

Let the region types be organized from finer to coarser, i.e., (R_I, R_E, R_U) and the same for the neighborhood types, i.e., (N_T, N_P) . For a set of agents C , remember that $S_{(\alpha, R_x, N_y)}(C)$ is the solution obtained from $\mathcal{P}_{(\alpha, R_x, N_y)}$ restricted to the agents in C . Formally, we want to find the best solution for all agents which can be described by the recurrence

$$S_{(\alpha_j, R_x, N_y)}^*(A) = \sum_{C \in \mathcal{P}_{(\alpha, R_x, N_y)}} \min \begin{cases} cost(S_{(\alpha_j, R_x, N_y)}(C)) \\ cost(S_{(\alpha_j, R_{x-1}, N_y)}^*(C)) \\ cost(S_{(\alpha_j, R_x, N_{y-1})}^*(C)) \\ cost(S_{(\alpha_{j-1}, R_x, N_y)}^*(C)) \end{cases}$$

where, for each component $C \in \mathcal{P}_{(\alpha, R_x, N_y)}$ we choose the best solution which can be obtained by considering the current solution $S_{(\alpha_j, R_x, N_y)}(C)$ using current radius factor, region type and neighborhood type, and the best solutions which are finer for this set of agents C .

The Algorithm 1 shows the pseudocode of a dynamic programming algorithm called Refinement Search. The algorithm constructs solutions in a bottom up fashion, from the base case with the finest α , neighborhood type, and region type to the coarsest ones. For each component C in the standard solution $S_{(\alpha, R_x, N_y)}$ of an iteration of loops (1.1 - 1.3), the algorithm checks if in a finer solution, the agents in C were covered with a smaller cost. The best solution for C among the current solution and the finer ones, is set on line 1.8.

5 GRASP Heuristic

In this section we present our GRASP based heuristic, which uses the Greedy Randomized Adaptive Algorithm described in Section 5.2 to generate initial solutions. After finding an initial solution, it performs a local search with the algorithm described in Section 5.3. The GRASP heuristic repeats this process until a stopping criteria occurs, which can be a maximum running time or a maximum number of iterations. Then, it returns the best solution found. The meta-heuristic GRASP was introduced by Feo and Resende [6] and it has been successfully applied to several combinatorial optimization problems [17]. More information is available in Resende and Ribeiro [17].

5.1 Basic Definitions for the GRASP Heuristic

The GRASP heuristic uses a contracted graph to find a partition of the agents set. This contracted graph is built using a measure of distance between each pair of agents, thus, we propose several ways to determine this measure of distance.

Given an input (G, c, T) of the R-SMCP, each agent $a \in A$ has a pickup vertex p_a and a delivery vertex d_a . The contracted graph $G_A = (A, E_A)$ is a complete weighted graph, in which each agent $a \in A$ corresponds to a single vertex and the distance between agents a and b , is defined according to one of the following six types of distances:

Distance 1 is the minimum cost of edges necessary to complete the cycle containing edges (p_a, d_a) and (p_b, d_b) . More formally, $\text{dist}_1(a, b) = \min\{c(p_a, p_b) + c(d_a, d_b), c(p_a, d_b) + c(d_a, p_b)\}$. This distance is symmetric and satisfies the triangle inequality.

Distance 2 is 0 if $a = b$, or the minimum cost of building a cycle that contains the pickup and delivery vertices of agents a and b . More formally, for $a \neq b$,

$$\text{dist}_2(a, b) = \min \left\{ \begin{array}{l} \text{dist}_1(a, b) + c(p_a, d_a) + c(p_b, d_b), \\ c(p_a, d_b) + c(d_b, d_a) + c(p_b, d_a) + c(p_a, p_b) \end{array} \right\}$$

This distance is symmetric and satisfies the triangle inequality.

Distance 3 is 0 if $a = b$, or the minimum cost to connect pickup and delivery vertices of a to the pickup or delivery vertex of b . More formally, for $a \neq b$,

$$\text{dist}_3(a, b) = \min \left\{ \begin{array}{l} c(p_a, p_b) + c(d_a, p_b), \\ c(p_a, d_b) + c(d_a, d_b) \end{array} \right\}$$

This distance is not symmetric and does not satisfy the triangle inequality.

Distance 4 is the symmetric version of dist_3 , i.e., the minimum cost between applying dist_3 to (a, b) and to (b, a) . More formally, $\text{dist}_4(a, b) = \min\{\text{dist}_3(a, b), \text{dist}_3(b, a)\}$. This distance is symmetric, but does not satisfy the triangle inequality.

Distance 5 is the minimum cost of edges necessary to connect a vertex from agent a to a vertex from agent b . More formally, $\text{dist}_5(a, b) = \min\{c(p_a, p_b), c(p_a, d_b), c(d_a, p_b), c(d_a, d_b)\}$. This distance is symmetric, but does not satisfy the triangle inequality.

Distance 6 is the cost of a minimum path on the contracted graph using Distance 5 as edges costs. More formally, let G_{A5} be the contracted graph using Distance 5, $\text{dist}_6(a, b) = \min_{P \in \mathcal{P}}\{c(P) \mid a \in P, b \in P\}$, where \mathcal{P} is the set of all paths in G_{A5} and $c(P)$ is the cost of path P . This distance is symmetric and satisfies the triangle inequality.

5.2 Greedy Randomized Adaptive Algorithm

The Greedy Randomized Adaptive algorithm, presented in Algorithm 2, creates n different partitions of the agents set, starting with a partition containing just one cluster with all agents. The algorithm iteratively constructs a new partition, increasing by one the number of clusters in it, until a final partition containing n clusters with isolated agents is created. For each partition, the algorithm creates a cycle cover by using the TSP heuristics described in Section 3 to construct a cycle for each cluster in the partition. Among all cycle covers created, the one with minimum cost is returned as solution. The algorithm starts by choosing at random an initial head (line 2.1) and by creating an initial cluster C_1 containing all vertices (line 2.3). Then, it computes the cost of the corresponding cycle cover of this initial clustering (line 2.4) and sets this solution as the best one so far. In any iteration of the main loop (lines 2.5 - 2.17), we begin with a partition with clusters C_1, \dots, C_{k-1} and we want to construct a new partition C_1, \dots, C_k , with one more cluster. Each cluster C_i has a special vertex denominated head, denoted by h_i . The algorithm proceeds by choosing a new head h_k , which is chosen as the agent that is farthest from its current head (lines 2.6 - 2.7). Then the algorithm re-creates the clusters C_1, \dots, C_k containing only their respective heads and, for each agent a which is not a head, it is re-assigned to the cluster of minimum distance (lines 2.9–2.14). The distance

Algorithm 2: Greedy Randomized Adaptive.

Input : (G, c, A) , a distance function dist , and a threshold T .
Output : A cycle cover \mathcal{C} .

```

2.1 initial_head  $\leftarrow$  a random vertice  $\in A$ 
2.2 initial_head becomes the head of cluster  $C_1$ ;
2.3 All agents are assigned to  $C_1$ ;
2.4 best_cover  $\leftarrow$  a cycle cover computed from  $C_1$ ;
2.5 for  $k \leftarrow 2$  to  $n$  do
2.6   new_head  $\leftarrow$  the agent that is farthest from its closest head;
2.7   new_head becomes a head of cluster  $C_k$ ;
2.8   Remove all agents that are not a head from their clusters and put them on a list  $L$ 
2.9   while there are elements in  $L$  do
2.10     Let min_dist and max_dist be the minimum and maximum distance of any agent in  $L$ 
2.11     to any cluster, respectively;
2.11     Create RCL list with the agents that are in a distance less than or equal to
2.11     (min_dist +  $T(\text{max\_dist} - \text{min\_dist})$ );
2.12     Randomly choose an agent  $a$  in RCL;
2.13     Put the  $a$  in its closest cluster;
2.14     Remove  $a$  from  $L$ ;
2.15   Let  $\mathcal{C}$  be a cycle cover computed from clusters  $C_1$  to  $C_k$ ;
2.16   if cost of  $\mathcal{C}$  is less than cost of best_cover then
2.17     best_cover  $\leftarrow$   $\mathcal{C}$ 
2.18 return best_cover

```

of agent a to a cluster C is defined as $\text{dist}(a, C) = \min_{b \in C} \{\text{dist}(a, b)\}$. Since we want a greedy randomized initial solution, instead of attributing each agent to the closest cluster in a deterministic greedy fashion, the heuristic creates a Restricted Candidate List (RCL) with the agents that are close to some cluster. More precisely, the RCL is created as follows, let *min_dist* and *max_dist* be the minimum and maximum distance, respectively, of any agent to any cluster. Given a threshold T , which is a value between 0 and 1, the algorithm constructs the RCL (lines 2.10 - 2.11) as the set containing every agent whose distance to a cluster is at most ($\text{min_dist} + T(\text{max_dist} - \text{min_dist})$). Then the algorithm chooses, uniformly at random, one of these agents, and assigns it to its closest cluster (lines 2.12 - 2.13). Notice that the minimum distance of the remaining agents in L has to be updated, since they may now become closer to the cluster where agent a was inserted. The algorithm proceeds until all agents are assigned to some cluster in $\{C_1, \dots, C_k\}$. Finally, the algorithm obtains a solution for the R-SMCP by running a TSP heuristic for each cluster (line 2.15) and it saves this solution if it is better than the current best one (lines 2.16 - 2.17).

5.3 Local Search

The Local Search algorithm begins with a current solution and then generates a set of neighbour solutions. If one of these has a cost reduction compared to the current solution, then the algorithm updates the current solution with the best neighbour solution found. It repeats this process until there is no significant improvement.

The algorithm used to find the best solution among the neighbour solutions is presented in Algorithm 3. The set of neighbor solutions \mathcal{S} initially contains only the current solution (line 3.1). For each agent a , the algorithm builds several new solutions as follows. In one of the new solutions agent a is removed from its current cycle and is left alone in an isolated cycle (line 3.3). The other solutions are constructed by removing agent a from its current cycle and inserting it in another cycle (lines 3.5 - 3.6). In the Computational Results section we use a first improvement variant of this algorithm, where it returns the first generated

Algorithm 3: Find Best Neighbor Solution

Input : (G, c, A) and an initial solution S' .
Output : A solution.

3.1 Let \mathcal{S} be a set of solutions initially containing only S'

3.2 **foreach** a *in* A **do**

3.3 $S \leftarrow \text{isolate}(S', a)$

3.4 Add S to \mathcal{S}

3.5 **foreach** cycle C *in* S' **do**

3.6 $S \leftarrow \text{swap_cycle}(S', C, a)$

3.7 Add S to \mathcal{S}

3.8 **return** the best solution in \mathcal{S}

solution that improves the current one S' . The first improvement variant obtained better results since the complete local search has to explore a costly neighbourhood, where for each swap of agent a , we need to run the TSP heuristics to rebuild a cycle.

6 Computational Results

All computations were performed in a single thread of an Intel Core™i7-4790 processor at 3.60GHz with 16GB of RAM, with Linux. The algorithms were implemented in C++, using the Graph library Lemon [14]. To solve the linear relaxation we used the Gurobi solver [10].

6.1 Instances

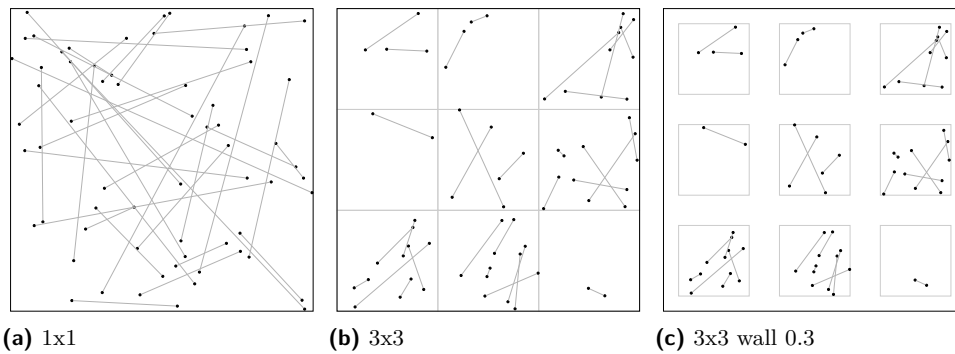
Since, to the best of our knowledge, there is no previous work on the R-SMCP, we have tested our algorithms on two types of instances: type 1 is a set of instances from the multi-commodity one-to-one pickup-and-delivery traveling salesman problem (m-PDTSP), and type 2 is a set of newly random generated instances.

Hernández-Pérez and Salazar-González [12] generated a set of instances to the m-PDTSP. For each instance, they generated $2n - 2$ uniformly random points with coordinates from -500 to 500 , a vertex in position 0 with coordinates $(0, 0)$ and a vertex in position $2n - 1$ also with coordinates $(0, 0)$ (corresponding to Class 3 of [12]). We only take into account the vertex distribution of these instances. For each $i \in \{0, \dots, n - 1\}$, we consider vertex i as a pickup point of an agent and $i + n$ as its corresponding delivery point. The instances have 6, 11 and 16 agents, with a total of 210 instances. This set of instances is the type 1.

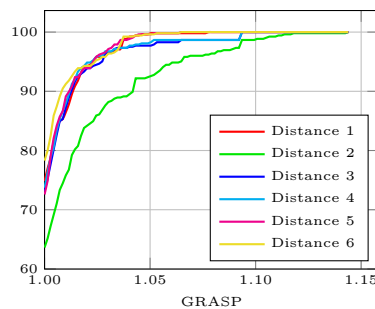
We also generated a set of instances having 16, 32, 64, 128 and 256 agents, where vertices corresponds to points distributed in a square of dimensions 100000×100000 . The square is divided in 1×1 , 2×2 , 3×3 , 4×4 and 5×5 frames, and each pair of pickup and delivery is in the same frame. The space between frames, which we call a wall, has 0%, 10%, 20%, 30% or 40% of the frame's size. The wall can be seen as a rectangle separating the different frames (see Figure 1). The location of each point is chosen uniformly at random. For each combination of number of agents, number of frames, and wall size, 3 instances were generated with different seeds. Notice that for the instances with division 1×1 there is no wall. Therefore, we generated a set of 315 instances. This set of instances is the type 2 and it can be found in the Laboratory of Optimization and Combinatorics website [15].

6.2 Final Results

We first compared the GRASP algorithm in different versions, each version using a different distance measure between agents (see Section 5.1). Figure 2 shows the performance profiles [3] comparing the different versions of the GRASP algorithm. This graphic is build as follows.



■ **Figure 1** Instances randomly generated with 16 agents, in (a) a 1×1 frame with no wall, in (b) a 3×3 frame with 0% wall and in (c) a 3×3 frame with 30% wall.

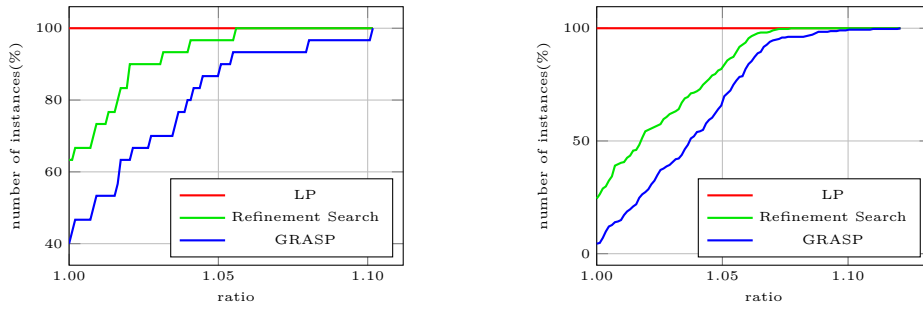


■ **Figure 2** Comparison between the GRASP algorithm running with different distances. The line that appears closer to the upper-left corner gives the best results, in this case distance 6.

First of all, we execute each version of the algorithm on all instances, obtaining the solution cost of each algorithm version for each instance. The performance profile graphic consists of a curve for each algorithm version. A curve of an algorithm version is constructed in the following manner: for each instance, we compute the ratio between the versions's solution cost and the best solution obtained for that instance among all algorithm versions.

The performance profile of an algorithm version is a plot of the cost ratios (x axes) versus the percentage of instances that this algorithm version could solve within at most that ratio (y axes). From Figure 2 we see that, except for the version using distance 2, all versions solved more or less 75% of the instances with ratio 1, meaning that they found 75% of the best solutions. In fact, the version using distance 6 solved almost 80% of the instances with ratio 1, and 100% of the instances with ratio 1.05, meaning that it found 80% of the best solutions and the other solutions had a cost at most 5% higher than the best solutions found. So we decide to use distance 6 in the GRASP algorithm. As for the threshold T of the adaptive method, defined on Section 5.2, we tested the values of 0.2, 0.4, 0.6, 0.8, and 1, and in general the best results were obtained with $T = 0.6$.

The GRASP algorithm was set to run for $\log_2(n)$ iterations, where n is the number of agents in that instance. Figure 3 presents the performance profiles of the GRASP algorithm and the Refinement Search. We also included the results of the lower bound obtained by solving the LP. For the type 1 instances, the Refinement Search found optimal solutions for approximately 60% of the instances, while GRASP found 40% of optimal solutions, and the Refinement Search found solutions to all instances with cost at most 6% higher than the lower bound given by the LP solution. For the type 2 instances we obtained similar results.



(a) Comparison for instances of type 1.

(b) Comparison for instances of type 2.

■ **Figure 3** Comparison between GRASP and Refinement Search heuristics.

■ **Table 1** Results separated by instance classes.

Classes	# inst	GAP(%)		time (s)	
		RS	GRASP	RS	GRASP
m-PDTSP	210	0.76	1.99	0.02	0.01
1x1	15	4.64	5.58	42.02	306.35
2x2	75	2.99	3.79	30.69	89.09
3x3	75	2.39	3.75	31.15	63.25
4x4	75	1.82	3.44	29.93	78.58
5x5	75	1.56	3.43	29.56	86.52
W0.0	75	3.72	5.13	34.45	290.09
W0.1	60	3.27	4.91	30.83	85.98
W0.2	60	2.00	3.45	29.59	9.04
W0.3	60	1.11	2.35	29.16	7.83
W0.4	60	1.06	2.30	29.54	7.93
rg-016	63	0.38	1.81	0.03	0.01
rg-032	63	1.16	2.69	0.22	0.10
rg-064	63	2.44	4.19	1.98	1.34
rg-128	63	3.51	4.69	15.62	26.71
rg-256	63	4.04	5.11	136.59	422.69
total average		1.69	3.02	18.54	54.11

In Table 1 we present a comparison of the Refinement Search (RS) and GRASP separating the results by the type and properties of the instances. In this table, in each line, we present the average GAP, and time, of the instances of a certain class compared to the lower bound obtained by solving the LP. The line m-PDTSP contains the results of the instances of type 1. For the type 2 instances, we sub-divided it depending on some properties. The “1 × 1” class contains results of the instances with one frame, while the “2 × 2” contains the results of the instances with 4 frames and so on. The “W0.x” class contains the results of the instances with wall separation of $(10 \times x)\%$ of the size of the frame. In the last lines, the “rg- n ” classes, we separated the instances by the number of agents n . The Refinement Search algorithm consistently obtained the lowest gaps and, in general, was also faster than the GRASP algorithm. We observe that the cost of the Refinement Search solution was, on average, 1.29% lower than the cost of the GRASP solution. We conjecture that Refinement Search performs better than GRASP because the latter works with agents as vertices of a modified graph, while the former deals with the original graph with pairs of vertices.

References

- 1 Jose Caceres-Cruz, Pol Arias, Daniel Guimarans, Daniel Riera, and Angel A Juan. Rich vehicle routing problem: Survey. *ACM Computing Surveys (CSUR)*, 47(2):32, 2015.
- 2 William J Cook, WH Cunningham, WR Pulleyblank, and A Schrijver. *Combinatorial optimization*. Springer, 2009.
- 3 Elizabeth D Dolan and Jorge J Moré. Benchmarking optimization software with performance profiles. *Mathematical programming*, 91(2):201–213, 2002.
- 4 Özlem Ergun, Gültekin Kuyzu, and Martin W. P. Savelsbergh. Reducing truckload transportation costs through collaboration. *Transportation Science*, 41(2):206–221, 2007.
- 5 Özlem Ergun, Gültekin Kuyzu, and Martin W. P. Savelsbergh. Shipper collaboration. *Computers & OR*, 34(6):1551–1560, 2007.
- 6 Thomas A Feo and Mauricio GC Resende. A probabilistic heuristic for a computationally difficult set covering problem. *Operations research letters*, 8(2):67–71, 1989.
- 7 Michel X. Goemans and David P. Williamson. A general approximation technique for constrained forest problems. *SIAM J. Comput.*, 24(2):296–317, 1995.
- 8 Ralph E Gomory and Tien Chung Hu. Multi-terminal network flows. *Journal of the Society for Industrial and Applied Mathematics*, 9(4):551–570, 1961.
- 9 Martin Grötschel, László Lovász, and Alexander Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1(2):169–197, 1981.
- 10 Gurobi Optimization, Inc. Gurobi optimizer reference manual, 2016. URL: <http://www.gurobi.com>.
- 11 Dan Gusfield. Very simple methods for all pairs network flow analysis. *SIAM Journal on Computing*, 19(1):143–155, 1990.
- 12 Hipólito Hernández-Pérez and Juan-José Salazar-González. The multi-commodity one-to-one pickup-and-delivery traveling salesman problem. *European Journal of Operational Research*, 196(3):987–995, 2009.
- 13 Ismail Karaoglan, Fulya Altiparmak, Imdat Kara, and Berna Dengiz. The location-routing problem with simultaneous pickup and delivery: Formulations and a heuristic approach. *Omega*, 40(4):465–477, 2012.
- 14 Lemon. Library for efficient modeling and optimization in networks, 2016. URL: <http://lemon.cs.elte.hu/trac/lemon/>.
- 15 LOCo - UNICAMP. Laboratory of optimization and combinatorics, 2016. URL: <http://www.loco.ic.unicamp.br>.
- 16 Sophie N. Parragh, Karl F. Doerner, and Richard F. Hartl. A survey on pickup and delivery problems. *Journal für Betriebswirtschaft*, 58(1):21–51, 2008.
- 17 Mauricio GC Resende and Celso C Ribeiro. *Optimization by GRASP: Greedy Randomized Adaptive Search Procedures*. Springer, 2016.
- 18 Juan-José Salazar-González. The Steiner cycle polytope. *European Journal of Operational Research*, 147(3):671–679, 2003.
- 19 Monika Steinová. Approximability of the minimum Steiner cycle problem. *Computing and Informatics*, 29(6+):1349–1357, 2012.
- 20 Paolo Toth and Daniele Vigo. *Vehicle routing: problems, methods, and applications*, volume 18. Siam, 2014.

Real-Time Traffic Assignment Using Fast Queries in Customizable Contraction Hierarchies

Valentin Buchhold

Karlsruhe Institute of Technology, Germany

Peter Sanders

Karlsruhe Institute of Technology, Germany

Dorothea Wagner

Karlsruhe Institute of Technology, Germany

Abstract

Given an urban road network and a set of origin-destination (OD) pairs, the traffic assignment problem asks for the traffic flow on each road segment. A common solution employs a feasible-direction method, where the direction-finding step requires many shortest-path computations. In this paper, we significantly accelerate the computation of flow patterns, enabling interactive transportation and urban planning applications. We achieve this by revisiting and carefully engineering known speedup techniques for shortest paths, and combining them with customizable contraction hierarchies. In particular, our accelerated elimination tree search is more than an order of magnitude faster for local queries than the original algorithm, and our centralized search speeds up batched point-to-point shortest paths by a factor of up to 6. These optimizations are independent of traffic assignment and can be generally used for (batched) point-to-point queries. In contrast to prior work, our evaluation uses real-world data for all parts of the problem. On a metropolitan area encompassing more than 2.7 million inhabitants, we reduce the flow-pattern computation for a typical two-hour morning peak from 76.5 to 10.5 seconds on one core, and 4.3 seconds on four cores. This represents a speedup of 18 over the state of the art, and three orders of magnitude over the Dijkstra-based baseline.

2012 ACM Subject Classification Theory of computation → Shortest paths

Keywords and phrases traffic assignment, equilibrium flow pattern, customizable contraction hierarchies, batched shortest paths

Digital Object Identifier 10.4230/LIPIcs.SEA.2018.27

Acknowledgements We thank Peter Vortisch for providing the Stuttgart instance, and Lukas Barth and Ben Strasser for interesting discussions.

1 Introduction

The number of drivers traveling along a road segment within a given period is the result of many individual decisions. The common behavioral assumption in practice is that motorists driving between a given origin and destination choose the path with the minimum travel time (known as *Wardrop's first rule* [39]). This seems natural, since travel is usually not a goal in itself, but entails disutility. However, the travel time on a path depends on the route choice of all other drivers, who themselves are trying to choose minimum travel time routes. Due to congestion, the travel time on a road segment increases with the traffic flow on it. As a result, some drivers choose at some point alternative routes, which can also get congested, and so on. When no driver can improve his travel time by unilaterally changing routes, each



© Valentin Buchhold, Peter Sanders, and Dorothea Wagner;
licensed under Creative Commons License CC-BY

17th International Symposium on Experimental Algorithms (SEA 2018).

Editor: Gianlorenzo D'Angelo; Article No. 27; pp. 27:1–27:15

Leibniz International Proceedings in Informatics



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

route used between a given origin and destination has the same travel time. This condition is known as the *user equilibrium*, and the flow pattern is called the *equilibrium flow pattern* [37].

We study the efficient computation of equilibrium flow patterns in road networks. More formally, given an urban road network and a set of origin-destination (OD) pairs, we want to compute the traffic flow on each road segment at equilibrium. This is known as *traffic assignment*, and is one of the major problems facing transportation engineers and urban planners [37]. In this paper, we accelerate the process of traffic assignment significantly (by a factor of 18). Our goal is twofold. The short-term objective is to enable *interactive* transportation and urban planning applications. The long-term aim is to develop a real-time demand-responsive public transit system, which makes use of a traffic assignment procedure as a subroutine. There, we decrease (rather than increase) the travel cost per individual as the flow increases, since increased flow makes public transit more cost-effective.

Related Work. The traffic assignment problem has been studied for over 60 years, and has motivated extensive research in the operations research community. The formulation as a mathematical program first appeared in 1956 [5]. A common solution employs the Frank-Wolfe algorithm [17], a feasible-direction method for solving quadratic programs with linear constraints. The application of the Frank-Wolfe method to the traffic assignment problem was first proposed in the late 1960s [6, 31], and the solution was implemented and experimentally evaluated for the first time in 1975 [25]. The textbook by Sheffi [37] offers a comprehensive introduction to the traffic assignment problem and considers research published until 1985. More recent developments are covered by Patriksson [32]. Peeta et al. [33] survey recent advances in *dynamic* traffic assignment, recognizing time variations in traffic flows and conditions during the period of analysis. Finally, Babonneau et al. [1] collect benchmark instances previously used in the literature, however, their largest instance is still an order of magnitude smaller than the benchmark instance used in this paper.

The performance of the Frank-Wolfe algorithm is clearly dominated by the direction-finding step, which requires a large number of shortest-path computations. The past decade has seen intense research on speedup techniques [2] for Dijkstra’s algorithm [13], which rely on a slow preprocessing phase to enable fast queries. One of the most prominent and versatile techniques among these are contraction hierarchies (CH) [18], which exploit the inherent hierarchy of the network. A fairly recent development are customizable speedup techniques [8, 12, 14], which split preprocessing into a slow metric-independent part, taking only the graph structure into account, and a fast metric-dependent part (the *customization*), incorporating new edge weights (the *metric*). CRP [8] and customizable CHs [12] are the most prominent among them. A common approach to accelerate one-to-all queries is to bundle together multiple shortest-path computations in a single search [20, 8, 7, 4, 40].

To the best of our knowledge, there is only a single paper [26] that solves the traffic assignment problem using state-of-the-art shortest-path algorithms (plain CHs in this case).

Contribution and Outline. The contribution of this work is twofold. First, we accelerate the state of the art in the area of traffic assignment. On our main benchmark instance, we see a speedup of 18. This is more than three orders of magnitude faster than the Dijkstra-based baseline. However, the building blocks to achieve this are also independent contributions, not restricted to traffic assignment, but generally applicable in the area of route planning. Our two main building blocks are as follows. (1) Currently, there are two CCH query algorithms, one based on Dijkstra’s algorithm and one based on elimination trees (a structure encoding the CH search space of each vertex). We thoroughly reengineer the elimination tree search (Section 3), providing a unified query algorithm that combines the good local-query

performance of the Dijkstra-based search with the good global-query performance of the elimination tree search. (2) We introduce a *centralized* elimination tree search for computing batched point-to-point shortest paths fast (Section 4). While there is a large amount of work on one-to-all, one-to-many, many-to-many, and point-of-interest queries [7, 9, 15, 16, 22, 10], we are the first that accelerate batched *point-to-point* shortest paths. All building blocks are extensively experimentally evaluated using solely real-world data (Section 5), whereas previous work fell back on synthetic OD-pairs [26].

2 Preliminaries

We now briefly review the three main algorithms we build upon. First, we describe the Frank-Wolfe algorithm for solving the traffic assignment problem. Then, we discuss two speedup techniques for Dijkstra’s algorithm, CHs and their customizable counterpart CCHs.

2.1 Traffic Assignment

Popular approaches [37] to the traffic assignment problem formulate a mathematical program, known as *Beckmann’s transformation* [5], whose solution is the equilibrium flow pattern. It is a convex minimization program with linear constraints. The *Frank-Wolfe algorithm* [17, 37], a feasible-direction method, is especially suitable for solving Beckmann’s transformation, since the direction-finding step can be implemented relatively efficiently. Being a feasible-direction method, it iteratively finds a feasible descent direction and advances by an optimal move size in the direction. An important subroutine of the Frank-Wolfe algorithm (when solving Beckmann’s transformation) is the *all-or-nothing assignment* procedure, which processes each OD-pair in turn and assigns one flow unit to each edge on the shortest travel time path.

The algorithm can be summarized as follows. (1) Perform an all-or-nothing assignment using free-flow travel times, yielding an initial solution. (2) Update the travel time on each edge according to the current solution (recall that the travel time increases with the traffic flow). (3) Perform an all-or-nothing assignment using the current travel times, yielding a set of auxiliary flows. (4) Perform a *line search* to determine the optimal move size α . (5) Set the new solution to a convex combination of the current solution and the auxiliary flows (according to α). (6) Check the stopping criterion, and terminate or go to step (2). In this paper we update travel times according to the modified Davidson function [30], perform the bisection method of Bolzano as line search, and stop after a predefined number of iterations.

2.2 Contraction Hierarchies

Contraction hierarchies (CH) [18] are a two-phase speedup technique to accelerate point-to-point shortest-path computations, which exploits the inherent hierarchy of road networks. The preprocessing phase heuristically orders the vertices by importance, and *contracts* them from least to most important. Intuitively, vertices that hit many shortest paths are considered more important, such as vertices on highways. To contract a vertex v , it is temporarily removed from the graph, and *shortcut* edges are added between its neighbors to preserve distances in the remaining graph (without v). Note that a shortcut is only needed if it represents the only shortest path between its endpoints, which can be checked by running a *witness search* (local Dijkstra) between its endpoints. The query phase runs a bidirectional Dijkstra on the augmented graph that only relaxes edges leading to vertices of higher ranks (importance). The stall-on-demand [18] optimization prunes the search at any vertex v with a suboptimal distance label, which can be checked by looking at upward edges coming into v .

Traffic Assignment Using CHs. The shortest-path computations are by far the most time-consuming part of the Frank-Wolfe algorithm. Carrying them out with the use of CHs (instead of Dijkstra’s algorithm) accelerates traffic assignments by two orders of magnitude [26]. Since the weight of each edge changes between two iterations, the CH is rebuilt from scratch in each iteration. Queries do not unpack shortcuts, but assign one flow unit to each (shortcut) edge on the packed path. After computing all paths, the shortcuts are unpacked in top-down fashion, with cumulated flow units propagated from shortcut to original edges.

2.3 Customizable CHs

Customizable contraction hierarchies (CCH) [12] are a three-phase technique, splitting CH preprocessing into a relatively slow metric-independent phase and a much faster customization phase. The metric-independent phase computes a *nested dissection order* [3, 19] on the vertices of the unweighted graph, and contracts them in this order without running witness searches (as they depend on the metric). As a result, it adds every potential shortcut. The customization phase computes the weights of the shortcuts by processing them in bottom-up fashion. To process a shortcut (u, v) , it enumerates all triangles $\langle u, v, w \rangle$ where w has lower rank than u and v , and checks if the path (u, w, v) improves the weight of (u, v) . After *basic customization*, one can optionally run *perfect witness searches* to remove superfluous edges.

There are two different query algorithms possible. First, one can run a standard CH search without modification. In addition, Dibbelt et al. [12] describe a query algorithm based on the *elimination tree* of the augmented graph. The parent of a vertex in the elimination tree is its lowest-ranked upward neighbor in the augmented graph. Bauer et al. [3] prove that the ancestors of a vertex v in the elimination tree are exactly the set of vertices in the CH search space of v . Hence, the elimination tree query algorithm explores the search space by traversing the elimination tree, avoiding a priority queue completely.

3 Accelerating Elimination Tree Searches

In the next section, we will devise a fast traffic assignment procedure based on customizable contraction hierarchies. While Dibbelt et al. [12] observe that the CCH query algorithm based on elimination trees achieves fastest query times for random queries (which tend to be long-range), it is slower by more than an order of magnitude than the Dijkstra-based query algorithm for local queries (see Section 5). However, the input of the traffic assignment problem consists of both local and long-range OD-pairs, requiring a query algorithm that can handle both types of queries well. Therefore, we review and carefully engineer the elimination tree search in this section. The result is a fast, unified CCH query algorithm, combining good performance for both local and long-range queries.

Given a source vertex s and a target vertex t , the original elimination tree search [12] works in five phases. First, we compute the lowest common ancestor (LCA) x of s and t in the elimination tree T rooted at the highest-ranked vertex r . This is done by enumerating the ancestors of s and t in increasing rank order until a common ancestor is found. Second, we revisit each vertex v on the s - x path in T , relaxing all *outgoing* upward edges of v . Third, we do the same for each vertex v on the t - x path in T , relaxing all *incoming* upward edges of v . Fourth, we visit each vertex v on the x - r path in T , relaxing all outgoing *and* incoming upward edges of v . Moreover, we check at each such vertex v whether the s - t path via v improves the tentative s - t distance. Fifth, we again revisit each vertex on the s - r and t - r path to reset its distance labels for the next shortest-path computation.

Phase Reduction. Our first optimization reduces the number of phases of the elimination tree search. We refrain from computing the LCA first, and then visiting each vertex from the source (target) to the LCA again. Instead, while we enumerate the ancestors of s and t in the same fashion as before, we immediately relax their edges. Moreover, we observe that the resetting phase is unnecessary. After relaxing the edges of a vertex, its distance labels are never read again. Therefore, we can safely reset them to ∞ right after relaxing the edges, avoiding the fifth phase completely. Note that we cannot reset parent pointers, since they may be needed afterwards. However, this is not an issue because resetting the distance labels suffices to decide whether a vertex has been visited before during the next query. With this optimization, each vertex is visited at most once, instead of up to three times as before.

Pruning Rule. The basic elimination tree search does not make use of pruning. Only when combined with the perfect witness search, Dibbelt et al. [12] employ the following basic pruning rule. Due to the removal of superfluous edges, a vertex may have an ancestor in the elimination tree that is not in its perfect search space. Such an ancestor will have a distance label of ∞ when visited during the search. To accelerate queries, Dibbelt et al. do not relax the edges of a vertex with a distance label of ∞ . We observe that a stricter pruning rule is possible. We do not relax edges of a vertex whose distance label exceeds the current tentative shortest-path distance, since those edges cannot possibly contribute to a shorter path. Despite its simplicity, this optimization accelerates the search quite drastically, by a factor of 15 for short-range queries (see Section 5). Moreover, our pruning rule does not require the perfect witness search, but can also be combined with the basic customization.

4 Accelerating Traffic Assignments by Fast Batched Shortest Paths

Previous work [26] applying speedup techniques to traffic assignment observed that the performance bottleneck depends on the traffic scenario under study. For short or off-peak periods, where there are few OD-pairs, preprocessing dominates the total running time. When there are many OD-pairs, as for long or peak periods, queries become the main bottleneck.

To decrease the preprocessing time, we apply the concept of customization to traffic assignment. Customizable speedup techniques [8, 12, 14] split preprocessing into a metric-independent part, taking only the graph structure into account, and a metric-dependent part (the *customization*), incorporating new edge weights (the *metric*). Since the graph topology does not change in all iterations of the traffic assignment procedure and only edge weights change, it suffices to run a fast customization in each iteration instead of an entire preprocessing. We build our accelerated traffic assignment upon customizable contraction hierarchies [12], which allows us to employ the hierarchy decomposition optimization from [26]. As basic query algorithm, we use the engineered elimination tree search from the previous section. To reduce the query time, the following sections introduce several optimization techniques for computing batched point-to-point shortest paths fast.

4.1 Reordering OD-pairs to Exploit Locality

Previous work processed the OD-pairs in no particular order. However, reordering the OD-pairs so that pairs with similar forward and reverse search spaces tend to be processed in succession improves memory locality and cache efficiency. We call two search spaces similar if their symmetric difference is small. Note that the size of the symmetric difference between the search spaces of u and v is equal to the distance between u and v in the elimination tree. Hence, we partition the elimination tree into as few cells with bounded diameter as possible,

assign IDs to cells according to the order in which they are reached during a DFS [29] on the elimination tree, and reorder OD-pairs lexicographically by the origin and destination cells.

We use a simple yet optimal greedy algorithm to partition the elimination tree into as few cells with diameter at most U as possible. Our algorithm repeatedly cuts out a subtree (with diameter at most U) and makes it a cell of its own. In order to do so, it maintains for each vertex v the height $h(v)$ of the remaining subtree T_v rooted at v , initialized to zero, and processes vertices in ascending rank order. To process v , we examine its children w_i in order of increasing height of T_{w_i} . If $h(v) + 1 + h(w_i) \leq U$, we set $h(v) = 1 + h(w_i)$. Otherwise, we cut out T_{w_i} , making it a cell of its own. We use $U = 40$ in our experiments.

4.2 Centralized Searches

Instead of processing similar OD-pairs *in succession*, processing them *at once* in a single search achieves additional speedup. The idea of bundling together multiple shortest-path computations was introduced in [20] and later used in [8, 7, 9, 4, 40]. However, in each case, centralized searches were only used for one-to-all and -many queries, and only combined with plain Dijkstra (and Bellman-Ford in [8]). Even (R)PHAST [7, 9] performs the CH searches sequentially, and bundles only the linear sweeps. We extend the idea to point-to-point queries, and combine it with CH searches, including appropriate stopping and pruning criteria.

The basic idea of centralized searches is to maintain k distance labels for each vertex u , laid out consecutively in memory. The i -th distance label represents the tentative distance from the i -th source to u . Initially, the i -th distance label of the i -th source is set to zero, and all remaining $kn - k$ distance labels to ∞ . When relaxing an edge (u, v) , we try to improve all k distance labels of v at once. Increasing k allows us to compute more shortest paths at once, however, it also evicts useful data from caches.

Dijkstra-Based Search. Initially, we insert all k sources (targets) into the queue of the forward (reverse) search. As keys, we can use many different values, for example the minimum over all k entries in a distance label or the minimum over the entries that were improved by the last edge relaxation. However, a preliminary experiment showed that using the minimum over *all* k entries clearly dominates the others, which is consistent with previous observations on related techniques [20]. We can stop the forward (reverse) search as soon as its queue is empty or the smallest queue entry exceeds the maximum over all k tentative shortest-path distances. When using stall-on-demand [18], we prune the forward (reverse) search at a vertex v when each of the k distance labels of v is suboptimal.

Elimination Tree Search. Computing multiple shortest paths in a single elimination tree search is more involved, since it uses no queues that can easily be initialized with multiple sources and targets. Instead, we equip the forward and reverse search each with a tournament tree [23]. Suppose we have k sorted sequences that are to be merged into a single output sequence, as in k -way mergesort. To do so, we have to repeatedly find the smallest from the leading elements in the k sequences. This can be done efficiently with a tournament tree.

In our case, the k sorted sequences are the paths in the elimination tree T from each source (target) to the root, and the single output sequence is the order in which we want to process the vertices during the search. More precisely, we initialize the tournament tree with all k sources (targets). Then, we extract a lowest-ranked vertex from the tournament tree, process it, and insert its parent in T into the tournament tree. We continue with a next-lowest-ranked vertex, until we reach the root of T . Note that in our case, unlike in mergesort, the sequences are implicit, and never stored explicitly.

As soon as two (or more) of the k paths in T converge at a common vertex, there are duplicates in the single output sequence. However, we want to process each vertex at most once. Therefore, whenever two or more paths converge, we block all but one of them, so that only one continues to move through the tournament tree. To do so, we insert for each path to be blocked a vertex with infinite rank into the tournament tree (instead of the next vertex on the path). We know that some paths converged, when we extract the very same vertex several times in succession from the tournament tree.

A clear advantage of the centralized elimination tree search is that it retains the *label-setting* property, i.e., each vertex and each edge is processed at most once. In contrast, the centralized Dijkstra-based search is a *label-correcting* algorithm. Note that one centralized elimination tree search is slower than k elimination tree searches by a factor of $\log k$ in O -notation (due to k -way merging), but outperforms them in practice (see Section 5).

4.3 Instruction-Level Parallelism

Modern CPUs have special registers and instructions that enable single-instruction multiple-data (SIMD) computations performing basic operations (e.g., additions, subtractions, shifts, compares, and data conversions) on multiple data items simultaneously [24]. We implemented versions of the centralized searches using SSE instructions (working with 128-bit registers), and additionally versions using AVX(2) instructions (manipulating 256-bit registers), requiring a processor based on Intel's Haswell or AMD's Excavator microarchitecture.

As an example, we describe how an AVX-accelerated edge relaxation (used in Dijkstra-based and elimination tree searches) works, assuming $k = 8$. Since we use 32-bit distance labels, all k labels of a vertex fit in a single 256-bit register. To relax an edge (u, v) , we copy all k distance labels of u to an AVX register, and broadcast the edge weight to all elements of another register. Then, we add both registers with a single instruction, and check with an AVX comparison whether any tentative distance improves the corresponding distance label of v . If so, we compute the packed minimum of the tentative distances and v 's distance labels. In the same fashion, we implement the other aspects (stopping and pruning criteria).

4.4 Core-Level Parallelism

Dibbelt et al. [12] introduce parallelization techniques for the triangle enumeration during customization. However, we observed that the perfect witness search building the upward and downward search graphs (which is difficult to parallelize) actually takes up 60% of the customization time. Hence, the speedup obtainable by parallelizing the customization phase is limited (less than a factor of 1.5). For simplicity, we stick to sequential customization.

In contrast, the shortest-path computations are easy to parallelize. Since the centralized computations are independent from one another, we can assign contiguous subsets of OD-pairs to distinct cores. We distribute the OD-pairs to cores in chunks of size 64. This maintains some locality even between centralized computations. Each core executes a chunk, then requesting another chunk until no chunk remains. Flow units on the (shortcut) edges are cumulated locally and aggregated after computing all paths. We observe an almost perfect speedup for the time spent on queries (see Section 5).

■ **Table 1** Traffic scenarios used for the evaluation of our traffic assignment procedures. We report for each scenario the period of analysis and the number of OD-pairs departing within that period.

scenario	analysis period	# OD-pairs
Tue30m	Tue., 7:00–7:30	118 933
Tue01h	Tue., 7:00–8:00	246 089
Tue02h	Tue., 7:00–9:00	478 098
Tue24h	a whole Tuesday	3 355 442
MonSun	a whole week	21 248 278

5 Experiments

Our publicly available code¹ is written in C++14 and compiled with the GNU compiler 7.3 using optimization level 3. We use 4-heaps [21] as priority queues. To ensure a correct implementation, we make extensive use of assertions (disabled during measurements), and check results against reference implementations such as Dijkstra’s algorithm. Our benchmark machine runs openSUSE Leap 42.3 (kernel 4.4.114), and has 128 GiB of DDR4-2133 RAM and an Intel Xeon E5-1630 v3 CPU, which has four cores clocked at 3.70 Ghz.

5.1 Inputs and Methodology

Our main instance is the metropolitan area of Stuttgart [36], Germany, encompassing more than 2.7 million inhabitants. The experiments were performed on the largest strongly connected component, consisting of 134 663 vertices and 307 759 edges. While this instance is significantly smaller than road networks studied before for evaluating point-to-point queries [2], it is the largest available to us that provides real-world capacities and OD-pairs, and is still an order of magnitude larger than the instances collected in [1]. Moreover, urban planners are usually interested in traffic assignments on metropolitan areas, not continents.

The OD-pairs were obtained from [27, 28], which was calibrated from a household travel survey [38] conducted in 2009/2010. The raw data contains about 51.8 million trips between 1174 traffic zones for a whole week, encompassing various modes of transportation such as pedestrian, bicycle, public transit and car. For our experiments we only considered car trips, and extracted five different traffic scenarios, as shown in Table 1. We chose a typical two-hour morning peak on a working day (Tuesday), and also included two smaller and two larger scenarios. While it may be unrealistic to compute a traffic assignment for a whole week (as the period of analysis would be too inhomogeneous), it shows the scalability of our procedures for tens of millions of OD-pairs. Note that we assume the actual origins and destinations to be uniformly distributed in the zone, and obtain OD-pairs by picking the origins and destinations uniformly at random from the zones according to the predicted trips.

Since our engineered elimination tree search is not restricted to traffic assignment, we evaluate it on the European road network, which is the standard benchmark instance for point-to-point queries [2]. It has 18 017 748 vertices and 42 560 275 edges, and was made available by PTV AG for the 9th DIMACS Implementation Challenge [11].

The CH preprocessing is borrowed from the open-source library RoutingKit². We compute nested dissection orders for CCHs using Inertial Flow [35], setting the balance parameter

¹ <https://github.com/vbuchhold/routing-framework>

² <https://github.com/RoutingKit/RoutingKit>

$b = 0.3$. The reported running times do not include partitioning time, as it suffices to partition a network only once, and reuse the resulting order for all traffic assignments on the same network. Partitioning the metropolitan area of Stuttgart takes less than two seconds (even on a single core). We always use perfect witness searches in combination with CCHs.

5.2 Elimination Tree Search

First we evaluate our engineered elimination tree search on its own. As most queries in the real world tend to be local, we use the established Dijkstra rank methodology [34], which considers local and long-range queries equally. In contrast, random queries tend to be long-range. The Dijkstra rank (with respect to a source s) of a vertex v is r if v is the r -th vertex settled by a Dijkstra search from s . We run 1000 point-to-point queries (without path unpacking) for each Dijkstra rank tested, with s picked uniformly at random. Figure 1 compares the performance of our accelerated elimination tree search (CCH-tree-fast), the original CCH query algorithms (CCH-Dij and CCH-tree), and the plain CH search on Europe with travel times. Note that CCH-tree is not really the original algorithm, but already uses our phase reduction optimization. CCH-tree-fast additionally uses our stricter pruning rule.

We observe that CCH-tree, while outperforming CCH-Dij on random queries [12], is actually much slower for most Dijkstra ranks, especially for the realistic ones. The reason is that the performance of CCH-tree is independent of the Dijkstra rank, since it always processes each vertex in the search space. However, our stricter pruning rule makes the algorithm sensitive to the Dijkstra rank, drastically speeding up short- and mid-range queries (by up to a factor of 15). As a result, CCH-tree-fast combines the good local-query performance of CCH-Dij with the good global-query performance of CCH-tree, and is faster than both on mid-range queries. It can be seen as a unified CCH query algorithm, replacing both original ones. Moreover, for many (realistic) Dijkstra ranks, it is about as fast as the non-customizable CH search. When optimizing travel *distances* (not shown in the figure), CCH-tree-fast even outperforms the CH search.

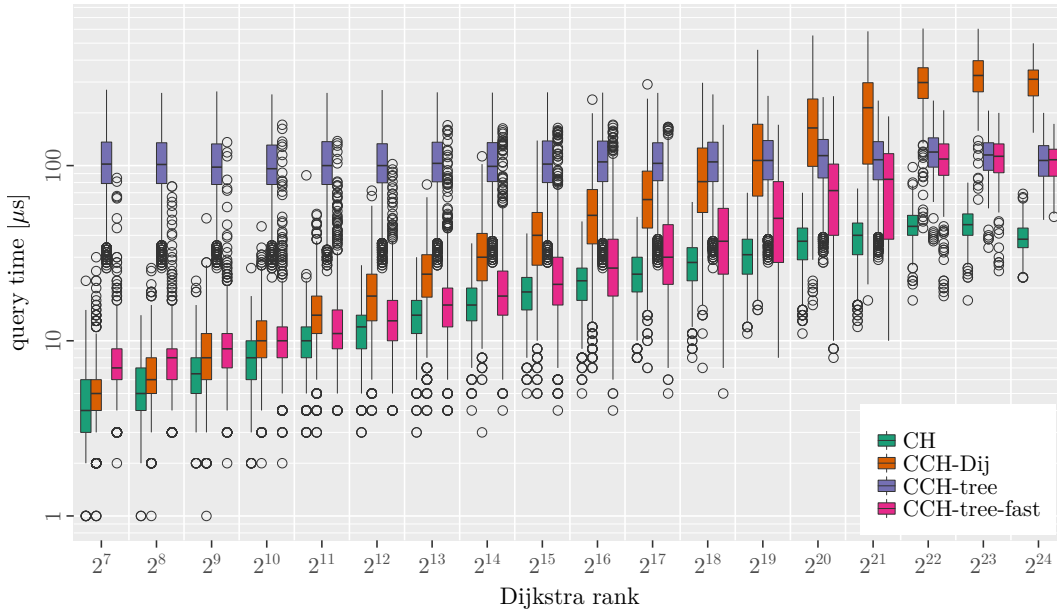
5.3 Traffic Assignment

We now evaluate the impact of our building blocks (customization, reordering OD-pairs, centralized searches, and parallelism at multi-core and instruction levels) on the performance of the traffic assignment procedure. As already mentioned, we run a predefined number of iterations for each scenario: twelve on Tue30m, Tue01h and Tue02h, six on Tue24h and MonSun. This choice is consistent with [37] and also justified by subsequent experiments.

Customization and Centralized Searches. Table 2 considers the influence of customization and of the centralized searches on the performance of the traffic assignment. For now, we use only a single core. The CCH-based procedures use the engineered elimination tree search.

Switching from plain to customizable CHs reduces the running time for all traffic scenarios. As expected, we obtain larger speedups for smaller scenarios (a factor of 3 on Tue30m), since preprocessing time dominates more in such scenarios. In contrast, reordering the OD-pairs so that similar OD-pairs are processed successively works better for larger scenarios, improving the running time on MonSun by about 20%.

The impact of computing multiple shortest paths at once without exploiting instruction-level parallelism is limited. However, when using SIMD instructions, centralized searches decrease the running time by up to another factor of 5.2. Increasing k allows us to compute more shortest paths at once, but it also evicts useful data from caches. Setting $k = 32$ seems



■ **Figure 1** Performance of our engineered elimination-tree search (CCH-tree-fast), the original CCH query algorithms (CCH-Dij and CCH-tree), and a CH. The input is Europe with travel times.

to be a good choice. Moreover, we observe that the centralized elimination searches achieve greater speedups than the Dijkstra-based ones, since they are label-setting. (Although the clustering approach described in Section 4 is tailored to the elimination tree search, preliminary experiments with unbiased clustering approaches not building upon the elimination tree showed a quite similar performance difference.)

Combining the optimizations, the traffic assignment procedure based on AVX-accelerated centralized elimination tree searches with $k = 32$ gives the best overall performance. It speeds up the state of the art by a factor of about 8 on all of our traffic scenarios. Compared to the Dijkstra-based baseline, this configuration is several hundred times faster.

Core-Level Parallelism. Table 3 shows how the traffic assignment procedure scales as the number of cores increases. We observe that the time spent on queries scales very well. With 4 cores, we gain a speedup of 3.5 for Tue24h and MonSun, and even our smallest scenario is accelerated by a factor of 2.7. In total, our multi-threaded centralized traffic assignment procedure decreases the running time on our main benchmark instance Tue02h from 76.5 to 4.3 seconds, a speedup of 18 over the state of the art.

For comparison, we also run the state of the art on four cores, parallelizing the shortest-path computations as described in Section 4. We observe that even on a single core, our procedure is always more than twice as fast as the parallelized state of the art. The difference between both parallelized versions is again a factor of about 8.

Convergence. Next, we evaluate how long the traffic assignment procedure takes to converge. For that we run a very large number of iterations of the procedure (300 in our case) and take the resulting flow pattern as the equilibrium situation. Figure 2 shows the average deviation of the OD-travel-costs in each iteration from the OD-travel-costs at equilibrium.

We observe that Tue30m, Tue01h, and Tue02h require more iterations to converge, since they are periods during the morning peak. In contrast, Tue24h and MonSun behave like

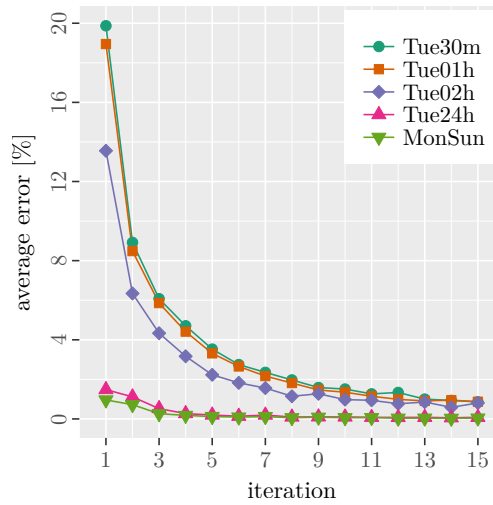
■ **Table 2** Impact of the centralized searches on the running time (in seconds) of the traffic assignment procedure for our scenarios. We evaluate the influence of using customizable CHs, reordering the OD-pairs (sorted), computing k shortest paths simultaneously, and using SSE and AVX instructions. The prior state of the art and our default configuration are highlighted in bold.

algo	sorted	k	SSE	AVX	Tue30m	Tue01h	Tue02h	Tue24h	MonSun
Dij	○	1	○	○	1857.27	3582.67	6028.48	20128.24	128993.50
CH	○	1	○	○	35.85	52.81	76.54	183.59	1082.20
CH	●	1	○	○	35.06	48.72	67.83	153.90	851.63
CH	●	4	○	○	34.47	45.75	62.20	130.07	656.66
CH	●	4	●	○	30.27	39.85	52.89	99.22	507.49
CH	●	8	○	○	38.04	50.80	71.11	151.24	763.45
CH	●	8	●	○	29.41	36.23	46.79	85.89	410.03
CH	●	8	○	●	29.64	36.33	45.51	81.98	397.91
CH	●	16	○	●	29.33	35.08	44.34	75.45	352.90
CH	●	32	○	●	29.87	37.30	46.71	72.85	322.04
CH	●	64	○	●	34.79	43.98	54.75	85.28	354.15
CCH	○	1	○	○	11.99	22.75	40.02	132.54	803.48
CCH	●	1	○	○	10.62	19.59	34.12	108.08	659.57
CCH	●	4	○	○	9.71	17.28	29.50	87.70	499.25
CCH	●	4	●	○	6.28	10.60	17.82	51.91	298.91
CCH	●	8	○	○	11.71	20.72	35.14	102.89	581.38
CCH	●	8	●	○	5.31	8.61	14.02	39.25	218.79
CCH	●	8	○	●	4.86	7.84	12.66	35.11	195.63
CCH	●	16	○	●	4.58	6.98	10.83	27.48	144.81
CCH	●	32	○	●	4.60	6.89	10.54	25.14	126.70
CCH	●	64	○	●	5.91	8.92	13.55	29.78	145.33

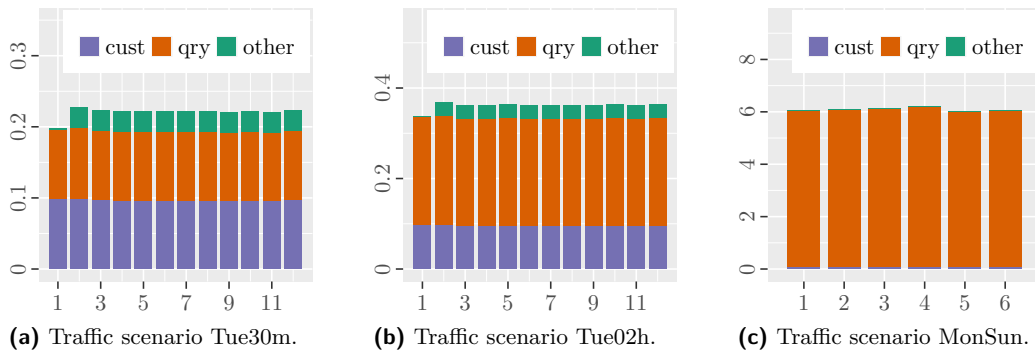
■ **Table 3** Impact of core-level parallelization on the performance of the traffic assignment procedure. We report for each scenario the time spent on queries and the total running time (both in seconds).

algo	# cores	Tue30m		Tue01h		Tue02h		Tue24h		MonSun	
		qry	total	qry	total	qry	total	qry	total	qry	total
CH	4	4.14	24.76	8.21	28.33	14.85	34.41	46.28	54.97	295.39	304.33
CCH	1	3.12	4.60	5.41	6.89	9.05	10.54	24.42	25.14	125.97	126.70
CCH	2	1.85	3.32	3.18	4.66	5.23	6.71	13.46	14.18	70.18	70.91
CCH	3	1.38	2.86	2.29	3.76	3.63	5.10	9.09	9.82	47.12	47.85
CCH	4	1.17	2.65	1.82	3.31	2.86	4.33	6.95	7.67	35.90	36.64

off-peak periods, since the traffic is considered to be uniformly distributed over the period of analysis. In relatively uncongested networks, the edge flows are in the range where the travel time functions are almost flat, the updated travel times are closer to the initial ones, and the equilibrium flow pattern is more similar to the initial solution [37]. The peak scenarios are close to equilibrium after about twelve iterations, the off-peak scenarios after about six iterations.



■ **Figure 2** Convergence of the traffic assignment procedure. The plot shows the average deviation of the OD-travel-costs in each iteration from the OD-travel-costs at equilibrium.



■ **Figure 3** Time in seconds (vertical) spent in each iteration (horizontal) for the multi-threaded traffic assignment procedure (using all 4 cores). For MonSun, customization and other work are hardly visible, since they take only 1.59% and 0.41% of the total time, respectively.

Time per Iteration. Figure 3 plots the running time (per phase) that our multi-threaded traffic assignment spends in each iteration. First, we observe that the procedure spends the same amount of time in each iteration. Although the inherent hierarchy of the network is weakened while computing an equilibrium flow pattern [26], this is expected since the performance of both CCH customization and queries is mostly metric-independent [12]. For our smallest scenario, customization takes 44% of the total time. This decreases to 27% for Tue02h, and to 2% for our largest scenario. All other work, such as the line search, the edge updates, and the convergence checks, is negligible (only 12% even for the smallest scenario).

6 Conclusion

We accelerated the computation of equilibrium flow patterns significantly. This was achieved by carefully engineering a number of building blocks, including customization, an improved CCH query algorithm, centralized searches, and parallelism at multi-core and instruction levels. Moreover, the improved, unified CCH query algorithm (replacing both original query algorithms) and the centralized elimination tree search are not restricted to the traffic

assignment problem, but generally applicable to (batched) point-to-point shortest paths. All building blocks were evaluated on real-world data used in production systems. On a metropolitan area encompassing more than 2.7 million inhabitants, we compute the flow pattern for a typical two-hour morning peak in merely 4.3 seconds, 18 times faster than the state of the art, and 1390 times faster than the Dijkstra-based baseline. This makes interactive urban transportation planning applications practical.

For traffic scenarios where the shortest-path computations are still the performance bottleneck of the traffic assignment procedure, it would be interesting to process only a sample of the demand in early iterations, and add more and more OD-pairs in subsequent iterations. In addition, we are interested in testing our traffic assignment procedure on benchmark instances that are even an order of magnitude larger than the one used in this work. Since we are not aware of any such real-world instances, we plan to work on *realistic* generators for synthetic OD-pairs. Finally, it would be interesting to study the efficient computation of time-dependent traffic flow profiles.

References

- 1 Frédéric Babonneau and Jean-Philippe Vial. Test instances for the traffic assignment problem. Technical report, Ordecys, 2008.
- 2 Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route planning in transportation networks. In *Algorithm Engineering: Selected Results and Surveys*, pages 19–80. Springer, 2016. doi:10.1007/978-3-319-49487-6_2.
- 3 Reinhard Bauer, Tobias Columbus, Ignaz Rutter, and Dorothea Wagner. Search-space size in contraction hierarchies. *Theoretical Computer Science*, 645:112–127, 2016. doi:10.1016/j.tcs.2016.07.003.
- 4 Reinhard Bauer and Daniel Delling. SHARC: Fast and robust unidirectional routing. *ACM Journal of Experimental Algorithmics*, 14:2.4:1–2.4:29, 2009. doi:10.1145/1498698.1537599.
- 5 Martin Beckmann, C. Bart McGuire, and Christopher B. Winsten. *Studies in the Economics of Transportation*. Yale University Press, 1956.
- 6 M. Bruynooghe, A. Gilbert, and M. Sakarovich. Une méthode d’affectation du traffic. In *Proceedings of the 4th International Symposium on the Theory of Road Traffic Flow*, 1968.
- 7 Daniel Delling, Andrew V. Goldberg, Andreas Nowatzky, and Renato F. Werneck. PHAST: Hardware-accelerated shortest path trees. *Journal of Parallel and Distributed Computing*, 73(7):940–952, 2013. doi:10.1016/j.jpdc.2012.02.007.
- 8 Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable route planning in road networks. *Transportation Science*, 51(2):566–591, 2017. doi:10.1287/trsc.2014.0579.
- 9 Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Faster batched shortest paths in road networks. In *Proceedings of the 11th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS’11)*, pages 52–63, 2011. doi:10.4230/OASICS.ATMOS.2011.52.
- 10 Daniel Delling and Renato F. Werneck. Customizable point-of-interest queries in road networks. *IEEE Transactions on Knowledge and Data Engineering*, 27(3):686–698, 2015. doi:10.1109/TKDE.2014.2345386.
- 11 Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*. American Mathematical Society, 2009.

- 12 Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Customizable contraction hierarchies. *ACM Journal of Experimental Algorithmics*, 21(1):1.5:1–1.5:49, 2016. doi:10.1145/2886843.
- 13 Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
- 14 Alexandros Efentakis and Dieter Pfoser. Optimizing landmark-based routing and preprocessing. In *Proceedings of the 6th ACM SIGSPATIAL International Workshop on Computational Transportation Science (IWCTS'13)*, pages 25–30, 2013. doi:10.1145/2533828.2533838.
- 15 Alexandros Efentakis and Dieter Pfoser. GRASP. Extending graph separators for the single-source shortest-path problem. In *Proceedings of the 22th Annual European Symposium on Algorithms (ESA'14)*, pages 358–370, 2014. doi:10.1007/978-3-662-44777-2_30.
- 16 Alexandros Efentakis, Dieter Pfoser, and Yannis Vassiliou. SALT. A unified framework for all shortest-path query variants on road networks. In *Proceedings of the 14th International Symposium on Experimental Algorithms (SEA'15)*, pages 298–311, 2015. doi:10.1007/978-3-319-20086-6_23.
- 17 Marguerite Frank and Philip Wolfe. An algorithm for quadratic programming. *Naval Research Logistics Quarterly*, 3(1-2):95–110, 1956. doi:10.1002/nav.3800030109.
- 18 Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact routing in large road networks using contraction hierarchies. *Transportation Science*, 46(3):388–404, 2012. doi:10.1287/trsc.1110.0401.
- 19 Alan George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, 1973. doi:10.1137/0710032.
- 20 Moritz Hilger, Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Fast point-to-point shortest path computations with arc-flags. In *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, pages 41–72. American Mathematical Society, 2009.
- 21 Donald B. Johnson. Priority queues with update and finding minimum spanning trees. *Information Processing Letters*, 4(3):53–57, 1975. doi:10.1016/0020-0190(75)90001-0.
- 22 Sebastian Knopp, Peter Sanders, Dominik Schultes, Frank Schulz, and Dorothea Wagner. Computing many-to-many shortest paths using highway hierarchies. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX'07)*, pages 36–45, 2007. doi:10.1137/1.9781611972870.4.
- 23 Donald E. Knuth. *The Art of Computer Programming: Sorting and Searching*. Addison-Wesley, 1998.
- 24 Daniel Kusswurm. *Modern X86 Assembly Language Programming: 32-bit, 64-bit, SSE, and AVX*. Apress, 2014.
- 25 Larry J. LeBlanc, Edward K. Morlok, and William P. Pierskalla. An efficient approach to solving the road network equilibrium traffic assignment problem. *Transportation Research*, 9(5):309–318, 1975. doi:10.1016/0041-1647(75)90030-1.
- 26 Dennis Luxen and Peter Sanders. Hierarchy decomposition for faster user equilibria on road networks. In *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*, pages 242–253, 2011. doi:10.1007/978-3-642-20662-7_21.
- 27 Nicolai Mallig, Martin Kagerbauer, and Peter Vortisch. mobitopp - A modular agent-based travel demand modelling framework. In *Proceedings of the 2nd International Workshop on Agent-based Mobility, Traffic and Transportation Models, Methodologies and Applications (ABMTRANS'13)*, pages 854–859, 2013. doi:10.1016/j.procs.2013.06.114.
- 28 Nicolai Mallig and Peter Vortisch. Modeling car passenger trips in mobitopp. In *Proceedings of the 4th International Workshop on Agent-based Mobility, Traffic and Transportation Models, Methodologies and Applications (ABMTRANS'15)*, pages 938–943, 2015. doi:10.1016/j.procs.2015.05.169.

- 29 Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer, 2008. doi:10.1007/978-3-540-77978-0.
- 30 Enock T. Mtoi and Ren Moses. Calibration and evaluation of link congestion functions: Applying intrinsic sensitivity of link speed as a practical consideration to heterogeneous facility types within urban network. *Journal of Transportation Technologies*, 4:141–149, 2014. doi:10.4236/jtts.2014.42014.
- 31 John D. Murchland. Road traffic distribution in equilibrium. In *Proceedings of Mathematical Methods in the Economic Sciences*, 1969.
- 32 Michael Patriksson. *The Traffic Assignment Problem: Models and Methods*. Topics in Transportation. VSP, 1994.
- 33 Srinivas Peeta and Athanasios K. Ziliaskopoulos. Foundations of dynamic traffic assignment: The past, the present and the future. *Networks and Spatial Economics*, 1(3-4):233–265, 2001. doi:10.1023/A:1012827724856.
- 34 Peter Sanders and Dominik Schultes. Highway hierarchies hasten exact shortest path queries. In *Proceedings of the 13th Annual European Symposium on Algorithms (ESA'05)*, pages 568–579, 2005. doi:10.1007/11561071_51.
- 35 Aaron Schild and Christian Sommer. On balanced separators in road networks. In *Proceedings of the 14th International Symposium on Experimental Algorithms (SEA'15)*, pages 286–297, 2015. doi:10.1007/978-3-319-20086-6_22.
- 36 Johannes Schlaich, Udo Heidl, and R. Pohlner. Verkehrsmodellierung für die Region Stuttgart: Schlussbericht. unpublished, 2011.
- 37 Yosef Sheffi. *Urban Transportation Networks: Equilibrium Analysis with Mathematical Programming Methods*. Prentice Hall, 1985.
- 38 Verband Region Stuttgart. Mobilität und Verkehr in der Region Stuttgart 2009/2010: Regionale Haushaltsbefragung zum Verkehrsverhalten. *Schriftenreihe Verband Region Stuttgart*, 29:1–138, 2011.
- 39 John G. Wardrop. Some theoretical aspects of road traffic research. In *Proceedings of the Institution of Civil Engineers*, pages 325–362, 1952. doi:10.1680/ipeds.1952.11259.
- 40 Hiroki Yanagisawa. A multi-source label-correcting algorithm for the all-pairs shortest paths problem. In *24th IEEE International Symposium on Parallel and Distributed Processing (IPDPS'10)*, pages 1–10, 2010. doi:10.1109/IPDPS.2010.5470362.

Engineering Motif Search for Large Motifs

Petteri Kaski

Department of Computer Science, Aalto University
Espoo, Finland
petteri.kaski@aalto.fi

Juho Lauri

Nokia Bell Labs
Dublin, Ireland
juho.lauri@nokia-bell-labs.com

Suhas Thejaswi

Department of Computer Science, Aalto University
Espoo, Finland
suhas.muniyappa@aalto.fi

Abstract

Given a vertex-colored graph H and a multiset M of colors as input, the *graph motif* problem asks us to decide whether H has a connected induced subgraph whose multiset of colors agrees with M . The graph motif problem is NP-complete but known to admit randomized algorithms based on *constrained multilinear sieving* over $\text{GF}(2^b)$ that run in time $O(2^k k^2 m M(2^b))$ and with a false-negative probability of at most $k/2^{b-1}$ for a connected m -edge input and a motif of size k . On modern CPU microarchitectures such algorithms have practical edge-linear scalability to inputs with billions of edges for small motif sizes, as demonstrated by Björklund, Kaski, Kowalik, and Lauri [ALENEX'15]. This scalability to large graphs prompts the dual question whether it is possible to scale to large motif sizes.

We present a *vertex-localized* variant of the constrained multilinear sieve that enables us to obtain, in time $O(2^k k^2 m M(2^b))$ and for every vertex simultaneously, whether the vertex participates in at least one match with the motif, with a per-vertex probability of at most $k/2^{b-1}$ for a false negative. Furthermore, the algorithm is easily vector-parallelizable for up to 2^k threads, and parallelizable for up to $2^k n$ threads, where n is the number of vertices in H . Here $M(2^b)$ is the time complexity to multiply in $\text{GF}(2^b)$.

We demonstrate with an open-source implementation that our variant of constrained multilinear sieving can be engineered for vector-parallel microarchitectures to yield hardware utilization that is bound by the available memory bandwidth.

Our main engineering contributions are (a) a version of the recurrence for tightly labeled arborescences that can be executed as a sequence of memory-and-arithmetic coalescent parallel workloads on multiple GPUs, and (b) a bit-sliced low-level implementation for arithmetic in characteristic 2 to support (a).

2012 ACM Subject Classification Mathematics of computing → Graph algorithms, Mathematics of computing → Probabilistic algorithms, Theory of computation → Parallel algorithms

Keywords and phrases algorithm engineering, constrained multilinear sieving, graph motif problem, multi-GPU, vector-parallel, vertex-localization

Digital Object Identifier 10.4230/LIPIcs.SEA.2018.28

Funding The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement 338077 "Theory and Practice of Advanced Search and Enumeration".



© Petteri Kaski, Juho Lauri, and Suhas Thejaswi;
licensed under Creative Commons License CC-BY

17th International Symposium on Experimental Algorithms (SEA 2018).

Editor: Gianlorenzo D'Angelo; Article No. 28; pp. 28:1–28:19

Leibniz International Proceedings in Informatics



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

Acknowledgements We gratefully acknowledge the use of computational resources provided by the Aalto Science-IT project at Aalto University and by CSC – IT Center for Science, Finland.

1 Introduction

Computer microarchitectures are increasingly *vector-parallel*, or, *single-instruction-multiple-data* (SIMD) parallel. While such parallelism can be harnessed to an impressive effect in many applications, a number of applications still remain where vector-parallel algorithm designs have not yet been deployed in a manner that seeks to utilize the maximum parallel bandwidth obtainable, especially in terms of memory bandwidth. Problems with (sparse) graph inputs are a particular case where high-bandwidth vectorization is not immediate, in particular because traversing the edges of a graph via adjacency lists produces a data-dependent¹ pattern of memory accesses that does not vectorize easily, unless the algorithm design is such that each traversal of an edge involves accesses to one or more long vectors of consecutive words in memory. The protagonist of this paper is the following NP-complete graph problem, which we will show admits such vectorizable algorithm designs.

The Graph Motif Problem. The *graph motif* problem asks, given a vertex-colored graph H (the *host* graph) and a multiset M of colors (the *motif* or the *query*) as input, whether H has a connected induced subgraph whose multiset of colors agrees with M . The set of vertices of such a connected induced subgraph is called a *match* to the query. The graph motif problem appears, for example, as a query problem for protein interaction networks in computational biology (cf. Lacroix, Fernandes, and Sagot [24] and Bruckner, Hüffner, Karp, Shamir, and Sharan [8]); we postpone a further discussion of earlier work to the end of this section.

Asymptotically the currently fastest algorithm designs for the graph motif problem are based on algebraic methods. For a connected host graph with m edges and a motif of size k , *constrained multilinear sieving* (cf. Björklund, Kaski, and Kowalik [6]) over the finite field $\text{GF}(2^b)$ enables a randomized algorithm that runs in time $O(2^k k^2 m M(2^b))$ and reports a false negative with probability at most $k/2^{b-1}$ (cf. Björklund, Kaski, Kowalik, and Lauri [7]). Here $M(2^b) = O(b \log b)$ is the time complexity of multiplication in $\text{GF}(2^b)$, cf. Lin, Al-Naffouri, Han, and Chung [25].

We observe in particular that the time complexity $O(2^k k^2 m M(2^b))$ of the Björklund–Kaski–Kowalik–Lauri design scales linearly in the number of edges m . Conversely, it is known that the exponential 2^k scaling in the motif size is the best possible unless there is a breakthrough in the complexity of the set cover problem (cf. Björklund, Kaski, and Kowalik [6, Theorem 6]).²

From a practical engineering perspective, constrained multilinear sieving is known to scale essentially in an edge-linear manner on modern CPU microarchitectures to graphs with hundreds of millions to billions of edges on a single compute node, when the motif size is small (cf. Björklund, Kaski, Kowalik, and Lauri [7]).

The present paper studies the dual question, namely what kind of empirical scalability can one obtain as a function of increasing motif size k . This question is motivated, for

¹ Indeed, while an adjacency list itself is read linearly from consecutive memory addresses, an algorithm typically must use the outcomes of such reads to address its further memory accesses, for example, to an array with one entry for each vertex in the graph.

² More precisely, an $O^*((2 - \epsilon)^k)$ -time design for some constant $\epsilon > 0$ for the graph motif problem would imply a $O^*((2 - \delta)^n)$ -time design for the set cover problem on a universe of size n for some constant $\delta > 0$.

example, when one is searching for a group of vertices with a specific color composition, but these vertices need not be immediately adjacent but rather some uncertain distance away from each other in the host graph. In this setting, the task can still be formulated as an instance of the graph motif problem,³ but this instance will have a somewhat larger size k for the motif. In this situation, theory tells us that it will be difficult to obtain a base algorithm design with better than 2^k scalability in k , so one is essentially forced to seek efficiency through implementation engineering. Here constrained multilinear sieving is an excellent base design for vector-parallelization, in particular since the algorithm evaluates the same multivariate polynomial P (defined by H) at 2^k distinct points (defined by the colors of H and M together with randomization). Furthermore, one can use a small field size 2^b and yet obtain good control on the probability of false negatives.

Our Contributions. Using the open-source CPU-based parallel implementation of Björklund, Kaski, Kowalik, and Lauri [7] as a starting point, we engineer an implementation of motif search that runs on compute nodes with one or more GPUs with performance that for large motif sizes achieves the empirical peak transfer bandwidth of the GPU on-device memory. Our present implementation is tailored for NVIDIA GPUs, but relies on design principles that generalize to other vectorized microarchitectures, including CPUs with vector units.

In more detail, our contributions are as follows:

- (i) *Vertex-localized sieving.* We develop a novel variant of the constrained multilinear sieve that operates simultaneously on a family of multivariate polynomials, one polynomial P_i for each vertex $i \in V(H)$, rather than a single polynomial P as in the original design. This re-design comes at asymptotically no extra cost and it enables us to *localize the outcome at vertices*, that is, with a single run of the sieve, we obtain *for every vertex* i the output whether there is at least one match that contains i , with a per-vertex probability of a false negative of at most $k/2^{b-1}$. This localization is advantageous in situations when the motif is large and the host has only isolated⁴ matches to the motif.
- (ii) *Coalescent recurrence for tightly labeled arborescences.* The most performance-critical aspect of the constrained multilinear sieve is the recurrence for the 2^k evaluations of each of the n polynomials P_i . We engineer a version of the recurrence that can be executed on multiple GPUs as a sequence of k workloads. For large k , these workloads are arithmetic-and-memory coalescent to the length of the available vectorization. A key principle is to design the memory layout of the recurrence so that each thread can work with the widest coalesced per-thread load and store instructions supported by the architecture. This is (i) to saturate the memory pipeline, and (ii) to supply each thread with enough local work that can be executed in low-latency per-thread registers to enable effective latency hiding. (Cf. Volkov [35] and Mei and Chu [27] for a discussion of latency hiding and memory hierarchies on GPUs.)
- (iii) *Bit-sliced arithmetic in characteristic 2.* Constrained multilinear sieving runs over a finite field $\text{GF}(2^b)$, which requires a fast implementation for finite-field multiplication. Asymptotically it is known that one can multiply in time $M(2^b) = O(b \log b)$, cf. Lin,

³ Potentially with generalization to each vertex being colored with a set of colors instead of a single color, to enable e.g. wild-card matches to accommodate for uncertainty.

⁴ In precise terms, when the vertices that are part of at least one match induce a subgraph whose each connected component is a match. In this situation, one run of the vertex-localized sieve produces *all matches* to a query, with total effort that scales *linearly* in the number of edges m . Indeed, we first run the vertex-localized sieve, and then run, for example, a depth-first search on the vertices contained in at least one match to identify the connected components.

Al-Naffouri, Han, and Chung [25]. Here we look at implementation for small values of b and rely on independent repetitions of the sieve to decrease the probability of false negatives. Obtaining a high-performance implementation presents a minor engineering obstacle due to the fact that the instruction set of e.g. NVIDIA GPUs does not directly support multiplication in characteristic 2, whereas modern CPUs implement instruction set extensions with such support (cf. Gueron and Kounavis [16]). We rely on software techniques and use *bit-slicing* (cf. Biham [3] and Rudra, Dubey, Jutla, Kumar, Rao, and Rohatgi [34]) to multiply in parallel in units of 32 elements of $\text{GF}(2^8)$ at a time using a simplified Mastrovito [26] multiplier implemented with Boolean word operations. We also experiment with other implementations for arithmetic in characteristic 2, but bit-slicing is clearly the fastest, yielding multiplication rates of more than 2.4 trillion $\text{GF}(2^8)$ -multiplications per second on an NVIDIA Tesla V100 SXM2 Accelerator, cf. Table 4 in the Appendix.

- (iv) *Open-source implementation.* To encourage and ease further contributions, we release our implementation as open-source software under the MIT License.⁵

Earlier Work. Motif search on graphs is a hard generalization of jumbled pattern matching on strings (cf. [9, 14, 15, 18]) that was introduced by Lacroix, Fernandes, and Sagot [24] in a bioinformatics context. Multiple variants and extensions of the base variant studied in the present paper have been introduced and studied in a number of works, including Bruckner, Hüffner, Karp, Shamir, and Sharan [8], Dondi, Fertin, and Vialette [11], Pinter and Zehavi [32, 33, 37], Björklund, Kaski, Kowalik [6], Bonnet and Sikora [12], and Zehavi [38].

From the perspective of parameterized algorithms [10], Fellows, Fertin, Hermelin, and Vialette [13] established that the graph motif problem (parameterized by the motif size k) is fixed-parameter tractable using the color-coding technique of Alon, Yuster, and Zwick [1]. The scaling $f(k)$ as a function of the parameter k was improved in a sequence of works [2, 6, 17, 21, 31], including the randomized $O^*(2.54^k)$ -time algorithm of Koutis [21], the randomized $O^*(2^k)$ -time algorithm of Björklund, Kaski, and Kowalik [6], and the deterministic $O^*(5.22^k)$ -time algorithm of Pinter, Scachnai, and Zehavi [31].⁶

Multilinear sieving and constrained multilinear sieving was developed in a sequence of works starting from pioneering work by Koutis [20], Williams [36], Koutis and Williams [22], and Koutis [21] on algebraic fingerprinting in group algebras (cf. Koutis and Williams [22]). The multivariate polynomial version of the sieve was developed by Björklund [4], Björklund, Husfeldt, Kaski, and Koivisto [5], and Björklund, Kaski, and Kowalik [6].

The generating polynomial P to capture connected sets of vertices in graphs can be traced back to Nederlof’s [28] insight on branching walks for space-efficient algebraization of the Steiner tree problem. Guillemot and Sikora [17] transported this insight to the graph motif problem. The generating polynomial was further enhanced by Björklund, Kaski, and Kowalik [6], and Björklund, Kaski, Kowalik, and Lauri [7].

2 Scalar Recurrences for Sieving with Localization

This section details all the scalar recurrences in our vertex-localized algorithm, where by *scalar* we mean an element of the finite field \mathbb{F}_{2^b} of size 2^b for a positive integer b . For an integer n , let us write $[n] = \{1, 2, \dots, n\}$. This section only describes the algorithm;

⁵ Available at: <https://github.com/pkaski/motif-localized>

⁶ Here the asymptotic notation $O^*(\cdot)$ suppresses a multiplicative factor polynomial in the input size.

for reasons of space, we will include the mathematical derivation of the algorithm and its correctness analysis in an extended version of this work.

The input to the algorithm consists of a vertex-colored host graph H and a motif M . Here H is an undirected simple graph with vertex set $V(H)$ and edge set $E(H)$, the vertex-coloring is a function $c : V(H) \rightarrow C$ for a set of colors C , and the motif is a function $M : C \rightarrow \mathbb{Z}_{\geq 0}$ with $\sum_{q \in C} M(q) = k$ for a positive integer k .

For convenience in what follows, let us assume that the vertices of H are numbered $1, 2, \dots, n$; that is, we assume that $V(H) = [n]$. Let us also introduce a set S_q of *shades* for each color $q \in C$, with $|S_q| = M(q)$ and $S_q \cap S_{q'} = \emptyset$ for all distinct $q, q' \in C$. For a vertex $i \in [n]$, let us write $\Gamma_H(i)$ for the set of vertices adjacent to i in H . Since H is simple, for each $j \in \Gamma_H(i)$ there is a unique edge $ij \in E(H)$ that joins i and j in H .

The algorithm now consists of the following five steps, where the key vectorizability property for implementation is highlighted in the main recurrence (d).

- (a) **Assign Random Values.** First, draw an independent uniform random value $\mu_{i,d} \in \mathbb{F}_{2^b}$ for each $i \in [n]$ and $d \in S_{c(i)}$. Then, draw an independent uniform random value $\nu_{d,\ell} \in \mathbb{F}_{2^b}$ for each $d \in \cup_{q \in C} S_q$ and $\ell \in [k]$. Finally, draw an independent uniform random value $\alpha_{s,(i,j)} \in \mathbb{F}_{2^b}$ for each $s = 2, 3, \dots, k$ and each orientation $(i, j) \in [n] \times [n]$ of an undirected edge $ij \in E(H)$ in H .
- (b) **Label Evaluation.** For each $i \in [n]$ and $\ell \in [k]$, compute

$$\zeta_{i,\ell} = \sum_{d \in S_{c(i)}} \mu_{i,d} \nu_{d,\ell} \in \mathbb{F}_{2^b}. \quad (1)$$

We can parallelize this step over i and ℓ as appropriate.

- (c) **Initialize Label-Sum Vectors.** For each subset $L \subseteq [k]$, compute the vector

$$\zeta^L = (\zeta_1^L, \zeta_2^L, \dots, \zeta_n^L) \in \mathbb{F}_{2^b}^n \quad (2)$$

given for all $i \in [n]$ by

$$\zeta_i^L = \sum_{\ell \in L} \zeta_{i,\ell} \in \mathbb{F}_{2^b}. \quad (3)$$

We can parallelize this step over L and i as appropriate.

- (d) **The Main Recurrence.** First, for $s = 1$ and each $i \in [n]$ and $L \subseteq [k]$, set

$$P_{i,1}(\zeta^L, \alpha) = \zeta_i^L. \quad (4)$$

Then, for each $s = 2, 3, \dots, k$, $i \in [n]$, and $L \subseteq [k]$, compute

$$P_{i,s}(\zeta^L, \alpha) = \sum_{j \in \Gamma_H(i)} \alpha_{s,(i,j)} \sum_{\substack{s_1+s_2=s \\ s_1, s_2 \geq 1}} P_{i,s_1}(\zeta^L, \alpha) P_{j,s_2}(\zeta^L, \alpha). \quad (5)$$

For each fixed value of s , we can parallelize this recurrence over i and L as appropriate. Furthermore, *the parallelization over L vectorizes*. That is, for each of the 2^k choices $L \subseteq [k]$, the index j ranges over precisely the same values in $\Gamma_H(i)$, and thus we can view the recurrence (5) as a recurrence *over vectors of length 2^k* , where the multiplication by $\alpha_{s,(i,j)}$ can be viewed as scalar-multiplication applied to a vector obtained as the sum of element-wise (Hadamard) products of vectors. This vectorizability is the gist of our GPU implementation described in the next section.

(e) **Sum at Each Vertex.** For each $i \in [n]$, take the sum over $L \subseteq [k]$ to obtain

$$Q_{i,k}(\mu, \nu, \alpha) = \sum_{L \subseteq [k]} P_{i,k}(\zeta^L, \alpha). \quad (6)$$

We can parallelize this recurrence over i and over L as appropriate; parallelization over L can use routine parallel aggregation techniques.

This algorithm has the property that $Q_{i,k}(\mu, \nu, \alpha) = 0$ with probability 1 when there is no match to M in H that contains the vertex i , and $Q_i(\mu, \nu, \alpha) = 0$ with probability at most $(2k - 1)/2^b$ when there is a match to M in H that contains the vertex i .

3 Engineering a Vector-Parallel Implementation

This section gives a high-level description of our implementation of the algorithm in §2. We intentionally avoid low-level details specific to a particular programming API such as CUDA for NVIDIA microarchitectures, with the understanding that the low-level details can be found in the accompanying source code. Our focus here is on principles that we believe generalize and support implementations on current and future vector-parallel architectures.

Workloads and Coalescence. It will be convenient to employ the following framework to describe at a high level how our implementation is structured.⁷ Suppose we run a parallel workload that consists of work by W independent parallel threads, which have been arranged into a tensor of *volume* W with r *modes*, to obtain a tensor of *shape*⁸

$$W_r \times W_{r-1} \times \cdots \times W_1$$

for positive integers W_1, W_2, \dots, W_r with $W = W_r W_{r-1} \cdots W_1$.

Each thread $t = 0, 1, \dots, W - 1$ in this workload can now be identified with its r -tuple of coordinates $(t_r, t_{r-1}, \dots, t_1)$ defined by $t = \sum_{j=1}^r t_j W_{j-1} W_{j-2} \cdots W_1$ and $t_j \in \{0, 1, \dots, W_j - 1\}$ for all $j = 1, 2, \dots, r$. In essence, the tuple $(t_r, t_{r-1}, \dots, t_1)$ represents the integer t as a mixed-base r -digit integer in the number system defined by the sequence $(W_r, W_{r-1}, \dots, W_1)$, where W_1 is the base of the least significant digit, W_2 is the base of the next least significant digit, and so on until W_r , which is the base of the most significant digit. To utilize vector-parallel hardware effectively, a key design principle is to ensure that the workload is *coalesced*, that is, any two threads t and t' that agree in all but possibly their c least significant coordinates are at all times executing the same instruction, and when this instruction is a memory access, the access is to units of memory at either *a single constant address* or *consecutive addresses* across the c least significant coordinates.⁹

Per-Thread Work Allocation and Memory Layout. When engineering an algorithm design for vector-parallel hardware with long latencies, among the most important considerations is to decide precisely what each thread in a parallel workload of W threads is going to do to

⁷ A reader familiar e.g. with NVIDIA CUDA API should have no difficulty translating this framework to NVIDIA-specific terminology e.g. in terms of grids of thread blocks.

⁸ We use the symbol “ \times ” to exclusively refer to Cartesian products and shapes of tensors, never for multiplication. For basic terminology on tensors, see Kolda and Bader [19].

⁹ Precisely how large values c and $W_c W_{c-1} \cdots W_1$ one needs depends on the width of the hardware vectorization. For example, in case of NVIDIA microarchitectures, one usually wants $W_c W_{c-1} \cdots W_1$ to be a positive multiple of 32 to ensure coalescent execution of warps.

saturate the hardware and to hide latency (cf. Volkov [35]). Three key objectives underlying such a decision are to

- (i) expose sufficient parallelism in the design to enable a large W to saturate the hardware;
- (ii) ensure coalescent execution by a careful ordering of modes in the workload; and
- (iii) make sure each thread works with enough local data to make use of low-latency storage available to each thread and/or to select groups of threads.¹⁰

Interleaved with the question of per-thread work allocation is the question how to implement the memory layout of the algorithm across the memory hierarchy so that

- (iv) local storage with lowest latency is the most frequently accessed type of storage;
- (v) when accessing high-latency storage, as much data as possible should be accessed with a single access to saturate the pipeline; and
- (vi) accesses to memory are coalesced.

Design Choices for Vertex-Localized Sieving. Let us now turn to how we implement the recurrences in §2 as parallel workloads together with their memory layouts. For reasons of space, we focus only on the GPU-side workloads; the CPU-side workloads are those of Björklund, Kaski, Kowalik, and Lauri [7], with only minor modifications to support vertex-localization.

Top-Level Structure. We execute the random assignment step (a) and the label evaluation step (b) on the host CPU(s), with parallelization over i . The subsequent steps (c,d,e) each vector-parallelize over the 2^k evaluations indexed by $L \subseteq [k]$, and these steps will be offloaded for execution on the available GPU(s) using a sequence of parallel workloads.

Workloads on the GPU. Since label-sum initialization (c) and the main recurrence (d) both vector-parallelize over $L \subseteq [k]$, it is natural to design the workloads so that a divisor D of 2^k appears in the least significant mode of each workload to enable coalescent execution when k is large enough.¹¹ Furthermore, since we want simultaneous localization at each $i \in [n]$, it follows that n is a natural mode of the workload.¹² Indeed, although execution along this mode will not be coalesced, each of the D threads working with the same vertex will follow the same pattern of data-dependent memory accesses when traversing the adjacency list associated with the vertex i to obtain each vertex $j \in \Gamma_H(i)$ in (5). Finally, to ensure sufficient local data for low-latency computations (cf. (iii) in §3), we design the workload so that each thread that implements (5) works with S out of the 2^k scalars, where S divides 2^k . Thus, we will execute workloads of shape

$$n \times D,$$

where each thread will work with S scalars, so that SD divides 2^k .¹³ When furthermore we have M devices (GPUs) available, where MSD divides 2^k , we execute workloads of shape

$$M \times n \times D.$$

¹⁰Here the registers locally available to each thread form the lowest-latency storage, whereas e.g. the shared memory available to each block of threads executing in a streaming multiprocessor in NVIDIA microarchitectures would form the next-lowest level of latency above the register file. Cf. Mei and Chu [27] for an empirical study of GPU memory architectures and their latency considerations.

¹¹For example, on NVIDIA microarchitectures it is desirable to have $D \geq 32$.

¹²We furthermore assume that n has been rounded up to the closest power of 2 by inserting isolated vertices.

¹³For example, our bit-sliced implementation for arithmetic on $\text{GF}(2^8)$ discussed in what follows assumes $S = 32$. With $D \geq 32$, we thus need $k \geq 10$ for coalescent execution.

with the mode of length M parallelized over the GPUs. That is, for each of the M GPUs we perform the following (asynchronously and in parallel on each GPU): First, we upload the evaluated labels (the output of (b)) from host memory to on-device memory on each GPU. Then, we iterate $\frac{2^k}{MSD}$ sequences of workloads, where each sequence consists of label-sum initialization (c) implemented with a single workload of shape $n \times D$, the main recurrence (d) implemented using a sequence of k workloads of shape $n \times D$, and the per-vertex sum (e) implemented as a standard batch-parallel sum that aggregates the nDS scalars output by (d) to n scalars. Finally, we download the n scalars to host memory. This results in Mn scalars downloaded from M GPUs in host memory, which we aggregate on the host to obtain n scalars, one scalar at each vertex, with parallelization over the vertices. The iteration over $\frac{2^k}{MSD}$ such sequences produces the final output (6) at each vertex $i \in [n]$.

Memory Layout on Each GPU. We now describe the memory layout used on each GPU. Since we execute workloads of $n \times D$ threads, and each thread works with S scalars, we want to access such S -scalar units of data with as-efficient-as-possible coalesced accesses (cf. §3 (v) and (vi)). Towards this end, suppose that S scalars occupy U words of memory and suppose the maximum amount of memory one thread can access (with a single load or store instruction) is A words, where A divides U . Then, to obtain coalesced memory accesses, we use a memory layout of shape

$$\frac{U}{A} \times n \times D \times A$$

and execute the loads/stores of S scalars per thread in groups of $\frac{U}{A}$ instructions that each load/store A words of data.¹⁴ Adjacency lists of vertices and scalar associations with (oriented) edges are implemented as simple contiguous arrays of words. Due to the $n \times D$ workload to implement (5), each group of D threads working on a vertex $i \in [n]$ will access the same element j of an adjacency list or oriented-edge-associated scalar $\alpha_{s,(i,j)}$, implying coalescent execution for large enough D .

4 Experiments and Conclusion

This section reports on experiments with our algorithm implementation that extends the CPU implementation of Björklund, Kaski, Kowalik, and Lauri [7] with a vector-parallel GPU implementation of our vertex-localized sieve (cf. §2 and §3). Our implementation is prepared with CUDA C [29] using the OpenMP API [30] for host-side parallelization and to enable parallelization across multiple GPUs. The running times of CPU experiments are measured using OpenMP wall-clock time interface and the running time of GPU experiments are measured using CUDA event API. Memory usage is tracked using wrapper functions for the standard memory allocation interfaces. Memory bandwidth is tracked by computing the total amount of memory read/written in bytes by each recurrence and dividing by the measured running time. Arithmetic bandwidth is tracked by computing the number of scalar multiplications in each recurrence and dividing by the measured running time.

Our experiments use one CPU and four GPU configurations with a full hardware and software description provided in Appendix A. Here we give a short overview of the CPU node and the main GPU node in the experiments:

¹⁴In concrete terms, for NVIDIA microarchitectures and our bit-sliced arithmetic for $\text{GF}(2^8)$ we choose $S = 32$, $A = 4$ and $U = 8$, measured in 32-bit words. That is, at CUDA level each thread reads/writes data in groups that consist of two coalesced `uint4`-accesses to global memory.

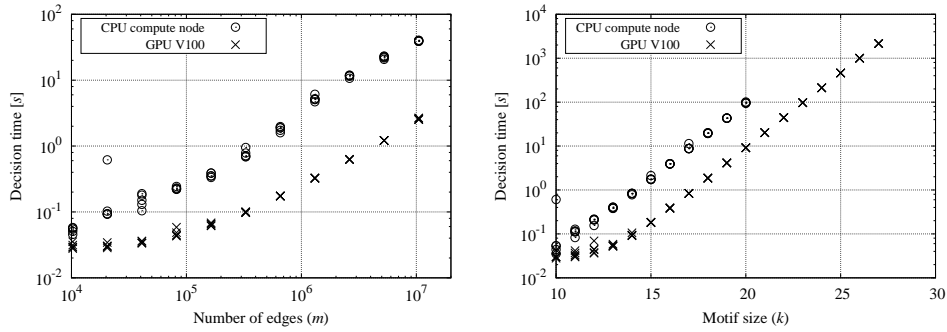
■ **Table 1** Speedup obtained with a single GPU and eight GPUs compared with a CPU-only implementation as we increase the motif size k . We perform experiments on five independent d -regular random graphs for each $n = 2^{10}$ fixed, $d = 20$ fixed and $k = 10, 11, \dots, 20$. The CPU-only experiments are performed on the CPU compute node with the $64 \times \text{GF}(2^8)$ bit-packed line type configured for the Björklund–Kaski–Kowalik–Lauri [7] implementation without vertex-localization. The GPU experiments with our implementation are configured with the $32 \times \text{GF}(2^8)$ bit-sliced line type and executed on the V100 GPU compute node (single V100 device and eight V100 devices). All the running times are in seconds. The column “CPU” displays in each row the minimum time over the five graphs, whereas the columns “GPU V100” and “ $8 \times \text{GPU V100}$ ” displays in each row the maximum time over the five graphs. The column “Speedup (GPU)” displays the ratio of the columns “CPU” and “GPU V100”, while the column “Speedup (Multi-GPU)” is the ratio of column “CPU” and “ $8 \times \text{GPU V100}$ ”.

k	CPU	GPU V100	$8 \times \text{GPU V100}$	Speedup (GPU)	Speedup (Multi-GPU)
10	0.0352 s	0.0432 s	0.0957 s	0.81	0.37
11	0.0828 s	0.0416 s	0.1180 s	1.99	0.70
12	0.1553 s	0.0696 s	0.0938 s	2.23	1.66
13	0.3808 s	0.0585 s	0.1046 s	6.51	3.64
14	0.7768 s	0.1062 s	0.1025 s	7.31	7.58
15	1.7244 s	0.1847 s	0.1111 s	9.33	15.52
16	3.9035 s	0.3968 s	0.1474 s	9.84	26.48
17	8.7340 s	0.8377 s	0.1906 s	10.43	45.82
18	19.3674 s	1.8950 s	0.3564 s	10.22	54.34
19	42.9873 s	4.1417 s	0.6480 s	10.38	66.34
20	94.2593 s	9.1468 s	1.2425 s	10.31	75.86

CPU compute node. An Apollo 6000 XL230a G9 blade server with two 2.6-GHz Intel Xeon E5-2690v3 CPUs (Haswell microarchitecture, 24 cores, 12 cores/CPU, no hyper-threading, 30 MiB L3 cache/CPU) and 128 GiB of main memory (8×16 GiB DDR4-2133 HP 752369-081).

V100 GPU compute node (NVIDIA DGX-1). An NVIDIA Tesla V100 SXM2 Accelerator device with one 1312-MHz NVIDIA GV100 GPU (Volta microarchitecture, 5120 cores, 80 SMs, 64 cores/SM) and 16384 MiB of on-device 4096-bit HBM2 with ECC enabled. The host is an NVIDIA DGX-1 with two 2.2-GHz Intel Xeon E5-2698v4 CPUs (Broadwell microarchitecture, 40 cores, 20 cores/CPU, hyper-threading enabled, 50 MiB L3 cache/CPU) and 512 GiB of main memory (16×32 GiB DDR4-2133 Samsung M393A4K40BB1-CRC). The host contains eight V100 devices.

Input Graphs. We use three synthetic graph topologies (regular, clique, and power-law) for our experiments, making use of the random graph generator from Björklund, Kaski, Kowalik, and Lauri [7]. The generator produces identical graph instances across all configurations to enable comparison between configurations. In addition, we use natural graph topologies obtained from the Koblenz Network Collection [23]. The following specific natural graphs from the Koblenz collection are considered in our experiments: Google [[web-Google](#)], Douban [[douban](#)], WordNet [[wordnet-words](#)], Stack Overflow [[stackexchange-stackoverflow](#)], Discogs [[discogs_affiliation](#)], MovieLens 10M [[movielens-10m_rating](#)], Hamsterster friendships [[petster-friendships-hamster](#)], Adolescent health [[moreno_health](#)], and Human protein (Stelzl) [[maayan-Stelzl](#)]; click on the text in brackets to follow the hyperlink. Each natural graph is preprocessed as in [7], i.e. (i) the vertices are randomly relabeled and (ii) to guarantee a unique match, a vertex is chosen uniformly at random and a monochromatic motif is placed on the first k vertices expanded by a depth-first search.



■ **Figure 1** Scalability and speedup of the vertex-localized implementation. We compare the running time of CPU-only and single-GPU implementations as we increase the number of edges m (left) for five independent d -regular random graphs for each $n = 2^{10}, 2^{11}, \dots, 2^{20}$, with $d = 20$ fixed and $k = 10$ fixed. We observe that our implementation scales linearly, as expected, with little variance between inputs except for small input sizes. For $n = 2^{20}$ with $k = 10$, our implementation on a single V100 GPU device is at least fourteen times faster than the CPU-only implementation on the CPU compute node. We compare the running time of the CPU-only and single-GPU implementations as we increase the motif size k (right) for five independent d -regular random graphs for each $n = 2^{20}$ fixed, $d = 20$ fixed and $k = 10, 11, \dots, 30$. We observe that our implementation scales exponentially with respect to the motif size, as expected, with little variance between inputs. For $n = 2^{10}$ and $k = 20$, our implementation on a single V100 GPU device is at least ten times faster than the CPU-only implementation on the CPU compute node. The CPU-only experiments are executed on the CPU compute node with the $64 \times \text{GF}(2^8)$ bit-packed line type configured for the Björklund–Kaski–Kowalik–Lauri [7] implementation without vertex-localization. The GPU experiments with our implementation are configured with the $32 \times \text{GF}(2^8)$ bit-sliced line type and executed on the V100 GPU compute node, using a single V100 device.

Scalability and Speedup. Our first set of experiments studies the scalability of our implementation and compares our implementation with the CPU-only parallel implementation of Björklund, Kaski, Kowalik, and Lauri [7]. We report the running time as a function of, (i) number of edges m , and (ii) the motif size k . The results of the experiments are displayed in Figures 1 and 2. In Table 1, we observe speedups of several tenfolds for a sufficiently large k for the GPU and multi-GPU implementations over the CPU-only implementation. Table 5 in Appendix A reports the speedup of GPU and multi-GPU implementations with respect to the CPU-only implementation for increasing number of edges, while Table 6 considers the speedup of the multi-GPU over the GPU implementation for even larger k . We observe a substantial benefit both from (a) offloading from the host to a GPU device, and (b) offloading to multiple GPU devices compared with a single device.

Topology-Invariance. Our final set of experiments studies the effect of graph topology to scaling. Figure 3 displays the running times for regular, clique, power-law and natural topologies as a function of increasing m and increasing k . We observe that the running time is essentially invariant across topologies for increasing k . However, for increasing m , we observe differences in performance between topologies, with the worst performance occurring for power-law topologies with small exponent α . This is due to our design choice to use an $n \times D$ workload for the main recurrence (d), whereby the n groups of (length- D -vectorized) threads operating on different vertices have running times that are proportional to the degree of each vertex. When many groups of high-degree vertices get scheduled on the same streaming multiprocessor (SM) of the GPU, this produces an uneven distribution of work across SMs

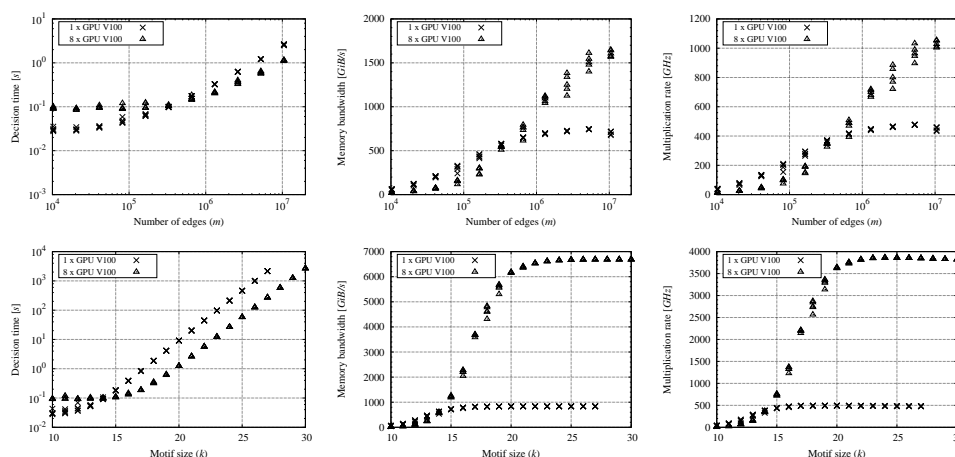
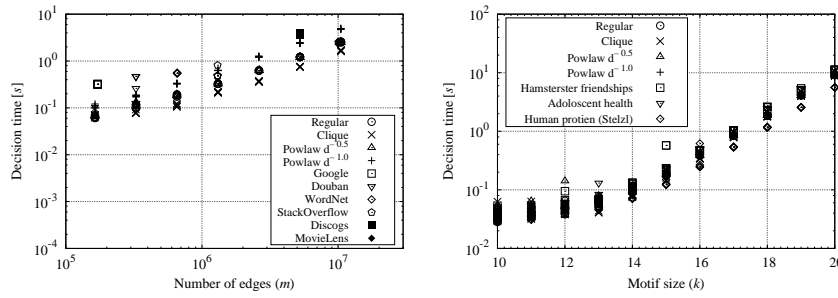


Figure 2 Scaling of our vertex-localized implementation as we increase the number of edges m (top row) and the motif size k (bottom row). For both rows, we display the runtime (left), memory bandwidth (middle), and arithmetic bandwidth (right). The experiments are configured with the $32 \times \text{GF}(2^8)$ bit-sliced line type and executed on the V100 GPU compute node. **Top row** experiments are run for five independent d -regular random graphs for each $n = 2^{10}, 2^{11}, \dots, 2^{20}$, $d = 20$ fixed and $k = 10$ fixed. For $n = 2^{20}$ with $k = 10$, the multi-GPU implementation executed on a configuration with eight V100 GPU devices is at least two times faster than the single-GPU implementation executed on a single V100 GPU device. We observe that the multi-GPU speedup becomes systematic only for large-enough m . For $n = 2^{20}$ and $k = 10$, with a single V100 device we obtain a memory bandwidth of at least 677 GiB/s and a simultaneous arithmetic bandwidth of more than 430 billion $\text{GF}(2^8)$ -multiplications per second. With eight V100 devices, we observe the results have more variance. For $n = 2^{20}$ and $k = 10$, we obtain at least 1567 GiB/s of memory bandwidth, and a simultaneous arithmetic bandwidth of at least 1003 billion $\text{GF}(2^8)$ -multiplications per second. **Bottom row** experiments are run for five independent d -regular random graphs for each $n = 2^{10}$, $d = 20$ fixed and $k = 10, 11, \dots, 30$. For $n = 2^{10}$ and $k = 20$, our implementation on a single V100 GPU device is at least ten times faster than the CPU-only implementation on the CPU compute node. For $n = 2^{10}$ and $20 \leq k \leq 27$, the multi-GPU implementation executed on a configuration with eight V100 GPU devices is at least seven times faster than the single-GPU implementation executed on a single V100 GPU device. For $n = 2^{10}$ and $k = 27$, with a single V100 device we obtain a memory bandwidth of at least 835 GiB/s and a simultaneous arithmetic bandwidth of more than 480 billion $\text{GF}(2^8)$ -multiplications per second. With eight V100 devices, we obtain at least 6680 GiB/s of memory bandwidth, and a simultaneous arithmetic bandwidth of at least 3820 billion $\text{GF}(2^8)$ -multiplications per second. We observe that for large k these memory bandwidths noticeably *exceed the measured peak transfer bandwidth for on-device global memory* in Table 3, which we believe is caused by the relatively small value of n and in-SM-caching of the $P_{i,s_1}(\zeta^L, \alpha)$ -values when executing the recurrence (5) across different $j \in \Gamma_H(i)$.

and delays the completion of the workload. Thus, our present implementation has topology-dependent performance, with the best performance obtained on graphs with a uniform distribution of vertex degrees. With nonuniform vertex degrees, a random permutation of the vertices (or, for example, a greedy bin-packing of the vertices by degree to the SMs) can be used to balance the load across SMs. Such load-balancing was not considered for the present implementation.



■ **Figure 3** Topology-invariance. We display the runtime of our vertex-localized implementation with respect to number of edges (left) with different graph topologies: (a) five independent synthetic graphs for each value of n ; and (b) five random relabelings of each natural graph. *Regular* graphs with $n = 2^{14}, 2^{15}, \dots, 2^{20}$, $d = 20$ fixed; *cliques* with $n = 2^{13}, 2^{14}, \dots, 2^{19}$, $d = 40$ fixed; *power-law* graphs with $n = 2^{14}, 2^{15}, \dots, 2^{20}$, $D = 20$, $w = 100$ for both $\alpha = -0.5$ and $\alpha = -1.0$. All instances have $k = 10$. We display the runtime of our implementation with respect to the motif size (right) for different graph topologies: (a) five independent synthetic graphs for each value of k ; and (b) five random relabelings of natural graph for each value of k . *Regular* graphs with $n = 2^{10}$, $d = 20$; *cliques* with $n = 2^{10}$, $d = 40$; *power-law* graphs with $n = 2^{10}$, $D = 20$, $w = 100$ for both $\alpha = -0.5$ and $\alpha = -1.0$. The motif size varies from $k = 10, 11, \dots, 20$. The experiments are configured with the $32 \times \text{GF}(2^8)$ bit-sliced line type and executed on the V100 GPU compute node. All experiments use a single V100 device. For the motif-size scaling, the running times are essentially invariant, as expected. However, for scaling with respect to number of edges m , power-law graphs with $\alpha = -1.0$ consume approximately three times the running time of the clique graphs, and this ratio becomes more systematic with increasing m . In particular we observe that our present implementation is not completely topology-invariant; graphs with uniform degree distribution have relatively better performance than the graphs with non-uniform degree distribution (cf. §4).

More Data and Experiments in the Appendix. The Appendix contains a more extensive set of experiments.¹⁵ For example, Figure 7 in Appendix A displays the running time of vertex-localization and decision-only variants. We observe that the overhead caused by vertex-localization is, as expected, negligible. In addition, the Appendix presents performance for our hardware configurations (Tables 2 and 3), arithmetic bandwidth measurements with different implementations of finite-field arithmetic (Table 4 and Figures 4, 5, and 6), and more detailed speedup tables (Tables 5 and 6).

Conclusion. This paper presented a vertex-localized variant of constrained multilinear sieving that enables algorithm engineering for vector-parallel microarchitectures such as single-GPU and multi-GPU configurations ranging from thousands to tens of thousands of cores, with tenfold to several tenfold speedups for large motif sizes compared with a carefully optimized parallel multi-CPU implementation [7]. The two key aspects of the present algorithm design that enable scalability are (i) the vectorization of the data-dependent memory accesses so that each traversal of an edge along an adjacency list results in memory accesses to long vectors of words at consecutive accesses when evaluating the main recurrence (cf. §2(d)), and (ii) the vectorization of the finite-field arithmetic in characteristic 2 through bit-slicing.

¹⁵ *Caveat on false negatives.* The current set of experiments is still lacking an experiment that studies the empirical false negative rate compared with the theoretical per-vertex probability of at most $(2k-1)/2^b$ for a false negative. Such an experiment is warranted since our implementation uses $b = 8$ and thus independent repetitions of the sieve must be used when one seeks to control the false negative rate over the vertices.

References

- 1 Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *J. ACM*, 42(4):844–856, 1995.
- 2 Nadja Betzler, Michael R. Fellows, Christian Komusiewicz, and Rolf Niedermeier. Parameterized algorithms and hardness results for some graph motif problems. In *Proceedings of the 19th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 5029 of *Lecture Notes in Computer Science*, pages 31–43. Springer, 2008.
- 3 Eli Biham. A fast new DES implementation in software. In *Proceedings of the 4th International Workshop on Fast Software Encryption (FSE)*, volume 1267 of *Lecture Notes in Computer Science*, pages 260–272. Springer, 1997.
- 4 Andreas Björklund. Determinant sums for undirected hamiltonicity. *SIAM J. Comput.*, 43(1):280–299, 2014.
- 5 Andreas Björklund, Thore Husfeldt, Petteri Kaski, and Mikko Koivisto. Narrow sieves for parameterized paths and packings. *J. Comput. Syst. Sci.*, 87:119–139, 2017.
- 6 Andreas Björklund, Petteri Kaski, and Łukasz Kowalik. Constrained multilinear detection and generalized graph motifs. *Algorithmica*, 74(2):947–967, 2016.
- 7 Andreas Björklund, Petteri Kaski, Łukasz Kowalik, and Juho Lauri. Engineering motif search for large graphs. In *Proceedings of the 17th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 104–118. SIAM, 2015.
- 8 Sharon Bruckner, Falk Hüffner, Richard M. Karp, Ron Shamir, and Roded Sharan. Topology-free querying of protein interaction networks. *Journal of Computational Biology*, 17(3):237–252, 2010.
- 9 Ferdinando Cicalese, Travis Gagie, Emanuele Giaquinta, Eduardo Sany Laber, Zsuzsanna Lipták, Romeo Rizzi, and Alexandru I. Tomescu. Indexes for jumbled pattern matching in strings, trees and graphs. In *Proceedings of the 20th International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 8214 of *Lecture Notes in Computer Science*, pages 56–63. Springer, 2013.
- 10 Marek Cygan, Fedor V. Fomin, Łukasz Kowalik, Daniel Lokshantov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015.
- 11 Riccardo Dondi, Guillaume Fertin, and Stéphane Vialette. Maximum motif problem in vertex-colored graphs. In *Proceedings of the 20th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 5577 of *Lecture Notes in Computer Science*, pages 221–235. Springer, 2009.
- 12 Édouard Bonnet and Florian Sikora. The graph motif problem parameterized by the structure of the input graph. *Discrete Applied Mathematics*, 231:78–94, 2017.
- 13 Michael R. Fellows, Guillaume Fertin, Danny Hermelin, and Stéphane Vialette. Upper and lower bounds for finding connected motifs in vertex-colored graphs. *J. Comput. Syst. Sci.*, 77(4):799–811, 2011. doi:10.1016/j.jcss.2010.07.003.
- 14 Travis Gagie, Danny Hermelin, Gad M. Landau, and Oren Weimann. Binary jumbled pattern matching on trees and tree-like structures. In *Proceedings of the 21st Annual European Symposium on Algorithms (ESA)*, volume 8125 of *Lecture Notes in Computer Science*, pages 517–528. Springer, 2013.
- 15 Emanuele Giaquinta and Szymon Grabowski. New algorithms for binary jumbled pattern matching. *Inf. Process. Lett.*, 113(14-16):538–542, 2013.
- 16 Shay Gueron and Michael E. Kounavis. *Intel® Carry-Less Multiplication Instruction and its Usage for Computing the GCM Mode - Rev 2.02*. Intel Corporation, April 2014. [Link].
- 17 Sylvain Guillemot and Florian Sikora. Finding and counting vertex-colored subtrees. *Algorithmica*, 65(4):828–844, 2013.
- 18 Tomasz Kociumaka, Jakub Radoszewski, and Wojciech Rytter. Efficient indexes for jumbled pattern matching with constant-sized alphabet. In *Proceedings of the 21st Annual*

- European Symposium on Algorithms (ESA)*, volume 8125 of *Lecture Notes in Computer Science*, pages 625–636. Springer, 2013.
- 19 Tamara G. Kolda and Brett W. Bader. Tensor decompositions and applications. *SIAM Rev.*, 51(3):455–500, 2009.
 - 20 Ioannis Koutis. Faster algebraic algorithms for path and packing problems. In *Proceedings of the 35th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 5125 of *Lecture Notes in Computer Science*, pages 575–586. Springer, 2008.
 - 21 Ioannis Koutis. Constrained multilinear detection for faster functional motif discovery. *Inf. Process. Lett.*, 112(22):889–892, 2012. doi:10.1016/j.ip1.2012.08.008.
 - 22 Ioannis Koutis and Ryan Williams. Algebraic fingerprints for faster algorithms. *Commun. ACM*, 59(1):98–105, 2016.
 - 23 Jérôme Kunegis. KONECT: the Koblenz network collection. In *Proceedings of the 22nd International World Wide Web Conference (WWW)*, pages 1343–1350, 2013. [Link].
 - 24 Vincent Lacroix, Cristina G. Fernandes, and Marie-France Sagot. Motif search in graphs: Application to metabolic networks. *IEEE/ACM Trans. Comput. Biology Bioinform.*, 3(4):360–368, 2006. doi:10.1109/TCBB.2006.55.
 - 25 Sian-Jheng Lin, Tareq Y. Al-Naffouri, Yungshiang S. Han, and Wei-Ho Chung. Novel polynomial basis with fast Fourier transform and its application to Reed-Solomon erasure codes. *IEEE Trans. Inform. Theory*, 62(11):6284–6299, 2016.
 - 26 Edoardo Mastrovito. *VLSI Architectures for Computations in Galois Fields*. PhD thesis, Department of Electrical Engineering, Linköping University, 1991.
 - 27 Xinxin Mei and Xiaowen Chu. Dissecting GPU memory hierarchy through microbenchmarking. *IEEE Trans. Parallel Distrib. Syst.*, 28(1):72–86, 2017.
 - 28 Jesper Nederlof. Fast polynomial-space algorithms using inclusion-exclusion. *Algorithmica*, 65(4):868–884, 2013.
 - 29 NVIDIA Corporation. *CUDA C Programming Guide, Version 9*. [Link].
 - 30 OpenMP Architecture Review Board. *OpenMP Application Program Interface, Version 4.5 – November 2015*. [Link].
 - 31 Ron Y. Pinter, Hadas Shachnai, and Meirav Zehavi. Deterministic parameterized algorithms for the graph motif problem. *Discrete Applied Mathematics*, 213:162–178, 2016.
 - 32 Ron Y. Pinter and Meirav Zehavi. Partial information network queries. In *Proceedings of the 24th International Workshop on Combinatorial Algorithms (IWOCA)*, volume 8288 of *Lecture Notes in Computer Science*, pages 362–375. Springer, 2013.
 - 33 Ron Y. Pinter and Meirav Zehavi. Algorithms for topology-free and alignment network queries. *J. Discrete Algorithms*, 27:29–53, 2014.
 - 34 Atri Rudra, Pradeep K. Dubey, Charanjit S. Jutla, Vijay Kumar, Josyula R. Rao, and Pankaj Rohatgi. Efficient Rijndael encryption implementation with composite field arithmetic. In *Proceedings of the 3rd International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 2162 of *Lecture Notes in Computer Science*, pages 171–184. Springer, 2001.
 - 35 Vasily Volkov. *Understanding Latency Hiding on GPUs*. PhD thesis, Electrical Engineering and Computer Sciences, University of California at Berkeley, 2016. Technical Report No. UCB/EECS-2016-143.
 - 36 Ryan Williams. Finding paths of length k in $O^*(2^k)$ time. *Inf. Process. Lett.*, 109(6):315–318, 2009. doi:10.1016/j.ip1.2008.11.004.
 - 37 Meirav Zehavi. Parameterized algorithms for module motif. In *Proceedings of the 38th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, volume 8087 of *Lecture Notes in Computer Science*, pages 825–836. Springer, 2013.
 - 38 Meirav Zehavi. Parameterized algorithms for the module motif problem. *Inf. Comput.*, 251:179–193, 2016.

A Additional Experimental Results

Additional Hardware Configurations. Our experiments use one CPU and four GPU configurations. The hardware configurations of CPU compute node and V100 GPU compute node are reported in Section 4. Each experiment documents the hardware configuration and the number of GPU devices used for the experiment.

K40 GPU compute node. An NVIDIA Tesla K40t Accelerator device with one 745-MHz NVIDIA GK110B GPU (Kepler microarchitecture, 2880 cores, 15 SMs, 192 cores/SM) and 12288 MiB of on-device GDDR5-3004 memory with ECC enabled. The host is a Bullx B715 DLC blade server with two 2.1-GHz Intel Xeon E5-2620v2 CPUs (Ivy Bridge microarchitecture, 12 cores, 6 cores/CPU, no hyper-threading, 15 MiB L3 cache) and 32 GiB of main memory (8×4 GiB DDR3-1600 Samsung M393B5273DH0-CMA). The host and the GPU are connected by a 16-lane PCI Express 3.0 bus. The host contains two K40t devices.

K80 GPU compute node. An NVIDIA Tesla K80 Accelerator device with two 875-MHz NVIDIA Tesla GK210 GPUs (Kepler microarchitecture, 2496 cores, 13 SMs, 192 cores/SM), 12288 MiB of on-device GDDR5-3004 memory with ECC enabled. The host is a Dell PowerEdge C4130 machine with two 2.4-GHz Intel Xeon E5-2620v3 CPUs (Haswell microarchitecture, 12 cores, 6 cores/CPU, no hyper-threading, 15 MiB L3 cache/CPU) and 128 GiB of main memory (16×8 GiB DDR4-2133 Hynix HMA42GR7AFR4N-TF). The host and the GPU are connected by a 16-lane PCI Express 3.0 bus. The host contains four K80 devices.

P100 GPU compute node. An NVIDIA Tesla P100 Accelerator device with one 1189-MHz NVIDIA GP100 GPU (Pascal microarchitecture, 3584 cores, 56 SMs, 64 cores/SM) and 16384 MiB of on-device 4096-bit HBM2 with ECC enabled. The host is a Dell PowerEdge C4130 with two 2.54-GHz Intel Xeon E5-2680v3 CPUs (Haswell microarchitecture, 24 cores, 12 cores/CPU, no hyper-threading, 30 MiB L3 cache/CPU) and 256 GiB of main memory (16×16 GiB DDR4-2133 Hynix HMA82GR7MFR8N-UH). The host and the device are connected by a 16-lane PCI Express 3.0 bus. The host contains four P100 devices.

Baseline Performance. Tables 2, 3, and 4 in Appendix A report the empirical baseline performance of the hardware. The host (CPU) memory bandwidth is measured by operating on a two gibibyte array of 64-bit words. Each experiment is executed five times and the average of five iterations is reported. Arithmetic bandwidth¹⁶ is measured for different implementations of arithmetic in lines of S scalars per thread (cf. §3). The bit-sliced $32 \times \text{GF}(2^8)$ line type has the best bandwidth, and this line type is used in our subsequent experiments. The on-device (GPU) and device–host (device-to-host and host-to-device) memory transfer rates are measured using the bandwidth-test tool distributed in the CUDA Examples package using a single device at a time.

¹⁶The number of scalar multiplications per second.

■ **Table 2** Memory bandwidths of CPU compute node, P100 GPU compute node (host) and V100 compute node (host).

Benchmark	Single core	All cores
<i>CPU compute node</i>		
Read from linear addresses (consecutive 64-bit words)	9.13 GiB/s	46.93 GiB/s
Write to linear addresses (consecutive 64-bit words)	5.69 GiB/s	21.92 GiB/s
Read from random addresses (individual 64-bit words)	0.29 GiB/s	5.28 GiB/s
Read from random addresses (full cache lines)	1.46 GiB/s	19.94 GiB/s
<i>P100 GPU compute node (host memory)</i>		
Read from linear addresses (consecutive 64-bit words)	10.34 GiB/s	41.07 GiB/s
Write to linear addresses (consecutive 64-bit words)	8.43 GiB/s	22.91 GiB/s
Read from random addresses (individual 64-bit words)	0.52 GiB/s	5.53 GiB/s
Read from random addresses (full cache lines)	1.55 GiB/s	19.80 GiB/s
<i>V100 GPU compute node (host memory)</i>		
Read from linear addresses (consecutive 64-bit words)	9.58 GiB/s	36.39 GiB/s
Write to linear addresses (consecutive 64-bit words)	7.78 GiB/s	19.27 GiB/s
Read from random addresses (individual 64-bit words)	0.46 GiB/s	5.20 GiB/s
Read from random addresses (full cache lines)	1.45 GiB/s	20.14 GiB/s

■ **Table 3** Memory bandwidth of the P100 and V100 GPU compute nodes, using one of the four P100 devices and one of the eight V100 devices, respectively.

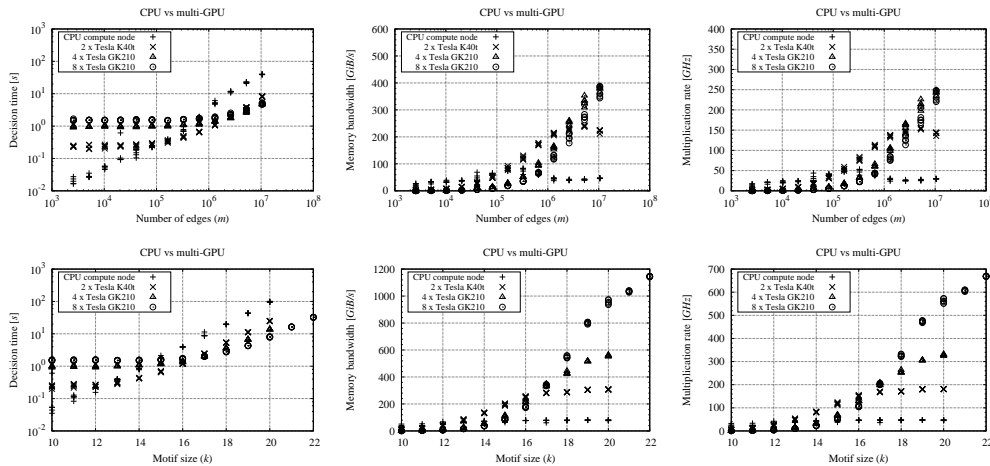
Node	Host to device	Device to host	Device to device
P100 GPU compute node	12.16 GiB/s	12.87 GiB/s	498.15 GiB/s
V100 GPU compute node	10.79 GiB/s	12.17 GiB/s	720.77 GiB/s

■ **Table 4** Arithmetic bandwidth of the P100 and V100 GPU compute nodes with different line implementations, using one of the P100 devices and one of the eight V100 devices, respectively.

Line type	Line multiplication	Scalar multiplication
<i>P100 GPU</i>		
1 × GF(2 ⁸) bit-packed line	82.04 GHz	82.68 GHz
4 × GF(2 ⁸) bit-packed line	328.20 GHz	331.09 GHz
16 × GF(2 ⁸) bit-packed line	348.30 GHz	348.30 GHz
32 × GF(2 ⁸) bit-sliced line	1462.62 GHz	1500.70 GHz
1 × GF(2 ⁸) lookup table	65.23 GHz	85.58 GHz
4 × GF(2 ⁸) lookup table	64.59 GHz	86.64 GHz
16 × GF(2 ⁸) lookup table	64.56 GHz	84.54 GHz
32 × GF(2 ⁸) lookup table	64.80 GHz	84.52 GHz
<i>V100 GPU</i>		
1 × GF(2 ⁸) bit-packed line	159.58 GHz	159.72 GHz
4 × GF(2 ⁸) bit-packed line	638.25 GHz	638.88 GHz
16 × GF(2 ⁸) bit-packed line	652.86 GHz	723.69 GHz
32 × GF(2 ⁸) bit-packed line	661.59 GHz	719.31 GHz
32 × GF(2 ⁸) bit-sliced line	2486.54 GHz	2441.10 GHz
1 × GF(2 ⁸) lookup table	410.52 GHz	496.94 GHz
4 × GF(2 ⁸) lookup table	408.91 GHz	510.29 GHz
16 × GF(2 ⁸) lookup table	409.91 GHz	531.27 GHz
32 × GF(2 ⁸) lookup table	409.93 GHz	528.28 GHz

■ **Table 5** Speedup obtained with a single GPU and eight GPUs compared with a CPU-only implementation as we increase the number of edges m . We perform experiments on five independent random d -regular graphs for each $n = 2^{10}, 2^{11}, \dots, 2^{20}$, with $d = 20$ fixed and $k = 10$ fixed. The CPU-only experiments are performed on the CPU compute node with the $64 \times \text{GF}(2^8)$ bit-packed line type configured for the Björklund–Kaski–Kowalik–Lauri [7] implementation without vertex-localization. The GPU experiments with our implementation are configured with the $32 \times \text{GF}(2^8)$ bit-sliced line type and executed on the V100 GPU compute node (single V100 device and eight V100 devices). All the running times are in seconds. The column “CPU” displays in each row the minimum time over the five graphs, whereas the columns “GPU V100” and “ $8 \times \text{GPU V100}$ ” displays in each row the maximum time over the five graphs. The column “Speedup (GPU)” displays the ratio of the columns “CPU” and “GPU V100”, while the column “Speedup (Multi-GPU)” is the ratio of column “CPU” and “ $8 \times \text{GPU V100}$ ”.

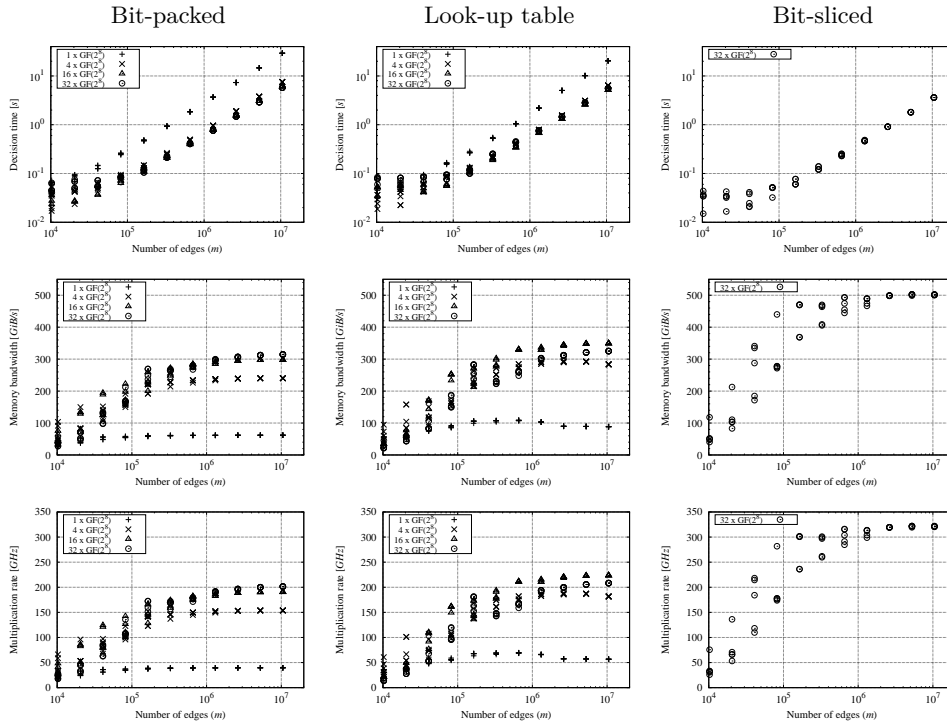
n	CPU	GPU V100	$8 \times \text{GPU V100}$	Speedup (GPU)	Speedup (Multi-GPU)
2^{10}	0.0456 s	0.0353 s	0.1009 s	1.29	0.45
2^{11}	0.0923 s	0.0344 s	0.0924 s	2.68	1.00
2^{12}	0.1042 s	0.0363 s	0.1081 s	2.87	0.96
2^{13}	0.2199 s	0.0586 s	0.1204 s	3.75	1.83
2^{14}	0.3339 s	0.0685 s	0.1242 s	4.88	2.69
2^{15}	0.6859 s	0.1014 s	0.1109 s	6.76	6.18
2^{16}	1.5949 s	0.1752 s	0.1842 s	9.10	8.66
2^{17}	4.7021 s	0.3292 s	0.2171 s	14.28	21.66
2^{18}	10.6986 s	0.6302 s	0.4022 s	16.98	26.60
2^{19}	20.7284 s	1.2187 s	0.6465 s	17.01	32.06
2^{20}	38.8648 s	2.6742 s	1.1552 s	14.53	33.64



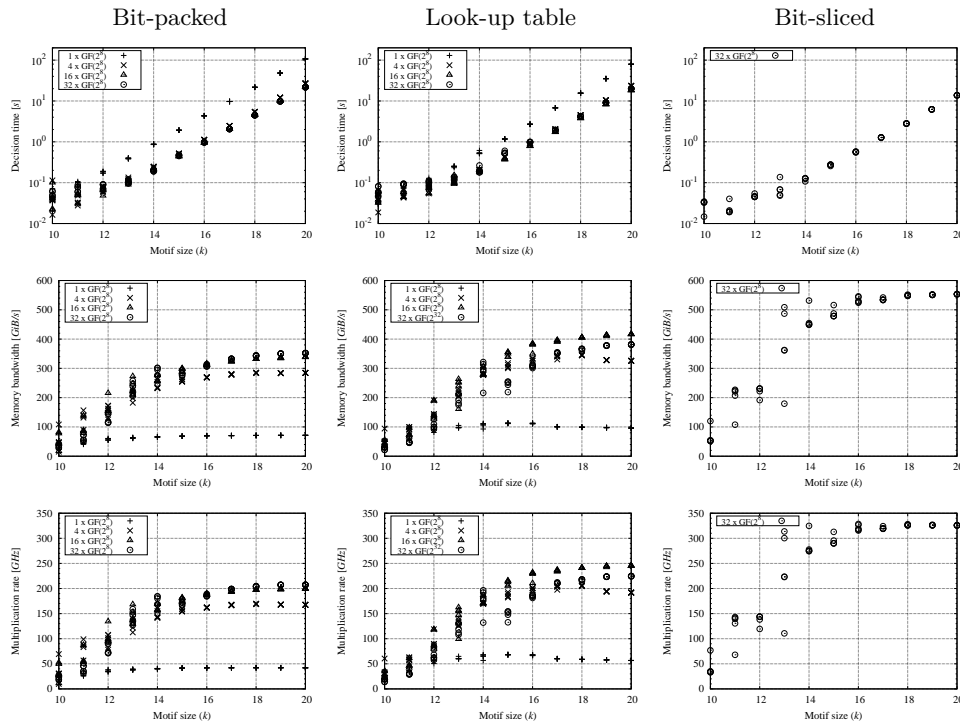
■ **Figure 4** Performance of the vertex-localized implementation in older GPU microarchitectures as we increase, (i) the number of edges m , and (ii) the motif size k . In the top row, we display the running time (left), memory bandwidth (center) and arithmetic bandwidth (right) for five independent d -regular random graphs with $n = 2^{10}, 2^{11}, \dots, 2^{19}$, $d = 20$ fixed and $k = 10$ fixed. In bottom row, we display the running time (left), memory bandwidth (center) and arithmetic bandwidth (right) for five independent d -regular random graphs with $n = 2^{10}$ fixed, $d = 20$ fixed and $k = 10, 11, \dots, 22$. Each configuration is reserved exclusively for the experiments at hand. The GPU experiments with our implementation are configured with the $32 \times \text{GF}(2^8)$ bit-sliced line type. The CPU-only experiments use the Björklund–Kaski–Kowalik–Lauri [7] implementation with the $64 \times \text{GF}(2^8)$ bit-packed line type.

■ **Table 6** Speedup obtained with eight GPUs compared with a single GPU as we increase the motif size k . We perform experiments on five independent graph inputs for each $n = 2^{10}$ fixed, $d = 20$ fixed and $k = 21, 22, \dots, 30$. The GPU experiments with our implementation are configured with the $32 \times \text{GF}(2^8)$ bit-sliced line type and executed on the V100 GPU compute node. All the running times are in seconds. The column “GPU V100” displays in each row the minimum time over the five graphs when using a single V100 device, whereas the column “ $8 \times \text{GPU V100}$ ” displays in each row the maximum time over the five graphs when using four V100 devices. The column “Speedup” displays the ratio of the columns “GPU V100” and “ $8 \times \text{GPU V100}$ ”.

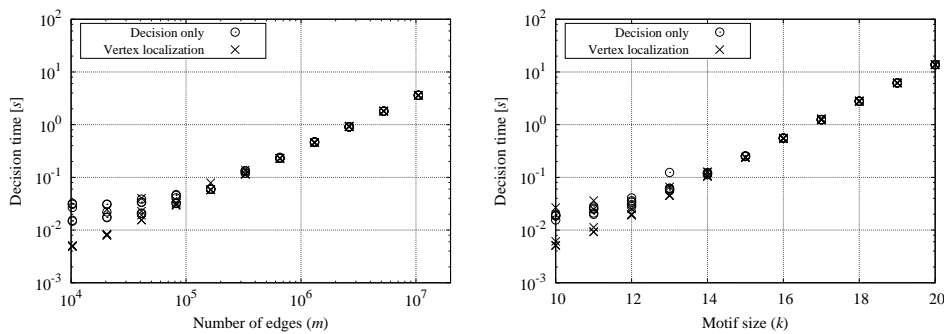
k	GPU V100	$8 \times \text{GPU V100}$	Speedup
21	20.1962 s	2.6556 s	7.61
22	44.4414 s	5.6852 s	7.82
23	97.2945 s	12.3074 s	7.91
24	212.3904 s	26.6797 s	7.96
25	461.7525 s	57.8421 s	7.98
26	1000.1718 s	125.0891 s	8.00
27	2160.4430 s	270.0623 s	8.00
28	-	581.5915 s	-
29	-	1249.4652 s	-
30	-	2676.9140 s	-



■ **Figure 5** Performance comparison of different scalar line types with increasing number of edges m . We display the runtime (top row), memory bandwidth (middle row) and arithmetic bandwidth (bottom row) of five independent d -regular random graphs for each $n = 2^8, 2^9, \dots, 2^{20}$, $d = 20$ fixed and $k = 10$ fixed. The experiments are configured with each line type and executed on the P100 GPU compute node. All experiments use a single P100 device. The bit-sliced $32 \times \text{GF}(2^8)$ line type has the best performance.



■ **Figure 6** Performance comparison of different scalar line types with increasing motif size k . We display the runtime (top row), memory bandwidth (middle row) and arithmetic bandwidth (bottom row) for five independent d -regular random graphs for each $n = 2^{10}$ fixed, $d = 20$ fixed and $k = 10, 11, \dots, 20$. The experiments are configured with each line type and executed on the P100 GPU compute node. All experiments use a single P100 device. The bit-sliced $32 \times \text{GF}(2^8)$ line type has the best performance.



■ **Figure 7** Overhead of vertex-localization. Here we compare our vertex-localized implementation against a separately prepared GPU implementation that uses the original Björklund–Kaski–Kowalik–Lauri design [7] without vertex-localization. We show scaling as a function of the number of edges (left) and the motif size (right). The left plot is the running time of five independent d -regular random graphs for each configuration of $n = 2^{10}, 2^{11}, \dots, 2^{20}$, $d = 20$ fixed and $k = 10$ fixed. The right plot is the running time of five independent d -regular random graphs for each $n = 2^{10}$ fixed, $d = 20$ fixed and $k = 10, 11, \dots, 20$. The experiments are configured with the $32 \times \text{GF}(2^8)$ bit-sliced line type and executed on the P100 GPU compute node. All experiments use a single P100 device. We observe that the overhead of vertex-localization is negligible, as expected.

Finding Hamiltonian Cycle in Graphs of Bounded Treewidth: Experimental Evaluation

Michał Ziobro

Theoretical Computer Science Department, Faculty of Mathematics and Computer Science,
Jagiellonian University, Kraków, Poland
michal.18.ziobro@student.uj.edu.pl

Marcin Pilipczuk

Institute of Informatics, University of Warsaw, Poland
malcin@mimuw.edu.pl

Abstract

The notion of treewidth, introduced by Robertson and Seymour in their seminal Graph Minors series, turned out to have tremendous impact on graph algorithmics. Many hard computational problems on graphs turn out to be efficiently solvable in graphs of bounded treewidth: graphs that can be swept with separators of bounded size. These efficient algorithms usually follow the dynamic programming paradigm.

In the recent years, we have seen a rapid and quite unexpected development of involved techniques for solving various computational problems in graphs of bounded treewidth. One of the most surprising directions is the development of algorithms for connectivity problems that have only single-exponential dependency (i.e., $2^{\mathcal{O}(t)}$) on the treewidth in the running time bound, as opposed to slightly superexponential (i.e., $2^{\mathcal{O}(t \log t)}$) stemming from more naive approaches. In this work, we perform a thorough experimental evaluation of these approaches in the context of one of the most classic connectivity problem, namely HAMILTONIAN CYCLE.

2012 ACM Subject Classification Theory of computation → Parameterized complexity and exact algorithms, Theory of computation → Graph algorithms analysis, Theory of computation → Dynamic programming

Keywords and phrases Empirical Evaluation of Algorithms, Treewidth, Hamiltonian Cycle

Digital Object Identifier 10.4230/LIPIcs.SEA.2018.29

Funding Supported by the “Recent trends in kernelization: theory and experimental evaluation” project, carried out within the Homing programme of the Foundation for Polish Science co-financed by the European Union under the European Regional Development Fund.

1 Introduction

The problem of finding HAMILTONIAN CYCLE in graph is one of the oldest and best known \mathcal{NP} -complete problems. It was intensely studied together with its more generic optimization version TRAVELING SALESMAN PROBLEM. Early and important result on this problem was dynamic algorithm invented independently by Bellman [2] and Held and Karp [16], running in time $O(2^n n^2)$. The exponential factor of this running time bound remains the best known for deterministic algorithms up to today, and a faster randomized Monte Carlo algorithm has been shown only very recently by Björklund [3]. Faster algorithms were also obtained for some special cases, like graphs with bounded degree [9, 4] or claw-free graphs [7].

An important class of graphs in which many combinatorial problems can be solved more efficiently, are graphs of bounded *treewidth*. Treewidth, introduced by Robertson and

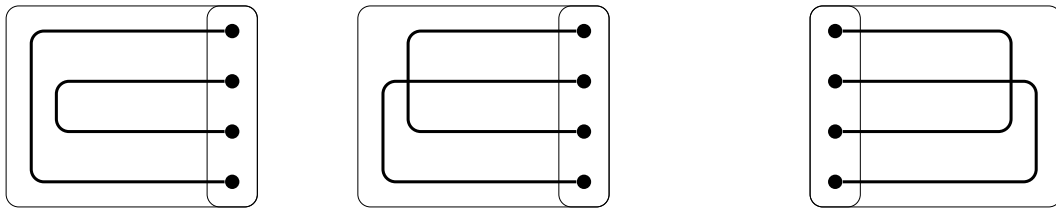


© Michał Ziobro and Marcin Pilipczuk;
licensed under Creative Commons License CC-BY
17th International Symposium on Experimental Algorithms (SEA 2018).

Editor: Gianlorenzo D’Angelo; Article No. 29; pp. 29:1–29:14



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



■ **Figure 1** A separator S with two possible partial solutions on the left. Only the first one forms a Hamiltonian cycle with the partial solution on the right, despite that in all of them the vertices on the separator have degree 1.

Seymour in their Graph Minors project [20], measures how the input graph resembles a tree, or how can be covered by a set of bounded-sized bags organized in tree-like structure which we call *tree decomposition*. It has proven to be very useful for dealing with \mathcal{NP} -hard problems; for example, given an n -vertex graph G and its tree decomposition of width t , one can solve the MAXIMUM INDEPENDENT SET problem in G in time $2^t \cdot t^{O(1)} \cdot n$. We refer to [8] for more examples of algorithms on graphs of bounded treewidth.

Essentially every algorithm for graphs of bounded treewidth follows the paradigm of dynamic programming: it gradually (in a bottom-to-top fashion on the tree decomposition) builds partial solutions in subgraphs of the input graph. Using the fact that a bag in a tree decomposition is a separator, in many combinatorial problems it suffices to keep only a bounded (in the width of the decomposition) number of partial solutions in each step of the algorithm. To illustrate this concept, consider a separation (A, B) in a graph G with $S = A \cap B$ (i.e., $A, B \subseteq V(G)$ are two sets with $A \cup B = V(G)$ and no edge between $A \setminus B$ and $B \setminus A$), and think of a dynamic programming algorithm that processed already the graph $G[A]$, but has not yet touched $B \setminus A$. Observe that a partial solution $X \subseteq A$ to the MAXIMUM INDEPENDENT SET problem interacts with $B \setminus A$ only via the set S . Consequently, it suffices to store, for every $X_S \subseteq S$, an independent set $X \subseteq A$ of maximum possible size satisfying $X \cap S = X_S$. If the separator S is of size at most t , it leads to 2^t bound on the size of the memoization table in the dynamic programming algorithm.

In the HAMILTONIAN CYCLE problem, the natural state space for the dynamic programming algorithm is a bit more complex. A partial solution in $G[A]$ would be a set of vertex-disjoint paths \mathcal{P} that all have endpoints in S and together visit every vertex of $A \setminus B$. To complete the partial solution \mathcal{P} to a Hamiltonian cycle H in G , it seems essential to remember not only which vertices of S are visited by \mathcal{P} and which are the endpoints of paths in \mathcal{P} , but also how the paths of \mathcal{P} pair up their endpoints in S (see Figure 1). This last piece of information leads to $2^{\theta(t \log t)}$ states for separator S of size t .

Up to late 2010, almost all known algorithms for combinatorial problems in graphs of bounded treewidth follow the naive approach outlined above, and researchers' effort focused mostly on speeding up computations in the so-called *join nodes* of the decomposition (see e.g. [22]).¹ In 2010, Lokshtanov, Marx, and Saurabh proved that many such algorithms have optimal dependency on treewidth [17] (under strong complexity assumptions) and provided a framework for proving similar lower bounds for complexities of the type $2^{\theta(t \log t)}$ [18]. However, providing such a tight lower bound for the connectivity problems such as HAMILTONIAN CYCLE in graphs of bounded treewidth remained elusive.

¹ A join node of a decomposition corresponds to a node of the underlying tree of the tree decomposition of degree at least 3; intuitively, it corresponds to a bounded-size separator that splits the graph into more than 2 pieces, and in the dynamic programming algorithm one needs to merge information from at least two of such pieces.

Quite unexpectedly, a year after it turned out that there is a reason for this lack of progress, and a Monte Carlo algorithm with running time $4^t n^{\mathcal{O}(1)}$ for finding a Hamiltonian cycle in a graph with a given tree decomposition of width t has been reported [10]. The work [10] introduced a framework called Cut&Count that provided randomized single-exponential (i.e., with running time bound of the form $2^{\mathcal{O}(t)} n^{\mathcal{O}(1)}$) algorithms for many connectivity problems in graphs of bounded treewidth. The key idea of the Cut&Count method is to replace the original connectivity requirement with a different counting-mod-2 task, and ensure correctness via the Isolation Lemma [19].

In following years, a good understanding of the aforementioned improvement has been obtained by Bodlaender et al [6]. In the language of HAMILTONIAN CYCLE, a linear algebra argument shows that it suffices only to keep 4^t partial solutions instead of the naive bound of $2^{\mathcal{O}(t \log t)}$; if the memoization table grows too large, an algorithm based on Gaussian elimination is able to prune provably unnecessary states. Cygan et al. [9] provided a better basis for the Gaussian elimination step and improved the bound for the number of states for HAMILTONIAN CYCLE to $(2 + \sqrt{2})^t$. Furthermore, in [9] a matching lower bound is shown. Due to the linear algebraic nature of the argument, this approach has been dubbed in the literature as the *rank-based approach*.

In [10], an involved fast convolution algorithm has been applied to obtain the $4^t n^{\mathcal{O}(1)}$ running time bound even in computations at join nodes. The need to execute Gaussian elimination in [6] and treat join nodes in a more direct fashion in both algorithms of [6, 9] yield worse theoretical running time bounds. Thus, the algorithm [10] remains theoretically fastest in graphs of bounded treewidth to this date.

Following a recent trend in multivariate algorithmics to experimentally evaluate parameterized algorithms (led by a growing popularity of the Parameterized Algorithms and Computational Experiments Challenge [12, 11]), in this work we thoroughly evaluate the aforementioned algorithms for HAMILTONIAN CYCLE. A direct inspiration for our work is the work of Fafianie et al [13] that provided an experimental comparison of the naive and rank-based approaches for STEINER TREE (i.e., without considering the Cut&Count approach). That is, in this work we include Cut&Count and we compare the following four approaches.

naive The aforementioned naive approach with $2^{\mathcal{O}(t \log t)}$ bound on the number of states.

rank-based The approach of [6], that is, the naive approach with pruning of the state space leading to 4^t size bound.

rand-based with improved basis The approach of [9], that is, the rank-based approach with the improved basis yielding the size bound $(2 + \sqrt{2})^t$.

Cut&Count The Cut&Count algorithm of [10].

As observed in [10], the application of the Isolation Lemma in the Cut&Count method yields a relatively high polynomial factor in the running time bound, but one can replace its usage with computations over a field of characteristic 2 and randomization via the Schwartz-Zippel lemma. This replacement leads again to linear dependency on the graph size in the running time bound. We follow this path. However, as has been overlooked in [10], the fast convolution algorithm at join nodes in the $4^t n^{\mathcal{O}(1)}$ -time algorithm does not support computations over a field of characteristic 2, as it requires division by 2. Our theoretical contribution in this paper is a method around this obstacle, essentially showing that it is sufficient to perform the convolution over the ring of polynomials $\mathbb{Z}[x]$. This is described in Section 2.4 and leads to the following conclusion.

► **Theorem 1.1.** *There exists a Monte Carlo algorithm that, given an n -vertex graph G together with its tree decomposition of width t , solves HAMILTONIAN CYCLE on G in time $4^t \cdot n \cdot (t \log n)^{\mathcal{O}(1)}$.*

In Section 2 we discuss implementation details of the algorithms. Section 3 discuss experiment setup and Section 4 discuss results. We conclude in Section 5.

2 Theory and implementation details

2.1 Tree decompositions

For more background on tree decompositions and dynamic programming algorithms using them, we refer to [8]. Here, we recall only the basic notions.

For a graph G , a *tree decomposition* is a pair (T, β) where T is a tree and β assigns to every node $t \in V(T)$ a set $\beta(t) \subseteq V(G)$ called a *bag* with the following invariants: (i) for every $v \in V(G)$, the set $\{t \in V(T) \mid v \in \beta(t)\}$ is nonempty and connected in T , (ii) for every $uv \in E(G)$ there exists $t \in V(T)$ with $u, v \in \beta(t)$. The width of the tree decomposition is the maximum size of a bag, minus one, and the treewidth of a graph is the minimum possible width of its tree decomposition.

As in multiple previous results, it is convenient to describe dynamic programming algorithms on a special type of decompositions, called *nice*. A *nice tree decomposition* is a rooted tree decomposition for which the bag of the root is empty and every node is of one of the following types:

Leaf node is a node t with no children and $\beta(t) = \emptyset$.

Introduce vertex node is a node t with unique child t' and a vertex v such that $\beta(t) = \beta(t') \uplus \{v\}$.

Forget vertex node is a node t with unique child t' and a vertex $v \in \beta(t')$ such that $\beta(t) = \beta(t') \setminus \{v\}$.

Join node is a node t with exactly two children t_1 and t_2 with $\beta(t) = \beta(t_1) = \beta(t_2)$.

For a node $t \in V(T)$, we define $\gamma_{\downarrow}(t)$ to be the union of $\beta(t')$ over t' being descendants of t in T . Furthermore, let G_t be the graph $G[\gamma_{\downarrow}(t)] - E(G[\beta(t)])$ (i.e., we exclude the edges inside the bag $\beta(t)$).

Additionally, in our case it is convenient to precede every **forget vertex node** with a sequence of **introduce edge nodes**. That is, for a **forget node** t with child t' and forgotten vertex v , we take $E_{t,v}$ to be the set of edges of G that connect v with vertices of $\beta(t) \setminus \{v\}$, subdivide the edge tt' in $E(T) \setminus E_{t,v}$ $|E_{t,v}|$ times, labelling the new nodes $\{t_e \mid e \in E_{t,v}\}$, and set $\beta(t_e) = \beta(t')$. The graphs G_{t_e} are defined as follows: if t'' is the unique child of t_e , then $G_{t_e} = G_{t''} \cup \{e\}$.

The intuition of this step is as follows: there is a significant difference between the graphs $G_{t'}$ and G_t , namely $E(G_t) = E(G_{t'}) \cup E_{t,v}$. We split this change into $|E_{t,v}|$ steps, adding edges one by one.

All our implementations start with preparing a nice tree decomposition with the **introduce edge nodes**.

2.2 Naive approach

Given a node t in a nice tree decomposition (T, β) , a *partial solution* is a family \mathcal{P} of vertex-disjoint paths in G_t such that (i) every vertex of $\gamma_{\downarrow}(t) \setminus \beta(t)$ is visited by some path in \mathcal{P} , and (ii) every path in \mathcal{P} has both its endpoints in $\beta(t)$. For a partial solution \mathcal{P} at node t , we define the following objects:

a bucket b is a function $b : \beta(t) \rightarrow \{0, 1, 2\}$ that assigns to every vertex $v \in \beta(t)$ its degree in the union of \mathcal{P} ;

a **pairing** E is a family of disjoint two-element subsets of $b^{-1}(1)$ that pairs up the endpoints of the same path in \mathcal{P} .

The pair (b, E) is the *state* of \mathcal{P} . The crucial observation is that among partial solutions with the same state, it suffices to memoize only one. Note that for a given bucket b with $\ell = |b^{-1}(1)|$, there are $(\ell - 1) \cdot (\ell - 3) \cdot 3 \cdot 1$ possible pairings, giving a $2^{\theta(|\beta(t)| \log |\beta(t)|)}$ bound on the number of different states.

With this observation, it is straightforward to design a dynamic programming algorithm that finds a Hamiltonian cycle in time $2^{\mathcal{O}(t \log t)} n$ given a tree decomposition of the input graph of width t . This is exactly the naive approach.

2.3 Rank based approach

The rank-based approach is strongly based on the naive one, with main change being a pruning on the number of possible pairings.

► **Theorem 2.1** ([6]). *For a fixed node t and bucket b , given a family \mathcal{E} of pairings, one can find a subfamily $\mathcal{E}^* \subseteq \mathcal{E}$ of size at most $2^{|b^{-1}(1)|-1}$ with the following property: for every Hamiltonian cycle H in G , if \mathcal{P} is its intersection with G_t and (b, E) is the state of \mathcal{P} , and $E \in \mathcal{E}$, then there exists $E^* \in \mathcal{E}^*$ such that for every partial solution \mathcal{P}^* with state (b, E^*) , the graph $(H - E(\mathcal{P})) \cup E(\mathcal{P}^*)$ is a Hamiltonian cycle as well.*

Furthermore, given b and \mathcal{E} , one can assign to every $E \in \mathcal{E}$ a $2^{|b^{-1}(1)|-1}$ -length 0-1 vector v_E such that the family \mathcal{E}^* is defined as the indices of any maximal independent (over \mathbb{F}_2) subfamily of $\{v_E | E \in \mathcal{E}\}$.

In other words, for a fixed bucket b , it is sufficient to keep only $2^{|b^{-1}(1)|-1}$ pairings, and pruning unnecessary pairings can be done via Gaussian elimination on a matrix with $|\mathcal{E}|$ rows and $2^{|b^{-1}(1)|-1}$ columns over the field \mathbb{F}_2 (the two-element field modulo 2).

In [9], Theorem 2.1 is improved with a different construction of vectors v_E that are of length $2^{|b^{-1}(1)|/2-1}$. Furthermore, [9] showed how to use the special structure of the vectors v_E to avoid Gaussian elimination at **introduce/forget vertex/edge nodes**, yielding $(2 + \sqrt{2})^p p^{\mathcal{O}(1)} n$ -time algorithms for graphs with a given *path* decomposition of width p (i.e., without any **join nodes**).

We implement the rank-based approach both with the vector construction of [6] and the improved one of [9]. Both implementations use Gaussian elimination, as it is not known how to avoid it at join nodes.

In the implementation, the core of the naive and rank-based approaches is the same. We use two variants of the implementations: keep track of partial solutions (so that the entire Hamiltonian cycle can be returned in the end) or, in order to save space, just remember a flag and a Hamiltonian cycle is found via self-reducibility. All implementations perform the same computations specific to the node type, which are straightforward in all cases. At join nodes, the algorithm first sorts the partial solutions by buckets and then tries to match the partial solutions only for buckets that fit each other (e.g., do not exceed the bound of 2 on a degree of a vertex).

After successfully computing the set of partial solutions for a current node the algorithm runs a reduce function. In the naive approach, it only deletes the duplicates by sorting set of partial solutions and checking if the two consecutive are same or not. In rank-based approach it divides all partial solutions into buckets (same as during processing the **join node**). For each bucket it computes the necessary matrix and performs Gaussian elimination on it to get a representative set of partial solutions.

To limit the effect of self-reducibility in case of the decision-only variant, we employ a problem-specific strategy. That is, we discover the Hamiltonian cycle edge-by-edge. For a path P in G with at least two edges, we can discover if G contains a Hamiltonian cycle containing P by deleting from G all edges of $E(G) \setminus E(P)$ that are incident to internal vertices of P , and run the decision algorithm on the obtained subgraph. Given a path P , we extend it one by one by doing a binary search over the next edge incident to an endpoint of P . This gives $\mathcal{O}(n \log \Delta)$ calls to the decision version of the problem for graphs with n vertices and maximum degree Δ .

2.4 Cut&Count approach

The main idea of the Cut&Count approach [10] is to replace search for a Hamiltonian cycle with counting the following objects: a cycle cover of the graph (i.e., a subset of edges where every vertex is of degree exactly two) with an assignment of every cycle to either left or right. In this manner, a fixed cycle cover with c cycles is counted 2^c times; if we additionally force one fixed vertex to be always on the left side, we get 2^{c-1} instead. That is, every Hamiltonian cycle is counted once, and every other cycle cover is counted an even number of times.

In [10], the Isolation Lemma [19] is employed to essentially reduce to the case when we solve instances with a unique Hamiltonian cycle. Then, the parity of the count described above indicates whether the graph contains a Hamiltonian cycle. However, this approach introduces a large polynomial overhead in the running time bound: first, because of the need for self-reducibility to discover the cycle (which we handle as in the previous section) and, second, because of the use of Isolation Lemma that adds an additional “weight” dimension to the dynamic programming memoization tables.

For the second overhead, as discussed [10], it can be remedied by, instead of using the Isolation Lemma, pick a field \mathbb{F} of characteristic 2 (i.e., a field of size 2^p for some integer p), associate with each edge $e \in E(G)$ a variable x_e , associate with each cycle cover a monomial being a product of the variables associated with the edges used in the cycle cover, and compute the sum of the monomials over all cycle covers and all left/right assignment, using a random assignment of values from \mathbb{F} to variables x_e . Then, if \mathbb{F} is large enough (larger than the maximum degree of the monomial, which is n), the Schwarz-Zippel lemma ensures that with good probability the result is nonzero if and only if the graph has a Hamiltonian cycle (i.e., there is a small probability of a false negative).

In our implementation, we follow this path, using a field of size 2^{64} . This size is large enough so that the failure probability is negligible. On the other hand, there exists an efficient implementation of operations on this field using the PCLMULQDQ processor instruction for multiplication. Our implementation of the field operations follow [5].

Furthermore, as discussed in the introduction, the choice of computations over $GF(2^{64})$ rather than arguably simpler counting algorithms via the Isolation Lemma resulted also in technical problems in handling **join nodes**. As observed in [10], a natural and direct approach to a **join node** with bag of size t runs in time $9^t t^{\mathcal{O}(1)}$. In [10], this is speeded up by an involved fast convolution approach, reducing the 9^t factor to 4^t . At heart of this approach lies an algorithm to quickly compute the following convolution.

Let $f, g : \mathbb{Z}_4^m \rightarrow R$ for some ring R and integer m . We define $f * g : \mathbb{Z}_4^m \rightarrow R$ as

$$(f * g)(x) = \sum_{y \in \mathbb{Z}_4^m} f(y)g(x - y).$$

Here, the addition in \mathbb{Z}_4^m is done coordinatewise. [10] developed a FFT-like approach to computing the above convolution, yielding the following.

► **Lemma 2.2** ([10]). *Given $f, g : \mathbb{Z}_4^m \rightarrow \mathbb{Z}$, the convolution $f * g$ can be computed in $4^m m^{\mathcal{O}(1)}$ operations on \mathbb{Z} on values of the order of $2^{\mathcal{O}(m)}$ times larger than the maximum absolute value of the input functions.*

However, the proof of the above lemma involves division by a factor of 4^m , making it inapplicable directly to $R = GF(2^{64})$. To circumvent this obstacle, we developed a new variant of Lemma 2.2, building on the internal structure of the field $GF(2^{64})$. Recall that a field $GF(2^p)$ can be defined as the ring $\mathbb{Z}[x]$ divided by the ideal generated by 2 and an irreducible (in $\mathbb{F}_2[x]$) polynomial Q of degree p .

► **Lemma 2.3.** *Let $p \geq 1$ and assume that the elements of field $GF(2^p)$ are given as polynomials from $\mathbb{F}_2[x]$ of degree less than p , and multiplication in $GF(2^p)$ is done modulo a known polynomial Q of degree p . Given two function $f, g : \mathbb{Z}_4^m \rightarrow GF(2^p)$, the convolution $f * g$ can be computed in time $4^m (pm)^{\mathcal{O}(1)}$.*

Proof. We follow the same algorithm as in the proof of Lemma 2.2 from [10], but treating the values of f and g as elements of $\mathbb{Z}[x]$, not $GF(2^{64})$. This allows the necessary division steps in the algorithm, and an inspection of the proof of [10] shows that the algorithm operates on $\mathcal{O}(m)$ -bit integers and polynomials of degree $\mathcal{O}(p)$. Then, at the very end, we reduce every resulting polynomial modulo 2 and modulo Q to obtain elements of $GF(2^{64})$. ◀

However, in the above we need to depart from the efficient implementation of operations in $GF(2^{64})$, and explicitly operate on polynomials in $\mathbb{Z}[x]$ of larger degree. While theoretically sound, this is expected to give a large overhead in experiments. Consequently, we test two variants of the Cut&Count algorithm: the one using a naive approach to the join nodes in time $9^t t^{\mathcal{O}(1)}$ and the one using Lemma 2.3.

To conclude the proof of Theorem 1.1, we observe that to ensure correctness with constant probability, the Cut&Count algorithm of [10] requires field $GF(2^p)$ with $p = \Omega(\log n)$.

3 Setup

3.1 Hardware and code

All of the computations were performed on a PC with an Intel Core i5-6500 processor and 16 GB of random-access memory. The operating system used during the experiments was Arch Linux. All implementations has been done in C++, the code is available at [1, 23].

3.2 Data sets

To evaluate our algorithms we decided to use well known set of HAMILTONIAN CYCLE instances from Flinders Hamiltonian Cycle Project [15] consisting of 1001 instances. To find tree decompositions of small with, we first applied our implementation of the minimum fill-in heuristic (cf. [14]). The heuristic returned tree decompositions of width at most 8 for 623 instances, and indicated that 30 more instances may have treewidth within ranges allowing usage of our HAMILTONIAN CYCLE algorithms.

We took the aforementioned 623 instances as our main benchmark. For sake of optimizing hyperparameters of our algorithms, we sampled a subset of 30 elements.

To the aforementioned 30 instances with larger but potentially tractable treewidth, we applied the heuristic of Ben Strasser [21] that won the second place in 2017 PACE Challenge [12]. This resulted in another 19 instances with tree decompositions of width between 17 and 29. Out of these instances, 15 turned out to be tractable by our algorithms.

Furthermore, we also sampled 7 random instances in the following way: starting from a Hamiltonian cycle C , we added a number of random edges with endpoints close on the cycle C (so that the treewidth is bounded). These instances are meant to generate many partial solutions at separator, and are expected to give large advantage to rank-based approaches.

To sum up, we operate on five data sets, all but the last being subsets of the Flinders Hamiltonian Cycle Project [15]:

set A is the whole set of graphs with small treewidth recognized by our heuristic (623 instances, treewidth at most 8),

set B is a subsample of A (30 instances, treewidth at most 8),

set C is the set of larger treewidth graphs with decompositions found by [21] (19 instances, treewidth between 17 and 29).

set D is the subset of the set C that turned out to be tractable by our implementations (15 instances, treewidth between 17 and 29).

set E is a set of 7 random graphs sampled as described above.

All instances from [15] are available through their webpage. At [1] we provide a list of the used instances in each set, the set E , and the used tree decompositions for sets C and D .

3.3 Fine-tuning the frequency of Gaussian elimination

As discussed in the introduction, in the rank-based approach the theoretical running time bound is worse than the one of Cut&Count approach partially due to the need of applying Gaussian elimination on the set of partial solutions. It is expected that the Gaussian elimination would also take substantial part of time resources in experiments.

In theory, the Gaussian elimination step is applied whenever the size of the set of partial solutions exceeds theoretical guarantees. However, in practice it seems reasonable that sometimes it pays off to apply this computationally expensive step less often; that is, allow the set of partial solutions to grow significantly beyond the theoretical bounds, and once in a while trim it at bulk with a single Gaussian elimination step. This intuition has been supported by the results of Fafianie et al [13] for the case of STEINER TREE.

Consequently, we start our experiments with fine-tuning the frequency of Gaussian elimination in both rank-based approaches we study. Since the width of the tree decomposition can play substantial role in deciding the optimal frequency, we do it separately for sets B and sets C .

In the next experiments, we use the optimum found frequencies for the algorithms based on both rank-based approaches. Note that the frequencies may differ between the low-treewidth regime (sets A and B) and the medium-treewidth regime (set C).

3.4 Comparison of the approaches

Having found the optimal frequency of the Gaussian elimination in the rank-based approaches, we run all four algorithms on every test in sets A , B , and C and compare results. In set C , every run has a timeout of 30 minutes. In set A , the timeout equals 10 minutes.

4 Results

In our experiments, it quickly became apparent that the variant of the naive and rank-based approach that stores all partial solutions (i.e., no self-reducibility) is significantly faster for small treewidth (sets A and B), while the self-reducibility one is faster for larger treewidth

■ **Table 1** Fine-tuning results for test set B . Note that the second and third columns correspond to compression guarantees of the two studied algorithms, respectively.

ℓ	$2^{\ell-1}$	$2^{\ell/2-1}$	τ	Total running time on set B (SS.ms)	
				rank-based 4^t	rank-based $(2 + \sqrt{2})^t$
4	8	2	3	1910.968	1318.385
			4	1827.949	1569.542
6	32	4	5	1901.457	1264.803
			7	1915.583	1286.522
			9	1960.515	1298.849
			11	1890.034	1316.813
			13	1876.483	1339.439
			15	1889.843	1401.620
8	128	8	17	1923.244	1425.338
			9	1896.748	1269.761
			18	1899.633	1290.696
			36	1996.629	1274.545
			72	1925.507	1268.261
			144	1863.837	1283.934

(sets C , D , and E). Thus, in what follows, we used the first one for experiments on small treewidth graph and the latter for larger treewidth graphs.

4.1 Fine-tuning the frequency of Gaussian elimination

4.1.1 Small treewidth

Recall that in sets A and B , the maximum size of the bag in the decomposition is 9. Consequently, in every state (b, E) the size of $b^{-1}(1)$ is at most 8 (as it must be even). The treatment of the states with $|b^{-1}(1)| \in \{0, 2\}$ does not use any of the involved rank-based techniques. Thus, we decided to separately fine-tune the frequency of applying the Gaussian elimination step to buckets with $|b^{-1}(1)|$ of size 4, 6, and 8 each. More formally, for $\ell \in \{4, 6, 8\}$ we fix a threshold τ and, for fixed bucket b with $|b^{-1}(1)| = \ell$ apply the Gaussian elimination step to the states (b, E) only if the number of these states is at least τ . While experimenting with one ℓ , the threshold for another sizes remains fixed. We perform tests on set B and report the total time used to find Hamiltonian cycles in all instances. The results are presented in Table 1.

From the results, it seems that lowering the frequency of Gaussian elimination does not help neither of the approaches, and evidently worsened the case for the improved base algorithm and $\ell = 4, 6$. The only exception seemed to be the case $\ell = 6$ and $\tau = 13$ for the worse base algorithm.

We see a number of good explanations for that. First, we think that case $\ell = 8$ appeared very rarely in the computations, and thus the impact of fine-tuning it has negligible effect in the overall result.

For the remaining cases, note that the matrices passed to the Gaussian elimination have at most 32 columns in the case of the first algorithm, and only 4 columns in the second. Thus, the Gaussian elimination step is very cheap in this regime of ℓ . Consequently, one does not gain much from lowering the frequency, while evidently losing by needing to maintain bigger memoization tables. This explains the worsening of the second algorithm for $\ell = 4, 6$

29:10 Finding Hamiltonian Cycle in Graphs of Bounded Treewidth

■ **Table 2** Fine-tuning results for test set D . The Gaussian elimination step is applied to buckets b with $\ell = |b^{-1}(1)|$ and at least $\alpha \cdot 2^{\ell/2-1}$ states (b, E) .

α	Total running time on set D (SS.ms)		α	Total running time on set D (SS.ms)	
	rank-based 4^t	rank-based $(2 + \sqrt{2})^t$		rank-based 4^t	rank-based $(2 + \sqrt{2})^t$
0.5	7363.078	2021.516	32	1802.851	1778.647
1	2704.165	1801.278	64	1797.416	1807.470
2	1925.618	1768.000	128	1794.877	1801.913
4	1813.478	1779.293	256	1801.104	1822.113
8	1792.872	1788.217	512	1795.312	1818.508
16	1806.994	1783.919	1024	1800.863	1800.698

and increasing τ .

However, one would expect that the first algorithm would also worsen with the increase of τ , but this is not supported by data. To explain this behavior, note that the values of τ used here are lower than the theoretical guarantees of the algorithm. Consequently, the pruning of the memoization tables in the first algorithm seem to give very little in these cases.

In other words, the pruning capabilities of the vectors v_E used by the first algorithm are much weaker for low values of ℓ than the capabilities of the second algorithm. This is most striking in the case $\ell = 4$: there are 3 pairings of a 4-element set; the first algorithm keeps all of them if present, while the second one notices that one is redundant and deletes it.

To sum up, the data indicates that decreasing the frequency of the Gaussian elimination step does not help for small values of ℓ , while the first algorithm with the worse pruning capabilities does not offer much pruning in this regime of values of ℓ .

4.1.2 Larger treewidth

For fine-tuning in graphs of larger treewidth, we use set D . Here, we propose slightly different threshold behavior: we fix a parameter α and, for fixed bucket b with $\ell = |b^{-1}(1)|$, we apply the Gaussian elimination step if the number of states (b, E) exceed $\alpha \cdot 2^{\ell/2-1}$ (i.e., α is a multiplicative parameter relative to the pruning size guarantee of the second algorithm). The results are gathered in Table 2.

The results indicate that a mild increase of the threshold (i.e., $\alpha = 2$) increases the speed of the second algorithm, while further increase of the threshold slowly worsens the bounds. For the first algorithm, the sweet spot seems to be slightly later, and further increase of the threshold does not necessarily worsen the algorithm.

The gain from mild increase in the case of both algorithms can be explained by the fact that for larger values of ℓ , the Gaussian elimination step starts to be costly. In the case of the first algorithm, we think that its pruning capabilities are limited for the Hamiltonian cycle problem, and thus further increase of the threshold does not change much.

To sum up, both algorithms definitely slow down if the Gaussian elimination step is done too frequently. The data showed optimum values $\alpha = 8$ for the first algorithm and $\alpha = 2$ for the second.

4.2 Comparison

As discussed in Section 2.4, we have implemented two variants of the Cut&Count algorithm: the one that uses the fast convolution at **join nodes** (Lemma 2.3) and the one that does it more naively in time bounded by $9^t t^{\mathcal{O}(1)}$.

■ **Table 3** Total running times for test set A (timeout 10 minutes per instance). The Cut&Count program did not finish within allotted time on 124 instances, the remainder solved all test cases. In the first row, we show total running time on all 499 tests solved by all programs.

	naive	rank-based 4^t	rank-based $(2 + \sqrt{2})^t$	Cut&Count
499 tests finished by all	5993.633	7383.249	5919.392	46650.101
all 623 tests	11532.153	13675.58	10278.827	-

■ **Table 4** Running times for test sets C and E . Hyphen means timeout (30 minutes). For the set C , the “tw” column indicates the width of the used tree decomposition (found by the algorithm of Strasser [21]). Tests where all algorithms were timeouted are not presented here.

test	$ V(G) $	$ E(G) $	tw	naive	rank-based 4^t	rank-based $(2 + \sqrt{2})^t$	Cut&Count
0074	462	756	28	38.737	109.655	110.040	-
0109	606	933	17	.063	.086	.085	.611
0110	606	925	17	.066	.089	.090	.471
0144	804	1256	21	.190	.253	.231	205.128
0145	804	1252	21	.137	.187	.186	3.549
0172	1002	1575	25	1.156	1.298	.554	-
0173	1002	1579	25	.459	.598	.475	215.115
0199	1200	1902	29	13.513	15.419	3.369	-
0200	1200	1902	26	3.673	6.900	1.544	-
0253	1578	2688	29	93.343	167.458	167.440	-
0268	1644	2767	25	36.449	70.157	69.111	-
0271	1662	2770	29	28.149	33.145	33.208	-
0272	1662	2863	25	554.271	1260.329	1230.722	-
0290	1770	3020	25	57.901	83.781	82.386	-
0298	1806	3071	23	10.035	18.611	18.492	-
E0001	360	566		371.775	-	64.390	-
E0002	600	886		204.197	-	28.882	-
E0003	700	1139		-	-	711.778	-
E0007	360	655		1575.475	-	328.191	-

We found out that the one with the fast convolution behaves very slowly even on small tests. This can be easily explained by the hidden complexity of ring computations inside Lemma 2.3. Consequently, while theoretically sound, we dropped it from further experiments and considered only the Cut&Count algorithm without the fast convolution.

For test set A , we have used a timeout of 10 minutes per instance. A CSV file with full results can be found on the project website [1]. Table 3 presents summary; the Cut&Count algorithm did not finish in time for 124 tests, and thus we compare its running time on the other 499 tests. For sets C and E , full results are in Table 4 (for set E only naive and improved rank-based algorithms were executed).

The first corollary from the results is that the Cut&Count approach does not turn out to be practical, and is heavily outperformed by other approaches. We see some good explanations for that. First, all other approaches are “positive-driven”: they keep only values in their memoization tables that correspond to found partial solutions, and in many cases there can be much fewer such partial solutions than the worst-case theoretical bound. In particular, these approaches can implicitly use some hidden structure of the input graph, such as planarity. The Cut&Count approach, on the other hand, relies on computing coefficients for partial cycle covers, and – even with our positive-driven implementation that keeps only nonzero elements – keeps track of much more partial solutions than the other approaches.

This effect is even stronger if one tries to use fast convolution at **join nodes**: the convolution fills up the entire table of 4^t values being polynomials, even if the input functions were sparse.

Second, the Cut&Count approach solves only a decision version of the problem, yielding large overhead from some self-reducibility application, while all other algorithms return the Hamiltonian cycle in question straight away.

For the other approaches, it is noticeable that the first rank-based approach (with 4^t guarantee on the size of the memoization tables) is clearly outperformed by the naive approach. That is, the cost of the Gaussian elimination step does not pay back in savings of the size of memoization tables. This can be explained as already discussed in the previous section: the vectors used in this algorithm are too weak to effectively prune the memoization tables, which is particularly visible on buckets b with small $\ell = |b^{-1}(1)|$.

Results from small treewidth graphs (set A) show also that the improved rank-based approach outperforms the naive one by roughly 10%. For larger treewidth (set C), the situation is more complicated: on some tests the rank-based approach outperforms the naive one by significant factor (0172, 0199, 0200), while sometimes it is opposite (0074, 0272). As expected, the artificially generated random instances gave big advantage to the rank-based approach.

A natural question is why we see only 10% increase despite significant asymptotic gain in the analysis ($2^{\mathcal{O}(t \log t)}$ vs $(2 + \sqrt{2})^t$). Apart from the obvious answers to this questions (the values of t we are studying are low for asymptotic analysis), we would like to point out another, problem-specific reason. The difference between the naive approach and the rank-based one is only within handling states for one fixed bucket b , and there are up to 3^t different buckets. Iterating over all non-empty buckets is a common part of both approaches, and can be responsible for most of their running time.

To sum up, the only approach competitive with the naive approach is the improved rank-based approach with the $(2 + \sqrt{2})^t$ guarantee on the size of memoization tables. However, its gain is limited, and there are multiple cases where the use of Gaussian elimination steps is not helpful at all.

5 Conclusions

We have experimentally evaluated multiple known approaches to solve HAMILTONIAN CYCLE in graphs of bounded treewidth. The results show that the Cut&Count approach is impractical, while the improved rank-based approach of [9] consistently outperforms the more generic one of [6]. Furthermore, the latter seem to help little and is outperformed by the naive solution.

The comparison between the naive solution and the improved rank-based one of [9] is more intricate. On graphs of small treewidth, the second one outperforms the first one by 10% margin. For larger treewidth, the results are rather indecisive.

The results indicate potential in the improved rank-based algorithm of [9] and point to the need of further theoretical study of this approach. In [9], the authors show how to perform pruning without the need of Gaussian elimination at **introduce/forget nodes**. The question of matching the $(2 + \sqrt{2})^t t^{\mathcal{O}(1)}$ running time bound for **join nodes** remains open, and a positive answer to this question may lead to significantly faster implementation. Also, we did not try to mix the Gaussian elimination steps at **join nodes** with the other steps at **introduce/forget nodes**.

Finally, we found it quite remarkable that 638 out of 1001 instances of Flinders Hamiltonian Cycle Challenge [15] (i.e., our sets A and D) could be solved with the naive bounded treewidth routine on a personal computer, while 623 out of them (our set A) have one-digit treewidth.

References

- 1 Recent trends in kernelization: theory and experimental evaluation — project website. 2018. <http://kernelization-experiments.mimuw.edu.pl>.
- 2 Richard Bellman. Combinatorial processes and dynamic programming. Technical report, RAND CORP SANTA MONICA CA, 1958.
- 3 Andreas Björklund. Determinant sums for undirected hamiltonicity. In *Foundations of Computer Science (FOCS), 2010 51st Annual IEEE Symposium on*, pages 173–182. IEEE, 2010.
- 4 Andreas Björklund, Thore Husfeldt, Petteri Kaski, and Mikko Koivisto. Trimmed moebius inversion and graphs of bounded degree. *Theory of Computing Systems*, 47(3):637–654, 2010.
- 5 Andreas Björklund, Petteri Kaski, Lukasz Kowalik, and Juho Lauri. Engineering motif search for large graphs. In *2015 Proceedings of the Seventeenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 104–118. SIAM, 2014.
- 6 Hans L. Bodlaender, Marek Cygan, Stefan Kratsch, and Jesper Nederlof. Deterministic single exponential time algorithms for connectivity problems parameterized by treewidth. *Inf. Comput.*, 243:86–111, 2015. doi:10.1016/j.ic.2014.12.008.
- 7 Hajo Broersma, Fedor V Fomin, Pim van’t Hof, and Daniël Paulusma. Fast exact algorithms for hamiltonicity in claw-free graphs. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 44–53. Springer, 2009.
- 8 Marek Cygan, Fedor V Fomin, Lukasz Kowalik, Daniel Lokshantov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015.
- 9 Marek Cygan, Stefan Kratsch, and Jesper Nederlof. Fast hamiltonicity checking via bases of perfect matchings. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 301–310. ACM, 2013.
- 10 Marek Cygan, Jesper Nederlof, Marcin Pilipczuk, Michał Pilipczuk, Joham MM van Rooij, and Jakub Onufry Wojtaszczyk. Solving connectivity problems parameterized by treewidth in single exponential time. In *Foundations of Computer Science (FOCS), 2011 IEEE 52nd Annual Symposium on*, pages 150–159. IEEE, 2011.
- 11 Holger Dell, Thore Husfeldt, Bart MP Jansen, Petteri Kaski, Christian Komusiewicz, and Frances A Rosamond. The first parameterized algorithms and computational experiments challenge. In *LIPICs-Leibniz International Proceedings in Informatics*, volume 63. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.
- 12 Holger Dell, Christian Komusiewicz, Nimrod Talmon, and Mathias Weller. The pace 2017 parameterized algorithms and computational experiments challenge: The second iteration.
- 13 Stefan Fafianie, Hans L Bodlaender, and Jesper Nederlof. Speeding up dynamic programming with representative sets: an experimental evaluation of algorithms for steiner tree on tree decompositions. *Algorithmica*, 71(3):636–660, 2015.
- 14 Serge Gaspers, Joachim Gudmundsson, Mitchell Jones, Julián Mestre, and Stefan Rümmele. Turbocharging treewidth heuristics. In Jiong Guo and Danny Hermelin, editors, *11th International Symposium on Parameterized and Exact Computation, IPEC 2016, August 24-26, 2016, Aarhus, Denmark*, volume 63 of *LIPICs*, pages 13:1–13:13. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPICs.IPEC.2016.13.
- 15 M. Haythorpe. FHCP challenge set, 2015. <http://fhcp.edu.au/fhpcs>.
- 16 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.
- 17 Daniel Lokshantov, Dániel Marx, and Saket Saurabh. Known algorithms on graphs on bounded treewidth are probably optimal. In Dana Randall, editor, *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2011, San*

- Francisco, California, USA, January 23-25, 2011, pages 777–789. SIAM, 2011. doi:10.1137/1.9781611973082.61.
- 18 Daniel Lokshantov, Dániel Marx, and Saket Saurabh. Slightly superexponential parameterized problems. In Dana Randall, editor, *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2011, San Francisco, California, USA, January 23-25, 2011*, pages 760–776. SIAM, 2011. doi:10.1137/1.9781611973082.60.
 - 19 Ketan Mulmuley, Umesh V. Vazirani, and Vijay V. Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 7(1):105–113, 1987. doi:10.1007/BF02579206.
 - 20 Neil Robertson and Paul D Seymour. Graph minors. III. Planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49–64, 1984.
 - 21 Ben Strasser. Computing tree decompositions with flowcutter: PACE 2017 submission. *CoRR*, abs/1709.08949, 2017. arXiv:1709.08949.
 - 22 Johan M. M. van Rooij, Hans L. Bodlaender, and Peter Rossmanith. Dynamic programming on tree decompositions using generalised fast subset convolution. In Amos Fiat and Peter Sanders, editors, *Algorithms - ESA 2009, 17th Annual European Symposium, Copenhagen, Denmark, September 7-9, 2009. Proceedings*, volume 5757 of *Lecture Notes in Computer Science*, pages 566–577. Springer, 2009. doi:10.1007/978-3-642-04128-0_51.
 - 23 Michał Ziobro and Marcin Pilipczuk. Finding Hamiltonian Cycle in graphs of bounded treewidth: Experimental evaluation. code repository, 2018. https://github.com/stalowyjez/hc_tw_experiments.


Isomorphism Test for Digraphs with Weighted Edges

Adolfo Piperno

Dipartimento di Informatica, La Sapienza Università di Roma

Via Salaria 113, I-00198 Rome (Italy)

piperno@di.uniroma1.it

 <https://orcid.org/0000-0001-5001-6308>

Abstract

Colour refinement is at the heart of all the most efficient graph isomorphism software packages. In this paper we present a method for extending the applicability of refinement algorithms to directed graphs with weighted edges. We use **Traces** as a reference software, but the proposed solution is easily transferrable to any other refinement-based graph isomorphism tool in the literature. We substantiate the claim that the performances of the original algorithm remain substantially unchanged by showing experiments for some classes of benchmark graphs.

2012 ACM Subject Classification Mathematics of computing → Graph theory, Computing methodologies → Combinatorial algorithms

Keywords and phrases Practical Graph Isomorphism, Weighted Directed Graphs, Partition Refinement

Digital Object Identifier 10.4230/LIPIcs.SEA.2018.30

Acknowledgements I am grateful to Brendan McKay for many helpful suggestions and discussions about the topics of this paper.

1 Introduction

An *isomorphism* between two graphs is a bijection between their vertex sets that preserves adjacency. An *automorphism* is an isomorphism from a graph to itself. The set of all automorphisms of a graph G form a group under composition called the *automorphism group* $\text{Aut}(G)$ whose *order* is $|\text{Aut}(G)|$. The graph isomorphism problem (GI) is that of determining whether there is an isomorphism between two given graphs. It is convenient to consider GI for vertex coloured graphs, in which case isomorphisms and automorphisms must preserve colours of vertices.

In this paper we will consider GI for coloured graphs and digraphs with weighted edges, in which case isomorphisms and automorphisms must preserve weights of edges, too. Quite surprisingly, none of the existing GI software packages is currently able to treat such class of graphs directly. Existing software can handle weighted digraphs by using layers (as in the **nauty** manual [15]) or by using unweighted gadgets to simulate weighted directed edges (see Figure 1). However, both methods multiply the size of the graph and so increase the running time and space significantly. We will use **Traces** [16, 19] as reference program, but the method that we are going to describe can be adapted to any other GI software.

The most successful GI packages are based on the *individualization-refinement* technique: they can treat graphs with a huge number of vertices and edges quite efficiently. During the computation, these programs spend most of the time in the operation of *colour refinement*, i.e. in the assignment of a minimal number of colours to vertices of the graph, in a way that vertices with the same colour have neighbours with the same colours. In every GI package,



© Adolfo Piperno;

licensed under Creative Commons License CC-BY

17th International Symposium on Experimental Algorithms (SEA 2018).

Editor: Gianlorenzo D'Angelo; Article No. 30; pp. 30:1–30:13

Leibniz International Proceedings in Informatics



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

the refinement routines have been the object of subsequent optimizations, sometimes over decades: to add new features to them may not be an easy task.

From their part, refinement algorithms spend most of the time in counting neighbours of vertices. At each iteration, a reference colour c is selected and vertices of the graph are classified according to the number of c -coloured neighbours they have.

In the case of graphs with weighted directed edges – and in the context of graph isomorphism – the main issue to be considered is that the notion of adjacency is not as immediate as in the case of simple graphs. In this setting, the classification of a vertex u must take into account not only weights of edges from u to vertices with the current reference colour, but also weights of their opposite edges, since isomorphisms and automorphisms are requested not only to preserve the out-neighbourhood of u but also its in-neighbourhood (see e.g. Theorem 13 of [3]). The main motivation of this paper is to go beyond these additional difficulties by keeping the counting mechanism of the refinement algorithm for simple graphs. Shortly, the solution we propose is the adoption of *internal weights* – to be used during the computation, only – which encode the information from both the weights of an edge and its opposite edge. We will treat the case of unweighted directed graphs by considering two distinguished weights 1 and 0 with the meaning of “arc” and “non-arc”, respectively.

In particular, it is our aim to show (and prove) that the ability to process weighted graphs and digraphs can be added to `Traces` in a very simple and conservative way: (i) by keeping the original data structures and by changing a minimal number of lines of the existing code; (ii) by introducing a negligible overhead – with respect to the whole computation – in preprocessing weights, just in the case of the new families of graphs; (iii) by preserving substantially the same performances in the case of simple graphs. The simplicity of the proposed solution stems from the fact that it exactly captures the additional complexities arising when using graphs with weighted edges.

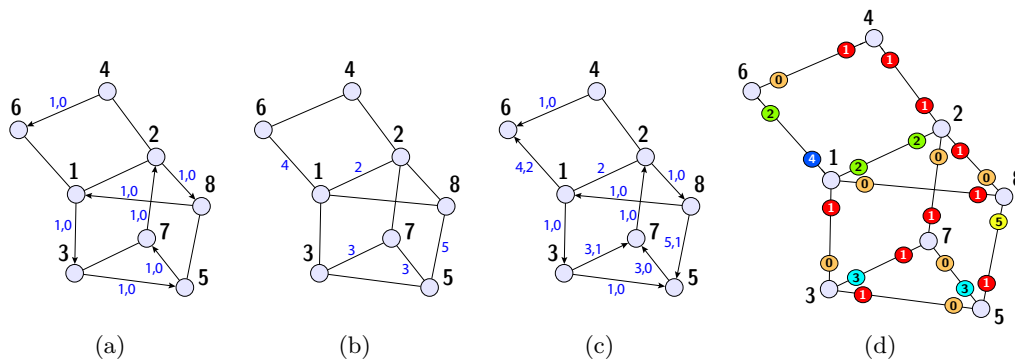
Towards the aim of the paper, in Section 2 we will briefly review the individualization-refinement technique and we will consider the issues in extending the method to the case of weighted digraphs; in Section 3 we will introduce internal weights, and we will prove their properties. The new algorithm will be presented in Section 4, together with a brief analysis of its complexity. Experimental results will be shown in Section 5.

2 Practical aspects of the graph isomorphism problem

The theoretical status of GI, which culminates with Babai’s recent quasi-polynomiality result [2], is outside the scope of this paper; a brief historical description can be found in [16].

From a practical perspective, the most successful approach to GI is the “individualization-refinement” method, which originates in [18, 5, 1] and was distributed in a software package, `nauty`, by McKay [13].

Basically, a *colouring (or partition) refinement* function classifies vertices of a graph G , in a way which is invariant under isomorphism, according to the classification of their neighbours. The sets of vertices with a given colour are called *cells*. Vertices with a specific colour (chosen in an isomorphism invariant way, again) are *individualized* one by one in order to distinguish them from other vertices in the same cell. This mechanism produces a search tree, whose nodes represent refined colourings, while branching is determined by the individualization step. Colourings in which all cells are singletons (called *discrete*) appear as leaves of the search tree. Equivalent discrete colourings induce automorphisms of the graph G . Pruning of the tree is obtained by excluding non-matching colourings and by the use of automorphisms. Comparing colourings also allows us to define a *best* leaf, which is used to canonically label the graph G , namely to produce a representative of exactly the isomorphism class of G .



■ **Figure 1** A directed graph (a), a weighted graph (b), a weighted digraph (c) and its encoding by means of a simple coloured graph (d). In these pictures, the arc (u, v) has label “ a, b ” when the weight of (u, v) is a and the weight of (v, u) is b . The simple edge (u, v) has label “ a ” when both the weight of (u, v) and that of (v, u) are $a \neq 0, 1$. The simple edge (u, v) has no label when both the weight of (u, v) and that of (v, u) are 1. The graph in (d) is obtained from the one in (c) by adding two vertices for each arc and colouring them according to their weight.

Software distributions based on the individualization-refinement technique such as *nauty*[13, 16, 14, 15], *Traces*[16, 14, 15, 19], *Bliss*[9, 10], *conauto*[12, 11] and *saucy* [6, 7] are the most efficient GI tools currently available, though Neuen and Schweitzer [17] have recently tailored classes of graphs which are not tractable by them.

2.1 Graphs and colourings

A *weighted digraph* is a triple $G = (V, E, w)$, where $(v, v) \notin E$ and $(u, v) \in E \Rightarrow (v, u) \in E, \forall u, v \in V$; $w : E \rightarrow \mathcal{W}$ is a function mapping arcs to elements of a finite set \mathcal{W} of possible *weights*. Note that loops can be easily represented in our setting by colouring vertices.

► **Remark.** Throughout the paper, we will assume without loss of generality that $\mathcal{W} \subseteq \mathbb{N}$, i.e. the set of weights is a finite set of natural numbers. In fact, for the purpose of isomorphism testing, it is the difference between weights which is relevant, rather than their actual value. Weighted graphs are in general directed graphs, since for any $u, v \in V$ and for any weight a , $w(u, v) = a \not\Rightarrow w(v, u) = a$. In order to represent an unweighted directed arc from u to v , we will always use the weights $0, 1 \in \mathcal{W}$ and impose $w(u, v) = 1, w(v, u) = 0$. We finally observe that graphs with multiple weighted edges can be represented in the present setting by suitable encodings of multiple arcs into single weighted arcs.

Let $\mathcal{G} = \mathcal{G}_n$ denote the set of graphs with vertex set $V = \{1, 2, \dots, n\}$. A *colouring* of V (or of $G \in \mathcal{G}$) is a surjective function π from V onto $\{1, 2, \dots, k\}$ for some k . The number of colours, i.e. k , is denoted by $|\pi|$. A *cell* of π is the set of vertices with some given colour. A *discrete colouring* is a colouring in which each cell is a singleton, in which case $|\pi| = n$. Note that a discrete colouring is a permutation of V .

If π, π' are colourings, then π' is *finer than or equal to* π (and π is *coarser than or equal to* π'), written $\pi' \preceq \pi$, if $\pi(v) < \pi(w) \Rightarrow \pi'(v) < \pi'(w)$ for all $v, w \in V$. This implies that each cell of π' is a subset of a cell of π , but the converse is not true.

A pair (G, π) , where π is a colouring of G , is called a *coloured graph*.

Let S_n denote the symmetric group acting on V . We indicate the action of elements of S_n by exponentiation. That is, for $v \in V$ and $g \in S_n$, v^g is the image of v under g . The same notation indicates the induced action on complex structures derived from V . In

particular, if $G = (V, E, w) \in \mathcal{G}$, then: (i) $G^g \in \mathcal{G}$ has u^g adjacent to v^g exactly when u and v are adjacent in G ; (ii) if π is a colouring of V , then π^g is the colouring with $\pi^g(v^g) = \pi(v)$ for each $v \in V$; (iii) w^g is such that $w^g(u^g, v^g) = w(u, v)$, for each $(u, v) \in E$; (iv) $(G, \pi)^g = ((V^g, E^g, w^g), \pi^g)$.

2.2 Graph isomorphism

Two coloured graphs $(G = (V, E, w), \pi), (G' = (V, E', w'), \pi')$ are *isomorphic* if there is $g \in S_n$ such that $(G', \pi') = (G, \pi)^g$, in which case we write $(G, \pi) \cong (G', \pi')$. Such a g is called an *isomorphism*. The *automorphism group* $\text{Aut}(G, \pi)$ is the group of isomorphisms of the coloured graph (G, π) to itself; that is,

$$\text{Aut}(G, \pi) = \{g \in S_n : (G, \pi)^g = (G, \pi)\}.$$

Let $\Pi = \Pi_n$ denote the set of colourings. A *canonical form* is a function

$$C : \mathcal{G} \times \Pi \rightarrow \mathcal{G} \times \Pi$$

such that, for all $G \in \mathcal{G}$, $\pi \in \Pi$ and $g \in S_n$,

$$C(G, \pi) \cong (G, \pi) \text{ and } C(G^g, \pi^g) = C(G, \pi). \quad (1)$$

In other words, it assigns to each coloured graph an isomorphic coloured graph that is a unique representative of its isomorphism class. It follows from the definition that $(G, \pi) \cong (G', \pi') \Leftrightarrow C(G, \pi) = C(G', \pi')$.

2.3 Refinement

We first review and discuss refinement for simple graphs.

► **Definition 1** (the simple graph case). Let $G \in \mathcal{G}$ be a simple graph.

1. A colouring of G is called *equitable* if any two vertices of the same colour are adjacent to the same number of vertices of each colour.
2. For every colouring π of G , a coarsest equitable colouring π' finer than π is called a (*colour*) *refinement* of π . It is well known that π' is unique up to the order of its cells.

An algorithm for computing π' appears in [13]. We summarize it in Algorithm 1. All refinement algorithms present in the literature are variants of this one. The paper of Berkholz, Bonsma and Grohe [3] has recently presented a deep analysis of refinement algorithms, establishing their complexity in $O((m+n) \log n)$ time, where n is the number of vertices and m the number of edges of the input graph.

► **Example 2.** A simple graph (left) and its colour refinement are shown in Figure 2. The rightmost colouring is obtained, by refining, after the individualization of vertex 10.

In Algorithm 1, the cell W causes the splitting of the cell X when two vertices in X have a different number of neighbours in W . We will call W the *reference cell*. The correctness of the algorithm is based on the fact that – at every iteration of the **while** loop – the sequence α contains at least cells which may cause any possible splitting of other cells. In particular, when the refinement function is called at the beginning of the computation, the sequence of all cells of the input colouring is assigned to α , while after an individualization step it is sufficient to refine the colouring by assigning to α only the cell which contains the individualized vertex.

Algorithm 1: Refinement algorithm.

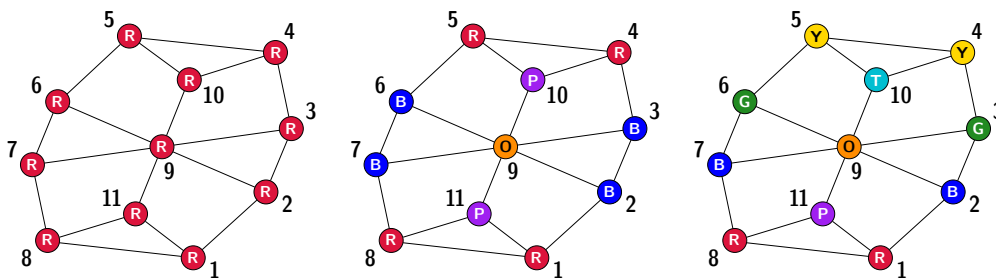
Data: π is the input colouring and α is a sequence of some cells of π .

Result: The final value of π is the output colouring.

```

while  $\alpha$  is not empty and  $\pi$  is not discrete do
  Remove some element  $W$  from  $\alpha$ ;
  Count the number of edges from vertices in  $W$  to each vertex;
  for each cell  $X$  of  $\pi$  do
    Let  $X_1, \dots, X_k$  be the fragments of  $X$  distinguished according
      to the counting of the previous step;
    Replace  $X$  by  $X_1, \dots, X_k$  in  $\pi$ ;
    if  $X \in \alpha$  then
      Replace  $X$  by  $X_1, \dots, X_k$  in  $\alpha$ ;
    else
      Add all but one of the largest of  $X_1, \dots, X_k$  to  $\alpha$ ;
    end
  end
end
end

```



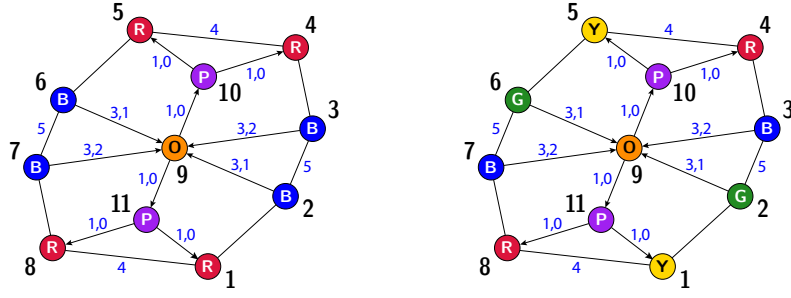
■ **Figure 2** Refinement (simple graphs); individualization of vertex 10 and refinement (right).

All the programs based on the individualization-refinement method spend most of their time in refining partitions; for its part, the refinement algorithm spends most of its time in counting neighbours of the reference cell. Therefore, the overall efficiency of the algorithm depends to a large degree on the efficiency of the colour refinement procedure. **Traces**, for instance, distinguishes several cases in the counting loop, according to different rates of density of the graph, and gives priority to singleton reference cells, in an isomorphism invariant way.

In this scenario, it is our aim to equip **Traces** with the additional resources needed to treat weighted graphs, *without making any change to the neighbour counting algorithms*. The main issue to be considered is that a cell must be split not only in conformity with the number of its outgoing edges falling into the reference cell, but also according to the weight of such edges and to the weight of their opposite edges (see e.g. the cell $\{2, 3, 6, 7\}$ in Figure 3).

► **Definition 3** (the weighted digraph case). Let $G = (V, E, w) \in \mathcal{G}$ be a weighted digraph with weights from $\mathcal{W} \subseteq \mathbb{N}$.

1. Let $u, v \in V$ be two distinct vertices of G . We say that u is (a, b) -adjacent to v if $w(u, v) = a$ and $w(v, u) = b$ and a, b are not both equal to 0. Therefore, if u is (a, b) -adjacent to v , then v is (b, a) -adjacent to u .



■ **Figure 3** Refinement (weighted digraphs): the splitting of the cell $\{2, 3, 6, 7\}$ into $\{2, 6\}\{3, 7\}$ is caused by the reference cell $\{9\}$, due to the weights of the edges *from* vertex 9 to 2, 3, 6, 7.

2. A colouring of G is called *equitable* if any two vertices of the same colour are (a, b) -adjacent to the same number of vertices of each colour, for any $(a, b) \in \mathcal{W} \times \mathcal{W}$.
3. For every colouring π of G a coarsest equitable colouring π' finer than π is called a *refinement* of π .

3 Internal weights

Let $G = (V, E, w) \in \mathcal{G}$ be a weighted digraph with weights from $\mathcal{W} \subseteq \mathbb{N}$. We assign *internal weights* to edges of G , with the aim of making the refinement phase similar as much as possible to that for simple graphs. We will prove that the order of the automorphism group of G with internal weights remains unchanged, and that a canonical form of G can be obtained at the end of the computation simply by restoring the original weights.

► **Definition 4.** We define the function \bar{w} which assigns *internal weights* to edges of G in two steps:

1. We define the function

$$\begin{aligned} \phi_w: E &\rightarrow \mathcal{W} \times \mathcal{W} \\ (u, v) &\mapsto (w(u, v), w(v, u)). \end{aligned} \tag{2}$$

and we denote by $\Phi_w = \{(w(u, v), w(v, u)) \mid (u, v) \in E\}$ the image of ϕ_w and by Φ_w^{lex} the lexicographically ordered sequence of elements of Φ_w .

2. Let $\bar{\mathcal{W}} = \{0, 1, \dots, |\Phi_w| - 1\}$. We define the function

$$\begin{aligned} \bar{w}: E &\rightarrow \bar{\mathcal{W}} \\ (u, v) &\mapsto \text{the index of } \phi_w(u, v) \text{ in } \Phi_w^{\text{lex}} \text{ (starting from 0)}. \end{aligned} \tag{3}$$

3. For any $G = (V, E, w) \in \mathcal{G}$, we denote $\bar{G} = (V, E, \bar{w})$.

► **Example 5.** In Figure 4, internal weights are assigned to the leftmost graph. For any edge (u, v) in the second column of the table, the corresponding entry $\mathbf{a}, \mathbf{b} \rightarrow \mathbf{i}$ in the first column shows that $w(u, v) = \mathbf{a}$, $w(v, u) = \mathbf{b}$ and $\bar{w}(u, v) = \mathbf{i}$. Therefore, the internal weight \mathbf{i} carries the information of both the weights of (u, v) and (v, u) .

For any pair of edges (u_1, v_1) and (u_2, v_2)

$$\text{(By (2))} \quad \phi_w(u_1, v_1) = \phi_w(u_2, v_2) \Leftrightarrow \phi_w(v_1, u_1) = \phi_w(v_2, u_2) \tag{4}$$

$$\text{(By (3))} \quad \bar{w}(u_1, v_1) = \bar{w}(u_2, v_2) \Leftrightarrow \phi_w(u_1, v_1) = \phi_w(u_2, v_2), \tag{5}$$

therefore the internal weight of an edge encodes both $w(u, v)$ and $w(v, u)$.

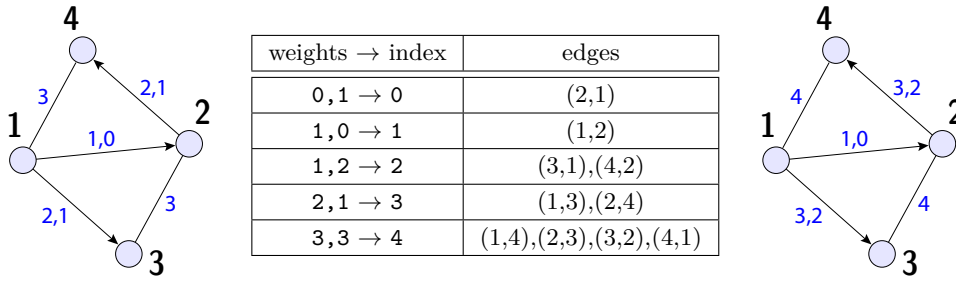


Figure 4 Assignment of internal weights.

► **Lemma 6.** Let $G = (V, E, w) \in \mathcal{G}$. Then

1. $\phi_w(u_1, v_1) = \phi_w(u_2, v_2) \Leftrightarrow \phi_{\bar{w}}(u_1, v_1) = \phi_{\bar{w}}(u_2, v_2)$.
2. $\phi_{\bar{w}}(u_1, v_1) = \phi_{\bar{w}}(u_2, v_2) \Leftrightarrow \phi_{\bar{w}}(v_1, u_1) = \phi_{\bar{w}}(v_2, u_2)$.

Proof. These follow from (4) and (5). ◀

► **Remark.** (Idempotency) For any $G = (V, E, w) \in \mathcal{G}$ we have $\overline{\overline{G}} = \overline{G}$.

In fact, by statement 2 of Lemma 6, for every $a \in \overline{\mathcal{W}}$ there is one and only one $b \in \overline{\mathcal{W}}$ such that $\phi_{\bar{w}}(u, v) = (a, b)$, for some $(u, v) \in E$. It follows that $\overline{\mathcal{W}} = \overline{\overline{\mathcal{W}}}$ and that the index of (a, b) in $\Phi_{\bar{w}}^{\text{lex}}$ is exactly a . Thus, $\bar{w} = \overline{\bar{w}}$. Note that $(a, b) \in \Phi_{\bar{w}} \Leftrightarrow (b, a) \in \Phi_{\bar{w}}$, therefore the set of pairs $\Phi_{\bar{w}}$ is a bijection on $\overline{\mathcal{W}}$.

► **Theorem 7.** Let $G, G_1, G_2 \in \mathcal{G}$. Then:

1. $\text{Aut}(G) = \text{Aut}(\overline{G})$.
2. $G_1 \cong G_2 \Rightarrow \overline{G_1} \cong \overline{G_2}$.

Proof. Both 1 and 2 follow from statement 1 of Lemma 6.

1. (\Rightarrow) Let $g \in \text{Aut}(G)$ be such that for some vertices u_1, v_1, u_2, v_2 we have $(u_1, v_1)^g = (u_2, v_2)$. Then $\phi_w(u_1, v_1) = \phi_w(u_2, v_2)$, since g preserves weights. By using 1 of Lemma 6 we obtain that $g \in \text{Aut}(\overline{G})$. The converse implication is proven similarly.
2. It is well known that we can decide the isomorphism of two graphs by comparing the order of their automorphism groups with the order of the automorphism group of their union graph. The theorem follows considering the union graph of G_1 and G_2 , applying the previous result. ◀

► **Remark.** The converse of statement 2 of Theorem 7 does not hold. A simple example can be derived as a consequence of the idempotency property. In fact, $\overline{\overline{G}} = \overline{G} \not\cong G \cong \overline{G}$. Consider as a further counterexample the graph G in Figure 4 (left), and replace weight 2 with 3 in all its occurrences, thus obtaining a graph G' not isomorphic to G . However, $\overline{G} = \overline{G'}$.

4 Refinement and isomorphism test

The use of internal weights allows refinements of weighted digraphs to be computed with an algorithm only slightly different from Algorithm 1, as shown in Algorithm 2. The counting loop is fractionated according to the internal weights of outgoing edges of elements of the reference cell W .

Algorithm 2: Refinement algorithm for weighted digraphs.

Data: π is the input colouring and α is a sequence of some cells of π .

Result: The final value of π is the output colouring.

```

while  $\alpha$  is not empty and  $\pi$  is not discrete do
  Remove some element  $W$  from  $\alpha$ ;
  for each internal weight  $z$  (in ascending order of out-arcs of vertices in  $W$  do
    Count the number of edges with weight  $z$  from vertices in  $W$  to each vertex;
    for each cell  $X$  of  $\pi$  do
      Let  $X_1, \dots, X_k$  be the fragments of  $X$  distinguished according
      to the counting of the previous step;
      Replace  $X$  by  $X_1, \dots, X_k$  in  $\pi$ ;
      if  $X \in \alpha$  then
        | Replace  $X$  by  $X_1, \dots, X_k$  in  $\alpha$ ;
      else
        | Add all but one of the largest of  $X_1, \dots, X_k$  to  $\alpha$ ;
      end
    end
  end
end
end

```

► **Theorem 8** (Correctness).

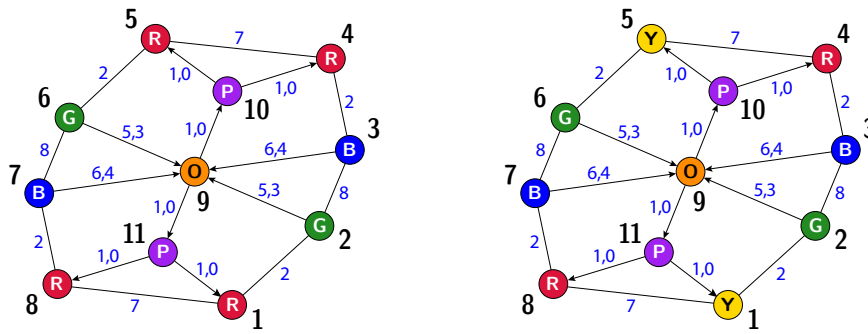
1. Given an internally weighted digraph G and a colouring π of G , the output colouring of Algorithm 2 is a refinement of π .
2. In the case of simple graphs, Algorithm 2 coincides with Algorithm 1.

Proof.

1. The proof follows the pattern of any similar proof in the literature, see e.g. [3]. In a nutshell, (i) the resulting colouring is as coarse as possible since any cell splitting executed by the algorithm is necessary; (ii) it is also sufficiently fine. In fact assume, towards a contradiction, that the final colouring has two cells W_1 and W_2 such that two vertices $u, v \in W_1$ have a different number of (a, b) -neighbours in W_2 , for some internal weights a, b . This is impossible if W_2 is present in the sequence α at the beginning of the computation. Therefore W_2 must have been derived by the splitting of some other cell W . Assume W_2 is not one of the largest cells coming from splitting W . In this case, W_2 is added to α and subsequently removed from it, thus causing u and v to be distributed into two different subcells of W_1 . Otherwise, if W_2 is not added to α after splitting W , then the remaining subcells of W – which are all added to α – cause the same splitting of W_2 (this is a classical result by Hopcroft [8]).
2. In the case of simple graphs, we can assume that only one weight is present. Therefore the highlighted loop in Algorithm 2 consists of only one iteration. ◀

4.1 Invariance by isomorphism and preprocessing

Let $G = (V, E, w) \in \mathcal{G}$ and let π be the initial colouring of G . Weights are chosen in the added loop of Algorithm 2 in ascending order, since this choice is invariant under isomorphism. In order to make the new algorithm easily usable in **Traces**, for each vertex v of G we consider the ordered sequence σ_v of internal weights of its outgoing edges and we store the neighbours



■ **Figure 5** σ_V -sequence colouring and refinement.

Algorithm 3: GI algorithm.

Data: A coloured graph $(G = (V, E, w), \pi)$.

Result: The order of $\text{Aut}(G, \pi)$ and the canonical labelling $C(G, \pi)$ of (G, π) .

- 1 Compute internal weights of G ;
 - 2 Make a copy of (V, E) ;
 - 3 Preprocess the new graph $G' = (V, E, \bar{w})$ by sorting the neighbours of each $v \in V$ according to the sequence σ_v and by considering the σ_V -refinement of π ;
 - 4 Split cells of π according to the order of vertices induced by the order of σ_V ;
 - 5 Run `WTraces` (namely, `Traces` with Algorithm 2 in place of Algorithm 1);
 - 6 Restore the original weights in the canonical labelling $C(G', \pi)$: if p is the permutation such that $C(G', \pi) = (G'^p, \pi^p)$, take $C(G, \pi) = (G^p, \pi^p)$.
-

of v according to this ordering. In addition, we denote $\sigma_V = \{\sigma_v \mid v \in V\}$ and we refine the colouring π by splitting each cell according to the lexicographic order of elements of σ_V . We observe that in a simple graph the counterpart of this splitting operation is the degree colouring, since in that case sequences in σ_V only differ in their length. At the end of this kind of preprocessing phase, if two vertices u and v appear in the same cell, then $\sigma_u = \sigma_v$ and internal weights of neighbours of u and v will immediately emerge in ascending order in the weight loop of Algorithm 2.

► **Example 9.** In Figure 5, the colouring of the leftmost graph is determined conforming to the ordering of σ_V . Two vertices with the same colour, e.g. 4 and 5, are such that $\sigma_4 = \sigma_5 = (0, 2, 7)$. We observe that the colouring is not equitable. In fact, 5 has 6 as neighbour, but 4, which appears in the same cell of 5, has no neighbour in the cell of 6. The cell $\{1, 4, 5, 8\}$ is split into $\{1, 5\}\{4, 8\}$ during the execution of Algorithm 2 as soon as the cell $\{2, 6\}$ is removed from α . More precisely, the splitting occurs when considering outgoing edges of elements of the cell $\{2, 6\}$ whose internal weight is 2. The result of the splitting operation is shown in the rightmost graph, whose colouring is equitable.

4.2 The new GI algorithm: analysis

Let $(G = (V, E, w), \pi)$ be a coloured graph with $n = |V|$ and $m = |E|$. Algorithm 3 summarizes the method to compute the order of the automorphism group and the canonical form of a weighted digraph that we have described in the previous sections. We observe that:

1. The computation of internal weights requires $O(n + m)$ time under the (reasonable) assumption that $\forall (u, v) \in E : w(u, v) < m$, $O(n + m \log m)$ time otherwise.

2. To make a copy of (V, E) requires $O(n + m)$ time.
3. The preprocessing phase requires $O(m)$ time for sorting the neighbours of vertices according to internal weights of outgoing edges, and $O(m)$ time to order the sequence σ_V . In fact, a radix sort can be used, which runs in $O(nn')$ time, where n' is the average length of sequences in σ_V . In our setting, $O(nn') = O(m)$ since the length of $\sigma_v \in \sigma_V$ is the out-degree of v .
4. For any colouring, **Traces** always maintains its inverse. Using this information, cells of π can be split in conformity to the ordering of σ_V in $O(n)$ time.

Recalling that the refinement function runs in time $O((m+n) \log n)$ and that $m \leq n(n-1)$, it follows that the additional computational effort of Algorithm 3 with respect to **Traces** is less than (or at least comparable to) one single call of the refinement function.

5 Experimental results

In the following figures, we present some experiments for a variety benchmark graphs. The graphs are taken from <http://pallini.di.uniroma1.it/Graphs.html>.

The times given are for a Macbook Pro with 3.1 GHz Intel i7 processor (16GB of RAM), using the LLVM compiler (version 9.0.0) and running in a single thread. The interested reader will find the binary codes at <http://pallini.di.uniroma1.it/Weights.html>, together with several other families of graphs.

We recall that **Traces** always computes the order and generators of the automorphism group of the input graph. At the user's request, it computes the canonical form of the graph, too.

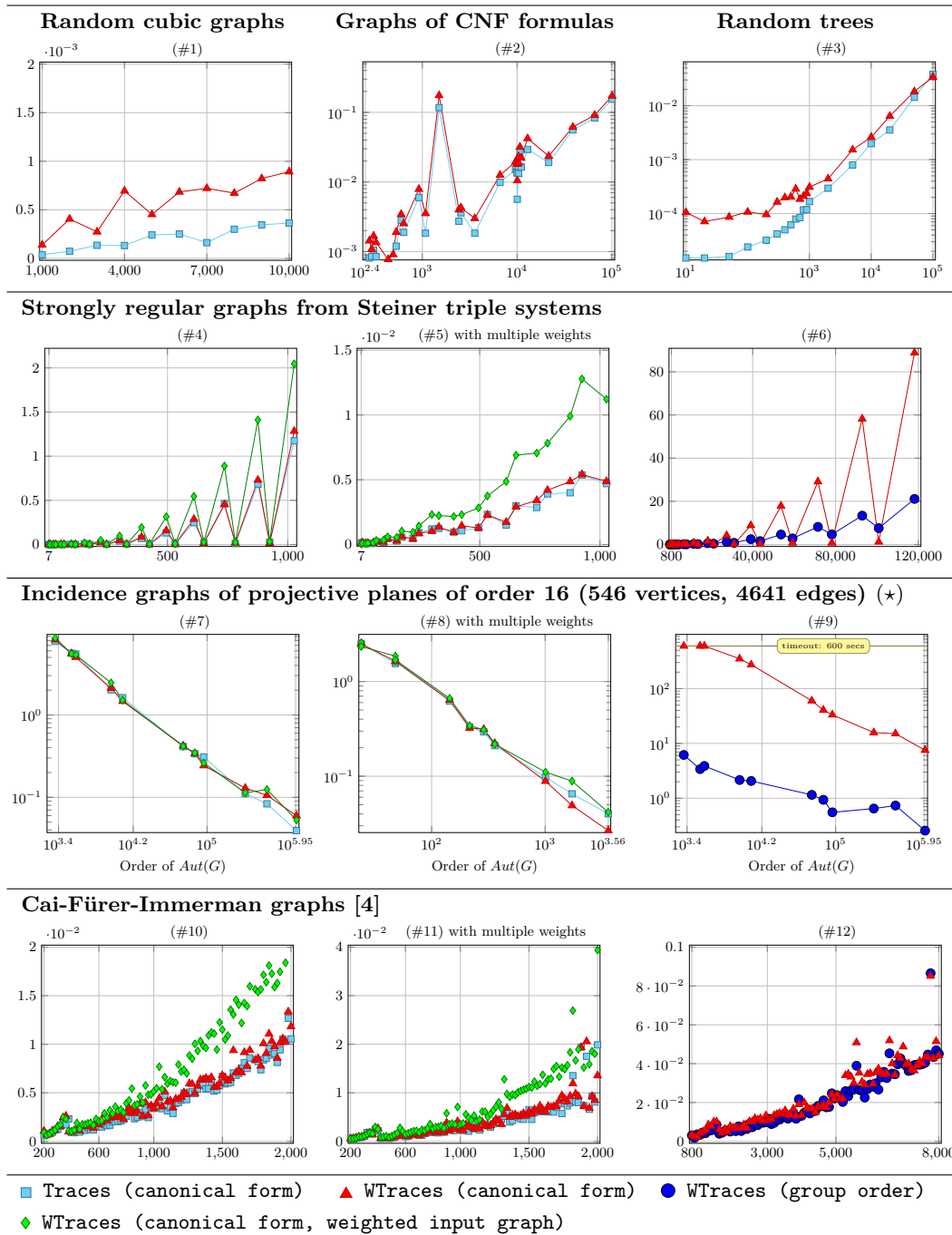
Easy graphs are processed multiple times to give more precise times. We usually start from the unit partition, except when specified in the pictures.

The execution of experimental tests with the assignment of random weights to arcs of graphs from some known relevant families does not give interesting benchmarks since the weight assignment usually breaks all the symmetries of the graph. In order to produce meaningful experiments, for each considered graph G , a weighted version G_w of G is built as follows: we consider the initial refined partition of G , say (W_1, \dots, W_m) , and to each arc (u, v) of G we assign the weight k if $v \in W_k$. This enables to force the program to consider the input as a weighted digraph, therefore executing all the additional steps described in the paper. Note that the graph G_w has the same automorphism group of G .

Four different experiments are reported:

- execution of the currently distributed version of **Traces** (v26r10), with canonical form;
- ▲ execution of **WTraces** (the new program) for a simple graph, with canonical form;
- ◆ execution of **WTraces** adding weights to the input graph, with canonical form;
- execution of **WTraces**, without canonical form.

All experiments show that **Traces** and **WTraces** have similar performances for simple unweighted graphs. In particular, plots #1-#3 in Figure 6 show that the extra computational cost becomes negligible as the number of vertices of the graph increases and (#7) as the graph becomes harder. Plots #4,#7,#10 show the the performance of **WTraces** for weighted digraphs, comparing them to their unweighted version. Due to the presence of the preprocessing overhead, some difference is found for very easy graphs, while the performances are similar for harder cases. The same holds in #5,#8,#11, where the initial colouring of the graph is obtained after individualizing one vertex, thus allowing more weights in the graph G_w .



■ **Figure 6** Performance comparison (horizontal: number of vertices (except (*)); vertical: time in seconds). (*) Incidence graphs of projective planes of order 16 are presented according to the order of their automorphism group.

Finally, plots #6,#9,#12 report the computation time of the simple coloured graphs associated to the weighted digraph reported in plots #4,#7,#10, according to the construction described in Figure 1 (c,d). These plots trivially show that the mentioned construction becomes unfeasible as the density of the graph increases.

6 Concluding remarks

We have presented a method which has enabled us to equip **Traces** with the ability of computing the order of the automorphism group and the canonical labelling of weighted digraphs. The correctness of the method has been proven in the paper. We have executed experimental tests which confirm that the performances of **Traces** remain substantially unchanged. In the case of unweighted digraphs, it would be interesting to compare the behaviour of the presented refinement algorithm with the one in [3]: the notion of (a, b) -adjacency seems to be stronger than the one used by the authors of that paper, since it not only allows for splitting cells according to the number of outgoing edges, but also in conformity with ingoing and undirected edges.

References

- 1 V.L. Arlazarov, I.I. Zuev, A.V. Uskov, and I.A. Faradzhev. An algorithm for the reduction of finite non-oriented graphs to canonical form. *USSR Computational Mathematics and Mathematical Physics*, 14(3):195–201, 1974.
- 2 László Babai. Graph isomorphism in quasipolynomial time [extended abstract]. In *Proceedings of the Forty-eighth Annual ACM Symposium on Theory of Computing*, STOC '16, pages 684–697, New York, NY, USA, 2016. ACM.
- 3 Christoph Berkholz, Paul Bonsma, and Martin Grohe. Tight lower and upper bounds for the complexity of canonical colour refinement. In Hans L. Bodlaender and Giuseppe F. Italiano, editors, *Algorithms – ESA 2013*, pages 145–156. Springer Berlin Heidelberg, 2013.
- 4 Jin-Yi Cai, Martin Fürer, and Neil Immerman. An optimal lower bound on the number of variables for graph identification. *Combinatorica*, 12(4):389–410, 1992.
- 5 D. G. Corneil and C. C. Gotlieb. An efficient algorithm for graph isomorphism. *J. ACM*, 17(1):51–64, 1970.
- 6 Paul T. Darga, Mark H. Liffiton, Karem A. Sakallah, and Igor L. Markov. Exploiting structure in symmetry detection for cnf. In *Proceedings of the 41st Annual Design Automation Conference*, DAC '04, pages 530–534, New York, NY, USA, 2004. ACM.
- 7 Paul T. Darga, Karem A. Sakallah, and Igor L. Markov. Faster symmetry discovery using sparsity of symmetries. In *Proceedings of the 45th Annual Design Automation Conference*, DAC '08, pages 149–154, New York, NY, USA, 2008. ACM.
- 8 John Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In Zvi Kohavi and Azaria Paz, editors, *Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.
- 9 Tommi Junttila and Petteri Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In *Proceedings of the Meeting on Algorithm Engineering & Experiments*, pages 135–149, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.
- 10 Tommi Junttila and Petteri Kaski. Conflict propagation and component recursion for canonical labeling. In Alberto Marchetti-Spaccamela and Michael Segal, editors, *Theory and Practice of Algorithms in (Computer) Systems*, pages 151–162. Springer Berlin Heidelberg, 2011.

- 11 José Luis López-Presa, Antonio Fernández Anta, and Luis Núñez Chiroque. Conauto-2.0: Fast isomorphism testing and automorphism group computation. *CoRR*, abs/1108.1060, 2011. [arXiv:1108.1060](https://arxiv.org/abs/1108.1060).
- 12 José Luis López-Presa and Antonio Fernández Anta. Fast algorithm for graph isomorphism testing. In Jan Vahrenhold, editor, *Experimental Algorithms*, pages 221–232. Springer Berlin Heidelberg, 2009.
- 13 Brendan D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1980.
- 14 Brendan D. McKay and Adolfo Piperno. *nauty* and *Traces*, software distribution web pages. <http://cs.anu.edu.au/~bdm/nauty> and <http://pallini.di.uniroma1.it>.
- 15 Brendan D. McKay and Adolfo Piperno. *nauty* and *Traces* user's guide (version 2.6). Available at <http://cs.anu.edu.au/~bdm/nauty> and <http://pallini.di.uniroma1.it>.
- 16 Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, ii. *Journal of Symbolic Computation*, 60:94–112, 2014.
- 17 Daniel Neuen and Pascal Schweitzer. Benchmark graphs for practical graph isomorphism. In *25th Annual European Symposium on Algorithms, ESA 2017, September 4-6, 2017, Vienna, Austria*, pages 60:1–60:14, 2017.
- 18 R. Parris and Ronald C. Read. A coding procedure for graphs. Technical report, Univ. of West Indies Computer Centre, 1969.
- 19 Adolfo Piperno. Search space contraction in canonical labeling of graphs (preliminary version). *CoRR*, abs/0804.4881, 2008. URL: <http://arxiv.org/abs/0804.4881>.

