Costas S. Iliopoulos
William F. Smyth (Eds.)

LNCS 7056

Combinatorial Algorithms

22nd International Workshop, IWOCA 2011
Victoria, BC, Canada, July 2011
Revised Selected Papers

Lecture Notes in Computer Science 7056

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison
Lancaster University, UK

Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler
University of Surrey, Guildford, UK

Jon M. Kleinberg
Cornell University, Ithaca, NY, USA

Alfred Kobsa
University of California, Irvine, CA, USA

Friedemann Mattern
ETH Zurich, Switzerland

John C. Mitchell
Stanford University, CA, USA

Moni Naor
Weizmann Institute of Science, Rehovot, Israel

Oscar Nierstrasz
University of Bern, Switzerland

C. Pandu Rangan
Indian Institute of Technology, Madras, India

Bernhard Steffen
TU Dortmund University, Germany

Madhu Sudan
Microsoft Research, Cambridge, MA, USA

Demetri Terzopoulos
University of California, Los Angeles, CA, USA

Doug Tygar
University of California, Berkeley, CA, USA

Gerhard Weikum
Max Planck Institute for Informatics, Saarbruecken, Germany

Costas S. Iliopoulos William F. Smyth (Eds.)

Combinatorial Algorithms

22nd International Workshop, IWOCA 2011
Vancouver, BC, Canada, July 20-22, 2011
Revised Selected Papers

Springer

Volume Editors

Costas S. Iliopoulos
King's College London
Department of Informatics
Strand, London WC2R 2LS, UK
E-mail: csi@dcs.kcl.ac.uk
and
Curtin University
Digital Ecosystems and Business Intelligence Institute
Perth WA 6845, Australia

William F. Smyth
McMaster University
Department of Computing and Software
Hamilton, ON L8S 4K1, Canada
E-mail: smyth@mcmaster.ca
and
Curtin University
Digital Ecosystems and Business Intelligence Institute
Perth WA 6845, Australia

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-25010-1 e-ISBN 978-3-642-25011-8
DOI 10.1007/978-3-642-25011-8
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011939494

CR Subject Classification (1998): G.2.1, G.2.2, I.1, I.3.5, F.2, E.1, E.4, H.1

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

© Springer-Verlag Berlin Heidelberg 2011
This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.
The use of general descriptive names, registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

This volume contains the papers presented at IWOCA 11: the 22nd International Workshop on Combinatorial Algorithms

The 22nd IWOCA was held July 20–22, 2011 on the green and spacious campus of the University of Victoria (UVic), itself located on green and spacious Vancouver Island, off the coast of British Columbia, a few scenic kilometers by ferry from the city of Vancouver. The meeting was sponsored and supported financially by the Pacific Institute for the Mathematical Sciences (PIMS); hosted by the UVic Department of Computer Science. The Local Arrangements Committee, cochaired by Wendy Myrvold and Venkatesh Srinivasan, did an outstanding job; the Program Committee was cochaired by Costas Iliopoulos and Bill Smyth; the intricacies of EasyChair were handled by German Tischler.

IWOCA descends from the original Australasian Workshop on Combinatorial Algorithms, first held in 1989, then renamed "International" in 2007 in response to consistent interest and support from researchers outside the Australasian region. The workshop's permanent website can be accessed at iwoca.org, where links to previous meetings, as well as to IWOCA 2011, can be found.

The IWOCA 2011 call for papers was distributed around the world, resulting in 71 submitted papers. The EasyChair system was used to facilitate management of submissions and refereeing, with three referees selected from the 40-member Program Committee assigned to each paper. A total of 30 papers were accepted, subject to revision, for presentation at the workshop.

The workshop also featured a problem session, chaired — in the absence of IWOCA Problems Cochairs Yuqing Lin and Zsuzsanna Liptak — by UVic graduate student Alejandro Erickson. Four invited talks were given by Tetsuo Asano on "Nearest Larger Neighbors Problem and Memory-Constrained Algorithms," Pavol Hell on "Graph Partitions," J. Ian Munro on "Creating a Partial Order and Finishing the Sort, with Graph Entropy" and Cenk Sahinalp on "Algorithmic Methods for Structural Variation Detection Among Multiple High-Throughput Sequenced Genomes."

The 51 registered participants at IWOCA 2011 hold appointments at institutions in 15 different countries on four continents (Asia, Australia, Europe, North America). The nations represented were: Australia (2), Canada (28), China (1), Czech Republic (2), Denmark (1), France (1), Germany (3), India (2), Israel (1), Iran (1), Italy (1), Japan (1), Russia (1), Taiwan (1), USA (5).

Atypical for IWOCA, the contributed talks were split into concurrent streams, A (Combinatorics) and B (Graph Theory). This strategy allowed 30-minute talks and so encouraged a relaxed atmosphere; still, one was often forced to choose between two attractive alternatives. Stream A included such topic areas as combinatorics on words, string algorithms, codes, Venn diagrams, set partitions;

Stream B dealt with several graph theory areas of current interest: Hamiltonian & Eulerian properties, graph drawing, coloring, dominating sets, spanning trees, and others.

We wish to thank the authors for their contributions: the quality of their papers made IWOCA exceptional this year. We would also like to thank the referees for their thorough, constructive and helpful comments and suggestions on the manuscripts.

August 2011

Costas S. Iliopoulos
Bill F. Smyth

Organization

Program Committee

Faisal Abu-Khzam	Lebanese American University, Lebanon
Amihood Amir	Bar-Ilan University and Johns Hopkins University, Israel/USA
Subramanian Arumugam	Kalasalingam University, India
Hideo Bannai	Kyushu University, Japan
Ljiljana Brankovic	University of Newcastle, UK
Gerth Stølting Brodal	Aarhus University, Dem
Charles Colbourn	Arizona State University, USA
Maxime Crochemore	King's College London, UK and Université Paris-Est, France
Diane Donovan	University of Queensland, Australia
Alan Frieze	Carnegie Mellon University, USA
Dalibor Froncek	University of Minnesota Duluth, USA
Roberto Grossi	Università di Pisa, Italy
Sylvie Hamel	University of Montreal, Canada
Jan Holub	Czech Technical University in Prague, Czech Republic
Seok-Hee Hong	University of Sydney, Australia
Costas Iliopoulos	King's College London, UK
Ralf Klasing	LaBRI - CNRS, France
Rao Kosaraju	Johns Hopkins University, USA
Marcin Kubica	Warsaw University, Poland
Anna Lubiw	University of Waterloo, Canada
Mirka Miller	University of Newcastle, UK
Laurent Mouchard	University of Rouen, France
Ian Munro	University of Waterloo, Canada
Wendy Myrvold	University of Victoria, Canada
Kunsoo Park	Seoul National University, Korea
Simon Puglisi	Royal Melbourne Institute of Technology, Australia
Rajeev Raman	University of Leicester, UK
Frank Ruskey	University of Victoria, Canada
Jeffrey Shallit	University of Waterloo, Canada
Michiel Smid	Carleton University, Canada
Bill Smyth	McMaster University, Canada
Iain Stewart	Durham University, UK
Gabor Tardos	Simon Fraser University, Canada
German Tischler	King's College London, UK

Alexander Tiskin	University of Warwick, UK
Eli Upfal	Brown University, USA
Lynette Van Zijl	Stellenbosch University, South Africa
Koichi Wada	Nagoya Institute of Technology, Japan
Sue Whitesides	University of Victoria, Canada
Christos Zaroliagis	CTI University of Patras, Greece

Additional Reviewers

Barbay, Jérémy
Battaglia, Giovanni
Beveridge, Andrew
Blin, Guillaume
Boeckenhauer, Hans-Joachim
Broersma, Hajo
Cadilhac, Michaël
Chauve, Cedric
Cooper, Colin
Cordasco, Gennaro
Erickson, Alejandro
Erlebach, Thomas
Fotakis, Dimitris
Foucaud, Florent
Franceschini, Gianni
Frieze, Alan
Golovach, Petr
Greene, John
Gupta, Anupam
Hahn, Gena
Hoffmann, Michael
Huang, Jing
Izumi, Taisuke
Izumi, Tomoko
Kalvoda, Tomas
Katayama, Yoshiaki
Klouda, Karel
Kontogiannis, Spyros
Korf, Richard
Kyncl, Jan
Langiu, Alessio
Loh, Po-Shen
Macgillivray, Gary
Mamakani, Khalegh
Marshall, Kim
Martin, Barnaby
Mcfarland, Robert
Merlini, Donatella
Mertzios, George B.
Moemke, Tobias
Ono, Hirotaka
Phanalasy, Oudone
Pineda-Villavicencio, Guillermo
Pissis, Solon
Prencipe, Giuseppe
Puglisi, Simon
Radoszewski, Jakub
Radzik, Tomasz
Razgon, Igor
Rylands, Leanne
Sau, Ignasi
Steinhofel, Kathleen
Teska, Jakub
Theodoridis, Evangelos
Tsichlas, Kostas
Vandin, Fabio
Vialette, Stéphane
Wallis, Wal
Yamashita, Yasushi
Yuster, Raphael

Table of Contents

Weighted Improper Colouring*

Julio Araujo[1,2], Jean-Claude Bermond[1], Frédéric Giroire[1], Frédéric Havet[1], Dorian Mazauric[1], and Remigiusz Modrzejewski[1]

[1] Mascotte, joint project I3S(CNRS/Univ. de Nice)/INRIA, France
[2] ParGO Research Group - Universidade Federal do Ceará - UFC, Brazil

Abstract. In this paper, we study a colouring problem motivated by a practical frequency assignment problem and up to our best knowledge new. In wireless networks, a node interferes with the other nodes the level of interference depending on numerous parameters: distance between the nodes, geographical topography, obstacles, etc. We model this with a weighted graph G where the weights on the edges represent the noise (interference) between the two end-nodes. The total interference in a node is then the sum of all the noises of the nodes emitting on the same frequency. A weighted t-improper k-colouring of G is a k-colouring of the nodes of G (assignment of k frequencies) such that the interference at each node does not exceed some threshold t. The Weighted Improper Colouring problem, that we consider here consists in determining the weighted t-improper chromatic number defined as the minimum integer k such that G admits a weighted t-improper k-colouring. We also consider the dual problem, denoted the Threshold Improper Colouring problem, where given a number k of colours (frequencies) we want to determine the minimum real t such that G admits a weighted t-improper k-colouring. We show that both problems are NP-hard and first present general upper bounds; in particular we show a generalisation of Lovász's Theorem for the weighted t-improper chromatic number. We then show how to transform an instance of the Threshold Improper Colouring problem into another equivalent one where the weights are either 1 or M, for a sufficient big value M. Motivated by the original application, we study a special interference model on various grids (square, triangular, hexagonal) where a node produces a noise of intensity 1 for its neighbours and a noise of intensity 1/2 for the nodes that are at distance 2. Consequently, the problem consists of determining the weighted t-improper chromatic number when G is the square of a grid and the weights of the edges are 1, if their end nodes are adjacent in the grid, and 1/2 otherwise. Finally, we model the problem using linear integer programming, propose and test heuristic and exact Branch-and-Bound algorithms on random cell-like graphs, namely the Poisson-Voronoi tessellations.

1 Introduction

Let $G = (V,E)$ be a graph. A *k-colouring* of G is a function $c : V \to \{1,\ldots,k\}$. The colouring c is *proper* if $uv \in E$ implies $c(u) \neq c(v)$. The *chromatic number* of G, denoted by $\chi(G)$, is the minimum integer k such that G admits a proper k-colouring. The goal

* This work was partially supported by région PACA, ANR Blanc AGAPE and ANR International Taiwan GRATEL.

C.S. Iliopoulos and W.F. Smyth (Eds.): IWOCA 2011, LNCS 7056, pp. 1–18, 2011.
© Springer-Verlag Berlin Heidelberg 2011

of the VERTEX COLOURING problem is to determine $\chi(G)$ for a given graph G. It is a well-known NP-hard problem [11].

A k-colouring c is *l-improper* if $|\{v \in N(u) \mid c(v) = c(u)\}| \leq l$ for all $u \in V$. Given a non-negative integer l, the *l-improper chromatic number* of a graph G, denoted by $\chi_l(G)$, is the minimum integer k such that G has an l-improper k-colouring. For given graph G and integer l, the IMPROPER COLOURING problem consists in determining $\chi_l(G)$ [14,6] and is also NP-hard. Indeed, if $l = 0$, observe that $\chi_0(G) = \chi(G)$. Consequently, VERTEX COLOURING is a particular case of IMPROPER COLOURING.

In this work we define and study a new variation of the improper colouring problem for edge-weighted graphs. Given an edge-weighted graph $G = (V, E, w)$, $w : E \to \mathbb{R}_+^*$, a threshold $t \in \mathbb{R}_+$, and a colouring c, we note the *interference* of a vertex w in this colouring as:

$$I_u(G, w, c) = \sum_{\{v \in N(u) | c(v) = c(u)\}} w(u, v).$$

We say that c is a *weighted t-improper k-colouring* of G if c is a k-colouring of G such that $I_u(G, w, c) \leq t$, for all $u \in V$.

Given a threshold $t \in \mathbb{R}_+^*$, the minimum integer k such that the graph G admits a weighted t-improper k-colouring is the *weighted t-improper chromatic number* of G, denoted by $\chi_t(G, w)$. Given an edge-weighted graph $G = (V, E, w)$ and a threshold $t \in \mathbb{R}_+^*$, determining $\chi_t(G, w)$ is the goal of the WEIGHTED IMPROPER COLOURING problem. Note that if $t = 0$ then $\chi_0(G, w) = \chi(G)$, and if $w(e) = 1$ for all $e \in E$, then $\chi_l(G, w) = \chi_l(G)$ for any positive integer l. Therefore, the WEIGHTED IMPROPER COLOURING problem is clearly NP-hard since it generalises VERTEX COLOURING and IMPROPER COLOURING.

On the other hand, we define the *minimum k-threshold* of G, denoted by $\omega_k(G, w)$ as the minimum real t such that G admits a weighted t-improper k-colouring. Then, for a given edge-weighted graph $G = (V, E, w)$ and a positive integer k, the THRESHOLD IMPROPER COLOURING problem consists of determining $\omega_k(G, w)$.

Motivation. Our initial motivation to these problems was the design of satellite antennas for multi-spot MFTDMA satellites [2]. In this technology, satellites transmit signals to areas on the ground called spots. These spots form a grid-like structure which is modelled by an hexagonal cell graph. To each spot is assigned a radio channel or colour. Spots are interfering with other spots having the same channel and a spot can use a colour only if the interference level does not exceed a given threshold t. The level of interference between two spots depends on their distance. The authors of [2] introduced a factor of mitigation γ and the interferences of remote spots are reduced by a factor $1 - \gamma$. When the interference level is too low, the nodes are considered to not interfere anymore. Considering such types of interferences, where nodes at distance at most i interfere, leads to the study of the i-th power of the graph modelling the network and a case of special interest is the power of grid graphs (see Section 3).

Related Work. Our problems are particular cases of the FREQUENCY ASSIGNMENT PROBLEM (FAP). FAP has several variations that were already studied in the literature (see [1] for a survey). In most of these variations, the main constraint to be satisfied is that if two vertices (mobile phones, antennas, spots, etc.) are close, then the difference

between the frequencies that are assigned to them must be greater than some function that usually depends on their distance.

There is a strong relationship between most of these variations and the $L(p_1,\ldots,p_d)$-LABELLING PROBLEM [15]. In this problem, the goal is to find a colouring of the vertices of a given graph G in such a way that the difference between the colours assigned to vertices at distance i must be at least p_i, for every $i = 1,\ldots,d$.

For some other variations, for each non-satisfied interference constraint a penalty must be paid. In particular, the goal of the MINIMUM INTERFERENCE ASSIGNMENT PROBLEM (MI-FAP) is to minimise the total penalties that must be paid, when the number of frequencies to be assigned is given. This problem can also be studied for only *co-channel interferences*, in which the penalties are applied only if the two vertices have the same frequency. However, MI-FAP under these constraints does not correspond to WEIGHTED IMPROPER COLOURING, because we consider the co-channel interference, i.e. penalties, just between each vertex and its neighbourhood.

The two closest related works we found in the literature are [13] and [7]. However, they both apply penalties over co-channel interference, but also to the *adjacent channel interference*, i.e. when the colours of adjacent vertices differ by one unit. Moreover, their results are not similar to ours. In [13], they propose an enumerative algorithm for the problem, while in [7] a Branch-and-Cut method is proposed and applied over some instances.

Results

In this article, we study both parameters $\chi_t(G,w)$ and $\omega_k(G,w)$. We first show that THRESHOLD IMPROPER COLOURING is NP-hard. Then we present general upper bounds; in particular we show a generalisation of Lovász' Theorem for $\chi_t(G,w)$. We then show how to transform an instance into an equivalent one where the weights are either 1 or M, for a sufficient big value M.

Motivated by the original application, we study a special interference model on various grids (square, triangular, hexagonal) where a node produces a noise of intensity 1 for its neighbours and a noise of intensity 1/2 for the nodes that are at distance 2. Consequently, the problem consists of determining $\chi_t(G,w)$ and $\omega_k(G,w)$, when G is the square of a grid and the weights of the edges are 1, if their end nodes are adjacent in the grid, and 1/2 otherwise.

Finally, we propose a heuristic and a Branch-and-Bound algorithm to solve the THRESHOLD IMPROPER COLOURING for general graphs. We compare them to an integer programming formulation on random cell-like graphs, namely Voronoi diagrams of

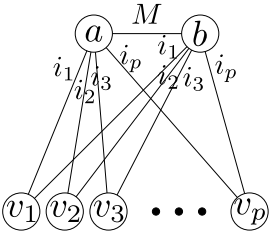


Fig. 1. Construction of $G(I,t)$ from an instance (I,t) of the PARTITION PROBLEM

random points of the plan. These graphs are classically used in the literature to model telecommunication networks [4, 8, 9].

2 General Results

In this section, we present some results for WEIGHTED IMPROPER COLOURING and THRESHOLD IMPROPER COLOURING for general graphs and general interference models.

2.1 NP-Completeness of THRESHOLD IMPROPER COLOURING

In this section, we prove that the decision problem associated to THRESHOLD IMPROPER COLOURING is NP-complete already for $k = 2$.

Theorem 1. *The following problem is NP-complete.*
Instance: *An edge-weighted graph $G = (V, E, w)$, $w : E \to \mathbb{R}_+^*$, a threshold $t \in \mathbb{R}_+$.*
Question: *Does G have a weighted t-improper 2-colouring?*

Proof. Given a 2-colouring c of G, one can test in $O(|E|)$-time if c is weighted t-improper by just checking, for each vertex v, if $I_v(G, w, c) \leq t$. Consequently, this problem is in NP.

Now we reduce the PARTITION problem [11] which is NP-complete, to it. In the PARTITION problem, given a set of p positive integers $I = \{i_1, \ldots, i_p\}$ and a threshold t, we want to decide if there is a partition of the elements of I into two sets A and B such that $\sum_{i_a \in A} i_a \leq t$ and $\sum_{i_b \in B} i_b \leq t$. We consider that $i_j \leq t$, for all $j \in \{1, \ldots, p\}$, and that $t \leq \sum_{j=1}^{p} i_j$, otherwise the answer for this problem is trivially no and yes, respectively.

Given an instance (I, t) of the PARTITION PROBLEM, let $G(I, t)$ be a graph whose vertex set is $V(G(I,t)) = \{v_j \mid j \in \{1, \ldots, p\}\} \cup \{a, b\}$ and whose edge set is $E(G(I,t)) = \{(a,b)\} \cup \{(a, v_j), (b, v_j) \mid j \in \{1, \ldots, p\}\}$ (see Figure 1). Define $M = 1 + \sum_{j=1}^{p} i_j$. Let $w : E(G(I,t)) \to \{i_1, \ldots, i_p, M\}$ be a weight function for the edges of $G(I,t)$ defined in the following way: $w(a,b) = M$ and $w(a, v_j) = w(b, v_j) = i_j$, for every $j \in \{1, \ldots p\}$.

We claim that (I, t) is a yes answer for the PARTITION PROBLEM if, and only if, $G(I,t)$ admits a weighted t-improper 2-colouring.

If (I, t) is a yes instance, let (A, B) be a partitioning such that $\sum_{i_a \in A} i_a \leq t$ and $\sum_{i_b \in B} i_b \leq t$. We claim that the following colouring c is a weighted t-improper 2-colouring of $G(I,t)$:

$$c(v) = \begin{cases} 1 & \text{if } v \in \{a\} \cup \{v_j \mid i_j \in A\}; \\ 2 & \text{otherwise.} \end{cases}$$

To verify this fact, observe that $I_a(G, w, c) = \sum_{i_j \in A} i_j \leq t$, that $I_b(G, w, c) = \sum_{i_j \in B} i_j \leq t$ and that $I_{v_j}(G, w, c) = i_j \leq t$, for each $j \in \{1 \ldots, p\}$.

Conversely, consider that $G(I,t)$ admits a weighted t-improper 2-colouring c. Remark that a and b must receive different colours since the weight of the edge (a,b) is $M > t$. Thus, assume that $c(a) = 1$ and that $c(b) = 2$. Let A be the subset of integers i_j, $j \in \{1, \ldots, p\}$, such that $c(v_j) = 1$ and $B = I \backslash A = \{i_j \mid \ c(v_j) = 2\}$. Observe that the sum of elements in A (resp. B) is equal to $I_a(G, w, c)$ (resp. $I_b(G, w, c)$) and they are both smaller or equal to t, since c is a weighted t-improper 2-colouring.

2.2 Bounds

Upper Bound for WEIGHTED IMPROPER COLOURING. It is a folklore result $\chi(G) \leq \Delta(G)+1$, for any graph G. Lovász [12] extended this result for IMPROPER COLOURING problem. He proved that $\chi_l(G) \leq \lceil \frac{\Delta(G)+1}{l+1} \rceil$. In what follows, we show an extension of these results to WEIGHTED IMPROPER COLOURING.

Given an edge-weighted graph $G=(V,E,w)$, $w : E \to \mathbb{R}_+^*$, and $v \in V$, let $d_w(v) = \sum_{u \in N(v)} w(u,v)$. Denote by $\Delta(G,w) = \max_{v \in V} d_w(v)$. Given a k-colouring $c : V \to \{1,\ldots,k\}$ of G, we denote $d^i_{w,c}(v) = \sum_{\{u \in N(v) | c(u)=i\}} w(u,v)$, for every vertex $v \in V$ and colour $i = 1,\ldots,k$. Note that $d^{c(v)}_{w,c}(v) = I_v(G,w,c)$. Finally, we denote $\gcd(w)$ the greatest common divisor of the weights of w. We use here the generalisation of the gcd to non-integer numbers (e.g. in $\mathbb{Q}$) where a number x is said to divide a number y if the fraction y/x is an integer. The important property of $\gcd(w)$ is that the difference between two interferences is a multiple of $\gcd(w)$; in particular, if for two vertices v and u, $d^i_{w,c}(v) > d^j_{w,c}(u)$, then $d^i_{w,c}(v) \geq d^j_{w,c}(u) + \gcd(w)$.

If t is not a multiple of the $\gcd(w)$, that is, there exists an integer $a \in \mathbb{Z}$ such that $a \gcd(w) < t < (a+1)\gcd(w)$, then $\chi^w_t(G) = \chi^w_{a \gcd(w)}(G)$.

Theorem 2. *Given an edge-weighted graph $G=(V,E,w)$, $w : E \to \mathbb{Q}_+^*$, and a threshold t multiple of* $\gcd(w)$*, then the following inequality holds:*

$$\chi_t(G,w) \leq \left\lceil \frac{\Delta(G,w)+\gcd(w)}{t+\gcd(w)} \right\rceil.$$

Proof. We say that a k-colouring c of G is *well-balanced* if c satisfies the following property:

Property 1. For any vertex $v \in V$, $I_v(G,w,c) \leq d^j_{w,c}(v)$, for every $j = 1,\ldots,k$.

If $k=1$ there is nothing to prove. Then, we prove that for any $k \geq 2$, there exists a well-balanced k-colouring of G. To prove this fact one may just colour G arbitrarily with k colours and then repeat the following procedure: if there exists a vertex v coloured i and a colour j such that $d^i_{w,c}(v) > d^j_{w,c}(v)$, then recolour v with colour j. Observe that this procedure neither increases (we just move a vertex from one colour to another) nor decreases (a vertex without neighbour on its colour is never moved) the number of colours within this process. Let W be the sum of the weights of the edges having the same colour in their endpoints. In this transformation, W has increased by $d^j_{w,c}(v)$ (edges that previously had colours i and j in their endpoints), but decreased by $d^i_{w,c}(v)$ (edges that previously had colour i in both of their endpoints). So, W has decreased by $d^i_{w,c}(v) - d^j_{w,c}(v) \geq \gcd(w)$. As $W \leq |E| \max_{e \in E} w(e)$ is finite, this procedure finishes and produces a well-balanced k-colouring of G.

Observe that in any well-balanced k-colouring c of a graph G, the following holds:

$$d_w(v) = \sum_{u \in N(v)} w(u,v) \geq k d^{c(v)}_{w,c}(v). \tag{1}$$

Let $k^* = \left\lceil \frac{\Delta(G,w)+\gcd(w)}{t+\gcd(w)} \right\rceil \geq 2$ and c^* be a well-balanced k^*-colouring of G. We claim that c^* is a weighted t-improper k^*-colouring of G.

By contradiction, suppose that there is a vertex v in G such that $c^*(v) = i$ and that $d^i_{w,c}(v) > t$. Since c^* is well-balanced, $d^j_{w,c}(v) > t$, for all $j = 1, \ldots, k^*$. By the definition of $\gcd(w)$ and as t is a multiple of $\gcd(w)$, it leads to $d^j_{w,c}(v) \geq t + \gcd(w)$ for all $j = 1, \ldots, k^*$. Combining this inequality with Inequality (1), we obtain:

$$\Delta(G,w) \geq d_w(v) \geq k^*(t + \gcd(w)),$$

giving

$$\Delta(G,w) \geq \Delta(G,w) + \gcd(w),$$

a contradiction. The result follows.

Note that when all weights are equal to one, we obtain the bound for the improper colouring derived in [12].

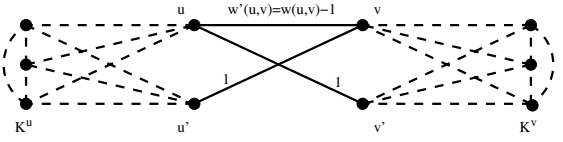


Fig. 2. Construction of G^{i+1} from G^i using edge (u,v) with $k = 4$. Dashed edges represent edges with infinite weights.

Brooks [5] proved that for a connected graph G, $\chi(G) = \Delta(G) + 1$ if, and only if, G is complete or an odd cycle. One could wonder for which edge-weighted graphs the bound we provide is tight. However, Correa et al. [6] already showed that it is NP-complete to determine if the improper chromatic number of a graph G attains the upper bound of Lovász, which is a particular case of WEIGHTED IMPROPER COLOURING and the bound we provided.

Upper Bound for THRESHOLD IMPROPER COLOURING. Let $G = (V,E,w)$, $w : E \to \mathbb{R}^*_+$, be an edge-weighted graph and k be a positive integer. Observe that, for the minimum k-threshold of G,

$$\omega_k(G,w) \leq \Delta(G,w) \leq \sum_{e \in E(G)} w(e).$$

In what follows, we improve this trivial upper bound.

Let $V' = \{u \in V, d(u) \geq k\}$ be the set of vertices with degree at least k. Set $G' = G - V'$.

Lemma 1. $\omega_k(G,w) = \omega_k(G',w)$

Proof. If there is a weighted t-improper k-colouring of G', then it is easy to get a weighted t-improper k-colouring of G choosing, for each vertex $u \in V \setminus V'$, a colour different from the colours of its neighbours. It is always possible because $d(u) \leq k-1$.

Conversely, if there is a weighted t-improper k-colouring of G, then there is a weighted t-improper k-colouring of G' by choosing, for every $v \in V'$, $c_{G'}(v) = c_G(v)$.

For the rest of the section, we only consider edge-weighted graphs with minimum degree at least k. For each $v \in V$, let $E^{k-1}_{min}(v)$ be the set of $d(v)-(k-1)$ least weighted edges incident to v.

Theorem 3. *Let $G=(V,E,w)$, $w : E \to \mathbb{R}^*_+$, be an edge-weighted graph and k be a positive integer. Then,*

$$\omega_k(G,w) \leq \max_{v \in V} w(E^{k-1}_{min}(v)),$$

where $w(E^{k-1}_{min}(v)) = \sum_{e \in E^{k-1}_{min}(v)} w(e)$.

Proof. Let $G^{k-1}_{min} = G[E \setminus \{\bigcup_{v \in V} E^{k-1}_{min}(v)\}]$. Observe that the maximum degree of a vertex in $G^{k-1}_{min} \leq k-1$. Consequently, G^{k-1}_{min} admits a proper k-colouring c of its vertices.

Observe that the maximum interference of a vertex v in G when G is coloured by the colouring c is at most $\max_{v \in V} w(E^{k-1}_{min}(v))$ and the result follows.

2.3 Transformation

In this section, we prove that the THRESHOLD IMPROPER COLOURING problem can be transformed into a problem mixing proper and improper colouring. More precisely, we prove the following:

Theorem 4. *Let $G_0=(V_0,E_0,w_0)$ be an edge-weighted graph such that, for every $e \in E$, $w(e) \in \mathbb{Z}^*_+$, and k be a positive integer. We can construct a graph $G^*=(V^*,E^*,w^*)$ such that $w^*(e) \in \{1,M\}$ for any $e \in E(G^*)$, satisfying $\omega_k(G_0,w_0) = \omega_k(G^*,w^*)$, where $M = 1+\sum_{e \in E(G)} w_0(e)$.*

Proof. Consider the function $f(G,w) = \sum_{\{e \in E(G) | w(e) \neq M\}} (w(e)-1)$.

If $f(G,w)=0$, all edges have weight either 1 or M and G has the desired property. In this case, $G^*=G$. Otherwise, we construct a graph G' and a function w' such that $\omega_k(G',w') = \omega_k(G,w)$, but $f(G',w') = f(G,w)-1$. By repeating this operation $f(G_0,w_0)$ times we get the required graph G^*.

In case $f(G,w)>0$, there exists an edge $e=(u,v) \in E(G)$ such that $2 \leq w(e) < M$. G' is obtained from G by adding two complete graphs on $k-1$ vertices K^u and K^v and two new vertices u' and v'. We join u and u' to all the vertices of K^u and v and v' to all the vertices of K^v. We assign weight M to all these edges. Note that, u and u' (v and v') always have the same colour, namely the remaining colour not used in K^u (resp. K^v).

We also add two edges uv' and $u'v$ both of weight 1. The edges of G keep their weight in G', except the edge $e=uv$ whose weight is decreased by one unit, i.e., $w'(e) = w(e)-1$. Thus, $f(G') = f(G)-1$ as we added only edges of weights 1 and M and we decreased the weight of e by one unit.

Now consider a weighted t-improper k-colouring c of G. We produce a weighted t-improper k-colouring c' to colour G' as follows: we keep the colours of all the vertices in G, we assign to u' (v') the same colour as u (resp., v), and we assign to K^u (K^v) the $k-1$ colours different from the one used in u (resp. v).

Conversely, from any weighted improper k-colouring c' of G', we get a weighted improper k-colouring c of G by just keeping the colours of the vertices that belong to G.

For such colourings c and c' we have that $I_x(G,w,c) = I_x(G',w',c')$, for any vertex x of G different from u and v. For $x \in K^u \cup K^v$, $I_x(G',w',c') = 0$. The neighbours of u with the same colour as u in G' are the same as in G, except possibly v' which has the same colour of u if, and only if, v has the same colour of u. Let $\varepsilon = 1$ if v has the same colour as u, otherwise $\varepsilon = 0$. As the weight of (u,v) decreases by one and we add the edge (u,v') of weight 1 in G', we get $I_u(G',w',c') = I_u(G,w,c) - \varepsilon + w'(u,v')\varepsilon = I_u(G,w,c)$. Similarly, $I_v(G',w',c') = I_v(G,w,c)$. Finally, $I_{u'}(G',w',c') = I_{v'}(G',w',c') = \varepsilon$. But $I_u(G',w',c') \geq (w(u,v)-1)\varepsilon$ and so $I_{u'}(G',w',c') \leq I_u(G',w',c')$ and $I_{v'}(G',w',c') \leq I_v(G',w',c')$. In summary, we have

$$\max_x I_x(G',w',c') = \max_x I_x(G,w,c)$$

and therefore $\omega_k(G,w) = \omega_k(G',w')$.

In the worst case, the number of vertices of G^* is $n + m(w_{max}-1)2k$ and the number of edges of G^* is $m + m(w_{max}-1)[(k+4)(k-1)+2]$ with $n = |V(G)|$, $m = |E(G)|$ and $w_{max} = \max_{e \in E(G)} w(e)$.

In conclusion, this construction allows to transform the THRESHOLD IMPROPER COLOURING problem into a problem mixing proper and improper colouring. Therefore the problem consists in finding the minimum l such that a (non-weighted) l-improper k-colouring of G^* exists with the constraint that some subgraphs of G^* must admit a proper colouring. The equivalence of the two problems is proved here only for integers weights, but it is possible to adapt the transformation to prove it for rational weights.

3 Squares of Particular Graphs

As mentioned in the introduction, WEIGHTED IMPROPER COLOURING is motivated by networks of antennas similar to grids [2]. In these networks, the noise generated by an antenna undergoes an attenuation with the distance it travels.

It is often modelled by a decreasing function of d, typically $1/d^\alpha$ or $1/(2^{d-1})$. Here we consider a simplified model where the noise between two neighbouring antennas is normalised to 1, between antennas at distance two is 1/2 and 0 when the distance is strictly greater than 2.

Studying this model of interference corresponds to study the WEIGHTED IMPROPER COLOURING of the square of the graph G, the graph obtained from G by joining every pair of vertices at distance 2, and to assign weights $w_2(e) = 1$, if $e \in E(G)$, and $w_2(e) = 1/2$, if $e \in E(G^2) - E(G)$. Observe that in this case the interesting threshold values are the non-negative multiples of 1/2.

In Figure 3 are given some examples of colouring for the square grid. In Figure 3(a) each vertex x has neither a neighbour nor a vertex at distance 2 coloured with its own colour, so $I_x(G^2,w_2,c) = 0$. In Figure 3(b) each vertex x has exactly one vertex of the same colour at distance 2, so $I_x(G^2,w_2,c) = 1/2$.

For any $t \in \mathbb{R}_+$, we determine the weighted t-improper chromatic number for the square of infinite paths, square grids, hexagonal grids and triangular grids under the interference model w_2. We also present lower and upper bounds for $\chi_t(T^2, w_2)$, for any tree T and any threshold t.

3.1 Infinite Paths and Trees

In this section, we characterise the weighted t-improper chromatic number of the square of an infinite path, for all positive real t. Moreover, we present lower and upper bounds for $\chi_t(T^2, w_2)$, for a given tree T.

Theorem 5. *Let $P = (V, E)$ be an infinite path. Then,*

$$\chi_t(P^2, w_2) = \begin{cases} 3, & \text{if } 0 \leq t < 1; \\ 2, & \text{if } 1 \leq t < 3; \\ 1, & \text{if } 3 \leq t. \end{cases}$$

Proof. Let $V = \{v_i \mid i \in \mathbb{Z}\}$ and $E = \{(v_{i-1}, v_i) \mid i \in \mathbb{Z}\}$. Each vertex of P has two neighbours and two vertices at distance two. Consequently, the first case $t \geq 3$ is trivial.

There is a 2-colouring c of (P^2, w_2) with maximum interference 1 by just colouring v_i with colour $i \ mod\ 2$. So $\chi_t(P^2, w_2) \leq 2$ if $t \geq 1$. We claim that there is no weighted 0.5-improper 2-colouring of (P^2, w_2). By contradiction, suppose that c is such a colouring. If $c(v_i) = 0$, for some $i \in \mathbb{Z}$, then $c(v_{i-1}) = c(v_{i+1}) = 1$ and $c(v_{i-2}) = c(v_{i+2}) = 0$. This is a contradiction because v_i would have interference 1.

Finally, the colouring $c(v_i) = i \ mod\ 3$, for every $i \in \mathbb{Z}$, is a feasible weighted 0-improper 3-colouring.

Theorem 6. *Let $T = (V, E)$ be a tree. Then, $\lceil \frac{\Delta(T) - \lfloor t \rfloor}{2t+1} \rceil + 1 \leq \chi_t(T^2, w_2) \leq \lceil \frac{\Delta(T)-1}{2t+1} \rceil + 2$.*

Proof. The lower bound is obtained by two simple observations. First, $\chi_t(H, w) \leq \chi_t(G, w)$, for any $H \subseteq G$. Let T be a tree and v be a node of maximum degree in T. Then, observe that the weighted t-improper chromatic number of the subgraph of T^2 induced by v and its neighbourhood is at least $\lceil \frac{\Delta(T) - \lfloor t \rfloor}{2t+1} \rceil + 1$. The colour of v can be assigned to at most $\lfloor t \rfloor$ vertices on its neighbourhood. Any other colour used in the neighbourhood of v cannot appear in more than $2t + 1$ vertices because each pair of vertices in the neighbourhood of v is at distance two.

Let us look now at the upper bound. Choose any node $r \in V$ to be its root. Colour r with colour 1. Then, by a pre-order traversal in the tree, for each visited node v colour all the children of v with the $\lceil \frac{\Delta(T)-1}{2t+1} \rceil$ colours different from the ones assigned to v and to its parent. This is a feasible weighted t-improper k-colouring of T^2, with $k \leq \lceil \frac{\Delta(T)-1}{2t+1} \rceil + 2$, since each vertex interferes with at most $2t$ vertices at distance two which are children of its parent.

3.2 Grids

In this section, we show the optimal values of $\chi_t(G^2, w_2)$, whenever G is an infinite square, or hexagonal or triangular grid, for all the possible values of t. The proofs of the theorems presented in this section can be found in the research report [3].

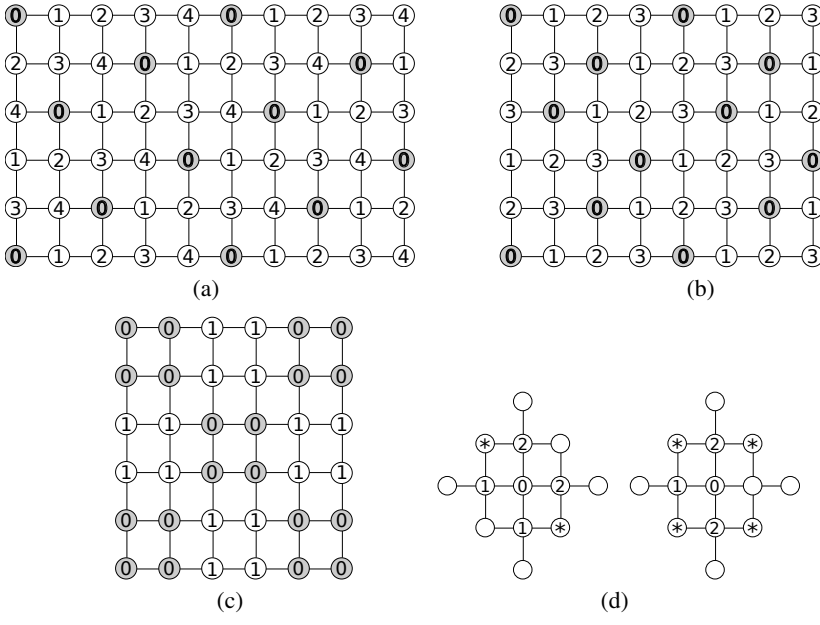


Fig. 3. Optimal colorings of G^2, for square grid G. Weighted 0-improper 5-colouring of G^2 in Figure 3(a), weighted 0.5-improper 4-colouring of G^2 in Figure 3(b) and weighted 3-improper 2-colouring of G^2 in 3(c). Figure 3(d) shows that there is no weighted 0.5-improper 3-colouring of G^2.

Square Grid. The square grid is the graph in which the vertices are all integer linear combinations $ae_1 + be_2$ of the two vectors $e_1 = (1,0)$ and $e_2 = (0,1)$, for any $a,b \in \mathbb{Z}$. Each vertex (a,b) has four neighbours: its *down neighbour* $(a-1,b)$, its *top neighbour* $(a+1,b)$, its *right neighbour* $(a,b+1)$ and its *left neighbour* $(a,b-1)$.

Theorem 7. *If G is an infinite square grid, then*

$$\chi_t(G^2, w_2) = \begin{cases} 5, & \text{if } t = 0; \\ 4, & \text{if } t = 0.5; \\ 3, & \text{if } 1 \leq t < 3; \\ 2, & \text{if } 3 \leq t < 8; \\ 1, & \text{if } 8 \leq t. \end{cases}$$

Proof. If $t = 0$, then the colour of vertex (a,b) must be different from the ones used on its four neighbours. Moreover, all the neighbours have different colours, as each pair of neighbours is at distance two. Consequently, at least 5 colours are needed. Figure 3(a) gives a a weighted 0-improper 5-colouring of G^2.

When $t = 0.5$, we claim that at least four colours are needed to colour G^2. The proof is by contradiction. Suppose that there exists a weighted 0.5-improper 3-colouring of it. Let (a,b) be a vertex coloured 0. No neighbour is coloured 0, otherwise (a,b) has interference 1. If three neighbours have the same colour, then each of them will have

interference 1. So two of its neighbours have to be coloured 1 and the two other ones 2 (see Figure 3(d)). Consider now the four nodes $(a-1,b-1)$, $(a-1,b+1)$, $(a+1,b-1)$ and $(a+1,b+1)$. For all configurations, at least two of these 4 vertices have to be coloured 0. But then (a,b) will have interference at least 1, a contradiction. A weighted 0.5-improper 4-colouring of G^2 is shown in Figure 3(b).

If $t = 1$, there exists a weighted 1-improper 3-colouring of G^2 given by the following construction: for $0 \leq j \leq 2$, let $A_j = \{(0,j) + a(3e_2) + b(e_1+e_2) \mid \forall a,b \in \mathbb{Z}\}$. For $0 \leq j \leq 2$, assign the colour j to all the vertices in A_j.

Now we prove by contradiction that for $t = 2.5$ we still need at least three colours in a weighted 2.5-improper colouring of G^2. Consider a weighted 2.5-improper 2-colouring of G^2 and let (a,b) be a vertex coloured 0. Vertex (a,b) has at most two neighbours of colour 0, otherwise it will have interference 3. We distinguish three cases:

1. Exactly one of its neighbours is coloured 0; let $(a,b-1)$ be this vertex. Then, the three other neighbours are coloured 1. Consider the two set of vertices $\{(a-1,b-1),(a-1,b+1),(a-2,b)\}$ and $\{(a+1,b-1),(a+1,b+1),(a+2,b)\}$; each of them has at least two vertices coloured 0, otherwise the vertex $(a,b+1)$ or $(a,b-1)$ will have interference 3. But then (a,b) having 4 vertices at distance 2 coloured 0 has interference 3, a contradiction.
2. Two neighbours of (a,b) are coloured 0.
   (a) These two neighbours are opposite, say $(a,b-1)$ and $(a,b+1)$. Consider again the two sets $\{(a-1,b-1),(a-1,b+1),(a-2,b)\}$ and $\{(a+1,b-1),(a+1,b+1),(a+2,b)\}$; they both contain at least one vertex of colour 0 and therefore (a,b) will have interference 3, a contradiction.
   (b) The two neighbours of colour 0 are of the form $(a,b-1)$ and $(a-1,b)$. Consider the two sets of vertices $\{(a+1,b-1),(a+1,b+1),(a+2,b)\}$ and $\{(a+1,b+1),(a-1,b+1),(a,b+2)\}$; these two sets contain at most one vertex of colour 0, otherwise (a,b) will have interference 3. So vertices $(a+1,b-1)$, $(a+2,b)$, $(a,b+2)$ and $(a-1,b+1)$ are of colour 1. Vertex $(a+1,b+1)$ is of colour 0, otherwise $(a+1,b)$ has interference 3. But then $(a,b-2)$ and $(a-1,b-1)$ are of colour 1, otherwise (a,b) will have interference 3. Thus, vertex $(a,b-1)$ has exactly one neighbour coloured 0 and we are again in Case 1.
3. All neighbours of (a,b) are coloured 1. If any of this neighbours has itself a neighbour (distinct from (a,b)) of colour 1, we are in case 1 or 2 for this neighbour. Therefore, all vertices at distance two from (a,b) have colour 0 and the interference in (a,b) is 4, a contradiction.

A weighted 3-improper 2-colouring of G^2 is given in Figure 3(c). Finally, since each vertex has 4 neighbours and 8 vertices at distance two, there is no weighted 7.5-improper 1-colouring of G^2 and, whenever $t \geq 8$, one colour suffices.

Hexagonal Grid. To define the hexagonal grid graph, there are many ways to define the system of coordinates. Here, we use grid coordinates as shown in Figure 4. The hexagonal grid graph is then the graph whose vertex set is the pairs of integers $(a,b) \in \mathbb{Z}^2$ and where each vertex (a,b) has 3 neighbours: $(a-1,b)$, $(a+1,b)$, and $(a,b+1)$ if $a+b$ is odd, or $(a,b-1)$ otherwise.

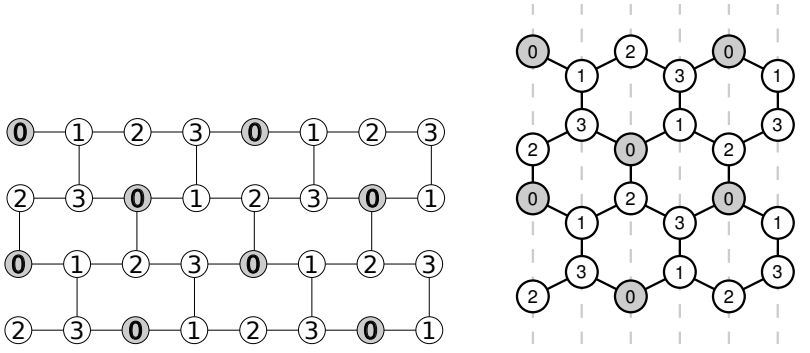
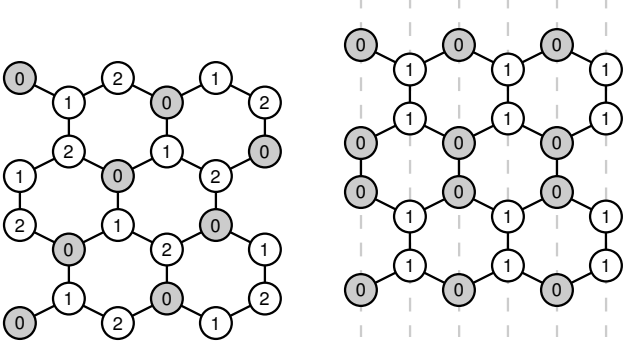


Fig. 4. Optimal construction with $t = 0$, $k = 4$. Left: Graph with coordinates. Right: Corresponding hexagonal grid in the euclidean space.



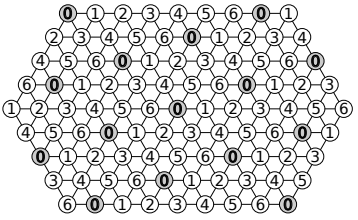
(a) $t = 1, k = 3$ (b) $t = 2, k = 2$

Fig. 5. Optimal constructions for the hexagonal grid

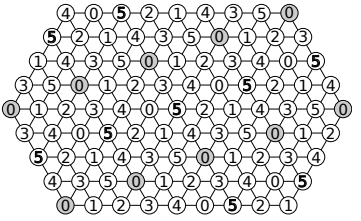
Theorem 8. *If G is an infinite hexagonal grid, then*

$$\chi_t(G^2, w_2) = \begin{cases} 4, & \text{if } 0 \le t < 1; \\ 3, & \text{if } 1 \le t < 2; \\ 2, & \text{if } 2 \le t < 6; \\ 1, & \text{if } 6 \le t. \end{cases}$$

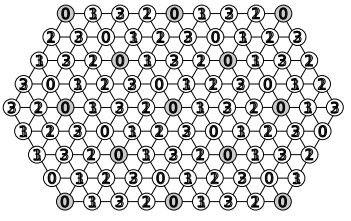
Triangular Grid. The triangular grid is graph whose vertices are all the integer linear combinations $ae_1 + be_2$ of the two vectors $e_1 = (\frac{\sqrt{3}}{2}, \frac{1}{2})$ and $e_2 = (0,1)$. Thus we may identify the vertices with the ordered pairs (a,b) of integers. Each vertex $v = (a,b)$ has six neighbours: its *left neighbour* $(a, b-1)$, its *right neighbour* $(a, b+1)$, its *left-up neighbour* $(a+1, b-1)$, its *right-up neighbour* $(a+1, b+1)$, its *left-down neighbour* $(a-1, b-1)$ and its *right-down neighbour* $(a-1, b+1)$.



(a) Weighted 0-improper 7-colouring of G^2.



(b) Weighted 0.5-improper 6-colouring of G^2.



(c) Weighted 1.5-improper 4-colouring of G^2.

Fig. 6. Constructions for the triangular grid

Theorem 9. *If G is an infinite triangular grid, then*

$$\chi_t(G^2, w_2) = \begin{cases} 7, & \text{if } t = 0; \\ 6, & \text{if } t = 0.5; \\ 5, & \text{if } t = 1; \\ 4, & \text{if } 1.5 \leq t < 3; \\ 3, & \text{if } 3 \leq t < 5; \\ 2, & \text{if } 5 \leq t < 12; \\ 1, & \text{if } 12 \leq t. \end{cases}$$

For determining the lower bounds for the cases in which $\chi_t(G^2, w_2)$ is equal to 2 and 3, the proofs involved too many subcases to be readable. Then, we used CPLEX with the integer programming formulations we present in Section 4 to validate them.

4 Integer Programs, Algorithms and Results

In this section, we look at how to solve the WEIGHTED IMPROPER COLOURING and THRESHOLD IMPROPER COLOURING for realistic instances. We consider Poisson-Voronoi tesselations as they are good models of antennas networks [4, 8, 9]. We present integer programming models for both problems. Then, we introduce two algorithmic approaches for THRESHOLD IMPROPER COLOURING: a simple greedy heuristic and a Branch-and-Bound algorithm.

4.1 Integer Programs and Algorithms

Integer Programming Models. Given an edge-graph $G = (V, E, w)$, $w : E \to \mathbb{R}_+^*$, and a positive real threshold t, we model WEIGHTED IMPROPER COLOURING by using two kinds of variables. Variable x_{ip} indicate if vertex i is coloured p and variable c^p indicate if colour p is used, for every $1 \leq i \leq n$ and $1 \leq p \leq l$, where l is an upper bound for the number of colours needed in an optimal weighted t-improper colouring of G (see Section 2). The model follows:

$$\begin{array}{rll} \min & \sum_p c^p & \\ \text{subject to} & & \\ & \sum_{j \neq i} w(i,j) x_{jp} \leq t + M(1 - x_{ip}) & (\forall i \in V, \forall p \in \{1, \ldots, l\}) \\ & c^p \geq x_{ip} & (\forall i \in V, \forall p \in \{1, \ldots, l\}) \\ & \sum_p x_{ip} = 1 & (\forall i \in V) \\ & x_{ip} \in \{0,1\} & (\forall i \in V, \forall p \in \{1, \ldots, l\}) \\ & c^p \in \{0,1\} & (\forall p \in \{1, \ldots, l\}) \end{array}$$

where M is a large integer. For instance, it is sufficient to choose $M > \sum_{(u,v) \in E} w(u,v)$.

For THRESHOLD IMPROPER COLOURING, given an edge-weighted graph $G = (V, E, w)$, $w : E \to \mathbb{R}_+^*$, and a positive integer k, the model we consider is:

$$\begin{array}{rll} \min & t & \\ \text{subject to} & & \\ & \sum_{j \neq i} w(i,j) x_{jp} \leq t + M(1 - x_{ip}) & (\forall i \in V, \forall p \in \{1, \ldots, k\}) \\ & \sum_p x_{ip} = 1 & (\forall i \in V) \\ & x_{ip} \in \{0,1\} & (\forall i \in V, \forall p \in \{1, \ldots, k\}) \end{array}$$

Levelling Heuristic. We develop a heuristic to solve THRESHOLD IMPROPER COLOURING. The idea is to try to level the distribution of interference over the vertices. Each vertex is coloured one after the other by the colour minimising the local interference. More precisely this is achieved by considering for the nodes not yet coloured the "current interference" i.e. the interference induced by the already coloured vertices.

Precisely, consider a vertex v not yet coloured and a colour $i \in \{1, \ldots, k\}$. We define the potential interference $I'_{v,i}$ as:

$$I'_{v,i} = \sum_{\{u \in N(v) \cap V_i | c(u) = i\}} w(u,v),$$

where V_i is the set of vertices that have already been assigned a colour. The order in which vertices are coloured is decided according to the total potential interference, defined as $I''_v = \sum_{i=1}^k I'_{v,i}$. The algorithm finds a feasible colouring in the first step and tries to improve it for p runs, where p is part of the input.

- The interference target is set $t_t = M$;
- while the number of runs is smaller than p;
  - all potential interferences are set to zero;
  - while there are still vertices to colour:
    - choose a vertex v randomly among the uncoloured vertices that have the maximum total potential interference;

    - try each colour i in the order of increasing potential interference $I'_{v,i}$:
        - if colouring v with i does not result in interference greater than t_t for v or any of its neighbours, colour v with i, else try a new colour;
        - if all colours resulted in excessive interferences, start new run.
- If all the vertices were successfully coloured, set $t_t = \max_{v \in V, i \in \{1,\ldots,k\}} I_v(G, w, c) - \gcd(w)$ and store the colouring as the best found.

As a *randomised greedy colouring* heuristic, it has to be run multiple times to achieve satisfactory results. This is not a practical issue due to low computational cost of each run. The local immutable colouring decision is taken in time $O(k)$. Then, after each such decision, the interference has to be propagated, which takes time linear in the vertex degree. This gives a computational complexity bound $O(kn\Delta)$.

Branch-and-Bound Algorithm. We also implemented a simple Branch-and-Bound algorithm inspired by the above heuristic. The order in which vertices are coloured is produced by a similar procedure to the one used in the above heuristic. In order to compute this order, we start by marking a random vertex and setting is as the first in a to colour list. Then, as long as there are unmarked vertices, we keep choosing a random vertex u among the unmarked vertices with biggest $\sum_{v \in N(u) \cap V_m} w(u,v)$, where V_m is the set of already marked vertices. Then we mark u and append it to the to order. A basic Branch-and-Bound colours vertices in the obtained order. Potential interference, as defined for the heuristic, is tracked with the additional step of decreasing the values when backing from a colouring. Colours are tried in the order of increasing potential interference. Thanks to that it produces results similar to the heuristic in a short time. On the other hand it is guaranteed to find the optimal solution in a finite time.

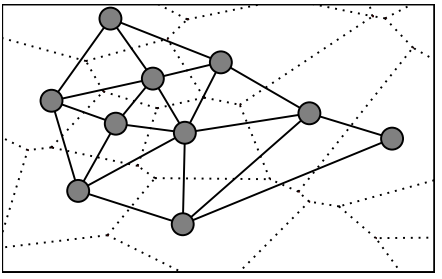
In the following, we compare the performance of these ILP models with the Levelling heuristic and the Branch-and-Bound algorithm .

4.2 Results

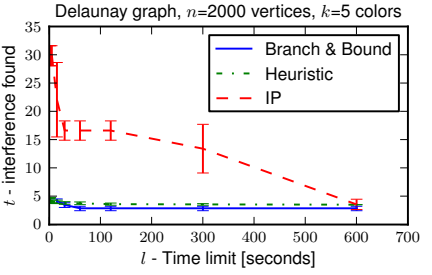
In this section, we look at the performances of the methods to solve the THRESHOLD IMPROPER COLOURING. We consider Delaunay graphs (dual of Voronoi diagram) for a set of random points. This kind of graph is a natural approximation of a network of irregular cells. The interference model is the one described in Section 3: adjacent nodes interfere by 1 and nodes at distance two interfere by 1/2.

Figure 7 shows a performance comparison of the above-mentioned algorithms. For all the plots, each data point represents an average over ten different graphs. The same graph is used for all values of colours and time limit. Therefore sub-figures 7(b) and 7(c) plot how results for a given problem instance get enhanced with increasing time limits. Plots 7(e) and 7(f) show decreasing interference along increasing the number of colours allowed. Finally plot 7(d) shows how well all the programs scale with increasing graph sizes.

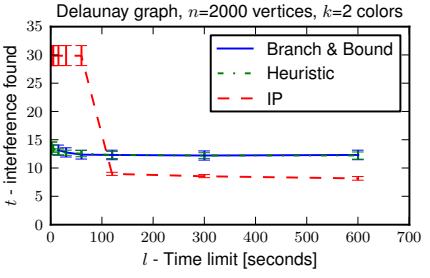
One immediate observation about both the heuristic and Branch-and-Bound algorithm is that they provide solutions in relatively short time. Despite their naive implementation in a high-level programming language, they tend to find near-optimal results in matter of seconds even for graphs of thousands of vertices. On the other hand, with



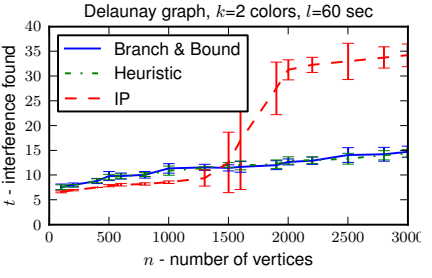
(a) Example Delaunay graph, dotted lines delimit corresponding Voronoi diagram cells



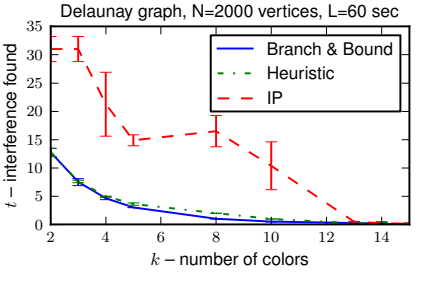
(b) Over time



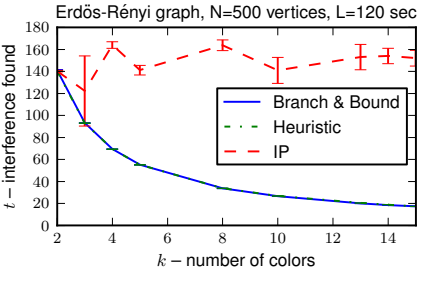
(c) Over time



(d) Over size



(e) Over colours



(f) Over colours

Fig. 7. Results comparison for Levelling heuristic, Branch-and-Bound algorithm and Integer Programme

limited time, they fail to improve up to optimal results, especially with a low number of allowed colours. Although it is easy to envision an implementation faster by orders of magnitude, this may still give little improvement — once a near-optimal solution is found, the Branch-and-Bound algorithm does not improve for a very long time (an example near-optimal solution found in around three minutes was not improved in over six days).

ILP solvers with good Branch-and-Cut implementations do not suffer from this problem. However, they can not take advantage of any specialised knowledge of the problem, only the basic integer programmming representation. Thus it takes much more time to produce first good results. Despite taking advantage of multi-core processing,

CPLEX — ILP solver used in this work, does not scale with increasing graph sizes as well as our simple algorithms. Furthermore, Figure 7(e) reveals one problem specific to integer programming. When increasing the number of allowed colours, obtaining small interferences gets easier. But this introduces additional constraints in the linear program, thus increasing the complexity for a solver.

Above observations are valid only for the very particular case of the simple interference function and very sparse graphs. The average degree in Delaunay graph converges to 6. Proposed algorithms also work quite well for denser graphs. Figure 7(f) plots interferences for different numbers of colours allowed found by the programs for an Erdös-Rényi graph with n=500 and p=0.1. This gives us an average degree of 50. Both Branch-and-Bound and heuristic programs achieve acceptable, and nearly identical, results. But the large number of constraints makes the linear program nearly inefficient.

5 Conclusion, Open Problems and Future Directions

In this paper, we introduced and studied a new colouring problem, WEIGHTED IMPROPER COLOURING . This problem is motivated by the design of telecommunication antenna network in which the interferences between two vertices depends on different factors and can take various values. For each vertex, the sum of the interference it receives should be less than a given threshold value.

We first give general bounds on the weighted-improper chromatic number. We then study the particular case of square, triangular and hexagonal grids. For these graphs, we provide their weighted-improper chromatic number for all possible values of t. Finally, we propose a heuristic and a Branch-and-Bound algorithm to find good solutions of the problem. We compare their results with the one of an integer program on cell-like networks, Poisson Voronoi tessellations.

Open Problems and Future Directions. Many problems remain to be solved :

- For the study of the grid graphs, we considered a specific function where vertex at distance one interfere by 1 and vertices at distance 2 by 1/2. Other weight functions should be considered. e.g. $1/d^2$ or $1/(2^{d-1})$, where d is the distance between vertices.
- Other families of graphs could be considered, for example hypercubes.
- Let $G = (V, E, w)$ be an edge-weighted graph where the weights are all equal to 1 or M. Let G_M be the subgraph of G induced by the edges of weight M; is it true that if $\Delta(G_M) << \Delta(G)$, then $\chi_t(G,w) \leq \chi_t(G) \leq \left\lceil \frac{\Delta(G,w)+1}{t+1} \right\rceil$? A similar result for $L(p,1)$-labelling [10] suggests it could be true.

References

1. Aardal, K.I., van Hoesel, S.P.M., Koster, A.M.C.A., Mannino, C., Sassano, A.: Models and solution techniques for frequency assignment problems. Annals of Operations Research 153(1), 79–129 (2007)

2. Alouf, S., Altman, E., Galtier, J., Lalande, J.F., Touati, C.: Quasi-optimal bandwidth allocation for multi-spot MFTDMA satellites. In: INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE, vol. 1, pp. 560–571. IEEE (2005)
3. Araujo, J., Bermond, J.-C., Giroire, F., Havet, F., Mazauric, D., Modrzejewski, R.: Weighted Improper Colouring. Research Report RR-7590, INRIA (April 2011)
4. Baccelli, F., Klein, M., Lebourges, M., Zuyev, S.: Stochastic geometry and architecture of communication networks. Telecom. Systems 7(1), 209–227 (1997)
5. Brooks, R.L.: On colouring the nodes of a network. Mathematical Proceedings of the Cambridge Philosophical Society 37(02), 194–197 (1941)
6. Correa, R., Havet, F., Sereni, J.-S.: About a Brooks-type theorem for improper colouring. Australasian Journal of Combinatorics 43, 219–230 (2009)
7. Fischetti, M., Lepschy, C., Minerva, G., Romanin-Jacur, G., Toto, E.: Frequency assignment in mobile radio systems using branch-and-cut techniques. European Journal of Operational Research 123(2), 241–255 (2000)
8. Gupta, P., Kumar, P.R.: The capacity of wireless networks. IEEE Transactions on Information Theory 46(2), 388–404 (2000)
9. Haenggi, M., Andrews, J.G., Baccelli, F., Dousse, O., Franceschetti, M.: Stochastic geometry and random graphs for the analysis and design of wireless networks. IEEE Journal on Selected Areas in Communications 27(7), 1029–1046 (2009)
10. Havet, F., Reed, B., Sereni, J.-S.: L(2,1)-labelling of graphs. In: Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2008, pp. 621–630. Society for Industrial and Applied Mathematics, Philadelphia (2008)
11. Karp, R.: Reducibility among combinatorial problems. In: Miller, R., Thatcher, J. (eds.) Complexity of Computer Computations, pp. 85–103. Plenum Press (1972)
12. Lovász, L.: On decompositions of graphs. Studia Sci. Math. Hungar. 1, 238–278 (1966)
13. Mannino, C., Sassano, A.: An enumerative algorithm for the frequency assignment problem. Discrete Applied Mathematics 129(1), 155–169 (2003)
14. Woodall, D.R.: Improper colorings of graphs. In: Nelson, R., Wilson, R.J. (eds.) Pitman Res. Notes Math. Ser., vol. 218, pp. 45–63. Longman Scientific and Technical (1990)
15. Yeh, R.K.: A survey on labeling graphs with a condition at distance two. Discrete Mathematics 306(12), 1217–1231 (2006)

Algorithmic Aspects of Dominator Colorings in Graphs

S. Arumugam[1,2] K. Raja Chandrasekar[1], Neeldhara Misra[3], Geevarghese Philip[3], and Saket Saurabh[3]

[1] National Centre for Advanced Research in Discrete Mathematics (n-CARDMATH), Kalasalingam University, Krishnankoil, India
{s.arumugam.klu,rajmath84}@gmail.com
[2] Conjoint Professor, School of Electrical Engineering and Computer Science The University of Newcastle, NSW 2308, Australia
[3] The Institute of Mathematical Sciences, Chennai, India
{neeldhara,gphilip,saket}@imsc.res.in

Abstract. In this paper we initiate a systematic study of a problem that has the flavor of two classical problems, namely Coloring and Domination, from the perspective of algorithms and complexity. A *dominator coloring* of a graph G is an assignment of colors to the vertices of G such that it is a proper coloring and every vertex dominates all the vertices of at least one color class. The minimum number of colors required for a dominator coloring of G is called the *dominator chromatic number* of G and is denoted by $\chi_d(G)$. In the Dominator Coloring (DC) problem, a graph G and a positive integer k are given as input and the objective is to check whether $\chi_d(G) \leq k$. We first show that unless P=NP, DC cannot be solved in polynomial time on bipartite, planar, or split graphs. This resolves an open problem posed by Chellali and Maffray [*Dominator Colorings in Some Classes of Graphs, Graphs and Combinatorics, 2011*] about the polynomial time solvability of DC on chordal graphs. We then complement these hardness results by showing that the problem is fixed parameter tractable (FPT) on chordal graphs and in graphs which exclude a fixed apex graph as a minor.

Keywords: Dominator Coloring, Fixed-Parameter Tractability, Chordal Graphs, Apex-Minor-Free Graphs.

1 Introduction

Dominating Set and Coloring are among the most fundamental problems in graph theory, algorithms and combinatorial optimization. Dominating Set asks for the minimum set of vertices such that every vertex of the graph not in this set has a neighbor in it. In Coloring we are asked to color the vertices with as few colors as possible, so that no edge is monochromatic, that is, both the endpoints of each edge receive different colors. These are classical NP-hard problems [17] and are well-studied from the point of view of approximation algorithms [12,23,25,26,27] and parameterized complexity [10,14,16]. Dominating

C.S. Iliopoulos and W.F. Smyth (Eds.): IWOCA 2011, LNCS 7056, pp. 19–30, 2011.
© Springer-Verlag Berlin Heidelberg 2011

Set and Coloring are "hard" problems from these perspectives. Thus, Dominating Set and Coloring are known to be W[2]-complete and para-NP complete, respectively, in parameterized complexity [10]. Further, $(1-o(1))\ln n$ and $n^{\epsilon}; \epsilon > 0$ are respective thresholds below which these problems cannot be approximated efficiently (unless NP has slightly super-polynomial time algorithm [12] or unless P=NP [27]).

Dominating Set and Coloring have a number of applications and this has led to the algorithmic study of numerous variants of these problems. Among the most well known ones are Connected Dominating Set, Independent Dominating Set, Perfect Code, List Coloring, Edge Coloring, Acyclic Edge Coloring and Choosability. Since both the problem and its variants are computationally hard problems, most of the research centers around algorithms in special classes of graphs like interval graphs, chordal graphs, planar graphs and H-minor free graphs. In this paper we initiate a systematic algorithmic study on the Dominator Coloring (DC) problem that has a flavor of both these classical problems. A *dominator coloring* of a graph G is an assignment of colors to the vertices of G such that it is a proper coloring (no edge is monochromatic) and every vertex dominates all vertices of at least one color class. The minimum number of colors required for a dominator coloring of G is called the *dominator chromatic number* of G and is denoted by $\chi_d(G)$. The problem we study is formally defined as follows.

Dominator Coloring (DC)

Input:	A graph G and an integer $k \geq 1$.
Parameter:	k.
Question:	Does there exist a dominator coloring of G using at most k colors?

Gera et al. [22] introduced the concept of dominator chromatic number, and a number of basic combinatorial and algorithmic results on DC have been obtained [20,21,22,24]. For example, it was observed by Gera et al. [22] that DC is NP-complete on general graphs by a simple reduction from 3-Coloring. More precisely, for any fixed $k \geq 4$, it is NP-complete to decide if a graph admits a dominator coloring with at most k colors [22]. In a recent paper Chellali and Maffray [6] show that unlike 3-Coloring, one can decide in polynomial time if a graph has dominator chromatic number 3. Furthermore, they show that the problem is polynomial time solvable on P_4 free graphs, and leave as a "challenging open problem" whether the problem can be solved in polynomial time on chordal graphs.

In this paper we do a thorough algorithmic study of this problem, analyzing both the classical complexity and the parameterized complexity. We begin by showing that unless P=NP, DC cannot be solved in polynomial time on bipartite, planar, or split graphs. The first two arguments are simple but make use of an unusual sequence of observations. The NP-completeness reduction on split graphs is quite involved. Since split graphs form a subclass of chordal graphs, this answers, in the negative, the open problem posed by Chellali and Maffray.

We complement our hardness results by showing that the problem is "fixed parameter tractable" on several of the graph classes mentioned above. Informally, a *parameterization* of a problem assigns an integer k to each input instance and a parameterized problem is *fixed-parameter tractable* (FPT) if there is an algorithm that solves the problem in time $f(k) \cdot |I|^{O(1)}$, where $|I|$ is the size of the input and f is an arbitrary computable function that depends only on the parameter k. We refer the interested reader to standard texts [10,14] on parameterized complexity. We show that DC is FPT on planar graphs, apex minor free graphs, split graphs and chordal graphs.

2 Preliminaries

All graphs in this article are finite and undirected, with neither loops nor multiple edges. n denotes the number of vertices in a graph, and m the number of edges. A subset $D \subseteq V$ of the vertex set V of a graph G is said to be a *dominating set* of G if every vertex in $V \setminus D$ is adjacent to some vertex in D. The *domination number* $\gamma(G)$ of G is the size of a smallest dominating set of G. A *proper coloring* of graph G is an assignment of colors to the vertices of G such that the two end vertices of any edge have different colors. The *chromatic number* $\chi(G)$ of G is the minimum number of colors required in a proper coloring of G. A *clique* is a graph in which there is an edge between every pair of vertices. The *clique number* $\omega(G)$ of G is the size of a largest clique which is a subgraph of G. We make use of the following known results.

Theorem 1. [20] *Let G be a connected graph. Then $max\{\chi(G), \gamma(G)\} \leq \chi_d(G) \leq \chi(G) + \gamma(G)$.*

Definition 1. *A* tree decomposition *of a (undirected) graph $G = (V, E)$ is a pair (X, U) where $U = (W, F)$ is a tree, and $X = (\{X_i \mid i \in W\})$ is a collection of subsets of V such that*

1. $\bigcup_{i \in W} X_i = V$,
2. *for each edge $vw \in E$, there is an $i \in W$ such that $v, w \in X_i$, and*
3. *for each $v \in V$, the set of vertices $\{i \mid v \in X_i\}$ forms a subtree of U.*

The width *of (X, U) is $\max_{i \in W}\{|X_i| - 1\}$. The* treewidth *$tw(G)$ of G is the minimum width over all the tree decompositions of G.*

Both our FPT algorithms make use of the fact that the DC problem can be expressed in Monadic Second Order Logic (MSOL) on graphs. The syntax of MSOL on graphs includes the logical connectives $\vee$, $\wedge$, $\neg$, $\Leftrightarrow$, $\Rightarrow$, variables for vertices, edges, sets of vertices and sets of edges, the quantifiers $\forall$, $\exists$ that can be applied to these variables, and the following five binary relations: (1) $u \in U$ where u is a vertex variable and U is a vertex set variable; (2) $d \in D$ where d is an edge variable and D is an edge set variable; (3) $\mathbf{inc}(d, u)$, where d is an edge variable, u is a vertex variable, and the interpretation is that the edge d is incident on the vertex u; (4) $\mathbf{adj}(u, v)$, where u and v are vertex variables

and the interpretation is that u and v are adjacent; (5) equality of variables representing vertices, edges, sets of vertices and sets of edges.

Many common graph and set-theoretic notions can be expressed in MSOL [5,8]. In particular, let $V_1, V_2, \ldots, V_k$ be a set of subsets of the vertex set $V(G)$ of a graph G. Then the following notions can be expressed in MSOL:

- $V_1, V_2, \ldots, V_k$ is a partition of $V(G)$:

$$\begin{aligned} Part(V(G); V_1, V_2, \ldots, V_k) \equiv \forall v \in V(G)[(v \in V_1 \vee v \in V_2 \vee \cdots \vee v \in V_k) \quad \wedge \\ (\neg(v \in V_1 \cap V_2)) \wedge (\neg(v \in V_1 \cap V_3)) \wedge \cdots \wedge (\neg(v \in V_{k-1} \cap V_k))] \quad \wedge \\ (\exists v \in V(G)[v \in V_1]) \wedge (\exists v \in V(G)[v \in V_2]) \wedge \cdots \wedge (\exists v \in V(G)[v \in V_k]) \end{aligned}$$

- V_i is an independent set in G:

$$IndSet(V_i) \equiv \forall u \in V_i[\forall v \in V_i[\neg \mathbf{adj}(u, v)]]$$

- Vertex v dominates all vertices in the set V_i:

$$Dom(v, V_i) \equiv \forall w \in V_i[\neg(w = v) \implies \mathbf{adj}(v, w)]$$

For a graph G and a positive integer k, we use $\varphi(G, k)$ to denote an MSOL formula which states that G has a dominator coloring with at most k colors:

$$\begin{aligned} \varphi(G, k) \equiv \exists V_1, V_2, \ldots, V_k \subseteq V(G)[Part(V(G); V_1, V_2, \ldots, V_k) \quad \wedge \\ IndSet(V_1) \wedge IndSet(V_2) \wedge \cdots \wedge IndSet(V_k) \quad \wedge \\ \forall v \in V(G)[Dom(v, V_1) \vee Dom(v, V_2) \vee \cdots \vee Dom(v, V_k)]] \end{aligned} \tag{1}$$

The following well known result states that every optimization problem expressible in MSOL has a linear time algorithm on graphs of bounded treewidth.

Proposition 1. [1,3,5,7,9] *Let φ be a property that is expressible in Monadic Second Order Logic. For any fixed positive integer t, there is an algorithm that, given a graph G of treewidth at most t as input, finds a largest (alternatively, smallest) set S of vertices of G that satisfies φ in time $f(t, |\varphi|)|V(G)|$ for a computable function $f()$.*

Since the size $|\varphi(G, k)|$ of the MSOL expression 1 is a function of k, we have

Theorem 2. *Given a graph G of treewidth t and a positive integer k as inputs, the* Dominator Coloring *problem can be solved in $f(t, k)|V(G)|$ time for a computable function $f()$.*

The operation of *contracting* an edge $\{u, v\}$ of a graph consists of replacing the two vertices u, v with a single vertex which is adjacent to all the former neighbours of u and v. A graph H is said to be a *contraction* of a graph G if H can be obtained from G by contracting zero or more edges of G. H is said to be a *minor* of G if H is a contraction of some subgraph of G. A graph G is said to be *apex* graph if there exists a vertex in G whose removal from G yields a planar

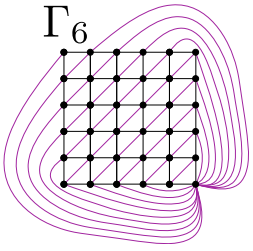


Fig. 1. The graph Γ_6

graph. A family $\mathcal{F}$ of graphs is said to be *apex minor free* if there is a specific apex graph H such that no graph in $\mathcal{F}$ has H as a minor. For instance, planar graphs are apex minor free since they exclude the apex graph K_5 as a minor. The treewidth of an apex minor free graph can be approximated to within a constant factor in polynomial time:

Proposition 2. [13, Theorem 6.4] *For any graph H, there is a constant w_H and a polynomial time algorithm which finds a tree decomposition of width at most $w_H t$ for any H-minor-free graph G of treewidth t.*

For $\ell \in \mathbb{N}$, Γ_ℓ is defined [15] to be the graph obtained from the $\ell \times \ell$-grid by (1) triangulating the internal faces such that all the internal vertices become of degree 6 and all non-corner external vertices are of degree 4, and (2) adding edges from one corner of degree two to all vertices of the external face. Figure 1 depicts Γ_6. Fomin et al. showed that any apex minor free graph of large treewidth contains a proportionately large Γ_ℓ as a contraction. More precisely:

Proposition 3. [15, Theorem 1] *For any apex graph H, there is a constant c_H such that every connected graph G which excludes H as a minor and has treewidth at least $c_H \ell$ contains Γ_ℓ as a contraction.*

3 Hardness Results

In this section we show that DC is NP-hard on very restricted classes of graphs. The only known hardness result for this problem is that it is NP-complete on general graphs [22]. In fact even determining whether there exists a dominator coloring of G using at most 4 colors is NP-complete. The proof is obtained by a reduction from 3-Coloring – checking whether an input graph is 3-colorable or not – to DC. Given an instance G to 3-Coloring, an instance G' for DC is obtained by adding a new vertex (universal vertex) and making it adjacent to every vertex of G. Now one can easily argue that G is 3 colorable if and only if G' has dominator coloring of size at most 4. Notice, however, that this simple reduction cannot be used to show that DC is NP-complete on restricted graph classes like planar graphs or split graphs or chordal graphs. We start with a few simple claims that we will make use of later.

Lemma 1. *Let $G = (V, E)$ be a graph. Given a proper a-coloring $\mathcal{C}$ of G and a dominating set D of G with $|D| = b$, we can find, in $O(|V| + |E|)$ time, a dominator coloring of G with at most $a + b$ colors.*

Proof. Let $\mathcal{C} = \{V_1, V_2, \ldots, V_a\}$ be a proper coloring of G and let D be a dominating set with $|D| = b$. Then $\mathcal{C}' = \{\{v\} : v \in D\} \cup \{V_i \cap (V - D) : V_i \in \mathcal{C}\}$ is a dominator coloring of G with at most $a + b$ colors. □

Corollary 1. [⋆][1] *If there exists an α-approximation algorithm for the chromatic number problem and a β-approximation algorithm for the domination number problem, then there exists an $(\alpha + \beta)$-approximation algorithm for the dominator chromatic number problem.*

Lemma 2. [⋆] *Let $\mathcal{F}$ be a class of graphs on which the Dominating Set problem is NP-complete. If the disjoint union of any two graphs in $\mathcal{F}$ is also in $\mathcal{F}$, then there is no polynomial time algorithm that finds a constant additive approximation for the Dominating Set problem on $\mathcal{F}$, unless $P = NP$.*

Corollary 2. [⋆] Dominator Coloring *on planar graphs cannot be solved in polynomial time, unless $P = NP$.*

Corollary 3. [⋆] Dominator Coloring *on bipartite graphs cannot be solved in polynomial time, unless $P = NP$.*

3.1 NP-Hardness of DC on Split Graphs

We now proceed to prove that the DC problem is NP-complete for split graphs. Our starting point is the following known characterization:

Theorem 3. [2] *Let G be a split graph with split partition (K, I) and $|K| = \omega(G)$, where K is a clique and I an independent set. Then $\chi_d(G) = \omega$ or $\omega + 1$. Further $\chi_d(G) = \omega$ if and only if there exists a dominating set D of G such that $D \subseteq K$ and every vertex v in I is nonadjacent to at least one vertex in $K \setminus D$.*

We exploit this characterization, and prove NP-completeness on split graphs by demonstrating the NP-completeness of the problem of checking if there exists a dominating set D of G such that $D \subseteq K$ and every vertex v in I is nonadjacent to at least one vertex in $K \setminus D$. We call this problem Split Graph Domination.

For showing Split Graph Domination NP-complete, we will need to define an intermediate problem called Partition Satisfiability, and demonstrate that it is NP-complete. We will then show that DC is NP-hard on split graphs by establishing a reduction from Partition Satisfiability.

Let ϕ be a CNF formula. Then we use $\mathcal{C}(\phi)$ to denote the set of clauses of ϕ. If C is a clause of ϕ, then we use $\nu(C)$ to denote the set of variables that appear in C. A clause is said to be all-positive (negative) if all the literals that appear in it are positive (negative).

[1] Due to space constraints, proofs of results marked with a [⋆] have been deferred to a longer version of the paper.

Definition 2 (Partition Normal Form). *A CNF formula ϕ over the variable set V is said to be in partition normal form if $\mathcal{C}(\phi)$ admits a partition into two parts $C_P(\phi)$ and $C_N(\phi)$ and there exists a bijection $f : C_P(\phi) \to C_N(\phi)$ such that for every $C \in C_P(\phi)$ the following conditions are satisfied:* (1) $\nu(C) \cup \nu(f(C)) = V$ *and* (2) $\nu(C) \cap \nu(f(C)) = \emptyset$. *Any clause in $C_P(\phi)$ is required to be an all-positive clause and any clause in $C_N(\phi)$ is required to be an all-negative clause.*

We are now ready to describe the problem Partition Satisfiability.

Partition Satisfiability
Input: A formula ϕ in CNF, over variables in V, given in partition normal form.
Question: Is ϕ satisfiable?

We establish the NP-completeness of Partition Satisfiability by a reduction from Disjoint Factors:

Disjoint Factors
Input: A word w over an alphabet Σ.
Question: For every $a \in \Sigma$, does there exist a substring w_a of w that begins and ends in a, such that for every $a, b \in \Sigma$, w_a and w_b do not overlap in w?

The problem of Disjoint Factors is known to be NP-complete [4]. Substrings that begin and end with the same letter a are referred to as *a-factors*.

Lemma 3. Partition Satisfiability *is* NP-complete.

Proof. Let $w = w_1 w_2 \dots w_n$ be an instance of Disjoint Factors over the alphabet

$$\Sigma = \{a_1, \dots, a_k\}.$$

For $1 \leq i < j \leq n$ and $1 \leq l \leq k$, we call the triplet (i, j, l) *valid* if the substring $w_i \dots w_j$ is an a_l-factor. Let $\mathcal{F}$ denote the set of valid triplets. We construct an instance of Partition Satisfiability as follows:

For every valid triplet (i, j, l), introduce the variable $P_l(i, j)$. For every $1 \leq l \leq k$, introduce the clause:

$$C_l := \left(\bigvee_{\{i,j \,:\, (i,j,l) \in \mathcal{F}\}} P_l(i, j) \right).$$

Let ϕ_{FACTOR} be the conjunction of the clauses thus formed: $\phi_{\text{FACTOR}} := C_1 \wedge C_2 \wedge \dots \wedge C_k$.

Further, for every i_1, j_1 and i_2, j_2 such that $1 \leq i_1 < j_1 \leq n$ and $1 \leq i_2 < j_2 \leq n$, and $[i_1, j_1] \cap [i_2, j_2] \neq \emptyset$, and there exist l_1, l_2; $1 \leq l_1, l_2 \leq k$, such that $(i_1, j_1, l_1) \in \mathcal{F}$ and $(i_2, j_2, l_2) \in \mathcal{F}$, we introduce the following clause:

$$C := \left(\overline{P_{l_1}(i_1, j_1)} \vee \overline{P_{l_2}(i_2, j_2)}\right)$$

Let $\mathcal{D}$ denote the set of clauses described above. Further, let ϕ_{DISJOINT} be the conjunction of these clauses: $\phi_{\text{DISJOINT}} := \bigwedge_{C \in \mathcal{D}} C$.

Claim. The formula: $\phi := \phi_{\text{DISJOINT}} \wedge \phi_{\text{FACTOR}}$ is satisfiable if and only if (w, Σ) is a YES-instance of DISJOINT FACTORS.

Proof. ($\Rightarrow$) Let χ be a satisfying assignment of ϕ. For all l, $1 \leq l \leq k$, there exists at least one pair (i, j), $1 \leq i < j \leq n$, such that χ sets $P_l(i, j)$ to 1. Indeed, if not, χ would fail to satisfy the clause C_l. Now, note that $w_i \dots w_j$ is a a_l-factor, since the variable $P_l(i, j)$ corresponds to a valid triplet.

We pick $w_i \dots w_j$ as the factor for a_l (if $P_l(i, j)$ is set to 1 by χ for more than one pair (i, j), then any one of these pairs will serve our purpose). It only remains to be seen that for $r, s \in \Sigma$, if $w_{i_1} \dots w_{j_1}$ is chosen as a a_r-factor, and $w_{i_2} \dots w_{j_2}$ is chosen as a a_s-factor, then $w_{i_1} \dots w_{j_1}$ and $w_{i_2} \dots w_{j_2}$ do not overlap in w. This is indeed the case, for if they did overlap, then it is easily checked that χ would fail to satisfy the clause: $\left(\overline{P_r(i_1, j_1)} \vee \overline{P_s(i_2, j_2)}\right)$.

($\Leftarrow$) If (w, Σ) is a YES-instance of DISJOINT FACTORS, then for every l, $1 \leq l \leq k$, there exist i, j; $1 \leq i < j \leq n$, such that $(i, j, l) \in \mathcal{F}$. We claim that setting all the "corresponding" $P_l(i, j)$ variables to 1 is a satisfying assignment for ϕ.

Indeed, every C_l is satisfied because there exists an a_l-factor for every l. Further, it is routine to verify that all clauses in $\mathcal{D}$ are satisfied because the chosen factors do not overlap in w. □

Now, it remains to construct from ϕ an equivalent formula ψ that is in partition normal form. To this end, we will use two new variables, $\{x, y\}$. Recall that we use V to denote the set of variables that appear in ϕ. For every clause C_l, define the clause $\hat{C}_l$ as: $\hat{C}_l := \left(\overline{x} \vee \overline{y} \vee \bigvee_{z \,\in\, V \setminus \nu(C_l)} \overline{z}\right)$.

Similarly, for every clause $C \in \mathcal{D}$, define $\hat{C}$ as: $\hat{C} := \left(x \vee y \vee \bigvee_{z \,\in\, V \setminus \nu(C)} z\right)$.

Let ψ be obtained by the conjunction of ϕ with the newly described clauses: $\psi := \phi \wedge \left(\bigwedge_{1 \leq l \leq k} \hat{C}_l\right) \wedge \left(\bigwedge_{C \in \mathcal{D}} \hat{C}\right)$.

Clearly, ψ is in partition normal form. The following partition of the clauses of ψ: $C_P = \{C_l \ : \ 1 \leq l \leq k\} \cup \{\hat{C} \ : \ C \in \mathcal{D}\}$ and $C_N = \{\hat{C}_l \ : \ 1 \leq l \leq k\} \cup \{C \ : \ C \in \mathcal{D}\}$ is a partition into all-positive and all-negative clauses. The bijection f defined as: $f(C_l) = \hat{C}_l$, for $1 \leq l \leq k$ and $f(\hat{C}) = C$, for $C \in \mathcal{D}$ is easily seen to be a bijection with the properties demanded by the definition of the partition normal form. We now arrive at our concluding claim:

Claim. ϕ is satisfiable if and only if ψ is satisfiable.

Proof. ($\Rightarrow$) Let χ be a satisfying assignment for ϕ. Extend ϕ to the new variables $\{x, y\}$ as follows: $\chi(x) = 1$ and $\chi(y) = 0$. It is easy to see that χ is satisfying for ψ.

($\Leftarrow$) This direction is immediate, as $\mathcal{C}(\phi) \subseteq \mathcal{C}(\psi)$. □

The proof that Partition Satisfiability is NP-hard follows when we put the two claims together: by appending the construction of ψ from ϕ to the formula ϕ obtained from the Disjoint Factors instance, we obtain an equivalent instance of Partition Satisfiability. This concludes the proof. We note that membership in NP is trivial — an assignment to the variables is clearly a certificate that can be verified in linear time. The lemma follows. □

Recall the Split Graph Domination problem that we introduced in the beginning of this section:

Split Graph Domination
Input: Split graph G with split partition (K, I) and $|K| = \omega$.
Question: Does there exist a dominating set D of G such that $D \subseteq K$ and every vertex v in I is nonadjacent to at least one vertex in $K \setminus D$?

We now turn to a proof that Split Graph Domination is NP-complete.

Theorem 4. Split Graph Domination *is* NP-complete.

Proof. It is straightforward to see that Split Graph Domination is in NP. We now prove that it is NP-hard by a reduction from Partition Satisfiability.

Given an instance ϕ (over the variables V) of Partition Satisfiability, we construct a split graph G with split partition (K, I) as follows. Introduce, for every variable in V, a vertex in K and for every all-positive clause of ϕ, a vertex in I: $K = \{v[x] \ : \ x \in V\}, I = \{u[C] \ : \ C \in C_P(\phi)\}$.

A pair of vertices $v[x]$ and $u[C]$ are adjacent if the variable x belongs to the clause C, that is, $x \in \nu(C)$. We also make all vertices in K pairwise adjacent and all vertices in I pairwise independent. This completes the construction.

Suppose ϕ admits a satisfying truth assignment χ. Let $D = \{v[x] \in K \ : \ \chi(x) = 1\}$. We now prove that this choice of D is a split dominating set. Consider $u[C] \in I$. There exists at least one $x \in V$ such that $x \in \nu(C)$ and $\chi(x) = 1$. Thus the corresponding vertex $v[x] \in D$, and $u[C]$ is dominated. Further, consider the all-negative clause $\hat{C}$ corresponding to C, that contains every variable in V that is not in $\nu(C)$. Since χ is a satisfying assignment, there is at least one $y \in V \setminus \nu(C)$ such that $\chi(y) = 0$. Clearly, $v[y] \notin D$, and $v[y]$ is not adjacent to $u[C]$.

Conversely, suppose there exists a dominating set $D \subseteq K$ such that each $u[C]$ in I is nonadjacent to at least one vertex in $K \setminus D$. Consider the following truth assignment χ for ϕ: $\chi(x) = 1$ if, and only if, $v[x] \in K \cap D$. We now prove that χ is a satisfying assignment. Consider any all-positive clause C. Since $u[C]$ was dominated by D, there exists a variable $x \in \nu(C)$ such that $v[x] \in D$, and thus $\chi(x) = 1$. Consider the corresponding all-negative clause $\hat{C}$. Since $K \setminus D$ contains at least one non-neighbor of $v[x]$, there exists a $y \notin \nu(C)$ such that $\chi(y) = 0$. Note that $y \notin \nu(C)$ implies that $y \in \nu(\hat{C})$. Recall that the assignment $\chi(y) = 0$ is then satisfying for $\hat{C}$, since $\hat{C}$ is an all-negative clause. □

From Theorem 3 and Theorem 4 we get

Theorem 5. *DC when restricted to split graphs is* NP-complete.

4 Parameterized Algorithms

In this section we investigate the fixed-parameter tractability of the DC problem in certain graph classes. Recall that it is NP-complete to decide if a graph admits a dominator coloring with at most 4 colors [22]. It follows that in *general* graphs, the DC problem cannot be solved even in time $n^{g(k)}$ for any function $g(k)$ — that is, DC does not belong to the complexity class XP — unless P=NP. Hence DC is not FPT in general graphs unless P=NP. As we show below, however, the problem is FPT in two important classes of graphs, namely apex-minor-free graphs (which include planar graphs as a special case) and chordal graphs. Recall that it is NP-complete to decide if a planar graph admits a proper 3-coloring [18]. As a consequence, the Graph Coloring problem parameterized by the number of colors is not even in XP in planar graphs. Our result for planar graphs thus brings out a marked difference in the parameterized complexity of these two problems when restricted to planar graphs.

Apex Minor Free Graphs. We now show that the Dominator Coloring problem is FPT on apex minor free graphs. This implies, as a special case, that the problem is FPT on planar graphs. We first show that if the treewidth of the input apex minor free graph is large, then the graph has no dominator coloring with a small number of colors.

Theorem 6. [⋆] *For any apex graph H, there is a constant d_H such that any connected graph G which excludes H as a minor and has treewidth at least $d_H\sqrt{k}$ has no dominator coloring with at most k colors.*

Let (G, k) be an instance of the Dominator Coloring problem, where G excludes the apex graph H as a minor. Let $t = w_H d_H \sqrt{k}$ where d_H, w_H are the constants of Theorem 6 and Proposition 2, respectively. To solve the problem on this instance, we invoke the approximation subroutine implied by Proposition 2 on the graph G. If this subroutine returns a tree decomposition with treewidth more than t, then we return NO as the answer. Otherwise we solve the problem using the algorithm of Theorem 2, and so we have:

Theorem 7. [⋆] *The* Dominator Coloring *problem is fixed parameter tractable on apex minor free graphs.*

Chordal Graphs and Split Graphs. We now show that the Dominator Coloring problem is FPT on chordal graphs. For a special class of chordal graphs, namely split graphs, we give an FPT algorithm which runs in time single-exponential in the parameter.

Theorem 8. *The* Dominator Coloring *problem is fixed parameter tractable on chordal graphs.*

Proof. Let (G, k) be an instance of the Dominator Coloring problem, where G is chordal. The algorithm first finds a largest clique in G. If the number of

vertices in this clique is more than k, then it returns NO as the answer. Otherwise it invokes the algorithm of Theorem 2 as a subroutine to solve the problem.

To see that this algorithm is correct, observe that if G contains a clique C with more than k vertices, then $\chi(G) > k$ since it requires more than k colors to properly color the subgraph C itself. It follows from Theorem 1 that $\chi_d(G) > k$, and so it is correct to return NO. A largest clique in a chordal graph can be found in linear time [19]. If the largest clique in G has size no larger than k, then — as is well known — the treewidth of G is at most $k-1$, and so the subroutine from Theorem 2 runs in at most $f((k-1),k)|V(G)| = g(k)|V(G)|$ time. Thus the algorithm solves the problem in FPT time. □

The Dominator Coloring problem can be solved in "fast" FPT time on split graphs:

Theorem 9. [⋆] *The* Dominator Coloring *problem can be solved in* $O(2^k \cdot n^2)$ *time on a split graph on* n *vertices.*

5 Conclusion and Scope

We derived several algorithmic results about the Dominator Coloring (DC) problem. We showed that the DC problem remains hard on several graph classes, including bipartite graphs, planar graphs, and split graphs. In the process we also answered, in the negative, an open problem by Chellali and Maffray [6] about the polynomial time solvability of DC on chordal graphs. Finally, we showed that though the problem cannot be solved in polynomial time on the aforementioned graph classes, it is FPT on apex minor free graphs and on chordal graphs. From Theorem 1 and from the fact that finding a constant additive approximation for the Dominating Set problem is W[2]-hard [11], it follows that the DC problem is W[2]-hard on bipartite graphs, and so also on the larger class of perfect graphs. An interesting problem which remains open is whether the DC problem is solvable in polynomial time on interval graphs.

References

1. Arnborg, S., Lagergren, J., Seese, D.: Easy problems for tree-decomposable graphs. Journal of Algorithms 12(2), 308–340 (1991)
2. Arumugam, S., Bagga, J., Chandrasekar, K.R.: On dominator colorings in graphs (2010) (manuscript)
3. Bodlaender, H.L.: A linear time algorithm for finding tree-decompositions of small treewidth. SIAM Journal on Computing 25, 1305–1317 (1996)
4. Bodlaender, H.L., Thomassé, S., Yeo, A.: Kernel Bounds for Disjoint Cycles and Disjoint Paths. In: Fiat, A., Sanders, P. (eds.) ESA 2009. LNCS, vol. 5757, pp. 635–646. Springer, Heidelberg (2009)
5. Borie, R.B., Parker, G.R., Tovey, C.A.: Automatic Generation of Linear-Time Algorithms from Predicate Calculus Descriptions of Problems on Recursively Constructed Graph Families. Algorithmica 7, 555–581 (1992)

6. Chellali, M., Maffray, F.: Dominator colorings in some classes of graphs. Graphs and Combinatorics, 1–11 (2011)
7. Courcelle, B.: The monadic second-order logic of graphs. i. recognizable sets of finite graphs. Information and Computation 85(1), 12–75 (1990)
8. Courcelle, B.: The expression of graph properties and graph transformations in monadic second-order logic. In: Rozenberg, G. (ed.) Handbook of Graph Grammars and Computing by Graph Transformations: Foundations, ch. 5, vol. 1. World Scientific (1997)
9. Courcelle, B., Mosbah, M.: Monadic second-order evaluations on tree-decomposable graphs. Theoretical Computer Science 109(1-2), 49–82 (1993)
10. Downey, R.G., Fellows, M.R.: Parameterized Complexity. Springer, New York (1999)
11. Downey, R.G., Fellows, M.R., McCartin, C., Rosamond, F.: Parameterized approximation of dominating set problems. Information Processing Letters 109(1), 68–70 (2008)
12. Feige, U.: A threshold of ln n for approximating set cover. Journal of the ACM 45(4), 634–652 (1998)
13. Feige, U., Hajiaghayi, M., Lee, J.R.: Improved approximation algorithms for minimum-weight vertex separators. SIAM Journal on Computing 38(2), 629–657 (2008)
14. Flum, J., Grohe, M.: Parameterized Complexity Theory. Texts in Theoretical Computer Science. An EATCS Series. Springer, Berlin (2006)
15. Fomin, F.V., Golovach, P., Thilikos, D.M.: Contraction Bidimensionality: The Accurate Picture. In: Fiat, A., Sanders, P. (eds.) ESA 2009. LNCS, vol. 5757, pp. 706–717. Springer, Heidelberg (2009)
16. Fomin, F.V., Thilikos, D.M.: Dominating sets in planar graphs: Branch-width and exponential speed-up. SIAM Journal on Computing 36(2), 281–309 (2006)
17. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP–Completeness. Freeman, San Francisco (1979)
18. Garey, M.R., Johnson, D.S., Stockmeyer, L.: Some simplified *NP*-complete graph problems. Theoretical Computer Science 1, 237–267 (1976)
19. Gavril, F.: Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. SIAM Journal on Computing 1(2), 180–187 (1972)
20. Gera, R.: On dominator coloring in graphs. In: Graph Theory Notes of New York, pp. 25–30. LII (2007)
21. Gera, R.: On the dominator colorings in bipartite graphs. In: ITNG, pp. 1–6. IEEE (2007)
22. Gera, R., Rasmussen, C., Horton, S.: Dominator colorings and safe clique partitions. Congressus Numerantium 181(7-9), 19–32 (2006)
23. Halldórsson, M.M.: A still better performance guarantee for approximate graph coloring. Information Processing Letters 45(1), 19–23 (1993)
24. Hedetniemi, S., Hedetniemi, S., McRae, A., Blair, J.: Dominator colorings of graphs (2006) (preprint)
25. Johnson, D.S.: Approximation algorithms for combinatorial problems. Journal of Computer and System Sciences 9(3), 256–278 (1974)
26. Lovász, L.: On the ratio of optimal integral and fractional covers. Discrete Mathematics 13, 383–390 (1975)
27. Lund, C., Yannakakis, M.: On the hardness of approximating minimization problems. Journal of the ACM 41(5), 960–981 (1994)

Parameterized Longest Previous Factor*

Richard Beal and Donald Adjeroh

West Virginia University,
Lane Department of Computer Science and Electrical Engineering,
Morgantown, WV 26506
r.beal@computer.org, don@csee.wvu.edu

Abstract. The longest previous factor (LPF) problem is defined for traditional strings exclusively from the constant alphabet Σ. A parameterized string (p-string) is a sophisticated string composed of symbols from a constant alphabet Σ and a parameter alphabet Π. We generalize the LPF problem to the parameterized longest previous factor (pLPF) problem defined for p-strings. Subsequently, we present a linear time solution to construct the $pLPF$ array. Given our pLPF algorithm, we show how to construct the $pLCP$ (parameterized longest common prefix) array in linear time. Our algorithm is further exploited to construct the standard LPF and LCP arrays all in linear time.

Keywords: parameterized suffix array, parameterized longest common prefix, p-string, p-match, LPF, LCP.

1 Introduction

Given an n-length traditional string $W = W[1]W[2]...W[n]$ from the alphabet Σ, the longest previous factor (LPF) problem is to determine the maximum length of a previously occurring factor for each suffix occurring in W. More formally, for any suffix u beginning at index i in the string W, the LPF problem is to identify the length of the longest factor between u and another suffix v at some position h before i in W: that is, $1 \leq h < i$. The LPF problem, introduced by Crochemore and Ilie [1], yields a data structure convenient for fundamental applications such as string compression [2] and detecting runs [3] within a string. In order to compute the LPF array, it is shown in [1] that the suffix array SA is useful to quickly identify the most lexicographically similar suffixes that constitute as previous factors for the chosen suffix in question. The use of SA expedites the work required to solve the LPF problem and likewise, is the *cornerstone* to solutions for many problems defined for traditional strings.

A generalization of traditional strings over an alphabet Σ is the parameterized string (p-string), introduced by Baker [4]. A p-string is a production of symbols from the alphabets Σ and Π, which represent the constant symbols and parameter symbols respectively. The parameterized pattern matching

* This work was partly supported by a grant from the National Historical Publications & Records Commission.

C.S. Iliopoulos and W.F. Smyth (Eds.): IWOCA 2011, LNCS 7056, pp. 31–43, 2011.
© Springer-Verlag Berlin Heidelberg 2011

(p-match) problem is to identify an equivalence between a pair of p-strings S and T when 1) the individual constant symbols match and 2) there exists a bijection between the parameter symbols of S and T. For example, the following p-strings that represent program statements $z{=}y * f/{+}{+}$y; and $a{=}b * f/{+}{+}b$; over the alphabets $\Sigma = \{*, /, +, =, ;\}$ and $\Pi = \{a, b, f, y, z\}$ satisfy both conditions and thus, the p-strings p-match. The motivation for addressing a problem in terms of p-strings is the range of problems that a single solution can address, including 1) exact pattern matching when $|\Pi| = 0$, 2) mapped matching (m-matching) when $|\Sigma| = 0$ [5], and clearly, 3) p-matching when $|\Sigma| > 0 \wedge |\Pi| > 0$. Prominent applications concerned with the p-match problem include detecting plagiarism in academia and industry, reporting similarities in biological sequences [6], discovering cloned code segments in a program [7], and even answering critical legal questions regarding the unauthorized use of intellectual property [8].

In this work, we introduce the parameterized longest previous factor (pLPF) for p-strings analogous to the LPF problem for traditional strings, which can similarly be used to study compression and duplication within p-strings. Given an n-length p-string $T = T[1]T[2]...T[n]$, the pLPF problem is to determine the longest parameterized suffix (p-suffix) v at position h for a p-suffix starting at i in T with $1 \leq h < i$. Our approach uses a parameterized suffix array (pSA) [9,10,11,12] for p-strings analogous to the traditional suffix array [13]. The major difficulty of the pLPF problem is that unlike traditional suffixes of a string, the p-suffixes are dynamic, varying with the starting position of the p-suffix. Thus, traditional LPF solutions cannot be directly applied to the pLPF problem.

Main Contributions: We generalize the LPF problem for traditional strings to the parameterized longest previous factor (pLPF) problem defined for p-strings. Then, we present a linear time algorithm for constructing the $pLPF$ data structure. Traditionally, the LPF problem is solved by using the longest common prefix (LCP) array. This was one approach used in [1]. In this work, we show how to go in the reverse direction: that is, given the pLPF solution, we now construct the $pLCP$ array. Further, we identify how to exploit our algorithm for the pLPF problem to construct the LPF and LCP arrays. Our main results are stated in the following theorems:

Theorem 1. *Given an n-length p-string T, $prevT =$* `prev`(T)*, the* `prev` *encoding of T, and pSA, the parameterized suffix array for T, the algorithm* `compute_pLPF` *constructs the $pLPF$ array in $O(n)$ time.*

Theorem 2. *Given an n-length p-string T, $prevT =$* `prev`(T)*, the* `prev` *encoding of T, and pSA, the parameterized suffix array for T, the* `compute_pLPF` *algorithm can be used to construct the $pLCP$ array in $O(n)$ time.*

2 Background / Related Work

Baker [7] identifies three types of pattern matching: 1) exact matching, 2) parameterized matching (p-match), and 3) matching with modifications. The first p-match breakthroughs, namely, the `prev` encoding and the parameterized suffix

tree (p-suffix tree) that demands the worst case construction time of $O(n(|\Pi| + \log(|\Pi| + |\Sigma|)))$, were introduced by Baker [4]. Like the traditional suffix tree [14,15,16], the p-suffix tree [4] implementation suffers from a large memory footprint. Other solutions that address the p-match problem without the space limitations of the p-suffix tree include the parameterized-KMP [5] and parameterized-BM [17], variants of traditional pattern matching approaches. Idury et al. [18] studied the multiple p-match problem using automata. The parameterized suffix array (p-suffix array) and the parameterized longest common prefix ($pLCP$) array combination is analogous to the suffix array and LCP array for traditional strings [13,14,15,16], which is both time and space efficient for pattern matching. Direct p-suffix array and $pLCP$ construction was first introduced by Deguchi et al. [10] for binary strings with $|\Pi| = 2$, which required $O(n)$ work. Deguchi and colleagues [9] later proposed the first approach to p-suffix sorting and $pLCP$ construction with an arbitrary alphabet size requiring $O(n^2)$ time in the worst case. We introduce new algorithms in [11,12] to p-suffix sort in linear time on average using coding methods from information theory.

Table 1. LPF calculation for string $W = AAABABAB\$$

i	$SA[i]$	$W[SA[i]...n]$	$LCP[i]$	$W[i...n]$	$LPF[i]$
1	9	\$	0	$AAABABAB\$$	0
2	1	$AAABABAB\$$	0	$AABABAB\$$	2
3	2	$AABABAB\$$	2	$ABABAB\$$	1
4	7	$AB\$$	1	$BABAB\$$	0
5	5	$ABAB\$$	2	$ABAB\$$	4
6	3	$ABABAB\$$	4	$BAB\$$	3
7	8	$B\$$	0	$AB\$$	2
8	6	$BAB\$$	1	$B\$$	1
9	4	$BABAB\$$	3	\$	0

In a novel application of the suffix array and the corresponding LCP array, Crochemore and Ilie [1] introduced the longest previous factor (LPF) problem for traditional strings. Table 1 shows an example LPF for a short sequence $W = AAABABAB\$$. For any suffix u beginning at index i in string W, the LPF problem is to identify the exact matching longest factor between u and another suffix v starting prior to index i in W. We note that this definition is similar to (though not the same as) the $Prior$ array used in [14]. Crochemore and Ilie [1] exploited the notion that the nearby elements within a suffix array are closely related en route to proposing a linear time solution to the LPF problem. They also proposed another linear time algorithm to compute the LPF array by using the LCP structure. The significance of an efficient solution to the LPF is that the resulting data structure simplifies computations in various string analysis procedures. Typical examples include computing the Lempel-Ziv factorization [2,19], which is fundamental in string compression algorithms such as the UNIX *gzip* utility [14,15] and in algorithms for detecting repeats in a

string [3]. Our motivation to study the LPF in terms of p-strings is the power of parameterization with relevance to various important applications.

3 Preliminaries

A string on an alphabet Σ is a production $T = T[1]T[2]...T[n]$ from Σ^n with $n = |T|$ the length of T. We will use the following string notations: $T[i]$ refers to the i^{th} symbol of string T, $T[i...j]$ refers to the substring $T[i]T[i+1]...T[j]$, and $T[i...n]$ refers to the i^{th} suffix of T: $T[i]T[i+1]...T[n]$. Parameterized pattern matching requires the finite alphabets Σ and Π. Alphabet Σ denotes the set of constant symbols while Π represents the set of parameter symbols. Alphabets are defined such that $\Sigma \cap \Pi = \emptyset$. Furthermore, we append the terminal symbol $\$ \notin \Sigma \cup \Pi$ to the end of all strings to clearly distinguish between suffixes. For practical purposes, we can assume that $|\Sigma| + |\Pi| \leq n$ since otherwise a single mapping can be used to enforce the condition.

Definition 1. Parameterized String (p-string): *A p-string is a production T of length n from $(\Sigma \cup \Pi)^*\$$.*

Consider the alphabet arrangements $\Sigma = \{A, B\}$ and $\Pi = \{w, x, y, z\}$. Example p-strings include $S = AxByABxy\$$, $T = AwBzABwz\$$, and $U = AyByAByy\$$.

Definition 2. ([4,10]) Parameterized Matching (p-match): *A pair of p-strings S and T are p-matches with $n = |S|$ if and only if $|S| = |T|$ and each $1 \leq i \leq n$ corresponds to one of the following:*

1. $S[i], T[i] \in (\Sigma \cup \{\$\}) \wedge S[i] = T[i]$
2. $S[i], T[i] \in \Pi \wedge ((a) \vee (b))$ */* parameter bijection */*
   (a) $S[i] \neq S[j], T[i] \neq T[j]$ **for any** $1 \leq j < i$
   (b) $S[i] = S[i-q]$ **iff** $T[i] = T[i-q]$ **for any** $1 \leq q < i$

In our example, we have a p-match between the p-strings S and T since every constant/terminal symbol matches and there exists a bijection of parameter symbols between S and T. U does not satisfy the parameter bijection to p-match with S or T. The process of p-matching leads to defining the `prev` encoding.

Definition 3. ([4,10]) Previous (`prev`) Encoding: *Given $\mathbb{Z}$ as the set of non-negative integers, the function* `prev` *: $(\Sigma \cup \Pi)^*\$ \rightarrow (\Sigma \cup \mathbb{Z})^*\$$ accepts a p-string T of length n and produces a string Q of length n that* 1) *encodes constant/terminal symbols with the same symbol and* 2) *encodes parameters to point to* **previous** *like-parameters. More formally, Q is constructed of individual $Q[i]$ with $1 \leq i \leq n$ where:*

$$Q[i] = \begin{cases} T[i], \textbf{ if } T[i] \in (\Sigma \cup \{\$\}) \\ 0, \textbf{ if } T[i] \in \Pi \wedge T[i] \neq T[j] \textbf{ for any } 1 \leq j < i \\ i - k, \textbf{ if } T[i] \in \Pi \wedge k = \max\{j | T[i] = T[j], 1 \leq j < i\} \end{cases}$$

For a p-string T of length n, the above $O(n)$ space $\texttt{prev}$ encoding requires the construction time of order $O(n \log(\min\{n, |\Pi|\}))$, which follows from the discussions of Baker [4,17] and Amir et al. [5] on the dependency of alphabet Π in p-match applications. Given an indexed alphabet and an auxiliary $O(|\Pi|)$ mapping structure, we can construct $\texttt{prev}$ in $O(n)$ time. Using Definition 3, our working examples evaluate to $\texttt{prev}(S) = A0B0AB54\$, \texttt{prev}(T) = A0B0AB54\$$, $\texttt{prev}(U) = A0B2AB31\$$. The relationship between p-strings and the lexicographical ordering of the $\texttt{prev}$ encoding is fundamental to the p-match problem.

Definition 4. $\texttt{prev}$ **Lexicographical Ordering:** *Given the p-strings S and T and two symbols s and t from the encodings $\texttt{prev}(S)$ and $\texttt{prev}(T)$ respectively, the relationships $=$, $\neq$, $<$, and $>$ refer to lexicographical ordering between s and t. We define the ordering of symbols from a $\texttt{prev}$ encoding of the production $(\Sigma \cup \mathbb{Z})^*\$$ to be $\$ < \zeta \in \mathbb{Z} < \sigma \in \Sigma$, where each ζ and σ is lexicographically sorted in their respective alphabets. The relationships $=$, $\neq$, $\prec$, and $\succ$ refer to the lexicographical ordering between strings. In the case of $\texttt{prev}(S)$ and $\texttt{prev}(T)$, $\texttt{prev}(S) \prec \texttt{prev}(T)$ when $\texttt{prev}(S)[1] = \texttt{prev}(T)[1], \texttt{prev}(S)[2] = \texttt{prev}(T)[2], ..., \texttt{prev}(S)[j-1] = \texttt{prev}(T)[j-1], \texttt{prev}(S)[j] < \texttt{prev}(T)[j]$ for some j, $j \geq 1$. Similarly, we can define $=_k$, $\neq_k$, $\prec_k$, and $\succ_k$ to refer to the lexicographical relationships between a pair of p-strings considering only the first $k \geq 0$ symbols.*

It is shown in [11,12] how to map a symbol in $\texttt{prev}$ to an integer based on the ordering of Definition 4 and subsequently, call the function $\texttt{in}(x, X)$ to answer alphabet membership questions of the form $x \in X$ in constant time. The following proposition essential to the p-matching problem is directly related to the established symbol ordering.

Proposition 1. ([4]) *Two p-strings S and T p-match when $\texttt{prev}(S) = \texttt{prev}(T)$. Also, $S \prec T$ when $\texttt{prev}(S) \prec \texttt{prev}(T)$ and $S \succ T$ when $\texttt{prev}(S) \succ \texttt{prev}(T)$.*

The example $\texttt{prev}$ encodings show a p-match between S and T since $\texttt{prev}(S) = A0B0AB54\$$ and $\texttt{prev}(T) = A0B0AB54\$$. Also, $U \succ S$ and $U \succ T$ since $\texttt{prev}(U) = A0B2AB31\$ \succ \texttt{prev}(S) = \texttt{prev}(T) = A0B0AB54\$$. We use the ordering established in Definition 4 to define the parameterized suffix array and the parameterized longest common prefix array.

Definition 5. Parameterized Suffix Array (pSA): *The pSA for a p-string T of length n maintains a lexicographical ordering of the indices i representing individual p-suffixes $\texttt{prev}(T[i...n])$ with $1 \leq i \leq n$, such that $\texttt{prev}(T[pSA[q]...n]) \prec \texttt{prev}(T[pSA[q+1]...n]) \forall q, 1 \leq q < n$.*

Definition 6. Parameterized Longest Common Prefix ($pLCP$) Array: *The $pLCP$ array for a p-string T of length n maintains the length of the longest common prefix between neighboring p-suffixes. We define $\texttt{plcp}(\alpha, \beta) = \max\{k | \texttt{prev}(\alpha) =_k \texttt{prev}(\beta)\}$. Then, $pLCP[1] = 0$ and $pLCP[i] = \max\{k \mid \texttt{plcp}(T[pSA[i]...n], T[pSA[i-1]...n])\}$, $2 \leq i \leq n$.*

For the example $T = AwBzABwz\$$ with $\texttt{prev}(T) = A0B0AB54\$$, we have $pSA = \{9, 8, 7, 4, 2, 1, 5, 6, 3\}$ and $pLCP = \{0, 0, 1, 1, 1, 0, 1, 0, 2\}$. The encoding `prev` is supplemented by the encoding `forw`.

Definition 7. ([11,12]) Forward (`forw`) Encoding: *Let the function* $\texttt{rev}(T)$ *reverse the p-string* T *and* $\texttt{repl}(T, x, y)$ *replace all occurrences in* T *of the symbol* x *with* y*. We define the function* `forw` *for the p-string* T *of length* n *as* $\texttt{forw}(T) = \texttt{rev}(\texttt{repl}(\texttt{prev}(\texttt{rev}(T)), 0, n))$.

For a p-string T of length n, the encoding `forw` 1) encodes constant/terminal symbols with the same symbol and 2) encodes each parameter p with the **forward** distance to the next occurrence of p or an unreachable forward distance n. Our definition of `forw` generates output mirroring the `fw` encoding used by Deguchi et al. [9,10]. The `forw` encodings in our example with $n = 9$ are $\texttt{forw}(S) = A5B4AB99\$$, $\texttt{forw}(T) = A5B4AB99\$$, $\texttt{forw}(U) = A2B3AB19\$$.

Definition 8. ([1]) Longest Previous Factor (LPF): *For an* n*-length traditional string* W*, the LPF is defined for each index* $1 \leq i \leq n$ *such that* $LPF[i] = \max(\{0\} \cup \{k \mid W[i...n] =_k W[h...n], 1 \leq h < i\})$.

The traditional string $W = AAABABAB\$$ yields $LPF = \{0, 2, 1, 0, 4, 3, 2, 1, 0\}$.

4 Parameterized LPF

We define the parameterized longest previous factor (pLPF) problem as follows to generalize the traditional LPF problem to p-strings.

Definition 9. Parameterized Longest Previous Factor ($pLPF$): *For a p-string* T *of length* n*, the pLPF array is defined for each index* $1 \leq i \leq n$ *to maintain the length of the* longest factor *between a p-suffix and a previous p-suffix occurring in* T*. More formally,* $pLPF[i] = \max(\{0\} \cup \{k \mid \texttt{prev}(T[i...n]) =_k \texttt{prev}(T[h...n]), 1 \leq h < i\})$.

The pLPF problem requires that we deal with p-suffixes, which are suffixes encoded with `prev`. This task is more demanding than the LPF for traditional strings because Lemma 1 indicates that we cannot guarantee the individual suffixes of a single `prev` encoding to be p-suffixes. Thus, the changing nature of the `prev` encoding poses a major challenge to efficient and correct construction of the $pLPF$ array using current algorithms that construct the LPF array for traditional strings. The proof is provided in [12] and omitted for space.

Lemma 1. *Given a p-string* T *of length* n*, the suffixes of* $\texttt{prev}(T)$ *are not necessarily the p-suffixes of* T*. More formally, if* $\pi \in \Pi$ *occurs more than once in* T*, then* $\exists i$*, s.t.* $\texttt{prev}(T[i...n]) \neq \texttt{prev}(T)[i...n], 1 \leq i \leq n$.

Consider the p-string $T = AAAwBxyyAAAzwwB\$$ using the previously defined alphabets. Table 2 shows the pLPF computation for the p-string T. We note the intricacies of Lemma 1 since simply using the traditional LPF algorithm 1)

Table 2. pLPF calculation for p-string $T = AAAwBxyyAAAzwwB\$$

i	$pSA[i]$	$pLCP[i]$	$\texttt{prev}(T[pSA[i]...n])$	$before_<[pSA[i]]$	$before_>[pSA[i]]$	$pLPF[i]$
1	16	0	\$	-1	6	0
2	6	0	$001AAA001B\$$	-1	4	2
3	12	3	$001B\$$	6	7	1
4	7	1	$01AAA001B\$$	6	4	0
5	13	2	$01B\$$	7	8	0
6	8	1	$0AAA001B\$$	7	4	1
7	14	1	$0B\$$	8	4	1
8	4	2	$0B001AAA091B\$$	-1	3	1
9	11	0	$A001B\$$	4	3	4
10	3	2	$A0B001AAA091B\$$	-1	2	3
11	10	1	$AA001B\$$	3	2	2
12	2	3	$AA0B001AAA091B\$$	-1	1	3
13	9	2	$AAA001B\$$	2	1	2
14	1	4	$AAA0B001AAA091B\$$	-1	-1	2
15	15	0	$B\$$	1	5	1
16	5	1	$B001AAA001B\$$	1	-1	0

with T yields $LPF = \{0, 2, 1, 0, 0, 0, 0, 1, 3, 2, 1, 0, 1, 2, 1, 0\}, 2)$ with $\texttt{prev}(T)$ produces $LPF = \{0, 2, 1, 0, 0, 1, 1, 0, 4, 3, 2, 1, 0, 1, 1, 0\}$, and 3) with $\texttt{forw}(T)$ generates $LPF = \{0, 2, 1, 0, 0, 0, 0, 1, 3, 2, 1, 3, 2, 1, 1, 0\}$, neither of which is the correct $pLPF$ array.

Crochemore and Ilie [1] efficiently solve the LPF problem for a traditional string W by exploiting the properties of the suffix array SA. They construct the arrays $prev_<[1...n]$ and $prev_>[1...n]$, which for each i in W maintain the suffix $h < i$ positioned respectively before and after suffix i in SA; when no such suffix exists, the element is denoted by -1. The conceptual idea to compute the $prev_<$ and $prev_>$ arrays in linear time via deletions in a doubly linked list of the SA was suggested in [1]. The algorithm is given in [12]. Furthermore, we will refer to $prev_<$ and $prev_>$ as $before_<$ and $before_>$ respectively, in order to avoid confusion with the `prev` encoding for p-strings. Then, $LPF[i]$ is the maximum q between $W[i...n] =_q W[before_<[i]...n]$ and $W[i...n] =_q W[before_>[i]...n]$. The magic of a linear time solution to constructing the LPF array is achieved through the computation of an element by *extending* the previous element, more formally $LPF[i] \geq LPF[i-1] - 1$, which is a variant of the extension property used in LCP construction proven by Kasai et al. [20]. We prove that this same property holds for the pLPF problem defined on p-strings.

Lemma 2. *The $pLPF$ for a p-string T of length n is such that $pLPF[i] \geq pLPF[i-1] - 1$ with $1 < i \leq n$.*

Proof. Consider $pLPF[i]$ at $i = 1$ by which Definition 9 requires that we find a previous factor at $1 \leq h < 1$ that does not exist; i.e., $pLPF[1] = 0$. At $i = 2$, indeed $pLPF[2] \geq pLPF[1] - 1 = -1$ is clearly true for all succeeding elements

Algorithm 1. pLPF computation

```
1  int[] compute_pLPF(int before<[], int before>[]) {
2     int  pLPF[n], pLPF<=0, pLPF>=0, i, j, k
3     for i = 1 to n {
4        j = max{0,pLPF<-1}
5        k = max{0,pLPF>-1}
6        if(before< ≠ null) pLPF< = Λ(i,before<[i],j)
7        if(before> ≠ null) pLPF> = Λ(i,before>[i],k)
8        pLPF[i] = max{pLPF<,pLPF>}
9     }return pLPF
10 }
```

Algorithm 2. p-matcher function Λ

```
1  int Λ(int a, int b, int q) {
2     boolean c = true
3     int x, y
4     if(b = −1) return 0
5     while(c ∧ (a+q) ≤ n ∧ (b+q) ≤ n) {
6        x = prevT[a+q], y = prevT[b+q]
7        if(in(x,Σ) ∧ in(y,Σ)){
8           if(x = y) q++
9           else c = false
10       }else if(in(x,ℤ) ∧ in(y,ℤ)){
11          if(q < x) x = 0
12          if(q < y) y = 0
13          if(x = y) q++
14          else c = false
15       }else c = false
16    }return q
17 }
```

in which a previous factor does not exist. For arbitrary $i = j$ with $1 < j < n$, suppose that the maximum length factor is at $g < j$ and without loss of generality, consider that the first $q \geq 2$ symbols match so that $\texttt{prev}(T[j...n]) =_q \texttt{prev}(T[g...n])$. Thus, $pLPF[j] = q$. Shifting the computation to $i = j+1$, we lose the symbols $\texttt{prev}(T[j])$ and $\texttt{prev}(T[g])$ in the p-suffixes at j and g respectively. By Proposition 1, $\texttt{prev}(T[j...j+q-1]) = \texttt{prev}(T[g...g+q-1]) \Rightarrow \texttt{prev}(T[j]) = \texttt{prev}(T[g])$ and as a consequence of the `prev` encoding in Definition 3 we have $\texttt{prev}(T[i...n]) =_{q-1} \texttt{prev}(T[g+1...n])$. Since we can guarantee that $\exists$ a factor with $(q-1)$ symbols for $pLPF[i]$ or possibly find another factor at h with $1 \leq h < i$ matching q or more symbols, the lemma holds. □

Lemma 2 permits us to adapt the algorithm `compute_LPF` given in [1] to p-strings. We introduce `compute_pLPF` in Algorithm 1 to construct the $pLPF$ array, which makes use of the p-matcher Λ in Algorithm 2 to properly handle the sophisticated

matching of p-suffixes, the dynamic suffixes under the `prev` encoding. The role of Λ is to *extend* the matches between the p-suffixes at a and b beyond the initial q symbols by directly comparing constant/terminal symbols and comparing the dynamically adjusted parameter encodings for each p-suffix.

Theorem 1. *Given an n-length p-string T, $prevT = \texttt{prev}(T)$, the* `prev` *encoding of T, and pSA, the parameterized suffix array for T, the algorithm* `compute_pLPF` *constructs the $pLPF$ array in $O(n)$ time.*

Proof. It follows from Lemma 2 that our algorithm exploits the properties of pLPF to correctly compute and extend factors, which requires $O(n)$ time. Computing the arrays $before_<$ and $before_>$ require $O(n)$ processing [12]. What remains now is to show that, between Algorithm 1 and Algorithm 2, the total number of times that the body of the **while** loop (lines 6-15 in Algorithm 2) will be executed is in $O(n)$. The number of iterations of the **while** loop is given by the number of matching symbol comparisons, namely the number of increments of the variable q, which identifies the shift required to compare the current symbol. Without loss of generality, suppose that the initial p-suffixes at position a and b are the longest suffixes at positions 1 and 2 in T of lengths n and $(n-1)$ respectively. In the worst case, $(n-1)$ of the symbols will match between these suffixes, by which each comparison that clearly requires $O(1)$ work, will increment q. Lemma 2 indicates that succeeding calculations, or calls to Λ, already match at least $(q-1)$ symbols that are not rematched and rather, the match is *extended*. Since the decreasing lengths of the succeeding suffixes at $3, 4, ..., n$ cannot *extend* the current q, no further matching or increments of q are needed. Hence, the **while** loop iterates a total of $O(n)$ times amortized across *all* of the n iterations of the **for** loop in Algorithm 1. Thus, the total work is $O(n)$. □

Our algorithm `compute_pLPF` is motivated by the `compute_LPF` algorithm in [1]. We also observe that similar pattern matching mechanisms as the one used between the **for** loop in Algorithm 1 and the **while** loop in Algorithm 2 exist in standard string processing, for example in computing the *border* array discussed in [15].

5 From pLPF to pLCP

Deguchi et al. [9,10] studied the problem of constructing the $pLCP$ array given the pSA. They showed that constructing the $pLCP$ array requires a non-trivial modification of the traditional LCP construction by Kasai et al. [20]. In [1], the LCP array was used as the basis for constructing the LPF array for traditional strings. Here, we present a simpler algorithm for constructing the $pLCP$ array. In particular, we show that, unlike in [1], it is possible to go the other way around: that is, given the pLPF solution, we now construct the $pLCP$ array. Later, we show that the same pLPF algorithm can be used to construct the LCP array and the LPF array for traditional strings. Crochemore and Ilie [1] identify that the traditional LPF array is a permutation of the well-studied LCP array. We observe the same relationship in terms of the $pLPF$ and $pLCP$ arrays.

Proposition 2. *The pLPF array is a permutation of pLCP.*

This observation allows us to view the $pLCP$ array from a different perspective. As a novel use of our `compute_pLPF` algorithm, we introduce a way to construct the $pLCP$ array in linear time. The key observation is that we can integrate the fact that the $pLCP$ occurs between neighboring p-suffixes and the fact that we preprocess the $before_<$ array, which for each i in the p-string T maintains the p-suffix $h < i$ positioned prior to the p-suffix i in pSA. We can also construct the array $after_<$ to maintain the p-suffix $j > i$ also positioned prior to the p-suffix i in pSA. Since h and j are both positioned prior to i in pSA, we can guarantee that either h or j must be the nearest neighbor to i. So, the maximum factor determines the nearest neighbor and thus, $pLCP[R[i]]$, where R is the inverse of pSA (see Algorithm 3). Theorem 2 shows that this computation is performed in linear time.

Algorithm 3. pLCP computation

```
1  int[] compute_pLCP(int before_<[], int after_<[]) {
2      int pLCP[n], M[n], R[n], i
3      for i = 1 to n
4          R[pSA[i]] = i
5      M = compute_pLPF(before_<, after_<)
6      for i = 1 to n
7          pLCP[R[i]] = M[i]
8      return pLCP
9  }
```

Theorem 2. *Given an n-length p-string T, $prevT$ = `prev`(T), the* `prev` *encoding of T, and pSA, the parameterized suffix array for T, the* `compute_pLPF` *algorithm can be used to construct the $pLCP$ array in $O(n)$ time.*

Proof. We can clearly relax the p-suffix selection restrictions enforced by the problem pLPF in Lemma 2 to exploit the idea of *extending* factors. Subsequently, only the parameters of Algorithms 1 and 2 impose such restrictions. Let $R[1...n]$ be the rank array, the inverse of pSA. We prove that the $pLCP$ is constructed with `compute_pLPF`$(before_<, after_<)$. Let $before_<[1...n]$ and $after_<[1...n]$ maintain, for all the i in T, the p-suffixes $h < i$ at position $R[h]$ in pSA and $j > i$ at position $R[j]$ in pSA, respectively, that are positioned prior to the p-suffix i at position $R[i]$ in pSA; when no such suffix exists, the element is denoted by -1. Without loss of generality, suppose that both h and j exist and $2 < i \leq n$, so we have either $R[j] = R[i]-1$ or $R[h] = R[i]-1$ as the neighboring p-suffix. So, $\max\{$`plcp`$($`prev`$(T[h...n]),$ `prev`$(T[i...n])),$ `plcp`$($`prev`$(T[j...n]),$ `prev`$(T[i...n]))\}$ distinguishes which p-suffix h or j is closer to i, identifying the nearest neighbor and in turn, $pLCP[R[i]]$. This statement is utilized in `compute_pLPF` exactly in terms of factors except that the value will be stored in $pLCP[i]$. So, after the computation using the call to `compute_pLPF` (line 5) in Algorithm 3, rearranging the resulting array using the rank array R (lines 6-7) produces the required

$pLCP$ array. We have yet to prove the time complexity. Since the parameter $after_<$ can be computed in $O(n)$ by deletions and indexing into a doubly linked list similar to $before_<$ [12] and since `compute_pLPF` executes in $O(n)$ time via Theorem 1, the theorem holds. □

Algorithm 4. Improved pLCP computation

```
1  int[] compute_pLCP() {
2      int pLCP[n], M[n], i
3      M[pSA[1]] = -1
4      for i = 2 to n
5          M[pSA[i]] = pSA[i-1]
6      M = compute_pLPF(M, null)
7      for i = 1 to n
8          pLCP[i] = M[pSA[i]]
9      return pLCP
10 }
```

For discussion purposes, Algorithm 3 uses a rank array R to index and preprocess the arrays $before_<$ and $after_<$ to determine the neighboring suffix, which can be found trivially with a p-suffix array, and thus, may be omitted for practical space. The improved solution is shown in Algorithm 4. For further improved space consumption, the implementation of Algorithm 4 may incorporate the LCP indexing contributions of [21]. In passing, we identify that upon the completion of line 6 in Algorithm 4, the M array is the permuted longest common prefix ($PLCP$) data structure observed in [22] for traditional strings.

6 From pLPF to LPF and LCP

The power of defining the pLPF problem in terms of p-strings is the generalization of a p-string production. We show in Theorems 3 and 4 that our `compute_pLPF` algorithm also computes the traditional LPF and LCP arrays.

Theorem 3. *Given an n-length traditional string W, the* `compute_pLPF` *algorithm constructs the LPF array in $O(n)$ time.*

Proof. Since $W[i] \in \Sigma \; \forall \; i, 1 \leq i < n$ and $W[n] \in \{\$\}$, then by Definition 1 we have $W \in (\Sigma \cup \Pi)^*\$$, which classifies W as a valid p-string. Given this, Theorem 1 proves that the construction of $pLPF$ for a p-string requires $O(n)$ time. In this special case, W consists of no such symbol $\pi \in \Pi$ so Lemma 1 identifies that $\texttt{prev}(W[i...n]) = \texttt{prev}(W)[i...n]$ and further $W = \texttt{prev}(W)$ by Definition 3, so $W[i...n] = \texttt{prev}(W)[i...n]$, which constrains the pLPF in Definition 9 to the LPF problem in Definition 8. Thus, from Theorem 1, `compute_pLPF` computes the LPF of W. □

Theorem 4. *Given an n-length traditional string W, the* `compute_pLCP` *algorithm constructs the LCP array in $O(n)$ time.*

Proof. In the same manner as Theorem 3, we may classify W as a valid p-string. Given this, Theorem 2 proves that the construction of $pLCP$ for a p-string requires $O(n)$ time. Mirroring the proof of Theorem 3, we have $W[i...n] = \texttt{prev}(W)[i...n]$, which constrains the pLCP in Definition 6 to the traditional LCP problem. Thus, from Theorem 2, `compute_pLCP` computes the LCP of W. □

7 Conclusion and Discussion

We introduce the parameterized longest previous factor (pLPF) problem for p-strings, which is analogous to the longest previous factor (LPF) problem defined for traditional strings. A linear time algorithm is provided to construct the $pLPF$ array for a given p-string. The advantage of implementing our solution `compute_pLPF` is that the algorithm may be used to compute the arrays $pLPF$, $pLCP$, LPF, LCP, or even the permuted LCP [22] in linear time, which are fundamental data structures preprocessed for the efficiency of countless pattern matching applications. Each of the proposed algorithms requires $O(n)$ worst case time and $O(n)$ worst case space. Since we provide construction algorithms for several data structures using the $pLPF$ construction as the groundwork, we are faced with the practical limitation that our algorithms are only as efficient as the `compute_pLPF` solution. We acknowledge that it is possible to use the techniques in [22,23,24] to improve the space consumption of the LCP array and similarly, the $pLCP$ data structure, since $pLCP$ is an array of integers analogous to the traditional LCP. Nonetheless, the significance of working though the LPF as an intermediate data structure is the straightforward and space efficient algorithm to construct the Lempel-Ziv (LZ) factorization [1,2,19]. Similarly, the $pLPF$ array can easily derive the LZ structure and allow us to study such applications as maximal runs in p-strings extended to source code plagiarism or redundancies in biological sequences.

References

1. Crochemore, M., Ilie, L.: Computing longest previous factor in linear time and applications. Inf. Process. Lett. 106(2), 75–80 (2008)
2. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. Inf. Theory 23(3), 337–343 (1977)
3. Main, M.: Detecting leftmost maximal periodicities. Discrete Appl. Math. 25(1-2), 145–153 (1989)
4. Baker, B.: A theory of parameterized pattern matching: Algorithms and applications. In: STOC 1993, pp. 71–80. ACM, New York (1993)
5. Amir, A., Farach, M., Muthukrishnan, S.: Alphabet dependence in parameterized matching. Inf. Process. Lett. 49, 111–115 (1994)
6. Shibuya, T.: Generalization of a suffix tree for RNA structural pattern matching. Algorithmica 39(1), 1–19 (2004)
7. Baker, B.: Finding clones with dup: Analysis of an experiment. IEEE Trans. Software Eng. 33(9), 608–621 (2007)
8. Zeidman, B.: Software v. software. IEEE Spectr. 47, 32–53 (2010)

9. Tomohiro, I., Deguchi, S., Bannai, H., Inenaga, S., Takeda, M.: Lightweight Parameterized Suffix Array Construction. In: Fiala, J., Kratochvíl, J., Miller, M. (eds.) IWOCA 2009. LNCS, vol. 5874, pp. 312–323. Springer, Heidelberg (2009)
10. Deguchi, S., Higashijima, F., Bannai, H., Inenaga, S., Takeda, M.: Parameterized suffix arrays for binary strings. In: PSC 2008, Czech Republic, pp. 84–94 (2008)
11. Beal, R., Adjeroh, D.: p-Suffix Sorting as Arithmetic Coding. In: Iliopoulos, C.S., Smyth, W.F. (eds.) IWOCA 2011. LNCS, vol. 7056, pp. 44–56. Springer, Heidelberg (2011)
12. Beal, R.: Parameterized Strings: Algorithms and Data Structures. MS Thesis. West Virginia University (2011)
13. Manber, U., Myers, G.: Suffix arrays: A new method for on-line string searches. SIAM J. Comput. 22, 935–948 (1993)
14. Gusfield, D.: Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. Cambridge University Press, Cambridge (1997)
15. Smyth, W.: Computing Patterns in Strings. Pearson, New York (2003)
16. Adjeroh, D., Bell, T., Mukherjee, A.: The Burrows-Wheeler Transform: Data Compression, Suffix Arrays and Pattern Matching. Springer, New York (2008)
17. Baker, B.: Parameterized pattern matching by Boyer-Moore-type algorithms. In: SODA 1995, pp. 541–550. ACM, Philadelphia (1995)
18. Idury, R., Schäffer, A.: Multiple matching of parameterized patterns. Theor. Comput. Sci. 154, 203–224 (1996)
19. Crochemore, M., Ilie, L., Smyth, W.: A simple algorithm for computing the Lempel Ziv factorization. In: DCC 2008, pp. 482–488 (2008)
20. Kasai, T., Lee, G., et al.: Linear-time Longest-common-prefix Computation in Suffix Arrays and its Applications. In: Amir, A., Landau, G.M. (eds.) CPM 2001. LNCS, vol. 2089, pp. 181–192. Springer, Heidelberg (2001)
21. Manzini, G.: Two Space Saving Tricks for Linear Time LCP Array Computation. In: Hagerup, T., Katajainen, J. (eds.) SWAT 2004. LNCS, vol. 3111, pp. 372–383. Springer, Heidelberg (2004)
22. Kärkkäinen, J., Manzini, G., Puglisi, S.: Permuted Longest-common-prefix Array. In: Kucherov, G., Ukkonen, E. (eds.) CPM 2009 Lille. LNCS, vol. 5577, pp. 181–192. Springer, Heidelberg (2009)
23. Puglisi, S., Turpin, A.: Space-time Tradeoffs for Longest-Common-prefix Array Computation. In: Hong, S.-H., Nagamochi, H., Fukunaga, T. (eds.) ISAAC 2008. LNCS, vol. 5369, pp. 124–135. Springer, Heidelberg (2008)
24. Fischer, J.: Wee LCP. Inf. Process. Lett. 110(8-9), 317–320 (2010)

p-Suffix Sorting as Arithmetic Coding*

Richard Beal and Donald Adjeroh

Lane Department of Computer Science and Electrical Engineering,
West Virginia University, Morgantown, WV 26506
r.beal@computer.org, don@csee.wvu.edu

Abstract. The challenge of direct parameterized suffix sorting (p-suffix sorting) for a parameterized string (p-string) is the dynamic nature of parameterized suffixes (p-suffixes). In this work, we propose transformative approaches to direct p-suffix sorting by generating and sorting lexicographically numeric fingerprints and arithmetic codes that correspond to individual p-suffixes. Our algorithm to p-suffix sort via fingerprints is the first theoretical linear time algorithm for p-suffix sorting for non-binary parameter alphabets, which assumes that each code is represented by a practical integer. We eliminate the key problems of fingerprints by introducing an algorithm that exploits the ordering of arithmetic codes to sort p-suffixes in linear time on average.

Keywords: parameterized suffix array, parameterized suffix sorting, arithmetic coding, fingerprints, p-string, p-match.

1 Introduction

Conventional pattern matching typically involves the matching of traditional strings over an alphabet Σ. Parameterized pattern matching using parameterized strings, introduced by Baker [1], attempts to answer pattern matching questions beyond its classical counterpart. A parameterized string (p-string) is a production of symbols from the alphabets Σ and Π, which represent the constant symbols and parameter symbols respectively. Given a pair of p-strings S and T, the parameterized pattern matching (p-match) problem is to verify whether the individual constant symbols match and whether there exists a bijection between the parameter symbols of S and T. If the two conditions are met, S is said to be a p-match of T. For example, there exists a p-match between the p-strings $z{=}y * f/{+}{+}\mathrm{y}$; and $a{=}b * f/{+}{+}b$; that represent program statements over the alphabets $\Sigma = \{*, /, +, =, ;\}$ and $\Pi = \{a, b, f, y, z\}$. Applications inherent to the p-matching problem include detecting plagiarism in academia and industry, reporting similarities in biological sequences [2], discovering cloned code segments in a program to assist with software maintenance [1], and answering critical legal questions regarding the unauthorized use of intellectual property [3].

* This work was partly supported by a grant from the National Historical Publications & Records Commission.

C.S. Iliopoulos and W.F. Smyth (Eds.): IWOCA 2011, LNCS 7056, pp. 44–56, 2011.
© Springer-Verlag Berlin Heidelberg 2011

Initial solutions to the p-match problem were based on the parameterized suffix tree (p-suffix tree) [1]. Idury et al. [4] studied the multiple p-match problem using automata. The physical space requirements of the p-suffix tree led to algorithms such as parameterized-KMP [5], parameterized-BM [6], and the parameterized suffix array (p-suffix array) [7,8]. Analogous to standard suffix sorting, the problem of parameterized suffix sorting (p-suffix sorting) is to sort all the n parameterized suffixes (p-suffixes) of an n-length p-string into a lexicographic order. The major difficulty is that unlike traditional suffixes of a string, the p-suffixes are dynamic, varying with the starting position of the p-suffix. Thus, standard suffix sorting approaches cannot be directly applied to the p-suffix sorting problem. Current approaches to directly construct the p-suffix array *without* a p-suffix tree for an n-length p-string from an arbitrary alphabet require $O(n^2)$ time in the worst case [7]. Such demands the need for alternative approaches to direct p-suffix sorting.

Main Contribution: We construct p-suffix arrays by generating and sorting codes that represent the individual p-suffixes of a p-string. We propose the first theoretical linear time claims to directly p-suffix sort p-strings from non-binary parameter alphabets. We state our main result in the following theorem:

Theorem 4. *Given a p-string T of length n, p-suffix-sorting of T can be accomplished in $O(n)$ time on average via parameterized arithmetic coding.*

2 Background / Related Work

Baker [1] defines pattern matching as either: 1) exact matching, 2) parameterized-matching, or 3) matching with modifications. A parameterized match (p-match) is a sophisticated matching scheme based on the composition of a parameterized string (p-string). A p-string is composed of symbols from a constant symbol alphabet Σ and a parameter alphabet Π. A pair of p-strings S and T of length n are said to p-match when the constant symbols $\sigma \in \Sigma$ match and there exists a bijection of parameter symbols $\pi \in \Pi$ between the pair of p-strings. Baker [1] offered the first p-match breakthroughs, namely, the `prev` encoding to detect a p-match and the parameterized suffix tree (p-suffix tree) analogous to the suffix tree for traditional strings [9,10,11]. The p-suffix tree is built on the `prev` encodings of the suffixes of the p-string, demanding $O(n(|\Pi|+\log(|\Pi|+|\Sigma|)))$ construction time in the worst case [1]. Improvements to the p-suffix tree construction were introduced by Kosaraju [12]. Other contributions in the area of parameterized suffix trees include constructon via randomized algorithms [13,14]. Like the traditional suffix tree [9,10,11], the p-suffix tree [1] implementation suffers from a large memory footprint. Other solutions that address the p-match problem without the space limitations of the p-suffix tree include the parameterized-KMP [5] and parameterized-BM [6], variants of traditional pattern matching approaches.

The native time and space efficiency of the suffix array led to the origination of the parameterized suffix array (p-suffix array). The p-suffix array is analogous to the suffix array for traditional strings introduced in [15]. Manber and Myers [15] show how to combine the suffix array and the LCP (longest common prefix)

array to competitively search for pattern $P = P[1...m]$ in a text $T = T[1...n]$ in $O(m + \log n)$ time. Direct p-suffix array construction was first introduced by Deguchi et al. [8] for binary strings with $|\Pi| = 2$ requiring $O(n)$ construction time through the assistance of a defined `fw` encoding. Deguchi and colleagues [7] later proposed the first approach to *direct* p-suffix sorting with an arbitrary alphabet size requiring $O(n^2)$ time in the worst case, *without* the assistance of a p-suffix tree. The parameterized longest common prefix ($pLCP$) array, analogous to the traditional LCP, was also defined and constructed in [7,8]. In this work, we propose efficient methods to the direct p-suffix sorting problem that avoid the large memory footprint of the p-suffix tree by using fingerprints and coding methods from information theory.

3 Preliminaries

A string on an alphabet Σ is a production $T = T[1]T[2]...T[n]$ from Σ^n with $n = |T|$ the length of T. We will use the following string notations: $T[i]$ refers to the i^{th} symbol of string T, $T[i...j]$ refers to the substring $T[i]T[i+1]...T[j]$, and $T[i...n]$ refers to the i^{th} suffix of T: $T[i]T[i+1]...T[n]$. The area of parameterized pattern matching defines the finite alphabets Σ and Π. Alphabet Σ denotes the set of constant symbols while Π represents the set of parameter symbols. Alphabets are defined such that $\Sigma \cap \Pi = \emptyset$. Furthermore, we append the terminal symbol $\$ \notin \Sigma \cup \Pi$ to the end of all strings to clearly distinguish between suffixes. For practical purposes, we can assume that $|\Sigma| + |\Pi| \leq n$ since, otherwise a single mapping can be used to enforce the condition.

Definition 1. Parameterized String (p-string): *A p-string is a production T of length n from $(\Sigma \cup \Pi)^*\$$.*

Consider the alphabet arrangements $\Sigma = \{A, B\}$ and $\Pi = \{w, x, y, z\}$. Example p-strings include $S = AxByABxy\$$, $T = AwBzABwz\$$, and $U = AyByAByy\$$.

Definition 2. ([1,8]) Parameterized Matching (p-match): *A pair of p-strings S and T are p-matches with $n = |S|$ if and only if $|S| = |T|$ and each $1 \leq i \leq n$ corresponds to one of the following:*

1. $S[i], T[i] \in (\Sigma \cup \{\$\}) \wedge S[i] = T[i]$
2. $S[i], T[i] \in \Pi \wedge ((a) \vee (b))$ */* parameter bijection */*
   (a) $S[i] \neq S[j], T[i] \neq T[j]$ **for any** $1 \leq j < i$
   (b) $S[i] = S[i-q]$ **iff** $T[i] = T[i-q]$ **for any** $1 \leq q < i$

In our example, we have a p-match between the p-strings S and T since every constant/terminal symbol matches and there exists a bijection of parameter symbols between S and T. U does not satisfy the parameter bijection to p-match with S or T. The process of p-matching leads to defining the `prev` encoding.

Definition 3. ([1,8]) Previous (prev) Encoding: *Given $\mathbb{Z}$ as the set of non-negative integers, the function* $\texttt{prev} : (\Sigma \cup \Pi)^*\$ \rightarrow (\Sigma \cup \mathbb{Z})^*\$$ *accepts a p-string T of length n and produces a string Q of length n that* 1) *encodes constant/terminal symbols with the same symbol and* 2) *encodes parameters to point to* **previous** *like-parameters. More formally, Q is constructed of individual $Q[i]$ with $1 \leq i \leq n$ where:*

$$Q[i] = \begin{cases} T[i], \textbf{ if } T[i] \in (\Sigma \cup \{\$\}) \\ 0, \textbf{ if } T[i] \in \Pi \wedge T[i] \neq T[j] \textbf{ for any } 1 \leq j < i \\ i - k, \textbf{ if } T[i] \in \Pi \wedge k = \max\{j | T[i] = T[j], 1 \leq j < i\} \end{cases}$$

For a p-string T of length n, the above $O(n)$ space prev encoding demands the worst case construction time $O(n \log(\min\{n, |\Pi|\}))$, which follows from the discussions of Baker [1,6] and Amir et al. [5] on the dependency of alphabet Π in p-match applications. Note that with an indexed alphabet and an auxiliary $O(|\Pi|)$ mapping structure, we can construct prev in $O(n)$ time. Using Definition 3, our examples evaluate to $\texttt{prev}(S) = A0B0AB54\$$, $\texttt{prev}(T) = A0B0AB54\$$, $\texttt{prev}(U) = A0B2AB31\$$. The relationship between p-strings and the lexicographical ordering of the prev encoding is fundamental to the p-match problem.

Definition 4. prev Lexicographical Ordering: *Given the p-strings S and T and two symbols s and t from the encodings* $\texttt{prev}(S)$ *and* $\texttt{prev}(T)$ *respectively, the relationships $=$, $\neq$, $<$, and $>$ refer to lexicographical ordering between s and t. We define the ordering of symbols from a* prev *encoding of the production* $(\Sigma \cup \mathbb{Z})^*\$$ *to be $\$ < \zeta \in \mathbb{Z} < \sigma \in \Sigma$, where each ζ and σ is lexicographically sorted in their respective alphabets. The relationships $=$, $\neq$, $\prec$, and $\succ$ refer to the lexicographical ordering between strings. In the case of* $\texttt{prev}(S)$ *and* $\texttt{prev}(T)$, $\texttt{prev}(S) \prec \texttt{prev}(T)$ *when* $\texttt{prev}(S)[1] = \texttt{prev}(T)[1], \texttt{prev}(S)[2] = \texttt{prev}(T)[2], ..., \texttt{prev}(S)[j-1] = \texttt{prev}(T)[j-1], \texttt{prev}(S)[j] < \texttt{prev}(T)[j]$ *for some j, $j \geq 1$. Similarly, we can define $=_k$, $\neq_k$, $\prec_k$, and $\succ_k$ to refer to the lexicographical relationships between a pair of p-strings considering only the first $k \geq 0$ symbols.*

The following proposition essential to the p-matching problem is directly related to the symbol ordering established in Definition 4.

Proposition 1. ([1]) *Two p-strings S and T p-match when* $\texttt{prev}(S) = \texttt{prev}(T)$. *Also, $S \prec T$ when* $\texttt{prev}(S) \prec \texttt{prev}(T)$ *and $S \succ T$ when* $\texttt{prev}(S) \succ \texttt{prev}(T)$.

The example prev encodings show a p-match between S and T since $\texttt{prev}(S) = A0B0AB54\$$ and $\texttt{prev}(T) = A0B0AB54\$$. Also, $U \succ S$ and $U \succ T$ since $\texttt{prev}(U) = A0B2AB31\$ > \texttt{prev}(S) = \texttt{prev}(T) = A0B0AB54\$$. We use the ordering established in Definition 4 to define the parameterized suffix array.

Definition 5. Parameterized Suffix Array (p-suffix array): *The p-suffix array (pSA) for a p-string T of length n maintains a lexicographical ordering of the indices i representing individual p-suffixes $prev(T[i...n])$ with $1 \leq i \leq n$, such that* $\texttt{prev}(T[pSA[q]...n]) \prec \texttt{prev}(T[pSA[q+1]...n]) \forall q, 1 \leq q < n$. *The act of constructing pSA is referred to as p-suffix sorting.*

In the working example using T, the p-suffix array $pSA = \{9, 8, 7, 4, 2, 1, 5, 6, 3\}$. The encoding `prev` is supplemented by the encoding `forw`.

Definition 6. Forward (`forw`) Encoding: *Let the function* $\texttt{rev}(T)$ *reverse the p-string* T *and* $\texttt{repl}(T, x, y)$ *replace all occurrences in* T *of the symbol* x *with* y*. We define the function* `forw` *for the p-string* T *of length* n *as* $\texttt{forw}(T) = \texttt{rev}(\texttt{repl}(\texttt{prev}(\texttt{rev}(T)), 0, n))$.

Essentially, `forw` performs the following on a p-string T of length n: 1) encodes constant/terminal symbols with the same symbol and 2) encodes each parameter p with the **forward** distance to the next occurrence of p or an unreachable forward distance n. Our definition of the `forw` encoding generates output mirroring the `fw` encoding used by Deguchi et al. [7,8]. Let $\mathbb{N}$ refer to the set of positive, non-zero integers. The function $\texttt{fw} : (\Sigma \cup \Pi)^* \rightarrow (\Sigma \cup \mathbb{N})^*$ produces an output encoding G with $\texttt{fw}(T) = G$ for each $1 \leq i \leq n$:

$$G[i] = \begin{cases} T[i], \textbf{ if } T[i] \in \Sigma \\ \infty, \textbf{ if } T[i] \in \Pi \wedge T[i] \neq T[j] \textbf{ for any } i < j \leq n \\ k - i, \textbf{ if } T[i] \in \Pi \wedge k = \min\{j | T[i] = T[j], i < j \leq n\} \end{cases}$$

The `forw` encodings in our example with $n = 9$ are $\texttt{forw}(S) = A5B4AB99\$$, $\texttt{forw}(T) = A5B4AB99\$$, $\texttt{forw}(U) = A2B3AB19\$$.

4 p-Suffix Sorting via Fingerprints

The magic of sorting the suffixes of a string T of length n from an alphabet Σ is rooted in the notion that individual suffixes are very closely related. Throughout this work, we are challenged with the reality that the p-suffix, more formally $\texttt{prev}(T[i...n])$, is not *naïvely* the suffix of the `prev` encoding of T, namely $\texttt{prev}(T)[i...n]$, which is formalized in Lemma 1. (Given space constraints, we omit the proofs of the lemmas, which are included in [16]).

Lemma 1. *Given a p-string* T *of length* n*, the suffixes of* `prev(T)` *are not necessarily the p-suffixes of* T*. More formally, if* $\pi \in \Pi$ *occurs more than once in* T*, then* $\exists i$*, s.t.* $\texttt{prev}(T[i...n]) \neq \texttt{prev}(T)[i...n], 1 \leq i \leq n$.

The centerpiece of this work is the idea that we can *directly* construct the p-suffix array *without* the large memory footprint of the p-suffix tree by handling the dynamically changing p-suffixes, which is fundamentally different from the standard suffix sorting approaches for traditional strings. To visually identify the difference between traditional suffixes and p-suffixes, consider the example $T = zAwz\$$ as a traditional string, in which the suffixes are methodically created by removing a symbol: $\boxed{zAwz\$} \rightarrow \boxed{Awz\$} \rightarrow \boxed{wz\$} \rightarrow \boxed{z\$} \rightarrow \boxed{\$}$. If we consider the same example $T = zAwz\$$ with $\Sigma = \{A\}$ and $\Pi = \{w, z\}$, then the p-suffixes defined under the `prev` encoding are dynamically changing: $\boxed{0A03\$} \rightarrow \boxed{A00\$} \rightarrow \boxed{00\$} \rightarrow \boxed{0\$} \rightarrow \boxed{\$}$.

Our idea is to modify the traditional Karp and Rabin (`KR`) fingerprinting scheme presented in [10,11,17] to accommodate the *changing* nature of p-suffixes.

The KR algorithm generates an integral KR "signature" or "fingerprint" code to represent a string using the lexicographical ordering of symbols [17]. By representing p-suffixes through numeric fingerprints we devise a mechanism to retain a "tangible" copy of the *changing* p-suffixes under the prev encoding. In this section, we assume that n is not too large. That is, the KR codes can fit into standard integer representations such as long long integer.

We now denote the following variables that are continually reused throughout this section for the working p-string T of length n: $prevT = \texttt{prev}(T)$, $forwT = \texttt{forw}(T)$, $max = \texttt{maxdist}(prevT)$ (see below), $R = |\Sigma| + max + 2$. Our fingerprinting approach relies on a lexicographical ordering implementation of Definition 4 to appropriately accommodate the prev alphabet $\Sigma \cup \mathbb{Z} \cup \{\$\}$. Our ordering scheme, function map, is formalized in Definition 7.

Definition 7. Mapping Function: *Let* $max = \texttt{maxdist}(prevT) = \max\{prevT[i] \mid prevT[i] \in \mathbb{Z}$ *for* $1 \leq i \leq n\}$. *Let function* $\alpha_i(x, X)$ *return the lexicographical order* $(1, 2, ..., |X|)$ *of the symbol* x *in alphabet* X. *We then define the function* $\texttt{map} : (\Sigma \cup \mathbb{Z} \cup \{\$\}) \rightarrow \mathbb{N}$ *to map a symbol, say* x, *in* $prevT$ *to an integer preserving the ordering of Definition 4. We also define the supplement function* $\texttt{in}(x, X)$ *to determine if* $x \in X$ *instantaneously based on the definition of* $\texttt{map}(x)$.

$$\texttt{map}(x) = \begin{cases} 1, \textbf{ if } x = \$ \\ \alpha_i(x, \mathbb{Z}) + 1, \textbf{ if } x \in \mathbb{Z} \\ \alpha_i(x, \Sigma) + max + 2, \textbf{ if } x \in \Sigma \end{cases}$$

$$\texttt{in}(x, X) = \begin{cases} true, \textbf{ if } X = \mathbb{Z} \wedge (1 < \texttt{map}(x) \leq max + 2) \\ true, \textbf{ if } X = (\Sigma \cup \{\$\}) \wedge (\texttt{map}(x) = 1 \vee \texttt{map}(x) > max + 2) \\ false, \textbf{ otherwise} \end{cases}$$

The function map is fundamental for the parameterized Karp-Rabin fingerprinting (pKR) algorithm, which generates parameterized Karp-Rabin (pKR) codes.

Definition 8. Parameterized Karp-Rabin (pKR) Function: *Let* $prevT_i = \texttt{prev}(T[i...n])$. *We define* $\texttt{pKR}(i) = \sum_{k=n}^{i} \left[R^{k-1} \times \texttt{map}(prevT_i[n-k+1])\right]$ *to return a fingerprint generated for the p-suffix beginning at index* i.

Table 1 shows example fingerprints using our pKR algorithm and also the standard algorithm KR for the string $T = AwBzABwz\$$. This example shows the true power of our pKR algorithm in that the ordering of the computed fingerprints for p-suffixes of T yields the correct p-suffix array $pSA = \{9, 8, 7, 4, 2, 1, 5, 6, 3\}$. We notice that using KR directly produces the array $\{1, 4, 5, 2, 3, 6, 7, 9, 8\}$, which is not the correct p-suffix array. The execution of function pKR may be *naïvely* cascaded to produce fingerprints for all n p-suffixes at positions $1 \leq i \leq n$ of p-string T requiring $O(n^2)$ time, which is a theoretical bottleneck. We can intelligently construct pKR codes for the p-suffixes of T by taking advantage of the relationship between p-suffixes and pKR codes. Consider q_i to be the pKR code for the p-suffix at position i. The code q_{i+1} can be used to compute the fingerprint for q_i for $i \geq 1$ by introducing a new symbol at position i. Lemmas 2 and 3 identify the adjustments that dynamically change the p-suffixes between the neighboring p-suffixes at i and $(i+1)$ when considering a symbol introduced at position i.

Table 1. Lexicographical ordering of p-suffixes with pKR, using $T = AwBzABwz\$$

i	pSA	$T[pSA[i]...n]$	prev$(T[pSA[i]...n])$	pKR$(pSA[i])$	KR$(pSA[i])$
1	9	\$	\$	43046721	43046721
2	8	$z\$$	0\$	90876411	263063295
3	7	$wz\$$	00\$	96190821	330556302
4	4	$zABwz\$$	$0AB04\$$	129298356	129593601
5	2	$wBzABwz\$$	$0B0AB54\$$	130740084	130740084
6	1	$AwBzABwz\$$	$A0B0AB54\$$	358900444	358900444
7	5	$ABwz\$$	$AB00\$$	388608030	391501431
8	6	$Bwz\$$	$B00\$$	398108358	424148967
9	3	$BzABwz\$$	$B0AB04\$$	401786973	401819778

Lemma 2. *Given p-string T, $prevT$ = prev(T), and $prevT[i+1...n]$ = prev$(T[i+1...n])$ where $T[i]$ is a constant, terminal, or the* only occurrence *of parameter $T[i]$ in $T[i...n]$, then $prevT[i...n]$ = prev$(T[i...n])$ if $prevT[i]$ = prev$(T[i])$.*

Lemma 3. *Given p-string T, $prevT$ = prev(T), $forwT$ = forw(T), and $prevT[i+1...n]$ = prev$(T[i+1...n])$ where $T[i] \in \Pi$ occurs multiple times in $T[i...n]$, then $prevT[i...n]$ = prev$(T[i...n])$ after* 1) *identifying the current parameter as the first occurrence of $T[i]$ ($prevT[i] = 0$) and* 2) *modifying the future occurrence of $T[i]$ ($prevT[i + forwT[i]] = forwT[i]$).*

We refer to a code generated by pKR for the p-suffix i as q_i. The transitions needed to compute a p-suffix i from a p-suffix $(i+1)$ formalized in Lemmas 2 and 3 are subsequently the requirements to compute code q_i from q_{i+1}. These transitions are consolidated into $\delta_{\texttt{pKR}}$ and shown to efficiently generate pKR codes.

Definition 9. Function $\delta_{\texttt{pKR}}$: *Let $\beta = forwT[i]$, $\lambda = (\texttt{map}(\beta) - \texttt{map}(0)) \times R^{n-\beta-1}$, and $B = \frac{q_{i+1} + \texttt{map}(\texttt{prev}(T[i]))R^n}{R}$. We define the function $\delta_{\texttt{pKR}}(i, q_{i+1})$ as follows to return the code q_i via a transition of the provided code q_{i+1} with the newly added symbol at position i.*

$$\delta_{\texttt{pKR}}(i, q_{i+1}) = \begin{cases} B, \textbf{ if } \texttt{in}(prevT[i], \Sigma \cup \{\$\}) \vee (\texttt{in}(prevT[i], \mathbb{Z}) \wedge forwT[i] \geq n) \\ B + \lambda, \textbf{ if } \texttt{in}(prevT[i], \mathbb{Z}) \wedge forwT[i] < n \end{cases}$$

Theorem 1. *Given a p-string T of length n and precalculated variables $prevT$ and $forwT$, function $\delta_{\texttt{pKR}}$ requires $O(n)$ time to generate fingerprints for all p-suffixes in T.*

Proof. The fingerprints are generated successively by the function calls $q_n = \delta_{\texttt{pKR}}(n, 0)$, $q_{n-1} = \delta_{\texttt{pKR}}(n-1, q_n)$,...,$q_1 = \delta_{\texttt{pKR}}(1, q_2)$. Either case of function $\delta_{\texttt{pKR}}$ may be computed in $O(1)$ time and is called sequentially a total of n times, once for each of the n p-suffixes. The overall time is $O(n)$. □

We introduce p_suffix_sort_pKR in Algorithm 1 to sort p-suffixes via the sorting of fingerprints through the transition function in Definition 9. Theorem 2 proves the time complexity of Algorithm 1.

Theorem 2. *Given a p-string T of length n, function* `p_suffix_sort_pKR` *sorts all the n p-suffixes of T in $O(n)$ time.*

Proof. We assume that the fingerprints for each p-suffix are practically represented by an integer code and each use of the code is accomplished in constant time. Thus, Section A) of `p_suffix_sort_pKR` follows from Theorem 1 to require $O(n)$ time. The radix sorting required in section B) requires $O(cn)$, where c is a constant. The loop in section C) clearly requires $O(n)$ time. Overall, `p_suffix_sort_pKR` requires $O(n)$ time. □

The idea used in Algorithm `p_suffix_sort_pKR` is novel, but assumes that the pKR fingerprints fit into practical integer representations. This assumption is primarily a limitation inherent to fingerprinting. It is well documented that Karp-Rabin integral fingerprints can be large and exceed the extremes of an integer with large strings and vast alphabets. The modulo operation discussed in [10,11,17] is used to handle this problem. However, the modulo operation will not preserve the lexicographical ordering between fingerprints and creates a new problem with respect to suffix sorting. Even if we use fingerprints to encode prefixes of p-suffixes, the codes can still be quite large with collisions. We extend our idea using arithmetic coding to address these limitations.

Algorithm 1. p-suffix sorting with fingerprints

```
1  struct pcode { int i, long long int pKR }
2  int[] p_suffix_sort_pKR(char T[]) {
3      pcode code[n], long long int pKR=0
4      int pSA[n], k
5      // A) -- generate the individual prev fingerprints
6      for k=n to 1 {
7          pKR=δ_pKR(k,pKR)
8          code[k]=(k,pKR)
9      }
10     // B) -- sort p-suffixes
11     radix_sort the pKR attribute of each pair in code
12     // C) -- retain p-suffix array
13     for k=1 to n
14         pSA[k]=code[k].i
15     return pSA
16 }
```

5 p-Suffix Sorting via Arithmetic Coding

Arithmetic coding compresses a string by recursively dividing up a real number line into intervals that account for the cumulative distribution function (*cdf*), which describes the probability space of each symbol. The interval for an arithmetic code AC is (lo, hi), where lo and hi are the low and high boundaries,

respectively. Any consistent choice in this region, say $\texttt{tag}(s) = \frac{s.hi+s.lo}{2}$, represents the arithmetic code and preserves the lexicographical ordering of strings. Arithmetic coding is further described in [18,19]. Recently, Adjeroh and Nan [20] used a novel application of Shannon-Fano-Elias codes from information theory to address the traditional suffix sorting problem. In the work, they generate arithmetic codes for *m-blocks*, or m-length prefixes of the suffixes, to maintain the ordering of m symbols. They show how to efficiently transition one *AC m-block* code at suffix i to construct the *m-block AC* at suffix $(i + 1)$ by removing the symbol at i and appending the symbol at $(i + m)$. The transitioning scheme is illustrated in Fig. 1. In terms of suffix sorting with arithmetic codes in [20], the suffixes are recursively partitioned and the generated *m-block* arithmetic codes are exploited to induce the ordering of the partitions in linear time. Extending the suffix sorting via arithmetic coding algorithm given in [20] to the p-suffix sorting problem is not straightforward because of the dynamic relationship between p-suffixes, identified in Lemma 1.

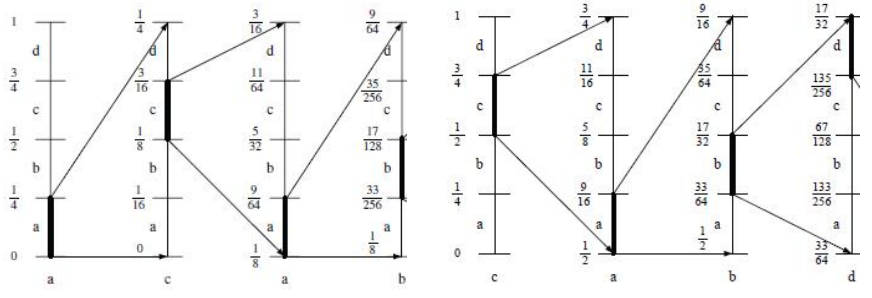


Fig. 1. Transitioning the *AC m-block* code from $\boxed{a}cab \rightarrow cab \rightarrow cab\boxed{d}$

Given an n-length p-string T, we can create a parameterized arithmetic code *pAC* via function `pAC` from Definition 10 for the *m-blocks*, or m-length prefixes, of the n p-suffixes of T. The distribution of symbols will impact the size of the intervals and hence the tag, but this does not change the order of the generated arithmetic codes. Thus, without loss of generality, we assume that each symbol $x \in (\Sigma \cup \mathbb{Z} \cup \{\$\})$ in the alphabet of a `prev` encoding to be equally probable, where p represents the probability of a symbol and the array *cdf* contains the values of the uniform cdf with respect to the neighboring lexicographical alphabet symbols. The following definition modifies the traditional `AC` algorithm to create an *m-block* arithmetic code for a p-suffix at position i in T.

Definition 10. Parameterized Arithmetic Coding (`pAC`) Function: *For an n-length p-string T, the function* `pAC` *in Algorithm* 2 *will generate an arithmetic code interval for the m-block of the p-suffix starting at position i.*

Table 2 shows the *pAC* codes for *m-blocks* of $m = 2, 3, n$ of p-string $T = AwBzABwz\$$. We notice that a "collision" occurs for two *pAC* codes using

Algorithm 2. Generating pAC for an m-length prefix of p-suffix i

```
1  struct AC { long double lo, long double hi }
2  AC pAC(int i, int m) {
3      int end=min{i+m−1,n}, k
4      char prevT_i[]=prev(T[i...end]), AC new=(0,0), old=(0,1)
5      for k=i to end {
6          new.hi=old.lo+(old.hi−old.lo)∗cdf[map(prevT_i[k−i+1])]
7          new.lo=old.lo+(old.hi−old.lo)∗cdf[map(prevT_i[k−i+1])−1]
8          old=new
9      }return new
10 }
```

Table 2. Lexicographical ordering of p-suffixes with pAC, using $T = AwBzABwz\$$

i	pSA	$T[pSA[i]...n]$	prev$(T[pSA[i]...n])$	tag(pAC$(pSA[i], m)$)		
				$m=2$	$m=3$	$m=n$
1	9	\$	\$	0.055556	0.055556	0.055556
2	8	$z\$$	0\$	0.117284	0.117284	0.117284
3	7	$wz\$$	00\$	0.129630	0.124143	0.124143
4	4	$zABwz\$$	$0AB04\$$	0.203704	0.209191	0.208743
5	2	$wBzABwz\$$	$0B0AB54\$$	0.216049	0.211934	0.212459
6	1	$AwBzABwz\$$	$A0B0AB54\$$	0.796296	0.801783	0.801384
7	5	$ABwz\$$	$AB00\$$	0.882716	0.878601	0.878076
8	6	$Bwz\$$	$B00\$$	0.907407	0.903292	0.902683
9	3	$BzABwz\$$	$B0AB04\$$	0.907407	0.911523	0.912083

$m = 2$ since the *m-blocks* are equivalent. Even though the pAC codes distinctly sort the n p-suffixes of T when m approaches n, we highlight that the practical limitation is arithmetic precision. See [18,20] for handling this problem.

In order to use the m-block codes, we must generate them efficiently. We denote the m-block arithmetic code at p-suffix i by pAC_i. The idea is to first use function pAC to compute pAC_1 and use this code to generate the remaining $(n-1)$ codes, namely pAC_2, pAC_3, ..., and pAC_n. Iteratively, we will need to adjust the arithmetic codes to 1) remove the old symbol and 2) add the new symbol. These cases are described below. The lemmas are similar in nature to Lemmas 2 and 3 and thus, are omitted for space.

Case 1: Removing a symbol s from an arithmetic code m-block requires us to simply delete s when $s \in \Sigma$ or $s \in \Pi$ and does not occur in the m-block. When $s \in \Pi$ and occurs later in the m-block, the code must accommodate for both the removed occurrence and the future occurrence of s.

Definition 11. Remove Symbol $\delta^-_{\texttt{pAC}}$ Transition: *Given the AC code A at m-block i with $i+m-1 \leq n$, $\delta^-_{\texttt{pAC}}$ supplies the transition to remove the symbol at*

position i and provide the new code A of the $(m\text{-}1)$-block at p-suffix $(i+1)$. Let $\beta = forwT[i]$, $j = i+\beta$, $e = \min\{i+m-1, n\}$, $\lambda = (\texttt{map}(\beta) - \texttt{map}(0)) \times p^{\beta+1}$, and $c = cdf[\texttt{map}(\texttt{prev}(T[i])) - 1]$.

$$\delta^{-}_{\texttt{pAC}}(i, A) = \begin{cases} \left(\frac{A.lo-c}{p}, \frac{A.hi-c}{p}\right), \textbf{if } (\texttt{in}(prevT[i], \mathbb{Z}) \wedge j > e) \vee \\ \qquad\qquad\qquad\qquad \texttt{in}(prevT[i], \Sigma \cup \{\$\}) \\ \left(\frac{A.lo-\lambda-c}{p}, \frac{A.hi-\lambda-c}{p}\right), \textbf{if } \texttt{in}(prevT[i], \mathbb{Z}) \wedge j \leq e \end{cases}$$

Case 2: Adding (i.e. appending) symbol s to an arithmetic code m-block requires us to simply append the code when $s \in \Sigma$ or $s \in \Pi$ and does not occur in the m-block. When $s \in \Pi$ and occurs previously in the m-block, the code must account for the new occurrence in terms of the previous occurrence of s.

Definition 12. Add Symbol $\delta^{+}_{\texttt{pAC}}$ Transition: *Given the AC code A at $(m\text{-}1)$-block $(i-m+1) \geq 1$, $\delta^{+}_{\texttt{pAC}}$ supplies the transition to add the symbol at position i and provide the new code A of the m-block at p-suffix $(i-m+1)$. Let $b = \max\{1, i-m+1\}$, $k = i - prevT[i]$, $\Delta = A.hi - A.lo$, $d = \Delta \times cdf[\texttt{map}(\texttt{prev}(T[i]))]$, $f = \Delta \times cdf[\texttt{map}(\texttt{prev}(T[i])) - 1]$, $v = \Delta \times cdf[\texttt{map}(prevT[i])]$, and $w = \Delta \times cdf[\texttt{map}(prevT[i]) - 1]$)*

$$\delta^{+}_{\texttt{pAC}}(i, A) = \begin{cases} (A.lo + f, A.lo + d), \textbf{if } (\texttt{in}(prevT[i], \mathbb{Z}) \wedge k < b) \vee \\ \qquad\qquad\qquad\qquad \texttt{in}(prevT[i], \Sigma \cup \{\$\}) \\ (A.lo + w, A.lo + v), \textbf{if } \texttt{in}(prevT[i], \mathbb{Z}) \wedge k \geq b \end{cases}$$

With the assistance of Definitions 11 and 12, we can efficiently generate the *m-block* codes for all n p-suffixes of T. Consider the p-string $T = zwzABA\$$, $\Sigma = \{A, B\}$, $\Pi = \{w, z\}$, and $m = 4$, we successively generate the *m-block* codes in the following fashion: $\boxed{0}\boxed{0}\boxed{2}A \overset{\delta^{-}_{\texttt{pAC}}}{\rightarrow} 00A \overset{\delta^{+}_{\texttt{pAC}}}{\rightarrow} 00A\boxed{B} \rightarrow \cdots$.

Theorem 3. *Given a p-string T of length n and precalculated variables $prevT$ and $forwT$, the pAC codes for all the m-length prefixes of the p-suffixes require $O(n)$ time to generate.*

Proof. Generating the first m-block code pAC_1 via $pAC_1 = \texttt{pAC}(1, m)$ will require $O(m)$ time. Iteratively, the neighboring pAC codes will be used to create the successive p-suffix codes. The first extension of code pAC_1 to create pAC_2 will require the removal of $prevT[1]$ via a call to $pAC_2 = \delta^{-}_{\texttt{pAC}}(1, pAC_1)$, which is $O(1)$ work, and the addition of symbol $prevT[2+m-1]$ via a call to $pAC_2 = \delta^{+}_{\texttt{pAC}}(2+m-1, pAC_2)$, which also demands $O(1)$ work. This process requiring two $O(1)$ steps is needed for the remaining $(n-1)$ m-block p-suffixes of T. The resulting time is $O(m+n)$. Since $m \leq n$, the theorem holds. □

The efficient preprocessing from Theorem 3 leads to our main result: an average case linear time algorithm for direct p-suffix sorting for non-binary parameter alphabets. We discuss the intricacies of worst case p-suffix array construction in the conclusions as an area for future work.

Theorem 4. *Given a p-string T of length n, p-suffix-sorting of T can be accomplished in $O(n)$ time on average via parameterized arithmetic coding.*

Proof. We can construct $\texttt{prev}(T)$ in $O(n)$ time given an indexed alphabet and an $O(|\Pi|)$ auxiliary data structure. The lexicographical ordering of the m-block pAC codes follow from the notion of arithmetic coding and Definition 7. From Theorem 3, we can create all the m-block $\texttt{pAC}$ codes in $O(n)$ time. Similar to [20], the individual floating-point codes may be converted to integer codes d_i in the range $[0, c(n-1)]$ by $d_i = \left\lfloor c(n-1)\frac{\texttt{tag}(pAC_i)-\texttt{tag}(pAC_{min})}{\texttt{tag}(pAC_{max})-\texttt{tag}(pAC_{min})} \right\rfloor$, where the constant $c > 1$ is chosen to best separate the d_i and values pAC_{min} and pAC_{max} correspond to the minimum and maximum pAC codes, respectively. From [21,22], we know that an n-length general string has a max longest common prefix of $O(\log_{|\Sigma|} n)$. Let $x \circ y$ be the string concatenation of x and y. Then, $Q = \texttt{prev}(T[1...n-1])\$\circ\texttt{prev}(T[2...n-1])\$\circ...\circ\texttt{prev}(T[n-2...n-1])\$\circ\$$ contains each individual p-suffix of T. Notice that Q is of length $|Q| = \frac{n(n+1)}{2} \in O(n^2)$ and since all p-suffixes are clearly represented, the symbols of Q may be mapped to a traditional string alphabet, allowing us to use the contribution of [21,22] to obtain the length of the maximum longest common prefix for an average string, which is of the same order $O(\log n^2) \in O(\log n)$. Then by choosing $m = O(\log n)$ and generating the m-block pAC codes, only the first $O(n)$ radix sort of the d_i codes is required to differentiate the p-suffixes of an average case string, demanding only $O(n)$ operations. □

6 Conclusion and Discussion

Approaching the direct p-suffix sorting problem by representing p-suffixes with fingerprints and arithmetic codes provides new mechanisms to handle the challenges of the p-string. We proposed a theoretical algorithm using fingerprints to p-suffix sort an n-length p-string in $O(n)$ time, with n and the alphabet size constrained in practice. Arithmetic codes were then used to propose an algorithm to p-suffix sort p-strings in linear time on average. In terms of direct suffix sorting, the time/space tradeoff varies with algorithms. For instance, the algorithm in [23] accomplishes in-place suffix sorting in super-linear time, using only space for the suffix array and text. On a practical note, our algorithms use space and computation to achieve linear time direct construction of the p-suffix array, improving on the time required by the approaches introduced in [7]. With respect to space, our algorithms use an array for the $\texttt{prev}$ encoding, which replaces the text, in addition to an array for pairs of elements representing the numeric codes and suffix indices. A future reseach problem is to address the worst case performance by identifying the intricate relationship between the dynamic nature of p-suffix partitions with induced sorting, the fundamental mechanism in worst case linear time suffix sorting of traditional strings [9,20,24,25].

References

1. Baker, B.: A theory of parameterized pattern matching: Algorithms and applications. In: STOC 1993, pp. 71–80. ACM, New York (1993)
2. Shibuya, T.: Generalization of a suffix tree for RNA structural pattern matching. Algorithmica 39(1), 1–19 (2004)
3. Zeidman, B.: Software v. software. IEEE Spectr. 47, 32–53 (2010)
4. Idury, R., Schäffer, A.: Multiple matching of parameterized patterns. Theor. Comput. Sci. 154, 203–224 (1996)
5. Amir, A., Farach, M., Muthukrishnan, S.: Alphabet dependence in parameterized matching. Inf. Process. Lett. 49, 111–115 (1994)
6. Baker, B.: Parameterized pattern matching by Boyer-Moore-type algorithms. In: SODA 1995, pp. 541–550. ACM, Philadelphia (1995)
7. Tomohiro, I., Deguchi, S., Bannai, H., Inenaga, S., Takeda, M.: Lightweight Parameterized Suffix Array Construction. In: Fiala, J., Kratochvíl, J., Miller, M. (eds.) IWOCA 2009. LNCS, vol. 5874, pp. 312–323. Springer, Heidelberg (2009)
8. Deguchi, S., Higashijima, F., Bannai, H., Inenaga, S., Takeda, M.: Parameterized suffix arrays for binary strings. In: PSC 2008, Czech Republic, pp. 84–94 (2008)
9. Adjeroh, D., Bell, T., Mukherjee, A.: The Burrows-Wheeler Transform: Data Compression, Suffix Arrays and Pattern Matching. Springer, New York (2008)
10. Gusfield, D.: Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. Cambridge University Press, Cambridge (1997)
11. Smyth, W.: Computing Patterns in Strings. Pearson, New York (2003)
12. Kosaraju, S.: Faster algorithms for the construction of parameterized suffix trees. In: FOCS 1995, pp. 631–637. ACM, Washington, DC (1995)
13. Cole, R., Hariharan, R.: Faster suffix tree construction with missing suffix links. In: STOC 2000, pp. 407–415. ACM, New York (2000)
14. Lee, T., Na, J.C., Park, K.: On-Line Construction of Parameterized Suffix Trees. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 31–38. Springer, Heidelberg (2009)
15. Manber, U., Myers, G.: Suffix arrays: A new method for on-line string searches. SIAM J. Comput. 22, 935–948 (1993)
16. Beal, R.: Parameterized Strings: Algorithms and Data Structures. MS Thesis. West Virginia University (2011)
17. Karp, R., Rabin, M.: Efficient randomized pattern-matching algorithms. IBM J. Res. Dev. 31, 249–260 (1987)
18. Moffat, A., Neal, R., Witten, I.: Arithmetic coding revisited. ACM Trans. Inf. Syst. 16, 256–294 (1995)
19. Cover, T., Thomas, J.: Elements of Information Theory. Wiley (1991)
20. Adjeroh, D., Nan, F.: Suffix sorting via Shannon-Fano-Elias codes. Algorithms 3(2), 145–167 (2010)
21. Karlin, S., Ghandour, G., et al.: New approaches for computer analysis of nucleic acid sequences. PNAS 80(18), 5660–5664 (1983)
22. Devroye, L., Szpankowski, W., Rais, B.: A note on the height of suffix trees. SIAM J. Comput. 21, 48–53 (1992)
23. Franceschini, G., Muthukrishnan, S.: In-Place Suffix Sorting. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 533–545. Springer, Heidelberg (2007)
24. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. J. ACM. 53, 918–936 (2006)
25. Manzini, G., Ferragina, P.: Engineering a lightweight suffix array construction algorithm. Algorithmca 40, 33–50 (2004)

Periods in Partial Words: An Algorithm*

Francine Blanchet-Sadri[1], Travis Mandel[2], and Gautam Sisodia[3]

[1] Department of Computer Science, University of North Carolina, P.O. Box 26170, Greensboro, NC 27402–6170, USA
blanchet@uncg.edu

[2] Department of Mathematics, The University of Texas at Austin, 1 University Station, C1200, Austin, TX 78712, USA

[3] Department of Mathematics, University of Washington, P.O. Box 354350, Seattle, WA 98195-4350, USA

Abstract. *Partial words* are finite sequences over a finite alphabet that may contain some holes. A variant of the celebrated Fine-Wilf theorem shows the existence of a bound $L = L(h, p, q)$ such that if a partial word of length at least L with h holes has periods p and q, then it has period $\gcd(p, q)$. In this paper, we associate a graph with each p- and q-periodic word, and study two types of vertex connectivity on such a graph: modified degree connectivity and r-set connectivity where $r = q \bmod p$. As a result, we give an algorithm for computing $L(h, p, q)$ in the general case.

1 Introduction

The problem of computing periods in *words*, or finite sequences of symbols from a finite alphabet, has important applications in several areas including data compression, coding, computational biology, string searching and pattern matching algorithms. Repeated patterns and related phenomena in words have played over the years a central role in the development of combinatorics on words [1], and have been highly valuable tools for the design and analysis of algorithms. In many practical applications, such as DNA sequence analysis, repetitions admit a certain variation between copies of the repeated pattern because of errors due to mutation, experiments, etc. Approximate repeated patterns, or repetitions where errors are allowed, are playing a central role in different variants of string searching and pattern matching problems [2]. *Partial words*, or finite sequences that may contain some holes, have acquired importance in this context. A *(strong) period* of a partial word u over an alphabet A is a positive integer p such that $u(i) = u(j)$ whenever $u(i), u(j) \in A$ and $i \equiv j \bmod p$ (in such a case, we call u *p-periodic*). In other words, p is a period of u if for all positions i and j congruent modulo p, the letters in these positions are the same or at least one of these positions is a hole.

* This material is based upon work supported by the National Science Foundation under Grant No. DMS–0452020.

C.S. Iliopoulos and W.F. Smyth (Eds.): IWOCA 2011, LNCS 7056, pp. 57–70, 2011.
© Springer-Verlag Berlin Heidelberg 2011

There are many fundamental results on periods of words. Among them is the well-known periodicity result of Fine and Wilf [3], which determines how long a p- and q-periodic word needs to be in order to also be $\gcd(p, q)$-periodic. More precisely, any word having two periods p, q and length at least $p + q - \gcd(p, q)$ has also $\gcd(p, q)$ as a period. Moreover, the length $p + q - \gcd(p, q)$ is optimal since counterexamples can be provided for shorter lengths, that is, there exists an *optimal* word of length $p+q-\gcd(p, q)-1$ having p and q as periods but not having $\gcd(p, q)$ as period [1]. Extensions of Fine and Wilf's result to more than two periods have been given. For instance, in [4], Constantinescu and Ilie give an extension for an arbitrary number of periods and prove that their lengths are optimal.

Fine and Wilf's result has been generalized to partial words [5,6,7,8,9,10,11]. Some of these papers are concerned with *weak* periodicity, a notion not discussed in this paper. The papers that are concerned with strong periodicity refer to the basic fact, proved by Shur and Konovalova (Gamzova) in [10], that for positive integers h, p and q, there exists a positive integer l such that a partial word u with h holes, two periods p and q, and length at least l has period $\gcd(p, q)$. The smallest such integer is called the optimal length and it will be denoted by $L(h, p, q)$. They gave a closed formula for the case where $h = 2$ (the cases $h = 0$ or $h = 1$ are implied by the results in [3,5]), while in [9], they gave a formula in the case where $p = 2$ as well as an optimal asymptotic bound for $L(h, p, q)$ in the case where h is "large." In [7], Blanchet-Sadri et al. gave closed formulas for the optimal lengths when q is "large," whose proofs are based on connectivity in the so-called (p, q)-periodic graphs. In this paper, we study two types of vertex connectivity in these graphs: the modified degree connectivity and r-set connectivity where $r = q \bmod p$. Although the graph-theoretical approach is not completely new, the paper gives insights into periodicity in partial words and provides an algorithm for determining $L(h, p, q)$ in *all* cases.

We end this section by reviewing basic concepts on partial words. Fixing a nonempty finite set of letters or an *alphabet* A, finite sequences of letters from A are called *(full) words* over A. The number of letters in a word u, or *length* of u, is denoted by $|u|$. The unique word of length 0, denoted by ε, is called the *empty* word. The set of all words over A of finite length is denoted by A^*. A *partial word* u of length n over A is a partial function $u : \{0, \ldots, n-1\} \to A$. For $0 \leq i < n$, if $u(i)$ is defined, then i belongs to the *domain* of u, denoted by $i \in D(u)$, otherwise i belongs to the *set of holes* of u, denoted by $i \in H(u)$. For convenience, we will refer to a partial word over A as a word over the enlarged alphabet $A_\diamond = A \cup \{\diamond\}$, where $\diamond \notin A$ represents a "do not know" symbol or hole.

2 (p, q)-Periodic Graphs

In this section, we discuss the fundamental property of periodicity, the goal of our paper which is to describe an algorithm to compute $L(h, p, q)$ in all cases, and some initial results. We can restrict our attention to the case where p and q are coprime, since it is well-known that the general case can be reduced to the coprime case (see, for example, [5,9]). Also, we assume without loss of generality

that $1 < p < q$. Fine and Wilf show that $L(0, p, q) = p + q - \gcd(p, q)$ [3], Berstel and Boasson that $L(1, p, q) = p + q$ [5], and Shur and Konovalova prove $L(2, p, q)$ to be $2p + q - \gcd(p, q)$ [10]. Other results include the following.

Theorem 1 ([7,9]). *For $0 \leq m < q$, $L(nq + m, 2, q) = (2n + 1)q + m + 1$.*

Theorem 2 ([7]). *If $q > x(p, h)$ where $x(p, h)$ is $p(\frac{h}{2})$ if h is even and $p(\frac{h+1}{2})$ if h is odd, then*

$$L(h, p, q) = \begin{cases} p(\frac{h+2}{2}) + q - \gcd(p, q), & \text{if } h \text{ is even;} \\ p(\frac{h+1}{2}) + q, & \text{if } h \text{ is odd.} \end{cases}$$

The problem of finding $L(h, p, q)$ is equivalent to a problem involving the vertex connectivity of certain graphs, as described in [7], which we now discuss. We can represent the periodic structure of a full word with two periods through a graph associated with the word. The (p, q)-*periodic graph* of size l is the graph $G = (V, E)$ where $V = \{0, 1, \ldots, l - 1\}$ and for $i, j \in V$, the pair $\{i, j\} \in E$ if and only if $i \equiv j \bmod p$ or $i \equiv j \bmod q$. The *degree* of a vertex $i \in V$, denoted $\mathbf{d}(i)$, is the number of vertices connected to i, that is,

$$\mathbf{d}(i) = \left\lfloor \frac{l - 1 - i \bmod p}{p} \right\rfloor + \left\lfloor \frac{l - 1 - i \bmod q}{q} \right\rfloor - \left\lfloor \frac{l - 1 - i \bmod pq}{pq} \right\rfloor.$$

The first term gives the number of p-connections, the second term the number of q-connections, and the third term the number of pq-connections.

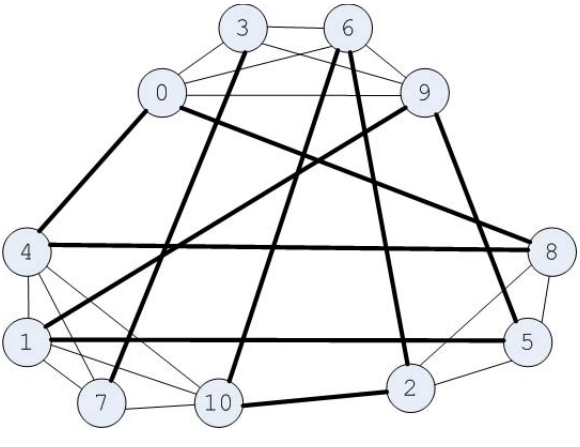


Fig. 1. The $(3, 4)$-periodic graph of size 11. The bold connections are q-edges, while the lighter ones are p-edges.

The (p, q)-periodic graph of size l can be thought to represent a full word of length l with periods p and q, with the vertices corresponding to positions of the

word, and the edges corresponding to the equalities between letters of the word forced by one of the periods. For example, we see that if the (p, q)-periodic graph of size l is connected, then a word of length l with periods p and q is 1-periodic, because there exists a path between every pair of vertices, thus the word is over a singleton alphabet. A graph has vertex connectivity κ if it can be disconnected with a suitable choice of κ vertex removals, but cannot be disconnected by any choice of $\kappa - 1$ vertex removals.

Note that a hole in a partial word u of length l with periods p and q corresponds to the removal of the associated vertex from the (p, q)-periodic graph of size l. Thus our search for $L(h, p, q)$ (when $\gcd(p, q) = 1$) can be restated in terms of vertex connectivity.

Lemma 1. *The length $L(h, p, q)$ is the smallest l such that the (p, q)-periodic graph of size l has vertex connectivity at least $h + 1$.*

If $G = (V, E)$ is the (p, q)-periodic graph of size l, then the *p-class of vertex i* is $\{j \in V \mid j \equiv i \bmod p\}$. A *$p$-connection* (or *$p$-edge*) is an edge $\{i, j\} \in E$ such that $i \equiv j \bmod p$. If an edge $\{i, j\}$ is a p-connection, then i and j are considered *p-connected*. Similar statements hold for q-classes, q-connections and pq-classes, pq-connections.

Throughout the paper, we will find it useful to group together p-classes whose smallest elements are congruent modulo r where $r = q \bmod p$. We do so by introducing the *r-set* of vertex i, where $i \in \{0, 1, \ldots, r - 1\}$, which is the set

$$\bigcup_{0 \le j < p \text{ and } j \equiv i \bmod r} p\text{-class of vertex } j = \bigcup_{j=0}^{\lfloor \frac{p-i-1}{r} \rfloor} p\text{-class of vertex } jr + i.$$

3 Connectivity in (p, q)-Periodic Graphs

Our algorithm to calculate $L(h, p, q)$ is based on (p, q)-periodic graphs. In this section, we discuss modified degree and $q \bmod p$-set connectivity in these graphs. Using Theorems 1 and 2, we can restrict our discussion to the case where $p \neq 2$ and $q \le p\lfloor \frac{h+1}{2} \rfloor$. Let $G = (V, E)$ be a graph. A *disconnection* of G is a partition $\{V_1, V_2, H\}$ of V (that is, $V = V_1 \cup H \cup V_2$ and V_1, V_2, H are mutually disjoint), such that neither V_1 nor V_2 is empty, and for $v_1 \in V_1$, $v_2 \in V_2$, $\{v_1, v_2\} \notin E$. An *optimal disconnection* is a disconnection such that the cardinality of H is κ, where κ is the vertex connectivity of G. The set H represents the vertices removed in a disconnection, while the sets V_1 and V_2 represent the vertices disconnected from each other in a disconnection.

If G is the (p, q)-periodic graph of size l for some p, q and l and $\{V_1, V_2, H\}$ is an optimal disconnection of G, note that we cannot disconnect G within a p-class since p-classes form complete subgraphs. In other words, a p-class cannot both contain elements in V_1 and V_2, that is, for a p-class C, either $C \subset V_1 \cup H$ or $C \subset V_2 \cup H$. We say that a disconnection $\{V_1, V_2, H\}$ of G disconnects a union of p-classes P if $V_1 \subset P$ and $P \subset V_1 \cup H$, or $V_2 \subset P$ and $P \subset V_2 \cup H$. Similarly, a q-class cannot both contain elements in V_1 and V_2.

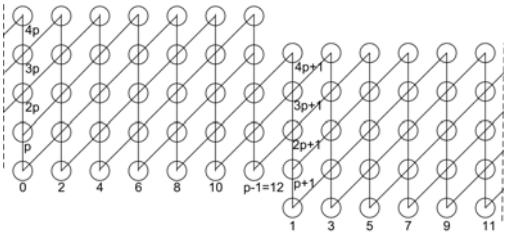


Fig. 2. A (p, q)-periodic graph where the vertical lines represent p-classes, while the diagonal lines represent q-classes. The q-edges wrap around at the dashed lines. All vertices in vertical and diagonal lines are connected to each other. In other words, lines represent several "normal" edges. In the graph, p-classes are grouped into two r-sets.

Suppose we want to disconnect a single p-class C from G. For a q-class C' of G, all of the vertices of C' within C or all of the vertices of C' outside of C must be removed. For $l \geq 2q$, a vertex $i \in C$ has q-connections with vertices outside of C. Each of these q-connections must be broken in order to disconnect C from G. The most efficient way to do so is to remove i itself, since i may have more than one q-connection. However, if we remove all of C from G, we have not formed a disconnection (V_1 or V_2 is empty). Thus, we do not remove the vertex in C contained in the smallest q-class in order to minimize the number of vertex removals required to disconnect C. So, if each vertex $i \in C$ is q-connected to some vertex j outside of C such that no other vertex in C is q-connected to j (no vertex in C is q-connected to i), then the most efficient way of disconnecting C from G is to disconnect a vertex of lowest degree in C. As long as $l \leq pq$, any two distinct vertices within a p-class belong to different q-classes. In this case, the most efficient way to disconnect a single p-class from G is to disconnect a single vertex of lowest degree in G (this is called a minimum degree disconnection).

When $l > pq$, vertices within the same p-class may belong to the same q-class (that is to say, vertices may be both p- and q-connected, or pq-connected). For a vertex i in V, vertices that are pq-connected to i share all other connections with i, and thus should not be counted in the number of vertices required to disconnect i as they are disconnected when i is disconnected. Thus, we introduce the idea of "modified" degree.

Let $G = (V, E)$ be the (p, q)-periodic graph of size l, and let $i \in V$. The *modified degree* of i, denoted $\mathbf{d}^*(i)$, is the number of vertices that are either p- or q-connected to i, but not pq-connected to i, that is,

$$\mathbf{d}^*(i) = \left\lfloor \frac{l - 1 - i \bmod p}{p} \right\rfloor + \left\lfloor \frac{l - 1 - i \bmod q}{q} \right\rfloor - 2 \left\lfloor \frac{l - 1 - i \bmod pq}{pq} \right\rfloor. \quad (1)$$

We subtract 2 times the number of pq-connections: once because we double counted them, and again because vertices that are pq-connected are connected to the same vertices, so disconnecting one vertex will also disconnect all the vertices pq-connected to it. Note that when $l \leq pq$, $\mathbf{d}(i) = \mathbf{d}^*(i)$. When $l > pq$, minimum

degree disconnections are replaced by minimum modified degree disconnections. For a (p, q)-periodic graph G, we define the *modified degree connectivity* of G, denoted κ_d, to be the smallest number of vertex removals required to make a minimum modified degree disconnection, and denote the minimum size of G such that $\kappa_d = h + 1$ by $l_d(h, p, q)$.

Usually, disconnecting more than one p-class takes more holes than individually disconnecting any one p-class, because in general, a set of p-classes has more connections with the rest of the graph than any single p-class. However, disconnecting entire r-sets may prove to be efficient when l is small, as the graph "bottlenecks" between r-sets (that is, fewer q-classes span r-sets than connect p-classes within an r-set). For a (p, q)-periodic graph G, we define the *r-set connectivity* of G, denoted κ_r, to be the smallest number of vertex removals required to make an r-set disconnection, and denote the minimum size of G such that $\kappa_r = h+1$ by $l_r(h, p, q)$. Thus, if G is the (p, q)-periodic graph of size l for $l > 2q$, then either a modified degree disconnection or an r-set disconnection will give an optimal disconnection of G.

Note that the sizes at which our graphs change connectivity are the optimal lengths in question. If the (p, q)-periodic graph of size l has vertex connectivity κ while the (p, q)-periodic graph of size $l + 1$ has vertex connectivity $\kappa + 1$, then $L(\kappa, p, q) = l + 1$. Similarly, if the (p, q)-periodic graph of size l has modified degree connectivity κ_d (respectively, r-set connectivity κ_r) while the (p, q)-periodic graph of size $l + 1$ has modified degree connectivity $\kappa_d + 1$ (respectively, r-set connectivity $\kappa_r + 1$), then $l_d(\kappa_d, p, q) = l + 1$ (respectively, $l_r(\kappa_r, p, q) = l + 1$).

4 *r*-Set Connectivity

Consider the (p, q)-periodic graph of size l where $q = mp + r$ with $0 < r < p$. Set $l = kp + r'$ where $0 \leq r' < p$. Figure 3 depicts a case in which $r' = 0$. We can see here that there are k rows in each r-set. In the columns on either side of any r-set we see that $m + 1$ vertices do not have q-connections to the adjacent r-set, so exactly $\beta = k - (m + 1)$ vertices are q-connected to the adjacent r-set. Consider two adjacent r-sets. Looking at the q-classes that connect these r-sets, we can see that the bottom m of these q-classes have 1 vertex in the left r-set. The next m q-classes have 2 vertices in the left r-set, and so on for the first $k - (m + 1)$ q-classes. The left side of the right r-set is anti-symmetric to this: the top m q-classes each have 1 vertex in the right r-set, and the next m q-classes each have 2 vertices and so on working down. When breaking these q-connections it is best to remove all the vertices from the smaller side of the q-class. Thus, for the bottom half of the q-classes we remove vertices from the left side, and for the top half we remove the same number of vertices from the right side. If $\beta = \gamma(2m) + \phi$ for $0 \leq \phi < 2m$, then we see that the number of vertices we must remove to separate these adjacent r-sets is

$$2m \sum_{i=1}^{\gamma} i + \phi(\gamma + 1) = 2m \frac{\gamma(\gamma + 1)}{2} + \phi(\gamma + 1).$$

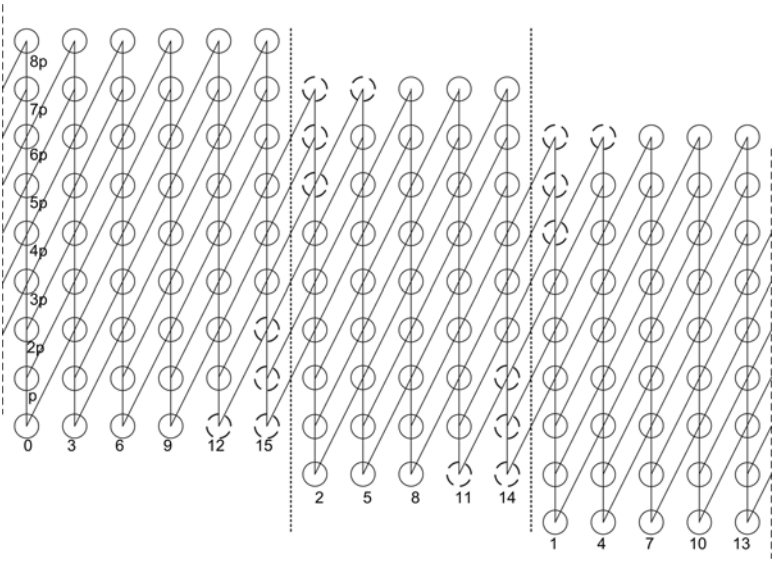


Fig. 3. An r-set disconnection for $p = 16$, $q = 35 = 2p + 3$, and $l = 9p$ (this length is not optimal). Here we are disconnecting the r-set of vertex 2 from the other r-sets.

Since an r-set disconnection requires separating adjacent r-sets twice, we have

$$\frac{\kappa_r}{2} = m\gamma(\gamma + 1) + \phi(\gamma + 1) = (m\gamma + \phi)(\gamma + 1).$$

Since γ is an integer and $\phi < 2m$, we can find γ in terms of κ_r and m by solving for when ϕ is equal to zero and then taking the floor. Using the quadratic formula,

$$\gamma = \left\lfloor \frac{\sqrt{m^2 + 2m\kappa_r} - m}{2m} \right\rfloor.$$

We solve for ϕ and find $\phi = \frac{\kappa_r}{2(\gamma+1)} - m\gamma$. From the definition of β we have $k = 2m\gamma + \phi + m + 1$. The length is never optimal when $r' = 0$ because κ_r only increases for nonzero values of r', as described below. We therefore want to select γ and ϕ such that they give us a value of κ_r that is strictly less than $h+1$. We will make room for the remaining vertex removals by adding r' vertices.

Now we need to calculate r' by determining at exactly which sizes the r-set connectivity actually increases. Starting with size $l = kp$, if we increase the size by r then the number of vertex removals required to break any r-set connection increases by 1 because between each connected pair of r-sets there is one more q-connection. Thus, the r-set connectivity increases by 2. Notice that every connected pair of r-sets requires the same number of vertex removals to separate them. Thus, if we remove the last vertex we added, then the r-set connectivity will have only increased by 1 from the previous size. After decreasing the size by one more vertex the r-set connectivity will be back down to where

it was for $l = kp$. The same thing happens if we add another r vertices and continue until we reach the r-set connectivity of the graph of size $l = (k+1)p$. If we have calculated k for a given p, q and h, then define δ to be the difference between the r-set connectivity that we are looking for and the r-set connectivity at length $l = kp$. Then $\delta = h + 1 - 2(m\gamma + \phi)(\gamma + 1)$, and we can calculate $r' = \left\lfloor \frac{\delta+1}{2} \right\rfloor r - \delta \bmod 2$. We arrive at the following theorem.

Theorem 3. *Let $q = mp + r$ where $0 < r < p$, and let $\beta = 2m\gamma + \phi$, where γ is the greatest integer strictly less than $\frac{\sqrt{m^2+2m(h+1)}-m}{2m}$ and ϕ is the greatest integer strictly less than $\frac{h+1}{2(\gamma+1)} - m\gamma$. Define $\delta = h+1-2(m\gamma+\phi)(\gamma+1)$. Then*

$$l_r(h,p,q) = (\beta + m + 1)p + \left\lfloor \frac{\delta+1}{2} \right\rfloor r - \delta \bmod 2.$$

Using this theorem we have calculated the lengths in Table 1. By comparing the

Table 1. Optimal lengths for r-set disconnections. The empty entries of the table are where $q > p\left\lfloor \frac{h+1}{2} \right\rfloor$, so r-set disconnections are not optimal (see Theorem 2).

	$h = 3$	$h = 4$	$h = 5$	$h = 6$	$h = 7$
$p < q < 2p$	$2p + q$	$3p + q - 1$	$3p + q$	$2p + 2q - 1$	$2p + 2q$
$2p < q < 3p$			$3p + q$	$4p + q - 1$	$4p + q$
$3p < q < 4p$					$4p + q$

lengths in Table 1 to the lengths that can be calculated using modified degree, r-set disconnections are only more efficient when $h = 4$ and $q < \frac{3p}{2}$. As we increase the length beyond the values shown in the table, experimental evidence suggests that r-set disconnections will continue to become less efficient because r-sets now gain q-connections faster than any pq-class gains connections.

5 Modified Degree Connectivity

To count the number of vertices we must remove to disconnect vertex i and all the vertices pq-connected to it, we use the formula in (1) for $\mathbf{d}^*(i)$.

Suppose $l = \tau pq + \omega$ for nonnegative integers τ and $\omega < pq$. If $\omega = 0$ then every vertex has the same modified degree: $\mathbf{d}^*(i) = (\tau q - 1) + (\tau p - 1) - 2(\tau - 1) = \tau(p + q - 2)$. If $\omega > 0$ then define G' to be the subgraph of the (p,q)-periodic graph G of size l that contains only the vertices in the last ω positions. Each of the last ω vertices has $\tau(p + q - 2)$ vertices in the first τpq positions to which it is either p-connected or q-connected but not pq-connected. Thus, the modified degree of a vertex i in G' is equal to $\tau(p + q - 2) + \mathbf{d}_{G'}(i)$, where $\mathbf{d}_{G'}(i)$ is the degree of i in G'. In other words, we can find the degree of the vertex i within the subgraph G', and add this degree to $\tau(p + q - 2)$ to get its modified degree in G. Thus, we have

$$\mathbf{d}^*(i) = \tau(p + q - 2) + \mathbf{d}^*_{G'}(i). \tag{2}$$

The positions of these last ω vertices modulo pq are all less than $\omega = l \bmod pq$, and any two positions in the same pq-class have the same modified degree. Thus we know that one of them will have the lowest modified degree of the graph.

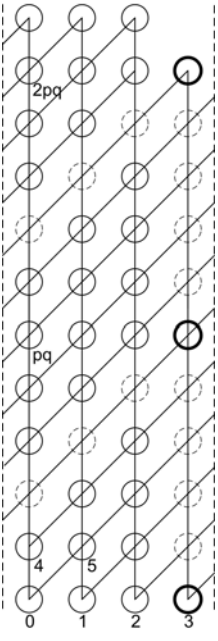


Fig. 4. The $(4, 5)$-periodic graph of size 47. This figure depicts an optimal disconnection where the dashed vertices are in H, the bold vertices are in V_2, and the rest of the vertices are in V_1. Notice that the vertices in V_2 have the minimal modified degree. They are all pq-connected to each other, and are p- or q-connected to the vertices in H. Increasing the size by 1 gives this pq-class one more p-connection, thereby increasing the connectivity of the graph by 1.

We want $\mathbf{d}^*(i) = h + 1$. Since τ is an integer and $\mathbf{d}^*_{G'}(i) < p + q - 2$, we can use the division algorithm and Equation (2) to get $\tau = \left\lfloor \frac{h+1}{p+q-2} \right\rfloor$ and $\mathbf{d}^*_{G'}(i) = (h+1) \bmod (p + q - 2)$. Recall that $l_d(h, p, q)$ is the smallest length at which the minimum modified degree is $h+1$. In other words, $l_d(h, p, q)$ is the optimal length $L(h, p, q)$ if we restrict ourselves to minimum modified degree disconnections. Note that we consider the degree of the empty graph to be zero. We now arrive at the following theorem.

Theorem 4. *The equality* $l_d(h, p, q) = \tau pq + \omega$ *holds, where* $0 \leq \omega < pq$. *More specifically,* $\tau = \left\lfloor \frac{h+1}{p+q-2} \right\rfloor$ *and*

$$\omega = \begin{cases} l_d((h+1) \bmod (p+q-2) - 1, p, q), & \text{if } (h+1) \bmod (p+q-2) \neq 0; \\ 0, & \text{otherwise.} \end{cases}$$

We have now reduced cases where $l_d(h,p,q) \geq pq$ to those where $l_d(h,p,q) < pq$, so now we will assume $l_d(h,p,q) < pq$. A vertex i in a graph of size l has $\left\lfloor \frac{l}{p} \right\rfloor - 1$ p-connections if $i \geq l \bmod p$ and $\left\lfloor \frac{l}{p} \right\rfloor$ p-connections if $i < l \bmod p$. Similarly, i has $\left\lfloor \frac{l}{q} \right\rfloor - 1$ q-connections if $i \geq l \bmod q$ and $\left\lfloor \frac{l}{q} \right\rfloor$ q-connections if $i < l \bmod q$. We add together the number of p-connections and the number of q-connections to find that the degree of i is $\left\lfloor \frac{l}{p} \right\rfloor$ plus $\left\lfloor \frac{l}{q} \right\rfloor$ minus either $0, 1$ or 2 depending on the value of i. We can assume that $l \geq p$ because there will never be an optimal length with $0 < l_d(h,p,q) \bmod pq < p$, since there are no p- or q- connections within this range. Thus we can assume that the vertex in the $p-1$ position exists, and we know that the $p-1$ position always satisfies the condition $p-1 \geq l \bmod p$. This allows us to make the following claim.

Theorem 5. *Define the function*

$$f(\omega,p,q) = \begin{cases} 2, & \text{if there exists } i \in [0..\omega-1] \text{ such that} \\ & i \bmod p \geq \omega \bmod p \text{ and } i \bmod q \geq \omega \bmod q; \\ 1, & \text{otherwise.} \end{cases}$$

Then the (p,q)-periodic graph of size ω has a modified degree connectivity $\kappa_d = \left\lfloor \frac{\omega}{p} \right\rfloor + \left\lfloor \frac{\omega}{q} \right\rfloor - f(\omega,p,q)$.

From this theorem we can see that κ_d increases whenever $f(\omega,p,q)$ changes from 2 to 1, or whenever ω increases to a multiple of either p or q while $f(\omega,p,q)$ stays constant.

Remark 1. If $l_d(h,p,q) = \omega$ and $f(\omega,p,q) = 2$, then $\omega = n_1 p$ or $\omega = n_2 q$ for some positive integers n_1 and n_2.

Since adding a new vertex never decreases the modified degree connectivity of these graphs, $f(\omega,p,q)$ can only change from 1 to 2 at multiples of p and q. If $\omega = n_1 p$ for a positive integer n_1, then a vertex in the $q-1$ q-class with position i satisfies $i > \omega \bmod p$ and $i > \omega \bmod q$, so $f(n_1 p,p,q) = 2$ for $n_1 p > q$ and $f(n_1 p,p,q) = 1$ for $n_1 p < q$. Similarly, $f(n_2 q,p,q) = 2$ for any positive n_2.

To calculate n_1 when $f(\omega,p,q) = 2$ we use the formula $\kappa_d = h+1 = \left\lfloor \frac{n_1 p}{p} \right\rfloor + \left\lfloor \frac{n_1 p}{q} \right\rfloor - 2$. We can solve as follows: $n_1 + \left\lfloor \frac{n_1 p}{q} \right\rfloor = (h+1)+2$ or $\left\lfloor n_1 \left(1 + \frac{p}{q}\right) \right\rfloor = h+3$. So if a solution exists, it is

$$n_1 = \left\lceil \frac{h+3}{1+\frac{p}{q}} \right\rceil . \tag{3}$$

If there is no solution for n_1 satisfying $n_1 + \left\lfloor \frac{n_1 p}{q} \right\rfloor - 2 = h+1$, then there must be a solution for n_2 satisfying $\kappa_d = h+1 = n_2 + \left\lfloor \frac{n_2 q}{p} \right\rfloor - 2$ and we calculate

$$n_2 = \left\lceil \frac{h+3}{1+\frac{q}{p}} \right\rceil . \tag{4}$$

We now consider the $f(\omega, p, q) = 1$ case. Note that $f(l, p, q) = 1$ for all $l < q$. For these cases vertices can only have p-connections, and we can see that $l_d(h, p, q) = (h + 2)p$ so long as $h + 2 \leq \left\lfloor \frac{q}{p} \right\rfloor$. For larger numbers of holes we must better characterize when vertices of lowest degree gain p- and q-connections. First, there is always a vertex of minimal degree in either the $p-1$ p-class or the $q-1$ q-class. This is because if we pick any other position that has minimal degree then we can increase this position without adding more p- or q-connections until it is in either the $p - 1$ p-class or the $q - 1$ q-class. Optimal lengths occur when these positions of minimal degree gain a new p- or q-connection.

Remark 2. If $l_d(h, p, q) = \omega$, $f(\omega, p, q) = 1$, and $h+2 > \left\lfloor \frac{q}{p} \right\rfloor$, then $\omega = n'_1 p + n'_2 q$ for some positive integers n'_1 and n'_2. For $\omega = n'_1 p + n'_2 q - 1$, the vertices of lowest degree are in the symmetric positions $n'_1 p - 1$ and $n'_2 q - 1$.

We now focus on finding these positions $n'_1 p - 1$ and $n'_2 q - 1$. If $f(\omega, p, q)$ changes from 2 to 1 when the $n'_1 p - 1$ vertex gains a q-connection, then we see from the definition of $f(\omega, p, q)$ that the $n'_1 p - 1$ vertex must have a larger value modulo q than the other vertices in the $p - 1$ p-class. Thus we can say that $(n'_1 p - 1) \bmod q > (n''_1 p - 1) \bmod q$ for all positive integers $n''_1 \neq n'_1$ where $n''_1 p < n'_1 p + n'_2 q$. Similarly we must have $(n'_2 q - 1) \bmod p > (n''_2 q - 1) \bmod p$ for all positive integers $n''_2 \neq n'_2$ where $n''_2 q < n'_1 p + n'_2 q$. Also, $n'_1 p + n'_2 q$ must fall between the $f(\omega, p, q) = 2$ solutions for $l_d(h - 1, p, q)$ and $l_d(h, p, q)$.

Algorithm 1. Find $l_d(h, p, q)$ when $1 < p < q$, $\gcd(p, q) = 1$ and $h < p + q - 2$

```
if h + 2 ≤ ⌊q/p⌋ then l_d(h,p,q) = (h + 2)p
else solve for f(ω,p,q) = 2 solutions for l_d(h − 1,p,q) and l_d(h,p,q)
    if the f(ω,p,q) = 2 value for l_d(h,p,q) is n_1 p then
        find the maximum value of n'_1 p mod q for 0 < n'_1 < n_1
        if the vertex in this position has a q-connection between
        f(ω,p,q) = 2 solutions for l_d(h − 1,p,q) and l_d(h,p,q) then
            l_d(h,p,q) is the position of this q-connection
        else l_d(h,p,q) = n_1 p
    if the f(ω,p,q) = 2 value for l_d(h,p,q) is n_2 q then
        find the maximum value of n'_2 q mod p for 0 < n'_2 < n_2
        if the vertex in this position has a p-connection between
        f(ω,p,q) = 2 solutions for l_d(h − 1,p,q) and l_d(h,p,q) then
            l_d(h,p,q) is the position of this p-connection
        else l_d(h,p,q) = n_2 q
```

For $m = \left\lfloor \frac{q}{p} \right\rfloor$, the $mp-1$ vertex has the lowest degree in a large number of cases when the length is less than pq (keep in mind that we can reduce any case to one where the length is less than pq). The following lemma identifies many of these cases. We then use this knowledge to find a large number of optimal lengths in the theorem that follows.

Lemma 2. *Let G be the (p,q)-periodic graph of size l, let $q = mp + r$ where $0 < r < p$, and let $l = nq + r_1$ where $0 \leq r_1 < q$. Let $mp \leq l \leq pq$. If $l \bmod q < mp$ or $nr - 1 < l \bmod p$, then the $mp-1$ vertex has minimum degree.*

Proof. We require $l \geq mp$ so the $mp-1$ vertex exists, and we require $l \leq pq$ so we do not have vertices that are both p- and q- connected to each other. We have that $l = nq + r_1 = n(mp + r) + r_1 = mnp + nr + r_1$, so $l \equiv (nr + r_1) \bmod p$. A vertex in the p-class of i has $\left\lfloor \frac{l}{p} \right\rfloor$ p-connections if $i < (nr + r_1) \bmod p$ or $\left\lfloor \frac{l}{p} \right\rfloor - 1$ p-connections if $i \geq (nr + r_1) \bmod p$. Similarly, the number of q-connections for a position in the q-class of j is n if $j < r_1$ or $n-1$ if $j \geq r_1$. The $mp-1$ vertex is in the p-class of $p-1$ so it always has $\left\lfloor \frac{l}{p} \right\rfloor - 1$ p-connections since $p - 1 \geq (nr + r_1) \bmod p$. The $mp-1$ vertex is in the q-class of $mp-1$ and so it has $n-1$ q-connections if $r_1 \leq mp - 1$ and has n q-connections if $mp \leq r_1 < q$. The degree of the $mp-1$ vertex is clearly minimum when $r_1 < mp$, that is, when $l \bmod q < mp$.

However, if $mp \leq r_1 \leq mp + s$ for some $0 \leq s < r$, then the vertices in the q-class of $mp + s$ have one fewer q-connection than any other vertex, and may have the same number of p-connections as the $mp-1$ vertex, giving them a lower degree than the $mp-1$ vertex. These vertices are of the form $(mp + s) + tq = mp + s + t(mp + r) = (t + 1)mp + tr + s$ for some nonnegative integer t satisfying $mp + s + tq \leq l - 1$. Thus a vertex $mp + s + tq$ falls in the p-class of $(tr + s) \bmod p$. Thus, vertices in the q-class of $mp + s$ have $\left\lfloor \frac{l}{p} \right\rfloor$ p-connections if and only if $(tr + s) \bmod p < l \bmod p$ for all integers $t \in \{0, \ldots, n-1\}$ and $s \in \{r_1 - mp, \ldots, r-1\}$. If this is the case then these vertices have one more p-connection than the $mp-1$ vertex and therefore do not have lower degree. Since $t \leq n-1$ and $s \leq r-1$, we have that $tr + s \leq nr - 1$. Note that if $nr - 1 < l \bmod p$, then $(tr+s) \bmod p = (tr+s) < l \bmod p$ for all $t \in \{0, \ldots, n-1\}$ and $s \in \{r_1 - mp, \ldots, r-1\}$. Thus, if $nr - 1 < l \bmod p$, then the $mp-1$ vertex has lowest degree in G. □

The following theorem gives $l_d(h,p,q)$ when the $mp-1$ vertex has the minimum degree in the graph of size $l_d(h,p,q) - 1$.

Theorem 6. *Let $q = mp + r$ where $0 < r < p$. Define $n_1 = \left\lceil \frac{h+3}{1+\frac{p}{q}} \right\rceil$ and $n_2 = \left\lceil \frac{h+3}{1+\frac{q}{p}} \right\rceil$, and define $\omega' = \min\{n_1 p, mp + (n_2 - 1)q\}$. Let $mp \leq \omega' \leq pq$. If $\omega' \bmod q < mp$ or $\lfloor \frac{\omega'}{q} \rfloor r - 1 < \omega' \bmod p$, then $l_d(h,p,q) = \omega'$.*

Proof. Let G denote the (p,q)-periodic graph of size l. If we restrict the size so that $mp \leq l \leq pq$ with $l \bmod q < mp$ or $nr - 1 < l \bmod p$, then by Lemma 2

the vertex $mp-1$ of G has lowest degree. Thus, within these ranges, optimal lengths occur whenever the $mp-1$ vertex gains a p- or q-connection. The $mp-1$ vertex gains a p-connection exactly when $l = n_1 p$ for an integer $n_1 > m$. We can calculate n_1 using Equation (3).

The $mp-1$ position gains a q-connection exactly when $l = mp+n_2' q$. This fits the form described in Remark 2 where $n_1' = m$. After using Equations (3) and 4 to calculate n_1 and n_2, we search for n_2' satisfying $\max\{(n_1-1)p, (n_2-1)q\} < mp + n_2' q < n_1 p$. The optimal length is then $mp + n_2' q$ if and only if such an n_2' exists. Since $mp < q$ and $n_2 q$ is the smallest multiple of q greater than $n_1 p$, any such n_2' satisfying the inequalities must be equal to $n_2 - 1$, where we calculate n_2 using Equation (4). We then know that $mp + n_2' q > \max\{(n_1-1)p, (n_2-1)q\}$, so we can now say that $mp + n_2' q$ is the optimal length if and only if it is less than $n_1 p$. Otherwise, $n_1 p$ is the optimal length. □

Algorithm 2. Find $L(h,p,q)$ when $1 < p < q$ and $\gcd(p,q) = 1$

```
if p = 2 then L(h,p,q) = (2⌊h/q⌋ + 1)q + h mod q + 1 by Thm 1
else
    if q > p⌊(h+1)/2⌋ then L(h,p,q) = p⌊(h+2)/2⌋ + q − (h+1) mod 2 by Thm 2
    else
        compute l_r(h,p,q) using Theorem 3
        compute l_d(h,p,q) using Theorem 4 (and Algorithm 1)
        L(h,p,q) = max{l_r(h,p,q), l_d(h,p,q)}
```

Theorem 7. *Given a number of holes h and two periods p and q, Algorithm 2 computes the optimal length $L(h,p,q)$. Computing $l_d(h,p,q)$ is linear in p and q and constant in h.*

6 Conclusion

Using the ideas of r-set and modified degree connectivities described in this paper, we have been able to answer conjectures in [7] (due to page restrictions however, we cannot provide these results here). Our methods can be used to prove closed formulas for any given number of holes. However, as the number of holes increases, the number of cases also increases. Our calculations show that an r-set disconnection is strictly more efficient than any modified degree disconnection, or $l_r(h,p,q) > l_d(h,p,q)$, if and only if $h = 4$ and $q < \frac{3p}{2}$, in which case, $L(h,p,q) = q + 3p - 1$. For instance, we have proved that if p and q are integers satisfying $2 < p < q$ and $\gcd(p,q) = 1$, then $L(3,p,q)$ is $p + 2q$ if $q < \frac{3p}{2}$, $4p$ if $\frac{3p}{2} < q < 2p$, and $2p + q$ if $q > 2p$. A topic of future research is to extend our approach to any number of periods. Moreover, a World Wide Web server interface has been established at

www.uncg.edu/cmp/research/finewilf4

for automated use of a program which given as input a number of holes h and two periods p and q, outputs $L(h, p, q)$ and an optimal word for that length.

References

1. Choffrut, C., Karhumäki, J.: Combinatorics of Words. In: Rozenberg, G., Salomaa, A. (eds.) Handbook of Formal Languages, vol. 1, pp. 329–438. Springer, Berlin (1997)
2. Smyth, W.F.: Computing Patterns in Strings. Pearson, Addison-Wesley (2003)
3. Fine, N.J., Wilf, H.S.: Uniqueness theorems for periodic functions. Proceedings of the American Mathematical Society 16, 109–114 (1965)
4. Constantinescu, S., Ilie, L.: Generalised Fine and Wilf's theorem for arbitrary number of periods. Theoretical Computer Science 339, 49–60 (2005)
5. Berstel, J., Boasson, L.: Partial words and a theorem of Fine and Wilf. Theoretical Computer Science 218, 135–141 (1999)
6. Blanchet-Sadri, F.: Algorithmic Combinatorics on Partial Words. Chapman & Hall/CRC Press, Boca Raton, FL (2008)
7. Blanchet-Sadri, F., Bal, D., Sisodia, G.: Graph connectivity, partial words, and a theorem of Fine and Wilf. Information and Computation 206(5), 676–693 (2008)
8. Halava, V., Harju, T., Kärki, T.: Interaction properties of relational periods. Discrete Mathematics and Theoretetical Computer Science 10, 87–112 (2008)
9. Shur, A.M., Gamzova, Y.V.: Partial words and the interaction property of periods. Izvestiya Rossiiskoi Akademii Nauk. Seriya Matematicheskaya 68(2), 191–214 (2004)
10. Shur, A.M., Konovalova, Y.V.: On the Periods of Partial Words. In: Sgall, J., Pultr, A., Kolman, P. (eds.) MFCS 2001. LNCS, vol. 2136, pp. 657–665. Springer, Heidelberg (2001)
11. Smyth, W.F., Wang, S.: A new approach to the periodicity lemma on strings with holes. Theoretical Computer Science 410, 4295–4302 (2009)

The 1-Neighbour Knapsack Problem

Glencora Borradaile[1,⋆], Brent Heeringa[2,⋆⋆], and Gordon Wilfong[3]

[1] Oregon State University
glencora@eecs.oregonstate.edu
[2] Williams College
heeringa@cs.williams.edu
[3] Bell Labs
gtw@research.bell-labs.com

Abstract. We study a constrained version of the knapsack problem in which dependencies between items are given by the adjacencies of a graph. In the *1-neighbour knapsack problem*, an item can be selected only if at least one of its neighbours is also selected. We give approximation algorithms and hardness results when the nodes have both uniform and arbitrary weight and profit functions, and when the dependency graph is directed and undirected.

1 Introduction

We consider the knapsack problem in the presence of constraints. The input is a graph $G = (V, E)$ where each vertex v has a *weight* $w(v)$ and a *profit* $p(v)$, and a knapsack of size k. We start with the usual knapsack goal—find a set of vertices of maximum profit whose total weight does not exceed k—and handle the additional requirement that a vertex can be selected only if *at least one* of its neighbours is also selected (vertices with no neighbours can always be selected). We call this the *1-neighbour knapsack problem*. We consider the problem with *general* (arbitrary) and *uniform* ($p(v) = w(v) = 1\ \forall v$) weights and profits, and with undirected and directed graphs. In the case of directed graphs, the neighbour constraint applies to the *out*-neighbours of a vertex.

Constrained knapsack problems have applications to scheduling, tool management, investment strategies and database storage [8,1,7]. There are also applications to network formation. For example, suppose a set of customers $C \subset V$ in a network $G = (V, E)$ wish to connect to a server, represented by a single sink $s \in V$. The server may activate each edge at a cost and each customer would result in a certain profit. The server wishes to activate a subset of the edges with cost within the server's budget. By introducing a vertex mid-edge with zero-profit and weight equal to the cost of the edge and giving each customer zero-weight, we convert this problem to a 1-neighbour knapsack problem.

⋆ Glencora Borradaile is supported by NSF grant CCF-0963921.
⋆⋆ Brent Heeringa is supported by NSF grant IIS-08125414.

C.S. Iliopoulos and W.F. Smyth (Eds.): IWOCA 2011, LNCS 7056, pp. 71–84, 2011.
© Springer-Verlag Berlin Heidelberg 2011

Results. We show that the four resulting problems

$$\{\text{general, uniform}\} \times \{\text{undirected, directed}\}$$

vary in complexity but admit several algorithmic approaches. We summarize our results in Table 1.

Table 1. Our results: upper and lower bounds on the approximation ratios for combinations of {general, uniform} × {undirected, directed}. For uniform, undirected, the bounds are running-times of optimal algorithms.

		Upper	Lower
Uniform	Undirected	linear-time exact	
	Directed	PTAS	NP-hard (strong sense)
General	Undirected	$\frac{(1-\varepsilon)}{2} \cdot (1 - 1/e^{1-\varepsilon})$	$1 - 1/e + \epsilon$
	Directed	*open*	$1/\Omega(\log^{1-\varepsilon} n)$

In Section 2 we describe a greedy algorithm that applies to the general 1-neighbour problem for both directed and undirected dependency graphs. The algorithm requires two oracles: one for finding a set of vertices with high profit and another for finding a set of vertices with high profit-to-weight ratio. In both cases, the total weight of the set cannot exceed the knapsack capacity and the subgraph defined by the vertices must adhere to a strict combinatorial structure which we define later. The algorithm achieves an approximation ratio of $(\alpha/2) \cdot (1 - 1/e^{\beta})$. The approximation ratios of the oracles determine the α and β terms respectively.

For the general, undirected 1-neighbour case, we give polynomial-time oracles that achieve $\alpha = \beta = (1 - \varepsilon)$ for any $\varepsilon > 0$. This yields a polynomial time $((1 - \varepsilon)/2) \cdot (1 - 1/e^{1-\varepsilon})$-approximation. We also show that no approximation ratio better than $1 - 1/e$ is possible (assuming P≠NP). This matches the upper bound up to (almost) a factor of 2. These results appear in Section 2.1.

In Section 2.2, we show that the general, directed 1-neighbour knapsack problem is $1/\Omega(\log^{1-\varepsilon} n)$-hard to approximate, even in DAGs.

In Section 3 we show that the uniform, directed 1-neighbour knapsack problem is NP-hard in the strong sense but that it has a polynomial-time approximation scheme (PTAS)[1]. Thus, as with general, undirected 1-neighbour problem, our upper and lower bounds are essentially matching.

Finally, in Section 4 we show that the uniform, undirected 1-neighbour knapsack problem affords a simple, linear-time solution.

Related Work. There is a tremendous amount of work on maximizing submodular functions under a single knapsack constraint [13], multiple knapsack

[1] A PTAS is an algorithm that, given a fixed constant $\varepsilon < 1$, runs in polynomial time and returns a solution within $1 - \varepsilon$ of optimal. The algorithm may be exponential in $1/\varepsilon$.

constraints [11], and both knapsack and matroid constraints [12,4]. While our profit function is submodular, the constraints given by the graph are not characterized by a matroid (our solutions, for example, are not closed downward). Thus, the 1-neighbour knapsack problem represents a class of knapsack problems with realistic constraints that are not captured by previous work.

As we show in Section 2.1, the general, undirected 1-neighbour knapsack problem generalizes several maximum coverage problems including the budgeted variant considered by Khuller, Moss, and Naor [9] which has a tight $(1 - 1/e)$-approximation unless P=NP. Our algorithm for the general 1-neighbour problem follows the approach taken by Khuller, Moss, and Naor but, because of the dependency graph, requires several new technical ideas. In particular, our analysis of the greedy step represents a non-trivial generalization of the standard greedy algorithm for submodular maximization.

Johnson and Niemi [7] give an FPTAS for knapsack problems on dependency graphs that are in-arborescences (these are directed trees in which every arc is directed toward a single root)[2]. This problem can be viewed as an instance of the general, directed 1-neighbour knapsack problem.

In a longer technical report [2] we explore a version of the constrained knapsack problem where an item may be selected only if *all* its neighbours are selected. This problem generalizes the subset-union knapsack problem (SUKP) [8], the precedence constrained knapsack problem (PCKP) [1], and the partially ordered knapsack problem (POK) [10].

Notation. We consider graphs G with n vertices $V(G)$ and m edges $E(G)$. Whether the graph is directed or undirected will be clear from context. We refer to edges of directed graphs as arcs. For an undirected graph, $N_G(v)$ denotes the neighbours of a vertex v in G. For a directed graph, $N_G(v)$ denotes the out-neighbours of v in G, or, more formally, $N_G(v) = \{u : vu \in E(G)\}$. Given a set of nodes X, $N_G^-(X)$ is the set of nodes not in X but that have a neighbour (or out-neighbour in the directed case) in X. That is, $N_G^-(X) = \{u : uv \in E(G), u \notin X, \text{ and } v \in X\}$. The degree (in undirected graphs) and out-degree (in directed graphs) of a vertex v in G is denoted $\delta_G(v)$. The subscript G will be dropped when the graph is clear from context. For a set of vertices *or* edges U, $G[U]$ is the graph induced on U.

For a directed graph G, $\mathcal{D}$ is the directed, acyclic graph (DAG) resulting from contracting maximal strongly-connected components (SCCs) of G. For each node $u \in V(\mathcal{D})$, let $V(u)$ be the set of vertices of G that are contracted to obtain u.

For convenience, extend any function f defined on items in a set X to any subset $A \subseteq X$ by letting $f(A) = \sum_{a \in A} f(a)$. If $f(a)$ is a set, then $f(A) = \bigcup_{a \in A} f(a)$. If f is defined over vertices, then we extend it to edges: $f(E) = f(V(E))$. For any knapsack problem, OPT is the set of vertices/items in an optimal solution.

[2] In their problem formulation, the constraints are given as an out-arborescences—directed trees in which every arc is directed away from a single root—and feasible solutions are subsets of vertices that are closed under the *predecessor* operation.

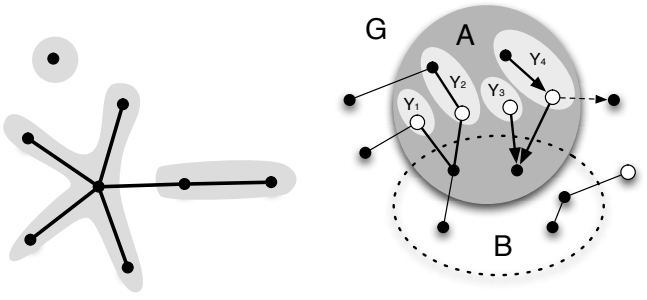


Fig. 1. (a) An undirected graph. If $\mathcal{H}$ is the family of star graphs, then the shaded regions give the only viable partition of the nodes—no other partition yields 1-neighbour sets. However, every *edge* viable with respect to $\mathcal{H}$. The singleton node is also viable since it is a 1-neighbour set for the graph. (b) A graph G with 1-neighbour sets A (dark shaded) and B (dotted). For convenience, we include both directed and undirected edges. The lightly shaded regions give a viable partition for $G[A \setminus B]$ and the white nodes denote $N_G^-(B)$. For the undirected case, Y_2 is viable for $G[A \setminus B]$, and since $|Y_2| = 2$, it is viable for $G[V(G) \setminus B]$. Y_1 is not viable for $G[V(G) \setminus B]$ but it is in $N_G^-(B)$. For the directed case, Y_3 is viable in $G[V(G) \setminus B]$ whereas Y_4 is a viable set only since we consider $G[V(G) \setminus B]$ with the dotted arc removed.

Viable Families and Viable Sets. A set of nodes U is a *1-neighbour set* for G if for every vertex $v \in U$, $|N_{G[U]}(v)| \geq \min\{\delta_G(v), 1\}$. That is, a 1-neighbour set is feasible with respect to the dependency graph. A family of graphs $\mathcal{H}$ is a *viable family* for G if, for any subgraph G' of G, there exists a partition $\mathcal{Y}_\mathcal{H}(G')$ of G' into 1-neighbour sets for G', such that for every $Y \in \mathcal{Y}_\mathcal{H}(G')$, there is a graph $H \in \mathcal{H}$ spanning $G[Y]$. For directed graphs, we take *spanning* to mean that H is a directed subgraph of $G[Y]$ and that Y and H contain the same number of nodes. For a graph G, we call $\mathcal{Y}_\mathcal{H}(G)$ a *viable partition* of G with respect to $\mathcal{H}$.

In Section 2.1 we show that star graphs form a viable family for any undirected dependency graph. That is, we show that any undirected graph can be partitioned into 1-neighbour sets that are stars. Fig. 1 (a) gives an example. In contrast, edges do not form a viable family since, for example, a simple path with 3 nodes cannot be partitioned into 1-neighbour sets that are edges. For DAGs, in-arborescences are a viable family but directed paths are not (consider a directed graph with 3 nodes u, v, w and two arcs (u, v) and (w, v)). Note that a viable family always contains a singleton vertex.

A 1-neighbour set U for G is *viable* with respect to $\mathcal{H}$ if there is a graph $H \in \mathcal{H}$ spanning $G[U]$. Note that the 1-neighbour sets in $\mathcal{Y}_\mathcal{H}(G)$ are, by definition, viable for G, but a viable set for G need not be in $\mathcal{Y}_\mathcal{H}(G)$. For example, if $\mathcal{H}$ is the family of stars and G is the undirected graph in Fig. 1 (a), then any edge is a viable set for G but the only viable partition is the shaded region. Note that if U is a viable set for G then it is also a viable set for any subgraph G' of G provided $U \subseteq V(G')$.

Viable families and viable sets play an essential role in our greedy algorithm for the general 1-neighbour knapsack problem. Viable families establish a set of structures over which our oracles can search. This restriction simplifies both the design and analysis of efficient oracles as well as couples the oracles to a shared family of graphs which, as we'll show later, is essential to our analysis. In essence, viable families provide a mechanism to coordinate the oracles into returning sets with roughly similar structure. Viable sets correctly capture the idea of an indivisible unit of choice in the greedy step. We formalize this with the following lemma which is illustrated in Fig. 1 (b).

Lemma 1. *Let G be a graph and $\mathcal{H}$ be a viable family for G. Let A and B be 1-neighbour sets for G. If $\mathcal{Y}_{\mathcal{H}}(C)$ is a viable partition of $G[C]$ where $C = A \setminus B$ then every set $Y \in \mathcal{Y}_{\mathcal{H}}(C)$ is either (i) a singleton node y such that $y \in N_G^-(B)$ (i.e., y has a neighbour in B), or (ii) a viable set for $G' = G[V(G) \setminus B]$ where, in the case that G is directed, G' contains no arc with a tail in $N_G^-(B)$.*

Proof. Let $\mathcal{Y}_{\mathcal{H}}(C)$ be a viable partition for $G[C]$ where $C = A \setminus B$ and A, B, G, G' and $\mathcal{H}$ are defined as above. If $|Y| = 1$ then let $Y = \{y\}$. If $\delta_G(y) = 0$ then Y is a viable set for G so it is viable set for G'. Otherwise, since A is a 1-neighbour set for G, y must have a neighbour in B so $y \in N_G^-(B)$. If $|Y| > 1$ then, provided G is undirected, Y is also a viable set in G so it is a viable set in G'. If G is directed, then Y may not be viable in G since it might contain a node z that is a sink in $G[C]$ but that is not a sink in G. However, in this case $c \in N_G^-(B)$ so it is a sink in G' since G' contains no arc with a tail in $N_G^-(B)$. Therefore, Y is viable for G'. □

2 The General 1-Neighbour Knapsack Problem

Here we give a greedy algorithm Greedy-1-Neighbour for the general 1-neighbour knapsack problem on both directed and undirected graphs. A formal description of our algorithm is available in Fig. 2. Greedy1-Neighbour relies on two oracles Best-Profit-Viable and Best-Ratio-Viable which find viable sets of nodes with respect to a fixed viable family $\mathcal{H}$. In each iteration i, we call Best-Ratio-Viable which, given the nodes not yet chosen by the algorithm, returns the highest profit-to-weight ratio, viable set S_i with weight not exceeding the remaining capacity. We also consider the set of nodes Z not in the knapsack, but with at least one neighbour already in the knapsack. Let s_i be the node with highest profit-to-weight ratio in Z not exceeding the remaining capacity. We greedily add either s_i or S_i to our knapsack U depending on which has higher profit-to-weight ratio. We continue until we can no longer add nodes to the knapsack.

For a viable family $\mathcal{H}$, if we can efficiently approximate the highest profit-to-weight ratio viable set to within a factor of β and if we can efficiently approximate the highest profit viable set to within a factor of α, then our greedy algorithm yields a polynomial time $\frac{\alpha}{2}(1 - 1/e^{\beta})$-approximation.

GREEDY-1-NEIGHBOUR(G, k) :

$S_{\max}$ = BEST-PROFIT-VIABLE(G, k)
$K = k$, $U = \emptyset$, $i = 1$, $G' = G$, $Z = \emptyset$
WHILE there is either a viable set in G' or a node in Z with weight $\leq K$
  S_i = BEST-RATIO-VIABLE(G', K)
  $s_i = \arg\max\{p(v)/w(v) \mid v \in Z\}$
  IF $p(s_i)/w(s_i) > p(S_i)/w(S_i)$
    $S_i = \{s_i\}$
  $G' = G[V(G') \setminus S_i]$
  $i = i + 1$, $U = U \cup V(S_i)$, $K = K - w(S_i)$
  $Z = N_G^-(U)$
  If G is directed, remove any arc in G' with a tail in Z
RETURN $\arg\max\{p(S_{\max}), p(U)\}$

Fig. 2. The GREEDY-1-NEIGHBOUR algorithm. In each iteration i, we greedily add either the viable set S_i or the node s_i to our knapsack U depending on which has higher profit-to-weight ratio. This continues until we can no longer add nodes to the knapsack.

Theorem 1. GREEDY-1-NEIGHBOUR *is a* $\frac{\alpha}{2}(1-\frac{1}{e^\beta})$*-approximation for the general 1-neighbour problem on directed and undirected graphs.*

Proof. Let OPT be the set of vertices in an optimal solution. In addition, let $U_i = \cup_{j=1}^{i} V(S_j)$ correspond to U after the first i iterations where $U_0 = \emptyset$. Let $\ell + 1$ be the first iteration in which there is either a node in $Z \cap \text{OPT}$ or a viable set in $\text{OPT} \setminus U_\ell$ whose profit-to-weight ratio is larger than $S_{\ell+1}$. Of these, let $\mathcal{S}_{\ell+1}$ be the node or set with highest profit-per-weight. For convenience, let $\mathcal{S}_i = S_i$ and $\mathcal{U}_i = U_i$ for $i = 1 \ldots \ell$, and $\mathcal{U}_{\ell+1} = \mathcal{U}_\ell \cup \mathcal{S}_{\ell+1}$. Notice that $\mathcal{U}_\ell$ is a feasible solution to our problem but that $\mathcal{U}_{\ell+1}$ is not since it contains $\mathcal{S}_{\ell+1}$ which has weight exceeding K. We *analyze* our algorithm with respect to $\mathcal{U}_{\ell+1}$.

Lemma 2. *For each iteration* $i = 1, \ldots, \ell + 1$*, the following holds:*

$$p(\mathcal{S}_i) \geq \beta \frac{w(\mathcal{S}_i)}{k} \left(p(\text{OPT}) - p(\mathcal{U}_{i-1})\right)$$

Proof. Fix an iteration i and let I be the graph induced by $\text{OPT} \setminus \mathcal{U}_{i-1}$. Since both OPT and $\mathcal{U}_{i-1}$ are 1-neighbour sets for G, by Lemma 1, each $Y \in \mathcal{Y}_{\mathcal{H}}(I)$ is either a viable set for G' (so it can be selected by BEST-RATIO-VIABLE) or a singleton vertex in $N_G^-(\mathcal{U}_{i-1})$ (which GREEDY-1-NEIGHBOUR always considers). Thus, if $i \leq \ell$, then by the greedy choice of the algorithm and approximation ratio of BEST-RATIO-VIABLE we have

$$\frac{p(\mathcal{S}_i)}{w(\mathcal{S}_i)} \geq \beta \frac{p(Y)}{w(Y)} \text{ for all } Y \in \mathcal{Y}_{\mathcal{H}}(I). \tag{1}$$

If $i = \ell + 1$ then $p(\mathcal{S}_{\ell+1})/w(\mathcal{S}_{\ell+1})$ is, by definition, at least as large as the profit-to-weight ratio of any $Y \in \mathcal{Y}$. It follows that for $i = 1, \ldots, \ell + 1$

$$
\begin{aligned}
p(\mathrm{OPT}) - p(\mathcal{U}_{i-1}) = \sum_{u \in V(I)} p(u) &\leq \frac{1}{\beta} \frac{p(\mathcal{S}_i)}{w(\mathcal{S}_i)} \sum_{u \in V(I)} w(u), \text{ by Eq. 1} \\
&\leq \frac{1}{\beta} \frac{p(\mathcal{S}_i)}{w(\mathcal{S}_i)} w(\mathrm{OPT}), \text{ since } I \text{ is a subset of OPT} \\
&\leq \frac{1}{\beta} \frac{k}{w(\mathcal{S}_i)} p(\mathcal{S}_i), \text{ since } w(\mathrm{OPT}) \leq k
\end{aligned}
$$

Rearranging gives Lemma 2. □

Lemma 3. *For $i = 1, \ldots, \ell + 1$, the following holds:*

$$
p(\mathcal{U}_i) \geq \left[1 - \prod_{j=1}^{i} \left(1 - \beta \frac{w(\mathcal{S}_j)}{k} \right) \right] p(\mathrm{OPT})
$$

Proof. We prove the lemma by induction on i. For $i = 1$, we need to show that

$$
p(\mathcal{U}_1) \geq \beta \frac{w(\mathcal{S}_1)}{k} p(\mathrm{OPT}). \tag{2}
$$

This follows immediately from Lemma 2 since $p(\mathcal{U}_0) = 0$ and $\mathcal{U}_1 = \mathcal{S}_1$. Suppose the lemma holds for iterations 1 through $i - 1$. Then it is easy to show that the inequality holds for iteration i by applying Lemma 2 and the inductive hypothesis. This completes the proof of Lemma 3. □

We are now ready to prove Theorem 1. Starting with the inequality in Lemma 3 and using the fact that adding $\mathcal{S}_{\ell+1}$ violates the knapsack constraint (so $w(\mathcal{U}_{\ell+1}) > k$) we have

$$
\begin{aligned}
p(\mathcal{U}_{\ell+1}) &\geq \left[1 - \prod_{j=1}^{\ell+1} \left(1 - \beta \frac{w(\mathcal{S}_j)}{k} \right) \right] p(\mathrm{OPT}) \\
&\geq \left[1 - \prod_{j=1}^{\ell+1} \left(1 - \beta \frac{w(\mathcal{S}_j)}{w(U_{\ell+1})} \right) \right] p(\mathrm{OPT}) \\
&\geq \left[1 - \left(1 - \frac{\beta}{\ell + 1} \right)^{\ell+1} \right] p(\mathrm{OPT}) \geq \left(1 - \frac{1}{e^{\beta}} \right) p(\mathrm{OPT})
\end{aligned}
$$

where the penultimate inequality follows because equal $w(\mathcal{S}_j)$ maximize the product. Since $S_{\max}$ is within a factor of α of the maximum profit viable set of weight $\leq k$ and $\mathcal{S}_{\ell+1}$ is contained in OPT, $p(S_{\max}) \geq \alpha \cdot p(\mathcal{S}_{\ell+1})$. Thus, we have $p(U) + p(S_{\max})/\alpha \geq p(\mathcal{U}_\ell) + p(\mathcal{S}_{\ell+1}) = p(\mathcal{U}_{\ell+1}) \geq \left(1 - \frac{1}{e^{\beta}}\right) p(\mathrm{OPT})$. Therefore $\max\{p(U), p(S_{\max})\} \geq \frac{\alpha}{2} \left(1 - \frac{1}{e^{\beta}}\right) p(\mathrm{OPT})$.

□

2.1 The General, Undirected 1-Neighbour Problem

Here we formally show that stars are a viable family for undirected graphs and describe polynomial-time implementations of Best-Profit-Viable and Best-Ratio-Viable that operate with respect to stars. Both oracles achieve an approximation ratio of $(1-\varepsilon)$ for any $\varepsilon > 0$. Combined with Greedy-1-Neighbour this yields a polynomial time $((1-\varepsilon)/2)\cdot(1-1/e^{1-\varepsilon})$-approximation for the general, undirected 1-neighbour problem. In addition, we show that this approximation is nearly tight by showing that the general, undirected 1-neighbour problem generalizes many coverage problems including the max k-cover and budgeted maximum coverage, neither of which have a $(1-1/e+\epsilon)$-approximation for any $\epsilon > 0$ unless P=NP.

Stars. For the rest of this section, we assume $\mathcal{H}$ is the family of star graphs (*i.e.* graphs composed of a center vertex u and a (possibly empty) set of edges all of which have u as an endpoint) so that given a graph G and a capacity k, Best-Profit-Viable returns the highest profit, viable star with weight at most k and Best-Ratio-Viable returns the highest profit-to-weight, viable star with weight at most k.

Lemma 4. *The nodes of any undirected constraint graph G can be partitioned into 1-neighbour sets that are stars.*

Proof. Let G_i be an arbitrary connected component of G. If $|V(G_i)| = 1$ then $V(G_i)$ is trivially a 1-neighbour set and the trivial star consisting of a single node is a spanning subgraph of G_i. If G_i is non-trivial then let T be any spanning tree of G_i and consider the following algorithm: while T contains a path P with $|P| > 2$, remove an interior edge of P from T. When the algorithm finishes, each path has at least one edge and at most two edges, so T is a set of non-trivial stars, each of which is a 1-neighbour set. □

Best-Profit-Viable. Finding the maximum profit, viable star of a graph G subject to a knapsack constraint k reduces to the traditional unconstrained knapsack problem which has a well-known FPTAS that runs in $O(n^3/\varepsilon)$ time [6,14]. Every vertex $v \in V(G)$ defines a knapsack problem: the items are $N_G(v)$ and the capacity is $k - w(v)$. Combining v with the solution returned by the FPTAS yields a candidate star. We consider the candidate star for each vertex and return the one with highest profit. Since we consider all possible star centers, Best-Profit-Viable runs in $O(n^4/\varepsilon)$ time and returns a viable star within a factor of $(1-\varepsilon)$ of optimal, for any $\varepsilon > 0$.

Best-Ratio-Viable. We again turn to the FPTAS for the standard knapsack problem. Our goal is to find a high profit-to-weight star in G with weight at most k. The standard FPTAS for the unconstrained knapsack problem builds a dynamic programing table T with n rows and nP' columns where n is the number of available items and P' is the maximum adjusted profit over all the items. Given an item v, its adjusted profit is $p'(v) = \lfloor \frac{p(v)}{(\varepsilon/n)\cdot P} \rfloor$ where P is

the true maximum profit over all the items. Each entry $T[i,p]$ gives the weight of the minimum weight subset over the first i items achieving profit p. An auxiliary data structure allows us to efficiently retrieve the corresponding subset.

Notice that, for any fixed profit p, $p/T[n,p]$ is the highest profit-to-weight ratio for that p. Therefore, for $1 \leq p \leq nP'$, the p maximizing $p/T[n,p]$ gives the highest profit-to-weight ratio of any feasible subset provided $T[n,p] \leq k$. Let S be this subset. We will show that $p(S)/w(S)$ is within a factor of $(1-\varepsilon)$ of OPT where OPT is the profit-to-weight ratio of the highest profit-to-weight ratio feasible subset S^*.

Letting $r(v) = p(v)/w(v)$ and $r'(v) = p'(v)/w(v)$, and following [14], we have

$$r(S^*) - ((\varepsilon/n) \cdot P) \cdot r'(S^*) \leq \varepsilon P/w(S^*)$$

since, for any item v, the difference between $p(v)$ and $((\varepsilon/n) \cdot P) \cdot p'(v)$ is at most $(\varepsilon/n) \cdot P$ and we can fit at most n items in our knapsack. Because $r'(S) \geq r'(S^*)$ and OPT is at least $P/w(S^*)$ we have

$$r(S) \geq (\varepsilon/n) \cdot P \cdot r'(S^*) \geq r(S^*) - \varepsilon P/w(S^*) \geq \text{OPT} - \varepsilon\text{OPT} = (1-\varepsilon)\text{OPT}.$$

Now, just as with Best-Profit-Viable, every vertex $v \in V(G)$ defines a knapsack instance where $N_G(V)$ is the set of items and $k - w(v)$ is the capacity. We run the modified FTPAS for knapsack on the instance defined by v and add v to the solution to produce a set of candidate stars. We return the star with highest profit-to-weight ratio. Since we consider all possible star centers, Best-Ratio-Viable runs in $O(n^4/\varepsilon)$ time and returns a viable star within a factor of $(1-\varepsilon)$ of optimal, for any $\varepsilon > 0$.

Why Stars? Besides some isolated vertices, our solution is a set of edges, but the edges are not necessarily vertex disjoint. Analyzing our greedy algorithm in terms of edges risks counting vertices multiple times. Partitioning into stars allows us to charge increases in the profit from the greedy step without this risk. In fact, stars are essentially the *simplest* structure meeting this requirement which is why we use them as our viable family.

General, Undirected 1-Neighbour Knapsack is APX-Complete. Here we show that it is NP-hard to approximate the general, undirected 1-neighbour knapsack problem to within a factor better than $1 - 1/e + \epsilon$ for any $\epsilon > 0$ via an approximation-preserving reduction from max k-cover [3]. An instance of max k-cover is a set cover instance $(S, \mathcal{R})$ where S is a ground set of n items and $\mathcal{R}$ is a collection of subsets of S. The goal is to cover as many items in S using at most k subsets from $\mathcal{R}$.

Theorem 2. *The general, undirected 1-neighbour knapsack problem has no* $1 - 1/e + \epsilon$*-approximation for any* $\epsilon > 0$ *unless P=NP.*

Proof. Given an instance of $(S, \mathcal{R})$ of max k-cover, build a bipartite graph $G = (U \cup V, E)$ where U has a node u_i for each $s_i \in S$ and V has a node v_j for each

set $R_j \in \mathcal{R}$. Add the edge $\{u_i, v_j\}$ to E if and only if $u_i \in R_j$. Assign profit $p(u_i) = 1$ and weight $w(u_i) = 0$ for each vertex $u_i \in U$ and profit $p(v_j) = 0$ and weight $w(u_i) = 1$ for each vertex $v_j \in V$. Since no pair of vertices in U have an edge and since every vertex in U has no weight, our strategy is to pick vertices from V and all their neighbours in U. Since every vertex of U has unit profit, we should choose the k vertices from V which collectively have the most neighbours. This is exactly the max k-cover problem. □

The max k-cover problem represents a class of *budgeted maximum coverage* (BMC) problems where the elements in the base set have unit profit (referred to as weights in [9]) and the cover sets have unit weight (referred to as costs in [9]). In fact, one can use the above reduction to represent an arbitrary BMC instance: form the same bipartite graph, assign the element weights in BMC as vertex profits in U, and finally assign the covering set costs in BMC as vertex weights in V.

2.2 General, Directed 1-Neighbour Knapsack Is Hard to Approximate

Here we consider the 1-neighbour knapsack problem where G is directed and has arbitrary profits and weights. We show via a reduction from *directed Steiner tree* (DST) that the general, directed 1-neighbour problem is hard to approximate within a factor of $1/\Omega(\log^{1-\varepsilon} n)$. Our result holds for DAGs. Because of this negative result, we also don't expect that good approximations exist for either Best-Profit-Viable and Best-Ratio-Viable for any family of viable graphs.

In the DST problem on DAGs we are given a DAG $G = (V, E)$ where each arc has an associated cost, a subset of t vertices called *terminals* and a root vertex $r \in V$. The goal is to find a minimum cost set of arcs that together connect r to all the terminals (*i.e.*, the arcs form an out-arborescence rooted at r). For all $\varepsilon > 0$, DST admits no $\log^{2-\varepsilon} n$-approximation algorithm unless $NP \subseteq ZTIME[n^{\text{poly}\log n}]$ [5]. This result holds even for very simple DAGs such as *leveled DAGs* in which r is the only root, r is at level 0, each arc goes from a vertex at level i to a vertex at level $i + 1$, and there are $O(\log n)$ levels. We use leveled DAGs in our proof of the following theorem.

Theorem 3. *The general, directed 1-neighbour knapsack problem is* $1/\Omega(\log^{1-\varepsilon} n)$*-hard to approximate unless* $NP \subseteq ZTIME[n^{\text{poly}\log n}]$.

Proof. Let D be an instance of DST where the underlying graph G is a leveled DAG with a single root r. Suppose there is a solution to D of cost C.

Claim. If there is an α-approximation algorithm for the general, directed 1-neighbour knapsack problem then a solution to D with cost $O(\alpha \log t) \times C$ can be found where t is the number of terminals in D.

Proof. Let $G = (V, A)$ be the DAG in instance D. We modify it to $G' = (V', A')$ where we split each arc $e \in A$ by placing a dummy vertex on e with weight equal

to the cost of e according to D and profit of 0. In addition, we also reverse the orientation of each arc. Finally, all other vertices are given weight 0 and terminals are assigned a profit of 1 while the non-terminal vertices of G are given a profit of 0. We create an instance N of the general, directed 1-neighbour knapsack problem consisting of G' and budget bound of C. By assumption, there is a solution to N with cost C and profit t. Therefore given N, an α-approximation algorithm would produce a set of arcs whose weight is at most C and includes at least t/α terminals. That is, it has a profit of at least t/α. Set the weights of dummy nodes to 0 on the arcs used in the solution. Then for all terminals included in this solution, set their profit to 0 and repeat. Standard set-cover analysis shows that after $O(\alpha \log t)$ repetitions, each terminal will have been connected to the root in at least one of the solutions. Therefore the union of all the arcs in these solutions has cost at most $O(\alpha \log t) \times C$ and connects all terminals to the root. □

Using the above claim, we'll show that if there is an α-approximation algorithm for the general, directed-1-neighbour problem then there is an $O(\alpha \log t)$-approximation algorithm for DST which implies the theorem. Let L be the total cost of the arcs in the instance of DST. For each $2^i < L$, take $C = 2^i$ and perform the procedure in the previous claim for $\alpha \log t$ iterations. If after these iterations all terminals are connected to the root then call the cost of the resulting arcs a valid cost. Finally, choose the smallest valid cost, say C' and C' will be no more than $2C_{\mathrm{OPT}}$ where C_{OPT} is the optimal cost of a solution for the DST instance. By the previous claim we have a solution whose cost is at most $2C_{\mathrm{OPT}} \times O(\alpha \log t)$. □

3 The Uniform, Directed 1-Neighbour Knapsack Problem

In this section, we give a PTAS for the uniform, directed 1-neighbour knapsack problem. We rule out an FPTAS by proving the following theorem, the proof of which appears in [2].

Theorem 4. *The uniform, directed 1-neighbour problem is strongly NP-hard.*

A PTAS for the Uniform, Directed 1-Neighbour Problem. Let U be a 1-neighbour set. Let A_U be a minimal set of arcs of G such that for every vertex $u \in U$, $\delta_{G[A_U]}(u) \geq \min\{\delta_G(u), 1\}$. That is, A_U is a *witness* to the feasibility of U as a 1-neighbour set. Since each node of U in $G[A_U]$ has out-degree 0 or 1, the structure of A_U has the following form.

Property 1. Each connected component of $G[A_U]$ is a cycle C and a collection of vertex-disjoint in-arborescences, each rooted at a node of C. C may be trivial, i.e., C may be a single vertex v, in which case $\delta_G(v) = 0$.

For a strongly connected component X, let $c(X)$ be the size of the shortest directed cycle in X with $c(X) = 1$ if and only if $|X| = 1$.

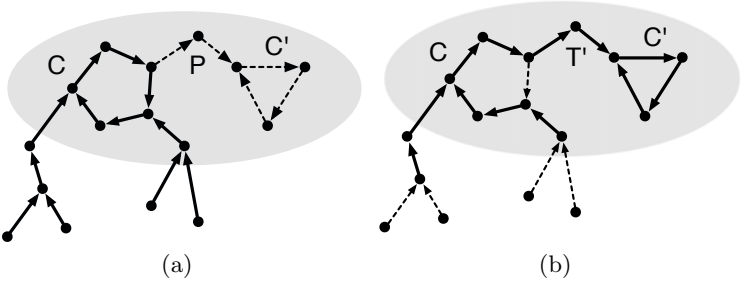


Fig. 3. Construction of a witness containing the smallest cycle of an SCC. The shaded region highlights the vertices of an SCC (edges not in C, C', or P are not depicted). The edges of the witness are solid. (a) The smallest cycle C' is not in the witness. (b) By removing an edge from C and leaf edges from the in-arborescences rooted on C, we create a witness that includes the smallest cycle C'.

Lemma 5. *There is an optimal 1-neighbour knapsack U and a witness A_U such that for each non-trivial, maximal SCC K of G, there is at most one cycle of A_U in K and this cycle is a smallest cycle of K.*

Proof. First we modify A_U so that it contains smallest cycles of maximal SCCs. We rely heavily on the structure of A_U guaranteed by Property 1. The idea is illustrated in Fig. 3.

Let C be a cycle of A_U and let K be the maximal SCC of G that contains C. Suppose C is not the smallest cycle of K or there is more than one cycle of A_U in K. Let H be the connected component of A_U containing C. Let C' be a smallest cycle of K. Let P be the shortest directed path from C to C'. Since C and C' are in a common SCC, P exists. Let T be an in-arborescence in G spanning P, C and H rooted at a vertex of C'.

Some vertices of $C' \cup P$ might already be in the 1-neighbour set U: let X be these vertices. Note that X and $V(H)$ are disjoint because of Property 1. Let T' be a sub-arborescence of T such that:

- T' has the same root as T, and
- $|V(T' \cup C') \cup X| = |V(H)| + |X|$.

Since $|V(T \cup C')| = |V(P \cup H \cup C')| \geq |V(H)| + |X|$ and $T \cup C'$ is connected, such an in-arborescence exists.

Let $B = (A_U \setminus H) \cup T' \cup C'$. Let B' be a witness spanning $V(B)$ contained in B that contains the arcs in C'. We have that B' has $|U|$ vertices and contains a smallest cycle of K.

We repeat this procedure for any SCC in our witness that contains a cycle of a maximal SCC of G that is not smallest or contains two cycles of a maximal SCC. □

To describe the algorithm, let $\mathcal{D} = (S, F)$ be the DAG of maximal SCCs of G and let $\varepsilon > 1/k$ be a fixed constant where k is the knapsack bound. (If $\varepsilon \leq 1/k$ then

the brute force algorithm which considers all subsets $V' \subseteq V(G)$ with $|V'| \leq k$ yields an acceptable bound for a PTAS.)

We say that $u \in S$ is *large* if $c(u) > \varepsilon\, k$, *petite* if $1 < c(u) \leq \varepsilon\, k$, or *tiny* if $c(u) = 1$. Let L, P, and T be the set of all large, petite and tiny SCCs respectively. Note that since $\varepsilon > 1/k$, for every $u \in L$, $c(u) > \varepsilon\, k > 1$.

UNIFORM-DIRECTED-1-NEIGHBOUR

$B = \emptyset$
For every subset $X \subseteq L$ such that $|X| \leq 1/\varepsilon$
  $D_X = \mathcal{D}[P \cup X]$.
  $Z = \{\text{tiny sinks of } \mathcal{D}\} \cup \{\text{petite sinks of } D_X\}$
  $P' = $ any maximal subset of Z such that $c(P') + c(X) \leq k$.
  $U = \bigcup_{K \in P' \cup X} \{V(C) \ : \ C \text{ is a smallest cycle of } K\}$
  Greedily add vertices to U such that U remains a 1-neighbour set until there are no more vertices to add or $|U| = k$. (Via a backwards search rooted at U.)
  $B = \arg\max\{|B|, |U|\}$
Return B.

Theorem 5. UNIFORM-DIRECTED-1-NEIGHBOUR *is a PTAS for the uniform, directed 1-neighbour knapsack problem.*

Proof. Let U^* be an optimal 1-neighbour knapsack and let A_{U^*} be its witness as guaranteed by Lemma 5. Let $\mathcal{L}, \mathcal{P}$, and $\mathcal{T}$ be the sets of large, petite, and tiny cycles in A_{U^*} respectively. By Lemma 5, each of these cycles is in a different maximal SCC and each cycle is a smallest cycle in its maximal SCC.

Let $\mathcal{L} = \{L_1, \ldots, L_\ell\}$ and let L^* be the set of large SCCs that intersect $L_1, \ldots, L_\ell$. Note that $|L^*| = \ell$. Since $k \geq |U^*| \geq \sum_{i=1}^{\ell} |L_i| > \ell\, \varepsilon\, k$ we have $\ell < 1/\varepsilon$. So, in some iteration of UNIFORM-DIRECTED-1-NEIGHBOUR, $X = L^*$. We analyze this iteration of the algorithm. There are two cases:

$P' = Z$. First we show that every vertex in U^* has a descendant in $X \cup P'$. Clearly if a vertex of U^* has a descendant in some $L_i \in \mathcal{L}$, it has a descendant in X. Suppose a vertex of U^* has a descendant in some $P_i \in \mathcal{P}$. P_i is within an SCC of D_X, and so it must have a descendant that is in a sink of D_X. Similarly, suppose a vertex of U^* has a descendant in some $T_i \in \mathcal{T}$. T_i is either a sink in $\mathcal{D}$ or has a descendant that is either a sink of $\mathcal{D}$ or a sink of D_X. All these sinks are contained in $X \cup P'$. Since every vertex of U^* can reach a vertex in $X \cup P'$, greedily adding to this set results in $|U| = |U^*|$ and the result of UNIFORM-DIRECTED-1-NEIGHBOUR is optimal.

$P' \neq Z$. For any sink $x \notin P'$, $c(P') + c(X) + c(x) > k$ but $c(x) \leq \varepsilon\, k$ by the definition of tiny and petite. So, $|U| \geq c(P') + c(X) > (1 - \varepsilon)k$, and the resulting solution is within $(1 - \varepsilon)$ of optimal.

The running time of UNIFORM-DIRECTED-1-NEIGHBOUR is $n^{O(1/\varepsilon)}$. It is dominated by the number of iterations, each of which can be executed in poly time. □

4 The Uniform, Undirected 1-Neighbour Problem

As our final result, we note that there is a relatively straightforward linear time algorithm for finding an optimal solution for instances of the uniform, undirected 1-neighbour knapsack problem. The algorithm essentially breaks the graph into connected components and then, using a counting argument, builds an optimal solution from the components. A proof of the following theorem appears in [2].

Theorem 6. *The uniform, undirected case has a linear-time solution.*

Acknowledgments. We thank Anupam Gupta for helpful discussions in showing hardness of approximation for general, directed 1-neighbour knapsack.

References

1. Boland, N., Fricke, C., Froyland, G., Sotirov, R.: Clique-based facets for the precedence constrained knapsack problem. Technical report. Tilburg University Repository, Netherlands (2005), `http://arno.uvt.nl/oai/wo.uvt.nl.cgi`
2. Borradaile, G., Heeringa, B., Wilfong, G.: The knapsack problem with neighbour constraints. CoRR, abs/0910.0777 (2011)
3. Feige, U.: A threshold of $\ln n$ for approximating set cover. J. ACM 45(4), 634–652 (1998)
4. Goundan, P.R., Schulz, A.S.: Revisiting the greedy approach to submodular set function maximization (2009) (preprint)
5. Halperin, E., Krauthgamer, R.: Polylogarithmic inapproximability. In: Proceedings of STOC, pp. 585–594 (2003)
6. Ibarra, O.H., Kim, C.E.: Fast approximation algorithms for the knapsack and sum of subset problems. J. ACM 22, 463–468 (1975)
7. Johnson, D.S., Niemi, K.A.: On knapsacks, partitions, and a new dynamic programming technique for trees. Mathematics of Operations Research, 1–14 (1983)
8. Kellerer, H., Pferschy, U., Pisinger, D.: Knapsack Problems. Springer, Heidelberg (2004)
9. Khuller, S., Moss, A., Naor(Seffi), J.: The budgeted maximum coverage problem. Inf. Process. Lett. 70(1), 39–45 (1999)
10. Kolliopoulos, S.G., Steiner, G.: Partially ordered knapsack and applications to scheduling. Discrete Applied Mathematics 155(8), 889–897 (2007)
11. Kulik, A., Shachnai, H., Tamir, T.: Maximizing submodular set functions subject to multiple linear constraints. In: Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2009, pp. 545–554. Society for Industrial and Applied Mathematics, Philadelphia (2009)
12. Lee, J., Mirrokni, V.S., Nagarajan, V., Sviridenko, M.: Non-monotone submodular maximization under matroid and knapsack constraints. In: Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, pp. 323–332. ACM, New York (2009)
13. Sviridenko, M.: A note on maximizing a submodular set function subject to a knapsack constraint. Operations Research Letters 32(1), 41–43 (2004)
14. Vazirani, V.: Approximation Algorithms. Springer, Berlin (2001)

A Golden Ratio Parameterized Algorithm for Cluster Editing

Sebastian Böcker

Lehrstuhl für Bioinformatik, Friedrich-Schiller-Universität Jena, Ernst-Abbe-Platz 2, 07743 Jena, Germany
sebastian.boecker@uni-jena.de

Abstract. The Cluster Editing problem asks to transform a graph by at most k edge modifications into a disjoint union of cliques. The problem is NP-complete, but several parameterized algorithms are known. We present a novel search tree algorithm for the problem, which improves running time from $O^*(1.76^k)$ to $O^*(1.62^k)$. In detail, we can show that we can always branch with branching vector $(2, 1)$ or better, resulting in the golden ratio as the base of the search tree size. Our algorithm uses a well-known transformation to the integer-weighted counterpart of the problem. To achieve our result, we combine three techniques: First, we show that zero-edges in the graph enforce structural features that allow us to branch more efficiently. Second, by repeatedly branching we can isolate vertices, releasing costs. Finally, we use a known characterization of graphs with few conflicts.

1 Introduction

Given an undirected graph G, the Cluster Editing problem asks for a minimal set of edge modifications such that the resulting graph is a vertex-disjoint union of cliques. In the corresponding Integer-Weighted Cluster Editing problem, we are given modification costs for each edge or non-edge, and we search for a set of edge modifications with minimum total weight. Here, one assumes that all edges have non-zero modification cost.

In application, the above task corresponds to clustering objects, that is, partitioning a set of objects into homogeneous and well-separated subsets. Similar objects are connected by an edge, and a cluster is a clique of the input graph. The input graph is corrupted and we have to clean (edit) the graph to reconstruct the clustering under the parsimony criterion. Clustering data still represents a key step of numerous life science problems. The weighted variant of the Cluster Editing problem has been frequently proposed for clustering biological entities such as proteins [18].

The Cluster Editing problem is NP-hard [13]. The parameterized complexity of Cluster Editing, using the number of edge modifications as parameter k, is well-studied, see also the FPT races column in [17]. A first algorithm with running time $O^*(2.27^k)$ [10] was improved to $O^*(1.92^k)$ by an extensive case analysis [9]. By transforming the problem to the integer-weighted

C.S. Iliopoulos and W.F. Smyth (Eds.): IWOCA 2011, LNCS 7056, pp. 85–95, 2011.
© Springer-Verlag Berlin Heidelberg 2011

variant, running time was advanced to $O^*(1.82^k)$ [1]. Using a characterization of graphs that do not contain many conflicts, results in the currently fastest algorithm with running time $O^*(1.76^k)$ [3]. There exist linear problem kernels for the unweighted [5] and the integer-weighted variant [4]. Recently, Cluster Editing with "don't care edges" (that is, edges whose modification cost is zero) has been shown to be fixed-parameter tractable [14]. To find exact solutions in practice, a combination of data reduction and Integer Linear Programming proved to be very efficient [2].

Our contributions. We present a new search tree algorithm for Cluster Editing with running time $O(1.62^k + k^2 + m + n)$ for m edges and n vertices, being the fastest known for the problem. The algorithm itself is rather simple, and is based on the merge branching introduced in [1]. We stress that our result only holds for the unweighted Cluster Editing problem, as general integer-weighted instances will not satisfy the "parity property" introduced below.

2 Preliminaries

A problem with input size n and parameter k is *fixed-parameter tractable* (FPT) if it can be solved in $O(f(k) \cdot p(n))$ time where f is any computable function and p is a polynomial. We naturally focus on the $f(k)$ factor, and sometimes adopt the $O^*(f(k))$ notation that suppresses polynomial factors. For a general introduction we refer to [7,15]; in particular, we assume familiarity with bounded search trees, branching vectors, and branching numbers. In the following, let n be the number of vertices, and k the number of edge modifications.

For brevity, we write uv as shorthand for an unordered pair $\{u, v\} \in \binom{V}{2}$. Let $s : \binom{V}{2} \to \mathbb{Z}$ be a *weight function* that encodes the input graph: For $s(uv) > 0$ a pair uv is an edge of the graph and has deletion cost $s(uv)$, while for $s(uv) < 0$, the pair uv is not an edge (a *non-edge*) of the graph and has insertion cost $-s(uv)$. Let $N(u)$ be the set of all vertices $v \in V$ such that $s(uv) > 0$. If $s(uv) = 0$, we call uv a *zero-edge*. We require that there are no zero-edges in the input graph. Nonetheless, zero-edges can appear in the course of computation and require additional attention when analyzing the algorithm.

When analyzing connected components we only consider edges of the graph. We say that $C \subseteq V$ is a *clique* in an integer-weighted graph if all pairs $uv \in \binom{C}{2}$ are edges. If all vertex pairs of a connected component are either edges or zero-edges, we call it a *weak clique*. Vertices uvw form a *conflict triple* in an integer-weighted graph if uv and vw are edges but uw is either a non-edge or a zero-edge. We distinguish two types of conflict triples uvw: if uw has weight zero then the conflict triple is called *weak*, whereas if uw is a non-edge then the conflict triple is called *strong*. If the integer-weighted graph contains no conflict triples then it is *transitive*, i.e. a disjoint union of weak cliques. But the converse is obviously not true, as the example of a single weak conflict triple shows: This graph is a weak clique but contains a (weak) conflict triple. To solve Weighted Cluster Editing we first identify all connected components of the input graph and calculate the best solutions for each component separately, because an

optimal solution never connects disconnected components. Furthermore, if the graph is decomposed during the course of the algorithm, then we recurse and treat each connected component individually.

An unweighted Cluster Editing instance can be encoded by assigning weights $s(uv) \in \{+1, -1\}$. In the resulting graph, all conflict triples are strong. During data reduction and branching, we may set pairs uv to "forbidden" or "permanent". Permanent edges can be merged immediately: *Merging* uv means replacing the vertices u and v with a single vertex u', and, for all vertices $w \in V \setminus \{u, v\}$, replacing pairs uw, vw with a single pair $u'w$. In this context, we say that we *join* vertex pairs uw and vw. The weight of the joined pair is $s(u'w) = s(uw) + s(vw)$. In case one of the pairs is an edge while the other is a non-edge, then we can decrease parameter k by $\min\{|s(uw)|, |s(vw)|\}$. Note that we may join any combination of two edges, non-edges, or zero-edges when merging two vertices. We stress that joined pairs can be zero-edges.

We encode a *forbidden pair* uv by setting $s(uv) = -\infty$. By definition, every forbidden pair uv is a non-edge, since $s(uv) < 0$. A forbidden pair uw can be part of a conflict triple uvw, which then is a strong conflict triple. Assume that we join pairs uv and uw where uw is forbidden and, hence, a non-edge. From the above definition, the resulting pair $u'w$ is forbidden, too, as $s(u'w) = s(uw) + s(vw) = -\infty + s(vw) = -\infty$ holds for all $s(vw) \in \mathbb{R} \cup \{-\infty\}$. Finally, if uw is forbidden and vw is an edge then k is decreased by $\min\{\infty, |s(vw)|\} = s(vw)$.

The following branching was proposed in [1]: We *branch on an edge* uv by recursively calling the algorithm two times, either removing uv and setting it to forbidden, or merging uv. If uv is part of at least one strong conflict triple, then merging uv will generate cost: As there is both an edge uw and a non-edge vw, we can reduce k by $\min\{s(uw), -s(vw)\}$. In case $s(uw) = -s(vw)$, joining uw and vw into $u'w$ results in $u'w$ being a zero-edge. At a later stage of the algorithm, this would prevent us from decreasing our parameter when joining the zero-edge $u'w$. To circumvent this problem, the following bookkeeping trick was introduced in [1]: We assume that joining uw and vw with $s(uw) = -s(vw)$ only reduces the parameter by $\min\{s(uw), -s(vw)\} - \frac{1}{2} = |s(uw)| - \frac{1}{2} \geq \frac{1}{2}$. If at a later stage we join this zero-edge with another pair, we decrease our parameter by the remaining $\frac{1}{2}$. So, both generating and destroying a zero-edge generates cost of at least $\frac{1}{2}$. Note that joining with a forbidden pair cannot create a zero-edge.

Assume that $s(vw) = -s(uw)$ with $|s(vw)| = |s(uw)| \geq 2$. Then, merging an edge uv in a conflict triple uvw will also generate a zero-edge, and generates cost of at least $\frac{3}{2}$. In our analysis, we sometimes concentrate on the case that $s(vw) = -s(uw) = \pm 1$, where merging uv has cost $\frac{1}{2}$. We do so only if it is absolutely obvious that $|s(vw)| = |s(uw)| \geq 2$ will result in the desired branching vector.

Our fixed-parameter algorithms require a cost limit k: In case a solution with cost $\leq k$ exists, the algorithm finds this solution; otherwise, "no solution" is returned. To find an optimal solution we call the algorithm repeatedly, increasing k.

3 Vertex Parities

We need a simple observation about the input graphs to reach an improved running time: An integer-weighted graph G with weight function $s : \binom{V}{2} \to \mathbb{Z}$ has the *parity property* if there is a *parity mapping* $p : V \to \{\text{EVEN}, \text{ODD}\}$ such that, for each pair uv, $s(uv)$ is odd if and only if both $p(u) = \text{ODD}$ and $p(v) = \text{ODD}$ holds. We ignore forbidden pairs in this definition, since $s(uv) = -\infty$ has no parity. Note that p is not necessarily unique, as demonstrated by a graph with two vertices and even edge weight. We infer a few simple observations from this definition: If $s(uv)$ is even, then either u or v or both must have EVEN parity. If u is EVEN then $s(uv)$ is even or uv is forbidden, for all $v \neq u$.

Clearly, an unweighted instance of Cluster Editing has the parity property, as we can set $p(u) = \text{ODD}$ for all vertices $u \in V$. The interesting observation is that a graph does not loose the parity property if we merge two vertices. Quite possibly, this results has been stated before in a different graph-theoretical context. We defer the simple, constructive proof to the full paper.

Lemma 1. *Assume that an integer-weighted graph G has the parity property. If we merge two vertices in G, then the resulting graph also has the parity property.*

If the input graph has the parity property then, after any sequence of merging operations, the resulting graph still has the parity property. This is particularly so for the edge branching from [1], as both operations (setting an edge to forbidden, or merging two vertices) preserve the parity property. For our branching, it is important to notice that a zero-edge has even parity, so the parity of at least one of its incident vertices must be EVEN.

4 Isolation and Vertices of Even Parity

Let $\varphi = \frac{1+\sqrt{5}}{2} = 1.61803\ldots$ be the *golden ratio*, satisfying $\varphi = 1 + \frac{1}{\varphi}$. One can easily see that a search tree algorithm with branching vector $(2, 1)$ results in a search tree of size $O(\varphi^k)$: This branching number is the positive root of $x^{-2} + x^{-1} - 1$, so $1 + x - x^2 = 0$, and dividing by x results in the definition of the golden ratio.

Our branching strategy is based on a series of lemmata, ensuring that either there is an edge to branch on, or that the remaining graph is "easy". Clearly, branching on an edge that is part of four or more conflict triples results in the desired branching vector. To this end, we concentrate on the critical case of three conflict triples. First, we consider the case of three strong conflict triples:

Lemma 2. *Let G be an integer-weighted graph that has the parity property. Assume that an edge uv is part of exactly three conflict triples, all of which are strong. Then, we can branch with branching number $\varphi = 1.61803\ldots$.*

We use this lemma to show that we can find an edge to branch on, if we can find an edge that is part of at least three conflict triples.

Lemma 3. *Let G be an integer-weighted graph that has the parity property. Assume that an edge uv is part of three or more conflict triples. Then, we can either find an edge with branching number φ, or we can reduce k without branching.*

The remainder of this section is devoted to proving these two central lemmata.

Proof (Lemma 2). We will show that we can find an edge to branch on, with branching vector $(1, 2)$ or better. In our reasoning, we will show that either, we have already reached the desired branching vector; or, we can infer certain structural properties about the instance.

Let a, b, c be the three vertices that are part of the three conflict triples with u, v. If $s(uv) \geq 2$ then branching on uv results in deletion cost $s(uv) \geq 2$ and merging cost $3 \cdot \frac{1}{2}$, so we reach branching vector $(2, \frac{3}{2})$ and we are done. If uvx with $x \in \{a, b, c\}$ is a conflict triple such that $s(vx) \geq 2$ or $s(ux) \leq -2$, then merging uv into u' will not create a new zero-edge incident to u'. So, branching on uv has branching vector $(1, 2 \cdot \frac{1}{2} + 1) = (1, 2)$, and we are done. The same argumentation holds for a conflict triple vux. In the following, we may assume that a, b, c are ODD, and that $s(uv) = 1$ and $|s(wx)| = 1$ holds for all $w \in \{u, v\}$ and $x \in \{a, b, c\}$; for all other cases, we have just shown that the desired branching vector can be reached.

Assume that u, v do not have a common neighbor, $N(u) \cup N(v) = \{u, v, a, b, c\}$. Then, merging u, v into u' generates three zero-edges $u'a, u'b, u'c$, and u' is isolated, $N(u') = \emptyset$. But then, we do not have to use bookkeeping for these edges, as $\{u'\}$ will also be a separated cluster of size one in the solution. So, branching on uv results in branching vector $(1, 3)$.

We will now use the same trick that the merged vertex u' can be isolated, but this is slightly more involved in case u, v have at least one common neighbor. Let $D := N(u) \cap N(v)$, then $N(u) \cup N(v) = D \cup \{u, v, a, b, c\}$ and $|D| \geq 1$. Our first step is to branch on uv: We delete uv with cost 1, and set it to forbidden.

Next, we merge u, v into a new vertex u'. This generates three zero edges $u'a, u'b, u'c$ with costs $\frac{3}{2}$. Here, $s(u'd) \geq 2$ holds for all $d \in D = \{d_1, \ldots, d_l\}$. We will now branch on all edges $u'd_j$ where the case that $u'd_j$ is deleted, is further analyzed. In detail, we either merge $u'd_i$ with costs $\frac{3}{2}$; or, we delete $u'd_i$ with cost 2 and branch on $u'd_{i+1}$, if $i < l$. Note that we either delete all $d_1, \ldots, d_l$, or we finally merge some $u'd_i$ with cost $\frac{3}{2}$. In the latter case, the total costs of this branch are $2(i-1) + \frac{3}{2}$. But in the very last case where all $d_1, \ldots, d_l$ are deleted, we separate u'. Hence, by the reasoning introduced above, we can "cash" cost $\frac{3}{2}$ we have put aside when generating the three zero-edges $u'a, u'b, u'c$. So, the costs of this final branch are $2l + \frac{3}{2}$. Recall that in all cases, we have additional cost $\frac{3}{2}$ for generating the three zero-edges. In total, we reach the partial branching vector $(0 + 3, 2 + 3, \ldots, 2l + 3) = (3, 5, 7, \ldots, 2l + 3)$.

We combine these two partial branching vectors into one branching vector $(1, 3, 5, 7, 9, \ldots, 2l + 3)$. We claim that any such branching vector corresponds to a branching number $x < \varphi$, and that the numbers converge towards φ. To this end, first note that $1/\varphi$ is the unique positive root of the polynomial $x^2 + x - 1$, that is the characteristic polynomial of branching vector $(2, 1)$. We analyze the

infinite series $f(x) := x^0 + x^2 + x^4 + \ldots$ that converges for all $|x| < 1$. Now, $x^2 \cdot f(x) = f(x) - 1$ and

$$(x^2 + x - 1) \cdot f(x) = f(x) - 1 + xf(x) - f(x) = xf(x) - 1.$$

So, for the series $g(x) := xf(x) - 1$ we have

$$g(x) = xf(x) - 1 = (x^2 + x - 1) \cdot f(x)$$

and, hence, $g(1/\varphi) = 0$. For the partial sums $S_l(x) := x^{2l+3} + x^{2l+1} + \cdots + x^3 + x^1 - 1$ we infer $S_l(x) < S_{l+1}(x)$ and $S_l(x) < g(x)$ for $x \in (0, \infty)$. Also, S_l is strictly increasing in $[0, \infty)$.

Note that any polynomial of the form $p(x) := a_n x^n + \cdots + a_1 x^1 - 1$ with $a_i \geq 0$ for all i, has exactly one positive root for $p \not\equiv -1$. This follows as p is continuous, $p'(x) > 0$ for all $x > 0$, so p is strictly increasing in $(0, \infty)$, $p(0) = -1$, and $\lim_{x \to \infty} p(x) = \infty$. Let x_l be the unique positive root of $S_l(x)$. With $S_l(x_{l+1}) < S_{l+1}(x_{l+1}) = 0$ we finally infer

$$x_1 > x_2 > x_3 > \cdots > 1/\varphi.$$

By definition, $1/x_l$ is the branching number for branching vector $(1, 3, 5, 7, 9, \ldots, 2l + 3)$, and we reach

$$1/x_1 < 1/x_2 < 1/x_3 < \cdots < \varphi.$$

Since the series S_l converges uniformly to g in the interval $[0, \alpha]$ for every $\alpha < 1$, we infer that $\lim_l 1/x_l = \varphi$ must hold, which concludes the proof of the lemma. □

Proof (Lemma 3). Again, we will show that either, we have already reached the desired branching vector $(1, 2)$ or better; or, we can infer certain structural properties about the instance.

If uv is part of four conflict triples then we reach branching vector $(1, 4 \cdot \frac{1}{2}) = (1, 2)$. If uv is part of three strong conflict triples then Lemma 2 guarantees branching number φ. So, assume that uv is part of exactly three conflict triples, and that uvw is a weak conflict triple, so uw is a zero-edge. As uv is part of three conflict triples, we can choose a, b such that $N(u) \triangle N(v) = \{w, a, b\}$. Clearly, for $s(uv) = 2$ we have branching vector $(2, \frac{3}{2})$, so we may assume $s(uv) = 1$. This implies that both u and v must have ODD parity. Since uw is a zero-edge, we infer that w has EVEN parity and, hence, that $s(vw) \geq 2$ holds. For our worst-case considerations, we may assume $s(vw) = 2$.

If vw is part of any additional conflict triples besides wvu, then we reach branching vector $(2, 1)$ for branching on vw: Deleting vw has cost 2, and merging vw then has cost $2 \cdot \frac{1}{2}$. The same holds true if v or w are incident to additional zero-edges besides uw. So, assume there are no zero-edges incident to v or w besides uw, and vx is an edge if and only if wx is an edge for all $x \neq u, v, w$. Let $X \subseteq V \setminus \{u, v, w\}$ be the set of vertices incident to v and, consequently, also to w. Let $X' := X \setminus \{a, b\}$, and note that this set can be empty. All $x \in X'$ are

also incident with u; otherwise, there is a fourth conflict triple for the edge uv. We infer $N(\{u, v, w\}) \subseteq \{u, v, w, a, b\} \cup X'$.

Choose an arbitrary $x \in X'$. If wx is part of an additional conflict triple besides wxu, or if x is incident to a zero-edge, then we again reach branching vector $(2, 1)$ for branching on wx: Deleting wx has cost 2 since w is EVEN, and merging wx has cost $2 \cdot \frac{1}{2}$. Hence, we infer three things: Firstly, each y adjacent to some $x \in X'$ is also adjacent to w and, hence, $y \in X$. So, $N(X') \subseteq \{u, v, w, a, b\} \cup X'$. Secondly, each pair $x, y \in X'$ must be connected by an edge. We distinguish three cases:

1. Assume $a, b \in X$, so va and vb are edges. In this case, u, v, w, a, b, X' form a connected component. If ab is a zero-edge or non-edge, then branching on wa results in branching vector $(2, 2 \cdot \frac{1}{2})$: although w, a, u do not form a conflict triple, merging wa still destroys the zero-edge uw. So, we may assume that ab is an edge. By the same reasoning, ax and bx must be edges, for all $x \in X'$. Next, $s(ux) = 1$ must hold for all $x \in X'$; otherwise, we can branch on ux with branching vector $(2, 3 \cdot \frac{1}{2})$. The cost of separating u from all other vertices is $|X'| + 1$, and the resulting graph consists of two cliques $\{u\}$ and $\{v, w, a, b\} \cup X'$. The cost of any other cut in this connected component is at least $|X'| + 3$ (for separating a or b), since w is adjacent to all vertices but u with edges of weight at least 2. The cost of transforming the connected component into a clique is $|s(ua)| + |s(ub)|$. So, we can test in constant time if one of the two possible transformations has cost at most k.
2. Assume $a \in X$ and $b \notin X$, so va and ub are edges. Then, $N(\{u, v, w, a\} \cup X') \subseteq \{u, v, w, a, b\} \cup X'$. For $s(ua) < -1$ we reach branching vector $(1, 2 \cdot \frac{1}{2} + 1)$ for branching on uv, as merging u, v will not generate a zero-edge incident to a and, hence, no bookkeeping is required. (Obviously, this includes the case that ua is forbidden.) So, $s(ua) \in \{0, 1\}$ must hold. Since bv is a non-edge, bw and bx for all $x \in X'$ are also non-edges. If $s(ub) \geq 2$ then branching on ub results in branching vector $(2, 1)$, as vub is a conflict triple. Now, one can easily see that no optimal solution can bisect v, w, a, X': For $X' = \emptyset$ a bisection of vertices v, w, a costs at least 3, and for $X' \neq \emptyset$ costs are at least 4. Given a solution that bisects v, w, a, X', we modify the solution by putting u, v, w, a, X' in a separate clique, with cost at most 1 for inserting ua, and cost 1 for removing ub. Clearly, this new solution has smaller total cost than the initial solution, so the initial solution cannot be optimal. Hence, we can merge v, w, a, X' without branching, generating cost of at least $\frac{1}{2}$ for destroying the zero-edge uw.
3. Assume $a, b \notin X$, so ua and ub are edges. Then, va and vb are non-edges, since no zero-edges can be incident to v. Similar to above, this implies that wa and wb, as well as ax and bx for all $x \in X'$, are non-edges, too: Otherwise, we can branch on vw or wx. If $s(ua) \geq 2$ then branching on uv results in branching vector $(1, 2)$. So, we infer $s(ua) = 1$ and, by symmetry, $s(ub) = 1$. Now, merging uv into some vertex u' results in a separated clique with

vertex set u', w, X that is not connected to the rest of the graph, and can be removed immediately. Hence, branching on uv leads to branching vector $(1, 2)$ as we do not have put away $2 \cdot \frac{1}{2}$ for potentially destroying zero-edges $u'a$ and $u'b$ later.

We have shown that we can find an edge that allows for the desired branching vectors, simplify the instance and reduce k without branching, or solve the remaining instance in constant time. □

5 Solving Remainder Instances

Assume that there is no edge in the graph that is part of three or more (weak or strong) conflict triples. We transform our weighted graph into an unweighted counterpart G_u, where zero-edges are counted as non-existing. This graph G_u is called the *type graph* of the weighted graph. Then, there is no edge uv in the unweighted graph G_u that is part of three conflict triples. Damaschke [6] characterizes such graphs: Let P_n, C_n, K_n be the chordless path, cycle, and clique on n vertices, respectively. Let $G + H$ denote the disjoint union of two graphs, and let $p \cdot G$ denote p disjoint copies of G. Let $G * H$ be the graph $G + H$ where, in addition, every vertex from G is adjacent to every vertex from H. Finally, the graph G^c has the same vertex set as G, and $\{u, v\}$ is an edge of G^c if and only if it is no edge of G. Now, Theorem 2 from [6] states:

Lemma 4. *Let G be a connected, unweighted graph such that no edge is part of three or more conflict triples. Then, G has at most six vertices, is a clique, a path, a cycle, or a graph of type $K_q * H$ for $q \geq 0$ and $H \in \{K_1 + K_1, C_5, P_4, K_1 + K_1 + K_1, K_2 + K_2, K_2 + K_1, (p \cdot K_2)^c\}$, $p \geq 2$.*

In fact, the characterization in [6] is slightly more complicated: To this end, note that $K_q * P_3 = K_{q+1} * (K_1 + K_1)$. Any non-edge in the type graph can be a non-edge or zero-edge in the weighted graph, and edges and non-edges can be arbitrarily weighted. We now show that we can efficiently solve all remaining, "simple" instances. This is similar to our argumentation in [3] but as we want to reach branching vector $(2, 1)$, our argumentation is slightly more involved. We defer the proof of Lemma 5 to the full version of this paper.

Lemma 5. *Let G be a connected graph that has the parity property. Assume that there is no edge that is part of three conflict triples. Then, we can find an edge with branching number φ; reduce k without branching; or, we can solve the instance in polynomial time.*

6 A Golden Ratio Base for Search Tree Size

Assume that G has the parity property. We want to show that we can either find an edge to branch on with branching number φ; decrease k without branching; or, solve the remaining instance in polynomial time. If there is an edge uv that

is part of at least three (weak or strong) conflict triples, we branch on this edge. By Lemma 3, doing so results in branching number φ, or we reduce k without branching, as desired. We can find an edge to branch on, in time $O(n^3)$. Similarly, we can perform all other tasks required for one step of the branching, in this time. If there is no edge uv that is part of at least three conflict triples, then Lemma 5 guarantees that we can branch with branching number φ; reduce k without branching; or, solve the instance in polynomial time. To compute minimum s-t-cuts as part of Lemma 5, we use the Goldberg-Tarjan algorithm [8] to compute a maximum s-t-flow in time $O(n^3)$, independent of edge weights. We reach:

Lemma 6. *Given an integer-weighted instance of the* Cluster Editing *problem with no zero-edges that satisfies the parity property, this instance can be solved in* $O(\varphi^k \cdot n^3)$ *time.*

We can combine this with the weighted kernel from [4] of size $O(k)$ with running time $O(n^2)$, resulting in running time $O(\varphi^k \cdot k^3 + n^2)$. To get rid of the multiplicative polynomial factor, we use interleaving [16]: Here, a small trick is required to make this kernel work with instances that may contain zero-edges; we defer the details to the full paper.

Theorem 1. *Given an integer-weighted instance of the* Cluster Editing *problem with no zero-edges that satisfies the parity property, this instance can be solved in* $O(\varphi^k + n^2)$ *time.*

Given an unweighted Cluster Editing instance, we first identify all critical cliques in time $O(m + n)$ for a graph with n vertices and m edges [12], and merge the vertices of each critical clique [1, 11]. The resulting integer-weighted instance has $O(k)$ vertices and no zero-edges, and satisfies the parity property. Using Theorem 1 we reach:

Theorem 2. Cluster Editing *can be solved in* $O(1.62^k + k^2 + m + n)$ *time.*

7 Conclusion

We have presented a parameterized algorithm for the Cluster Editing problem, that finally reaches the golden ratio as the base for the exponential growth of the running time. It is noticeable that search tree approaches plus additional structural observations still have a lot of potential to yield better FPT algorithms for well-known problems, even without extensive case handling. Note that the underlying edge branching is also very swift in practice, and can usually process instances with thousands of edge modifications in a matter of minutes [2].

The base $\varphi = \frac{1+\sqrt{5}}{2} = 1.61803\ldots$, resulting from branching vector $(2, 1)$, appears repeatedly in the analysis of advanced algorithms for the problem [1,3].

Hence, it is an interesting question for the future if we can get beyond the $O^*(\varphi^k)$ barrier. One possible extension lies in the split-off technique introduced in [3] for Cluster Deletion, even though it cannot be directly applied, as branching on a C_4 results in branching vector $(1, 1)$ for Cluster Editing. Improving upon the running time should not be problematic for the rather technical Lemma 5, though. Here, the open question is, which of these special cases are tractable (such as $H = K_1 + K_1$) and which are intractable (such as $H = K_1 + K_1 + K_1$), and what FPT algorithms can be derived for the hard ones.

References

1. Böcker, S., Briesemeister, S., Bui, Q.B.A., Truss, A.: Going weighted: Parameterized algorithms for cluster editing. Theor. Comput. Sci. 410(52), 5467–5480 (2009)
2. Böcker, S., Briesemeister, S., Klau, G.W.: Exact algorithms for cluster editing: Evaluation and experiments. Algorithmica 60(2), 316–334 (2011)
3. Böcker, S., Damaschke, P.: Even faster parameterized cluster deletion and cluster editing. Inform. Process. Lett. 111(14), 717–721 (2011)
4. Cao, Y., Chen, J.: Cluster Editing: Kernelization Based on Edge Cuts. In: Raman, V., Saurabh, S. (eds.) IPEC 2010. LNCS, vol. 6478, pp. 60–71. Springer, Heidelberg (2010)
5. Chen, J., Meng, J.: A $2k$ Kernel for the Cluster Editing Problem. In: Thai, M.T., Sahni, S. (eds.) COCOON 2010. LNCS, vol. 6196, pp. 459–468. Springer, Heidelberg (2010)
6. Damaschke, P.: Bounded-Degree Techniques Accelerate Some Parameterized Graph Algorithms. In: Chen, J., Fomin, F.V. (eds.) IWPEC 2009. LNCS, vol. 5917, pp. 98–109. Springer, Heidelberg (2009)
7. Downey, R.G., Fellows, M.R.: Parameterized Complexity. Springer, Heidelberg (1999)
8. Goldberg, A.V., Tarjan, R.E.: A new approach to the maximum-flow problem. J. ACM 35(4), 921–940 (1988)
9. Gramm, J., Guo, J., Hüffner, F., Niedermeier, R.: Automated generation of search tree algorithms for hard graph modification problems. Algorithmica 39(4), 321–347 (2004)
10. Gramm, J., Guo, J., Hüffner, F., Niedermeier, R.: Graph-modeled data clustering: Fixed-parameter algorithms for clique generation. Theor. Comput. Syst. 38(4), 373–392 (2005)
11. Guo, J.: A more effective linear kernelization for cluster editing. Theor. Comput. Sci. 410(8-10), 718–726 (2009)
12. Hsu, W.-L., Ma, T.-H.: Substitution Decomposition on Chordal Graphs and Applications. In: Hsu, W.-L., Lee, R.C.T. (eds.) ISA 1991. LNCS, vol. 557, pp. 52–60. Springer, Heidelberg (1991)
13. Křivánek, M., Morávek, J.: NP-hard problems in hierarchical-tree clustering. Acta. Inform. 23(3), 311–323 (1986)
14. Marx, D., Razgon, I.: Fixed-parameter tractability of multicut parameterized by the size of the cutset. In: Proc. of ACM Symposium on Theory of Computing, STOC 2011, pp. 469–478. ACM (2011), doi:10.1145/1993636.1993699

15. Niedermeier, R.: Invitation to Fixed-Parameter Algorithms. Oxford University Press (2006)
16. Niedermeier, R., Rossmanith, P.: A general method to speed up fixed-parameter-tractable algorithms. Inform. Process. Lett. 73, 125–129 (2000)
17. Rosamond, F. (ed.): FPT News: The Parameterized Complexity Newsletter (Since 2005), http://fpt.wikidot.com/
18. Wittkop, T., Emig, D., Lange, S., Rahmann, S., Albrecht, M., Morris, J.H., Böcker, S., Stoye, J., Baumbach, J.: Partitioning biological data with transitivity clustering. Nat. Methods 7(6), 419–420 (2010)

Stable Sets of Threshold-Based Cascades on the Erdős-Rényi Random Graphs

Ching-Lueh Chang[1,⋆] and Yuh-Dauh Lyuu[2,3,⋆⋆]

[1] Department of Computer Science and Engineering, Yuan Ze University, Taoyuan, Taiwan
clchang@saturn.yzu.edu.tw
[2] Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan
lyuu@csie.ntu.edu.tw
[3] Department of Finance, National Taiwan University, Taipei, Taiwan

Abstract. Consider the following reversible cascade on the Erdős-Rényi random graph $G(n,p)$. In round zero, a set of vertices, called the seeds, are active. For a given $\rho \in (0,1]$, a non-isolated vertex is activated (resp., deactivated) in round $t \in \mathbb{Z}^+$ if the fraction f of its neighboring vertices that were active in round $t-1$ satisfies $f \geq \rho$ (resp., $f < \rho$). An irreversible cascade is defined similarly except that active vertices cannot be deactivated. A set of vertices, S, is said to be stable if no vertex will ever change its state, from active to inactive or vice versa, once the set of active vertices equals S. For both the reversible and the irreversible cascades, we show that for any constant $\epsilon > 0$, all $p \in [(1+\epsilon)(\ln(e/\rho))/n, 1]$ and with probability $1 - n^{-\Omega(1)}$, every stable set of $G(n,p)$ has size $O(\lceil \rho n \rceil)$ or $n - O(\lceil \rho n \rceil)$.

1 Introduction

Let $G(V,E)$ be a simple undirected graph and $\rho \in (0,1]$, where each vertex of G can be in one of two states, active or inactive. Consider the following process, called the reversible cascade. In round zero, only a set of vertices, called the seeds, are active. Thereafter, a non-isolated vertex is activated (resp., deactivated) in a round if the fraction f of its neighboring vertices that were active in the previous round satisfies $f \geq \rho$ (resp., $f < \rho$). A set $S \subseteq V$ is said to be stable for the reversible cascade if no vertex will ever change its state once the set of active vertices equals S. The irreversible cascade and its stable sets are defined similarly except that the deactivations of vertices are prohibited.

To model socio-economic contagion amongst fully rational individuals, Morris [32] considers a countably infinite population in which each player has two

⋆ Supported in part by the National Science Council of Taiwan under grant 99-2218-E-155-014-MY2.

⋆⋆ Supported in part by the National Science Council of Taiwan under grant 97-2221-E-002-096-MY3 and Excellent Research Projects of National Taiwan University under grant 98R0062-05.

C.S. Iliopoulos and W.F. Smyth (Eds.): IWOCA 2011, LNCS 7056, pp. 96–105, 2011.
© Springer-Verlag Berlin Heidelberg 2011

strategies and a finite set of neighbors. In each round, a player chooses strategy 1 if and only if at least a $\rho \in (0, 1]$ fraction of its neighbors do likewise in the previous round. So the resulting model is the same as our reversible cascade except that $G(V, E)$ is now an infinite graph with finite degrees. Define the contagion threshold to be the largest ρ such that there exists a finite set of players whose initial choice of strategy 1 eventually leads all players to strategy 1. Morris proves several characterizations of the contagion threshold and an $1/2$ upper bound on it. For variants with partially rational players whose states are updated at random times governed by a Poisson process, much research studies the expected waiting time until everyone or almost everyone enters the same state [7, 14, 28, 43].

Consensus formation and periodic behavior are important aspects of the reversible cascade with $\rho = 1/2$ [34, 36, 44]. No matter what vertices are chosen as seeds, the set of active vertices in a simple undirected graph will eventually (1) stabilize or (2) coincide with one subset of V in even-numbered rounds and another subset in odd-numbered rounds [20, 21]. More general discrete-time dynamical systems also exhibit similar behavior [19, 29–31, 38, 39]. The reversible cascade with $\rho = 1/2$ as well as its slight variants also model the propagation of transient faults in majority-based fault-tolerant systems [17, 18, 35] and the evolution of host-pathogen systems [2–4, 22]. Flocchini et al. [17, 18] and Peleg [35] study the minimum number of seeds guaranteeing that (1) all vertices will be active after a finite number of rounds and (2) no active vertices will ever be deactivated in any round. For any simple undirected graph $G(V, E)$, Peleg [35] shows that $\Omega(\sqrt{|V|})$ seeds are needed for activating all vertices in two rounds. Berger [6] constructs a graph family for which a constant number of seeds can activate all vertices after a finite number of rounds. Agur et al. [2, 4] and Granville [22] derive exact formulas for the number of stable sets of a ring. They also count the number of cyclic binary strings with arbitrary restrictions on the numbers of consecutive 0's and 1's. For a survey of the above results, see [36].

Next, we turn to irreversible cascades. Luccio et al. [26] and Flocchini et al. [16–18] assume that each vertex is activated when the majority of its neighboring vertices are active, where the majority may assume the strict or the simple form. Their setup is suitable for modeling the propagation of permanent faults in majority-based fault-tolerant systems [15–18, 26, 36]. Bounds have been derived on the minimum number of seeds that can activate all vertices after a finite number of rounds for rings [16, 17], complete trees, butterflies, cube-connected cycles, shuffle-exchange graphs, DeBruijn graphs, hypercubes [16, 17, 26], tori [13, 18, 24, 25, 37], Erdős-Rényi random graphs [9, 11] or directed graphs without source vertices [1, 9, 10]. Bootstrap percolation analyzes the density of independently chosen seeds needed to activate all vertices at the end with high probability [5, 41].

Computational issues regarding irreversible cascades have also been studied. In particular, efficient algorithms are known for the problem of finding a minimum set of seeds activating all vertices after a finite number of rounds in a

tree [13, 40], cycle, complete graph and bipartite complete graph [40]. In contrast, many hardness results are known for the same problem and its variants in general graphs [12, 23, 24, 40].

For both the reversible and the irreversible cascades, this paper proves that for any constant $\epsilon > 0$, all $p \in [(1+\epsilon)(\ln(e/\rho))/n, 1]$ and with probability $1 - n^{-\Omega(1)}$, every stable set of $G(n,p)$ has size $O(\lceil \rho n \rceil)$ or $n - O(\lceil \rho n \rceil)$. With $\rho \to 0^+$, therefore, any stable set of $G(n,p)$ occupies either an $o(1)$ or a $1 - o(1)$ fraction of all vertices.

2 Definitions

Let $G(V,E)$ be a simple undirected graph [42]. For $X, Y \subseteq V$, define $e(X,Y)$ to be the number of edges with one endpoint in X and the other in Y. For a positive integer n and a real number $p \in [0,1]$, the Erdős-Rényi random graph $G(n,p)$ is a simple undirected graph with vertices $1, 2, \ldots, n$ where each of the possible $\binom{n}{2}$ edges appears independently with probability p [8]. For each $v \in \{1, 2, \ldots, n\}$, define $N(v) \subseteq [n] \setminus \{v\}$ to be the set of neighbors of v and $\deg(v) \equiv |N(v)|$. For convenience, define $[n] \equiv \{1, 2, \ldots, n\}$. Furthermore, define $2^{[n]}$ to be the power set of $[n]$, i.e., the set of all subsets of $[n]$.

Consider the following reversible cascade on the Erdős-Rényi random graph $G(n,p)$ whose vertices have two possible states, active or inactive. In round zero, only a set of vertices, called the seeds, are active. For a given $\rho \in (0,1]$, a non-isolated vertex is activated (resp., deactivated) in round $t \in \mathbb{Z}^+$ if the fraction f of its neighboring vertices that were active in round $t-1$ satisfies $f \geq \rho$ (resp., $f < \rho$). More precisely, a vertex with degree $d > 0$ is activated (resp., deactivated) in a round if at least (resp., less than) $\lceil \rho d \rceil$ of its neighboring vertices are active in the previous round. The irreversible cascade is defined similarly except that deactivations of vertices are prohibited. Define $\sigma_\rho^{\text{rev}} : 2^{[n]} \to 2^{[n]}$ to map the set of active vertices in a round to that in the next round, assuming the reversible cascade. Then define $\sigma_\rho^{\text{irr}} : 2^{[n]} \to 2^{[n]}$ similarly for the irreversible cascade. A set $S \subseteq [n]$ is said to be stable for the reversible (resp., irreversible) cascade if $\sigma_\rho^{\text{rev}}(S) = S$ (resp., $\sigma_\rho^{\text{irr}}(S) = S$). So a reversible or irreversible cascade stops evolving precisely when its set of active vertices is stable.

Below is a straightforward lemma.

Lemma 1. *For a simple undirected graph $G(V,E)$ and all disjoint $A, B \subseteq V$,*

$$e(A, A \cup B) \geq \sum_{v \in A} \frac{|N(v) \cap (A \cup B)|}{2}.$$

Proof. Each edge with an endpoint in A and the other in $B \setminus A$ contributes 1 to both $e(A, A \cup B)$ and $\sum_{v \in A} |N(v) \cap (A \cup B)|$. Each edge with both endpoints in A contributes 1 to $e(A, A \cup B)$ and 2 to $\sum_{v \in A} |N(v) \cap (A \cup B)|$. □

3 Stable Sets of the Erdős-Rényi Random Graphs

This section shows that for any constant $\epsilon > 0$, all $p \in [\,(1+\epsilon)\,(\ln(e/\rho))/n, 1]$, $\rho \in (\,0,1\,]$ and with probability $1 - n^{-\Omega(1)}$, every stable set of $G(n,p)$ has size $O(\lceil \rho n \rceil)$ or $n - O(\lceil \rho n \rceil)$.

Lemma 2. *Let $n \in \mathbb{Z}^+$, $p \in [\,0,1\,]$ and $\rho \in (\,0,1\,]$. Then*

$$\Pr\left[\,|\,\{v \in [\,n\,] \mid \deg(v) > 30pn\}\,| \geq \rho n\,\right] < \binom{n}{\lceil \rho n \rceil} 2^{-15pn\lceil \rho n \rceil},$$

where the probability is taken over the random graphs $G(n,p)$.

Proof. In the proof, all probabilities are taken over the random graphs $G(n,p)$. Clearly,

$$\begin{aligned} & \Pr\left[\,|\,\{v \in [\,n\,] \mid \deg(v) > 30pn\}\,| \geq \rho n\,\right] \\ = & \Pr\left[\,\exists X \subseteq [\,n\,], |\,X\,| = \lceil \rho n \rceil, \forall v \in X, \deg(v) > 30pn\,\right] \\ \leq & \sum_{X \subseteq [\,n\,], |\,X\,| = \lceil \rho n \rceil} \Pr\left[\,\forall v \in X, \deg(v) > 30pn\,\right]. \end{aligned} \tag{1}$$

Now fix an arbitrary $X \subseteq [\,n\,]$ with size $\lceil \rho n \rceil$. If $\deg(v) > 30pn$ for all $v \in X$, then Lemma 1 implies $e(X, [\,n\,]) > 15pn\,|\,X\,|$. Hence

$$\begin{aligned} & \Pr\left[\,\forall v \in X, \deg(v) > 30pn\,\right] \\ \leq & \Pr\left[\,e\,(X, [\,n\,]) > 15pn\,|\,X\,|\,\right] \\ < & 2^{-15pn\lceil \rho n \rceil}, \end{aligned}$$

where the last inequality follows from Chernoff's bound [33, Exercise 4.1]. This and inequality (1) complete the proof. □

Lemma 3. *For any $n \in \mathbb{Z}^+$, $p \in [\,0,1\,]$, $\rho \in [\,1/n, 1\,]$, $\xi \in \mathbb{Z}^+$ with $\xi > 30$ and $\eta \equiv 1 - (30/\xi)$,*

$$\begin{aligned} & \Pr\left[\exists X, U \subseteq [n], |X| = \lceil \rho n \rceil, \xi \lceil \rho n \rceil \leq |U| \leq n - \xi \lceil \rho n \rceil,\ \forall v \in [n] \setminus (U \cup X)\,,\ |N(v) \cap U| \leq 30\rho pn\right] \\ \leq & \ 2 \sum_{s=\xi\lceil \rho n \rceil}^{\lfloor n/2 \rfloor} \exp\left(2\rho n \ln \frac{e}{\rho} + s \ln \frac{en}{s} + \eta p s (n-s) \ln \frac{e^{-\eta}}{(1-\eta)^{1-\eta}}\right), \end{aligned}$$

where the probability is taken over the random graphs $G(n,p)$.

Proof. In the proof, all probabilities are taken over the random graphs $G(n,p)$. For any $X, U \subseteq [\,n\,]$ with $|\,U\,| \geq \xi\lceil \rho n \rceil$ and each $u \in [\,n\,] \setminus (U \cup X)$,

$$\begin{aligned} & \Pr\left[\,|\,N(u) \cap U\,| \leq 30\rho pn\,\right] \\ \leq & \Pr\left[\,|\,N(u) \cap U\,| \leq (1-\eta) \cdot E\left[\,|\,N(u) \cap U\,|\,\right]\right] \\ \leq & \left(\frac{e^{-\eta}}{(1-\eta)^{1-\eta}}\right)^{p\,|\,U\,|} \end{aligned}$$

by Chernoff's bound [27, Theorem 4.5]; hence, as the random variables $|N(v) \cap U|$ for $v \in [n] \setminus (U \cup X)$ are independent,

$$\Pr[\forall v \in [n] \setminus (U \cup X), |N(v) \cap U| \leq 30\rho pn] \leq \left(\frac{e^{-\eta}}{(1-\eta)^{1-\eta}}\right)^{p|U|(n-|U\cup X|)}. \quad (2)$$

If, furthermore, $|X| = \lceil \rho n \rceil$ and $|U| \leq n - \xi\lceil \rho n \rceil$, then

$$\left(\frac{e^{-\eta}}{(1-\eta)^{1-\eta}}\right)^{p|U|(n-|U\cup X|)} \leq \left(\frac{e^{-\eta}}{(1-\eta)^{1-\eta}}\right)^{\eta p|U|(n-|U|)} \quad (3)$$

by the easily verifiable fact that $n - |U \cup X| \geq \eta(n - |U|)$. Now,

$$\begin{aligned}
&\Pr[\exists X, U \subseteq [n], |X| = \lceil \rho n \rceil, \xi\lceil \rho n \rceil \leq |U| \leq n - \xi\lceil \rho n \rceil, \forall v \in [n] \setminus (U \cup X), |N(v) \cap U| \leq 30\rho pn] \\
&\leq \sum_{X \subseteq [n], |X| = \lceil \rho n \rceil} \sum_{U \subseteq [n], \xi\lceil \rho n \rceil \leq |U| \leq n - \xi\lceil \rho n \rceil} \Pr[\forall v \in [n] \setminus (U \cup X), |N(v) \cap U| \leq 30\rho pn] \\
&\leq \sum_{X \subseteq [n], |X| = \lceil \rho n \rceil} \sum_{U \subseteq [n], \xi\lceil \rho n \rceil \leq |U| \leq n - \xi\lceil \rho n \rceil} \left(\frac{e^{-\eta}}{(1-\eta)^{1-\eta}}\right)^{\eta p|U|(n-|U|)} \\
&\leq \binom{n}{\lceil \rho n \rceil} \sum_{U \subseteq [n], \xi\lceil \rho n \rceil \leq |U| \leq n - \xi\lceil \rho n \rceil} \left(\frac{e^{-\eta}}{(1-\eta)^{1-\eta}}\right)^{\eta p|U|(n-|U|)}, \quad (4)
\end{aligned}$$

where the second inequality follows from inequalities (2)–(3). Furthermore,

$$\begin{aligned}
&\sum_{U \subseteq [n], \xi\lceil \rho n \rceil \leq |U| \leq n - \xi\lceil \rho n \rceil} \left(\frac{e^{-\eta}}{(1-\eta)^{1-\eta}}\right)^{\eta p|U|(n-|U|)} \\
&= \sum_{s=\xi\lceil \rho n \rceil}^{n-\xi\lceil \rho n \rceil} \sum_{U \subseteq [n], |U|=s} \left(\frac{e^{-\eta}}{(1-\eta)^{1-\eta}}\right)^{\eta ps(n-s)} \\
&\leq 2 \sum_{s=\xi\lceil \rho n \rceil}^{\lfloor n/2 \rfloor} \binom{n}{s} \left(\frac{e^{-\eta}}{(1-\eta)^{1-\eta}}\right)^{\eta ps(n-s)} \quad (5) \\
&\leq 2 \sum_{s=\xi\lceil \rho n \rceil}^{\lfloor n/2 \rfloor} \exp\left(s \ln \frac{en}{s} + \eta ps(n-s) \ln \frac{e^{-\eta}}{(1-\eta)^{1-\eta}}\right), \quad (6)
\end{aligned}$$

where inequality (5) follows from $\binom{n}{s} = \binom{n}{n-s}$ and $s(n-s) = (n-s)s$. Inequalities (4)–(6) complete the proof. □

Theorem 4. *Let $\epsilon > 0$. Then there exists an integer $\xi > 30$ such that for any $n \in \mathbb{Z}^+$, $\rho \in [1/n, 1/\xi^2)$, $p \in [(1+\epsilon)(\ln(e/\rho))/n, 1]$, $\sigma_G \colon 2^{[n]} \to 2^{[n]}$ satisfying*

$$\forall U \subseteq [n], \ \{v \in [n] \mid |N_G(v) \cap U| > \rho \deg_G(v)\} \subseteq \sigma_G(U) \quad (7)$$

for each simple undirected graph G with vertex set $[n]$, and writing $\sigma \equiv \sigma_{G(n,p)}$,

$$\Pr[\exists U \subseteq [n], \xi\lceil \rho n \rceil \leq |U| \leq n - \xi\lceil \rho n \rceil, \ \sigma(U) \subseteq U] = n^{-\Omega(1)}, \quad (8)$$

where the probability is taken over over the random graphs $G(n,p)$. The hidden constants in the Ω notations are independent of n, p, ρ, ϵ and ξ.

Proof. We will leave ξ to be determined later; before that we only need $\xi > 30$ in the derivation. Define $\eta \equiv 1 - (30/\xi)$. As $p \in [\,(1+\epsilon)\,(\ln(e/\rho))/n, 1\,]$,

$$\exp\left(s\,\ln\frac{en}{s} + \eta ps(n-s)\,\ln\frac{e^{-\eta}}{(1-\eta)^{1-\eta}}\right)$$
$$\leq \exp\left(s\,\ln\frac{en}{s} + \eta\,(1+\epsilon)\left(\ln\frac{e}{\rho}\right)s\left(1-\frac{s}{n}\right)\ln\frac{e^{-\eta}}{(1-\eta)^{1-\eta}}\right), \tag{9}$$

$s \in (\,0, n\,]$. Define

$$g(s,n,\rho) \equiv s\,\ln\frac{en}{s} + \eta\,(1+\epsilon)\left(\ln\frac{e}{\rho}\right)s\left(1-\frac{s}{n}\right)\ln\frac{e^{-\eta}}{(1-\eta)^{1-\eta}}. \tag{10}$$

Elementary calculus and laborious calculations reveal the following properties of $g(s,n,\rho)$:

- $g(\xi\rho n, n, \rho) \leq -4\rho n\ln(e/\rho)$ provided that $\rho \in (0, 1/\xi^2)$ and

$$\xi\left(1+\eta\,(1+\epsilon)\left(1-\frac{1}{\xi}\right)\ln\frac{e^{-\eta}}{(1-\eta)^{1-\eta}}\right) < -4; \tag{11}$$

- $\partial g(s,n,\rho)/\partial s < 0$ for $s \in [\,\xi\rho n, \epsilon n/(4+4\epsilon)\,]$ provided that

$$1+\eta\,(1+\epsilon)\left(1-\frac{\epsilon}{2+2\epsilon}\right)\ln\frac{e^{-\eta}}{(1-\eta)^{1-\eta}} < 0; \tag{12}$$

- $g(s,n,\rho) \leq -3n$ for $s \in [\,\epsilon n/(4+4\epsilon), n/2\,]$ provided that $\rho \in (0, 1/\xi^2)$ and

$$\frac{1}{2}\ln\left(\frac{e(4+4\epsilon)}{\epsilon}\right) + \eta\,(1+\epsilon)\left(\ln\left(e\xi^2\right)\right)\,\frac{\epsilon}{4+4\epsilon}\left(1-\frac{\epsilon}{4+4\epsilon}\right)\ln\frac{e^{-\eta}}{(1-\eta)^{1-\eta}} < -3. \tag{13}$$

By elementary calculus and $\eta = 1 - (30/\xi)$,

$$\lim_{\xi\to\infty}\ln\frac{e^{-\eta}}{(1-\eta)^{1-\eta}} = -1.$$

Therefore, with laborious calculations, inequalities (11)–(13) hold in the limit as $\xi \to \infty$. Hence there exists a real number $C(\epsilon) > 30$, depending only on ϵ, such that inequalities (11)–(13) hold for $\xi \geq C(\epsilon)$. From now on, we assume that $\xi \geq C(\epsilon)$ and $\rho \in [\,1/n, 1/\xi^2\,)$. So the derived properties on $g(s,n,\rho)$ give

$$\max_{s\in[\,\xi\rho n, n/2\,]} g(s,n,\rho) \leq \max\left\{-4\rho n\ln\frac{e}{\rho}, -3n\right\}. \tag{14}$$

By Lemma 3 and inequalities (9)–(10) and (14),

$$\Pr\left[\exists X, U \subseteq [n], |X| = \lceil\rho n\rceil, \xi\lceil\rho n\rceil \leq |U| \leq n - \xi\lceil\rho n\rceil, \forall v \in [n]\setminus(U\cup X), |N(v)\cap U| \leq 30\rho pn\right]$$
$$\leq 2\sum_{s=\xi\lceil\rho n\rceil}^{\lfloor n/2\rfloor}\exp\left(\max\left\{-2\rho n\ln\frac{e}{\rho}, -3n + 2\rho n\,\ln\frac{e}{\rho}\right\}\right)$$
$$= O\left(\frac{1}{n}\right). \tag{15}$$

Let

$$Y \equiv \{v \in [\,n\,] \mid \deg(v) > 30pn\}$$

be the set of vertices with degrees greater than $30pn$. By Lemma 2,

$$\Pr[\,|\,Y\,| \geq \rho n\,] \leq \binom{n}{\lceil \rho n \rceil} 2^{-15pn\lceil \rho n \rceil} = n^{-\Omega(1)}. \tag{16}$$

For any $U \subseteq [\,n\,]$ with $\sigma(U) \subseteq U$ and $v \in [\,n\,] \setminus U$, $|\,N(v) \cap U\,| \leq \rho \deg(v)$ by relation (7); if, furthermore, $v \notin Y$, then

$$|\,N(v) \cap U\,| \leq 30\rho pn. \tag{17}$$

Therefore,

$$\begin{aligned}
&\Pr[(|\,Y\,| < \lceil \rho n \rceil) \wedge (\exists U \subseteq [\,n\,], \xi\lceil \rho n \rceil \leq |\,U\,| \leq n - \xi\lceil \rho n \rceil, \sigma(U) \subseteq U)\,] \\
&\leq \Pr[(|Y| < \lceil \rho n \rceil) \wedge (\exists U \subseteq [n], \xi\lceil \rho n \rceil \leq |U| \leq n - \xi\lceil \rho n \rceil, \forall v \in [n] \setminus (U \cup Y)\,, |N(v) \cap U| \leq 30\rho pn)] \\
&\leq O\left(\frac{1}{n}\right),
\end{aligned} \tag{18}$$

where the last inequality follows from inequality (15). Summing up inequalities (16) and (18) proves Eq. (8). □

As a remark, in Theorem 4, σ_G is nonrandom for each undirected graph $G(V, E)$, whereas $\sigma_{G(n,p)}$ depends on the underlying random graph $G(n, p)$.

As ξ depends only on ϵ in Theorem 4, we may take $\xi\lceil \rho n \rceil = O(\lceil \rho n \rceil)$ in Eq. (8) when $\epsilon > 0$ is a constant, as done below.

Theorem 5. *Let $\epsilon > 0$ be a constant, $n \in \mathbb{Z}^+$, $\rho \in (\,0, 1\,]$ and $p \in [\,(1 + \epsilon)\,(\ln(e/\rho))/n, 1\,]$. Assume that $\sigma_G \colon 2^{[\,n\,]} \to 2^{[\,n\,]}$ satisfies relation (7) for each simple undirected graph G with vertex set $[\,n\,]$. Then, writing $\sigma \equiv \sigma_{G(n,p)}$,*

$$\Pr[\,\forall S \subseteq [\,n\,], (\sigma(S) = S) \Rightarrow (|\,S\,| = O(\lceil \rho n \rceil)) \vee (|\,S\,| = n - O(\lceil \rho n \rceil))\,] = 1 - n^{-\Omega(1)}, \tag{19}$$

where the probability is taken over the random graphs $G(n, p)$. The hidden constants in the O and Ω notations are independent of n, p, ρ and S.

Proof. Assume without loss of generality that $\rho \geq 1/n$. Let ξ be as in Theorem 4, which is a constant because ϵ is now a constant. The case of $\rho \in [\,1/n, 1/\xi^2\,)$ is an immediate consequence of Theorem 4. For the case of $\rho \geq 1/\xi^2 = \Omega(1)$, Eq. (19) trivially holds. □

We now have the following corollary on the stable sets for the reversible and the irreversible cascades.

Corollary 6. *Let $\epsilon > 0$ be a constant. For any $n \in \mathbb{Z}^+$, $\rho \in (\,0, 1\,]$, $p \in [\,(1 + \epsilon)\,(\ln(e/\rho))/n, 1\,]$ and with probabilities taken over the random graphs $G(n, p)$,*

$$\Pr[\,\text{every stable set of } G(n,p) \text{ has size } O(\lceil \rho n \rceil) \text{ or } n - O(\lceil \rho n \rceil)\,] = 1 - n^{-\Omega(1)} \tag{20}$$

for both the reversible and the irreversible cascades. The hidden constants in the O and Ω notations are independent of n, p and ρ.

Proof. Immediate from Theorem 5 and the fact that relation (7) holds with $\sigma \in \{\sigma_\rho^{\mathrm{rev}}, \sigma_\rho^{\mathrm{irr}}\}$. □

For the irreversible cascades on $G(n,p)$ with $p \in [\,(1+\epsilon)\,(\ln\,(e/\rho))/n, 1\,]$, Corollary 6 implies the following polynomial-time algorithm for finding with probability $1-o(1)$ a set of $O(\lceil \rho n \rceil)$ seeds activating all vertices at the end: First, pick a set S of $C\lceil \rho n \rceil$ seeds *arbitrarily*, where $C > 0$ is a sufficiently large constant. Second, pick all the vertices in $[\,n\,] \setminus \sigma_\rho^{\mathrm{irr}}(S)$ also as seeds. The number of seeds thus picked is $O(\lceil \rho n \rceil)$ with probability $1-o(1)$ because, by Corollary 6, an irreversible cascade with $C\lceil \rho n \rceil$ seeds cannot stop activating vertices until at least $n - C\lceil \rho n \rceil$ vertices are activated. It is asymptotically optimal for $p \in [\,\beta\,(\ln(e/\rho))/(\rho n), 1\,]$, where $\beta > 0$ is a sufficiently large constant [11]. We note that results of Ackerman et al. [1] can also be used to show the existence of $O(\lceil \rho n \rceil)$ seeds activating all vertices at the end for the irreversible cascades with $p \in [\,(1+\epsilon)\,(\ln\,(e/\rho))/n, 1\,]$.

The next theorem shows that the range $p \in [\,(1+\epsilon)\,(\ln\,(e/\rho))/n, 1\,]$ in Corollary 6 cannot be widened to $p \in [\,(1-\epsilon)\,(\ln\,(e/\rho))/n, 1\,]$. The proof follows a standard analysis on the number of isolated vertices of the Erdős-Rényi random graphs.

Theorem 7. *Let $\epsilon \in (0,1)$ be a constant. For any $n \in \mathbb{Z}^+$, $\rho \in [\,1/n, 1\,]$, $p \in [\,0, (1-\epsilon)\,(\ln\,(e/\rho))/n\,]$ and with probabilities taken over the random graphs $G(n,p)$,*

$$\Pr\left[\, G(n,p) \text{ has stable sets of sizes } \Omega\left(\rho^{1-\epsilon/2}n\right) \text{ and } n - \Omega\left(\rho^{1-\epsilon/2}n\right)\,\right] = 1 - o(1)$$

for both the reversible and the irreversible cascades. The hidden constants in the Ω notations are independent of p.

Proof. It is implicit in [42, Theorem 8.5.22] that for $p \in [\,0, (1-\epsilon)\,(\ln\,(e/\rho))/n\,]$, the number of isolated vertices of $G(n,p)$ is $\Omega\left(\rho^{1-\epsilon/2}n\right)$ with probability $1-o(1)$. The theorem follows because both the set of isolated vertices and that of non-isolated vertices are stable for either the reversible or the irreversible cascade.

References

[1] Ackerman, E., Ben-Zwi, O., Wolfovitz, G.: Combinatorial model and bounds for target set selection. Theoretical Computer Science (forthcoming 2010), doi:10.1016/j.tcs.2010.08.021

[2] Agur, Z.: Resilience and variability in pathogens and hosts. IMA Journal on Mathematical Medicine and Biology 4(4), 295–307 (1987)

[3] Agur, Z.: Fixed points of majority rule cellular automata with application to plasticity and precision of the immune system. Complex Systems 5(3), 351–357 (1991)

[4] Agur, Z., Fraenkel, A.S., Klein, S.T.: The number of fixed points of the majority rule. Discrete Mathematics 70(3), 295–302 (1988)

[5] Balogh, J., Bollobás, B., Morris, R.: Bootstrap percolation in high dimensions. Combinatorics, Probability and Computing 19(5-6), 643–692 (2010)

[6] Berger, E.: Dynamic monopolies of constant size. Journal of Combinatorial Theory Series B 83(2), 191–200 (2001)
[7] Blume, L.E.: The statistical mechanics of strategic interaction. Games and Economic Behavior 5(3), 387–424 (1993)
[8] Bollobás, B.: Random Graphs, 2nd edn. Cambridge University Press (2001)
[9] Chang, C.-L., Lyuu, Y.-D.: Spreading messages. Theoretical Computer Science 410(27-29), 2714–2724 (2009)
[10] Chang, C.-L., Lyuu, Y.-D.: Bounding the Number of Tolerable Faults in Majority-Based Systems. In: Calamoneri, T., Diaz, J. (eds.) CIAC 2010. LNCS, vol. 6078, pp. 109–119. Springer, Heidelberg (2010)
[11] Chang, C.-L., Lyuu, Y.-D.: Spreading of messages in random graphs. Theory of Computing Systems 48(2), 389–401 (2011)
[12] Chen, N.: On the approximability of influence in social networks. In: Proceedings of the 19th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 1029–1037 (2008)
[13] Dreyer, P.A., Roberts, F.S.: Irreversible k-threshold processes: Graph-theoretical threshold models of the spread of disease and of opinion. Discrete Applied Mathematics 157(7), 1615–1627 (2009)
[14] Ellison, G.: Learning, local interaction, and coordination. Econometrica 61(5), 1047–1071 (1993)
[15] Flocchini, P.: Contamination and decontamination in majority-based systems. Journal of Cellular Automata 4(3), 183–200 (2009)
[16] Flocchini, P., Geurts, F., Santoro, N.: Optimal irreversible dynamos in chordal rings. Discrete Applied Mathematics 113(1), 23–42 (2001)
[17] Flocchini, P., Královič, R., Ružička, P., Roncato, A., Santoro, N.: On time versus size for monotone dynamic monopolies in regular topologies. Journal of Discrete Algorithms 1(2), 129–150 (2003)
[18] Flocchini, P., Lodi, E., Luccio, F., Pagli, L., Santoro, N.: Dynamic monopolies in tori. Discrete Applied Mathematics 137(2), 197–212 (2004)
[19] Ginosar, Y., Holzman, R.: The majority action on infinite graphs: Strings and puppets. Discrete Mathematics 215(1-3), 59–71 (2000)
[20] Goles, E., Olivos, J.: Periodic behavior of generalized threshold functions. Discrete Mathematics 30(2), 187–189 (1980)
[21] Goles-Chacc, E., Fogelman-Soulie, F., Pellegrin, D.: Decreasing energy functions as a tool for studying threshold networks. Discrete Applied Mathematics 12(3), 261–277 (1985)
[22] Granville, A.: On a paper by Agur, Fraenkel and Klein. Discrete Mathematics 94(2), 147–151 (1991)
[23] Kempe, D., Kleinberg, J., Tardos, É.: Maximizing the spread of influence through a social network. In: Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 137–146 (2003)
[24] Kynčl, J., Lidický, B., Vyskočil, T.: Irreversible 2-conversion set is NP-complete. Technical Report KAM-DIMATIA Series 2009-933, Department of Applied Mathematics, Charles University, Prague, Czech Republic (2009)
[25] Luccio, F.: Almost exact minimum feedback vertex set in meshes and butterflies. Information Processing Letters 66(2), 59–64 (1998)
[26] Luccio, F., Pagli, L., Sanossian, H.: Irreversible dynamos in butterflies. In: Proceedings of the 6th International Colloquium on Structural Information and Communication Complexity, pp. 204–218 (1999)

[27] Mitzenmacher, M., Upfal, E.: Probability and Computing: Randomized Algorithms and Probabilistic Analysis. Cambridge University Press (2005)
[28] Montanari, A., Saberi, A.: Convergence to equilibrium in local interaction games. In: Proceedings of the 50th Annual IEEE Symposium on Foundations of Computer Science, pp. 303–312 (2009)
[29] Moran, G.: Parametrization for stationary patterns of the r-majority operators on 0-1 sequences. Discrete Mathematics 132(1-3), 175–195 (1994)
[30] Moran, G.: The r-majority vote action on 0-1 sequences. Discrete Mathematics 132(1-3), 145–174 (1994)
[31] Moran, G.: On the period-two property of the majority operator in infinite graphs. Transactions of the American Mathematical Society 347(5), 1649–1667 (1995)
[32] Morris, S.: Contagion. Review of Economic Studies 67(1), 57–78 (2000)
[33] Motwani, R., Raghavan, P.: Randomized Algorithms. Cambridge University Press (1995)
[34] Mustafa, N.H., Pekec, A.: Majority Consensus and the Local Majority Rule. In: Yu, Y., Spirakis, P.G., van Leeuwen, J. (eds.) ICALP 2001. LNCS, vol. 2076, pp. 530–542. Springer, Heidelberg (2001)
[35] Peleg, D.: Size bounds for dynamic monopolies. Discrete Applied Mathematics 86(2-3), 263–273 (1998)
[36] Peleg, D.: Local majorities, coalitions and monopolies in graphs: A review. Theoretical Computer Science 282(2), 231–257 (2002)
[37] Pike, D.A., Zou, Y.: Decycling Cartesian products of two cycles. SIAM Journal on Discrete Mathematics 19(3), 651–663 (2005)
[38] Poljak, S., Sura, M.: On periodical behavior in societies with symmetric influences. Combinatorica 3(1), 119–121 (1983)
[39] Poljak, S., Turzik, D.: On an application of convexity to discrete systems. Discrete Applied Mathematics 13(1), 27–32 (1986)
[40] Reddy, T.V.T., Krishna, D.S., Rangan, C.P.: Variants of spreading messages. In: Proceedings of the 4th Workshop on Algorithms and Computation, pp. 240–251 (2010)
[41] Stauffer, D., Aharony, A.: Introduction to Percolation Theory, 2nd edn. Taylor & Francis (1994)
[42] West, D.B.: Introduction to Graph Theory, 3rd edn. Prentice-Hall, Upper Saddle River (2007)
[43] Young, H.P.: The diffusion of innovations in social networks. In: Blume, L.E., Durlauf, S.N. (eds.) Economy as an Evolving Complex System. Proceedings Volume in the Santa Fe Institute Studies in the Sciences of Complexity, vol. 3, pp. 267–282. Oxford University Press, New York (2006)
[44] Zollman, K.J.S.: Social structure and the effects of conformity. Humanities, Social Sciences and Law 172(3), 317–340 (2008)

How Not to Characterize Planar-Emulable Graphs

Markus Chimani[1,⋆], Martin Derka[2,⋆⋆],
Petr Hliněný[2,⋆⋆⋆], and Matěj Klusáček[2,⋆⋆⋆]

[1] Algorithm Engineering, Friedrich-Schiller-University Jena, Germany
markus.chimani@uni-jena.de
[2] Faculty of Informatics, Masaryk University Brno, Czech Republic
{hlineny,xderka,xklusac1}@fi.muni.cz

Abstract. We investigate the question of which graphs have *planar emulators* (a locally-surjective homomorphism from some finite planar graph)—a problem raised in Fellows' thesis (1985) and conceptually related to the better known planar cover conjecture by Negami (1986). For over two decades, the planar emulator problem lived poorly in a shadow of Negami's conjecture—which is still open—as the two were considered equivalent. But, in the end of 2008, a surprising construction by Rieck and Yamashita falsified the natural "planar emulator conjecture", and thus opened a whole new research field. We present further results and constructions which show how far the planar-emulability concept is from planar-coverability, and that the traditional idea of likening it to projective embeddability is actually very out-of-place. We also present several positive partial characterizations of planar-emulable graphs.

1 Introduction

A graph G has a *planar emulator (cover)* H if H is a finite planar graph and there exists a homomorphism from H onto G that is locally surjective (bijective, respectively). In such a case we also say that G is *planar-emulable (-coverable)*. See Def. 2.1 for a precise definition, and Fig. 1 for a simple example. Informally, every vertex of G is *represented* by one or more vertices in H such that the following holds: Whenever two nodes v and u are adjacent in G, any node representing v in H has *at least one* (in case of an emulator) or *exactly one* (in case of a cover) adjacent node in H that represents u. Conversely, no node representing v in H has a neighbor representing u if v, u are nonadjacent in G.

Coarsely speaking, the mutually similar concepts of planar covers and planar emulators both "preserve" the local structure of a graph G while "gaining" planarity for it. Of course, the central question is which nonplanar graphs do have planar covers or emulators.

⋆ M. Chimani has been funded by a Carl-Zeiss-Foundation juniorprofessorship.
⋆⋆ M. Derka has been supported by Masaryk University internal grant for students.
⋆⋆⋆ Supported by the Czech science foundation; grants P202/11/0196 and GIG/11/E023.

C.S. Iliopoulos and W.F. Smyth (Eds.): IWOCA 2011, LNCS 7056, pp. 106–120, 2011.
© Springer-Verlag Berlin Heidelberg 2011

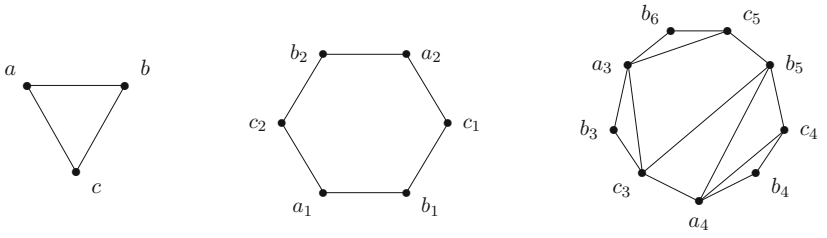


Fig. 1. Examples of a planar cover (center) and a planar emulator (right) of the triangle $G = K_3$ (left). We simply denote by a_j, $j = 1, 2, \ldots$ the vertices representing a of G, and analogically with b, c.

The two concepts emerged independently from works of Fellows [5,6] (emulator) and Negami [16,17,18] (cover). On the one hand, the class of planar-coverable graphs is relatively well understood. At least, we have the following:

Conjecture 1.1 (Negami [17], 1988). *A graph has a (finite) planar cover if and only if it embeds in the projective plane.*

Yet, this natural (see below) and firmly believed conjecture is still open today despite of more than 20 years of intensive research. See [11] for a recent survey.

On the other hand, it was no less natural to assume [5,6] that the property of being planar-emulable coincides with planar-coverability. By definition, the latter immediately implies the former. For the other direction, it was highly counterintuitive to assume that, having more than one neighbors in H representing the same adjacent vertex of G, could ever help to gain planarity of H —such "additional" edges seem to go against Euler's bound on the number of edges of a planar graph. Hence, it was widely believed:

Conjecture 1.2 (Fellows [6], 1988, falsified 2008). *A graph has a (finite) planar emulator if and only if it embeds in the projective plane.*

Perhaps due to similarity to covers, no significant effort to specifically study planar-emulable graphs occurred during the next 20 years after Fellows' manuscript [6].

Today, however, we know of one important difference between the two cases: Conjecture 1.2 is false! In 2008, Rieck and Yamashita [19] proved the truly unexpected breakthrough result that there are graphs which have planar emulators, but no planar covers and do not embed in the projective plane; see Theorem 2.4. This finding naturally ignited a new research direction, on which we report herein. We show that the class of planar-emulable graphs is, in fact, *much larger* than the class of planar-coverable ones; that the concept of projective embeddability seems very out-of-place in the context of planar emulators; and generally, how poorly planar emulators are yet understood.

Apart from its pure graph theoretic appeal, research regarding planar emulators and covers may in fact have algorithmic consequences as well: While

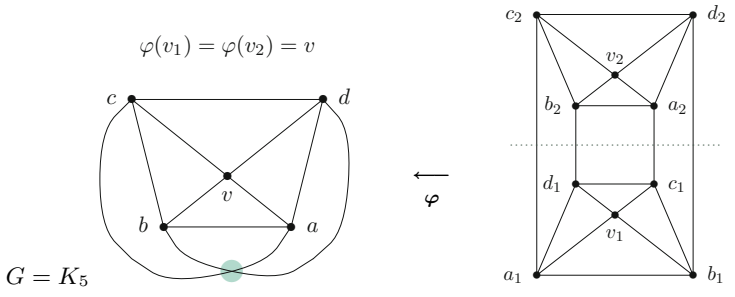


Fig. 2. The graph $G = K_5$ (left) and its two-fold planar cover (right) via a homomorphism φ. The cover is obtained for a "crosscap-less" drawing of G and its mirror image.

Negami's main interest [16] was of pure graph theoretic nature, Fellows [5, and personal communication] considered computing motivation for emulators. Additionally, we would like to sketch another potential algorithmic connection; there are problems that are NP-hard for general graphs, but polynomial-time solvable for planar graphs (e.g., maximum cut), or where the polynomial complexity drops when considering planar graphs (e.g., maximum flow). Yet, the precise breaking point is usually not well understood. Considering such problems for planar-emulable or planar-coverable graphs may give more insight into the problems' intrinsic complexities. Before this can be investigated, however, these classes first have to be reasonably well understood themselves. Our paper aims at improving upon this latter aspect of planar emulators.

This paper is organized as follows: Section 2 discusses all the major prior findings w.r.t. covers and emulators, including the aforementioned result by Rieck and Yamashita. Then, Theorem 2.5 presents our main new improvement. Section 3 reviews some necessary basic properties and tools, most of which have been previously sketched in [6]. In Section 4 we give previously unknown emulator constructions, proving Theorem 2.5 and also showing how unrelated emulators are from covers. We would particularly like to mention a very small and nicely-structured emulator of the notoriously difficult graph $K_{1,2,2,2}$ in Fig. 8. Finally, in Section 5 we study how far one can get in the pursuit to characterize planar-emulable graphs with the structural tools previously used in [12] for covers, and where the current limits are.

Due to space restrictions, many arguments and constructions have to be skipped in this paper, and we refer to the long preprint version [2] for the rest.

2 On Planar Covers and Emulators

We restate the problem on a more formal level. All considered graphs are simple, finite, and undirected. A *projective plane* is the simplest nonorientable surface—a plane with one crosscap (informally, a place in which a bunch of selected edges

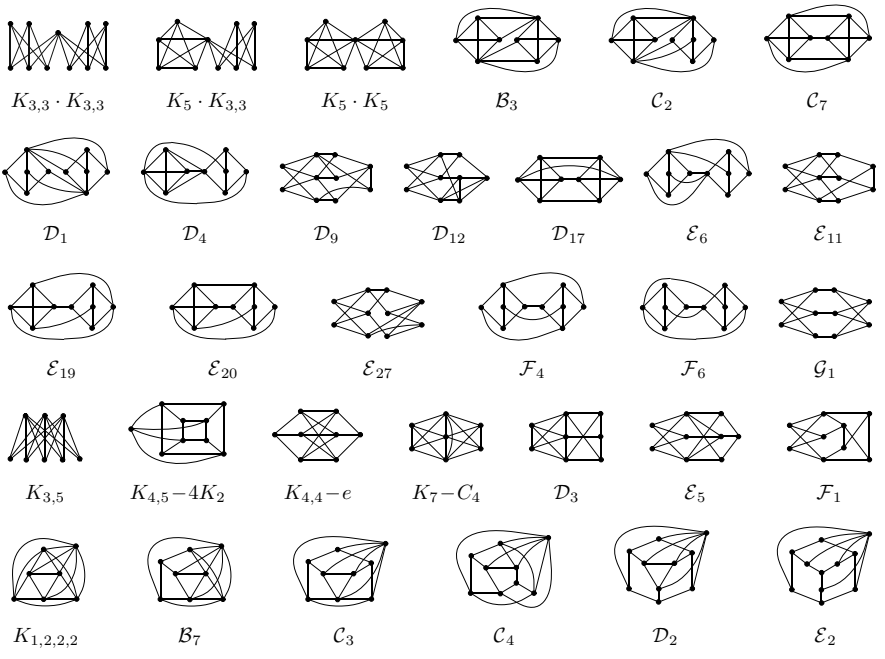


Fig. 3. The 32 connected projective forbidden minors. (The three disconnected ones, $K_5 + K_5$, $K_5 + K_{3,3}$, $K_{3,3} + K_{3,3}$, are skipped since they are not important here.)

of an embedded graph may "cross" each other). A graph *homomorphism* of H into G is a mapping $h : V(H) \to V(G)$ such that, for every edge $\{u, v\} \in E(H)$, we have $\{h(u), h(v)\} \in E(G)$.

Definition 2.1. *A graph G has a* planar emulator (cover) *H if H is a planar finite graph and there exists a graph homomorphism $\varphi : V(H) \to V(G)$ such that, for every vertex $v \in V(H)$, the neighbors of v in H are mapped by φ surjectively (bijectively) onto the neighbors of $\varphi(v)$ in G. The homomorphism φ is called an* emulator (cover) projection.

One immediately obtains the following two claims:

Lemma 2.2. *a) If H is a planar cover of G, then H is also a planar emulator of G. The converse is not true in general.*
b) If G embeds in the projective plane, then G has a two-fold planar cover (i.e., $|\varphi^{-1}(u)| = 2$ for all $u \in V(G)$); cf. [16]. See also Fig. 2.

These two claims, together with some knowledge about universal coverings in topology, make Conjectures 1.1 and 1.2 sound very plausible. To precisely describe the motivation for our research direction in planar emulators, we briefly comment on the methods that have been used in the investigation of planar-coverable graphs, too.

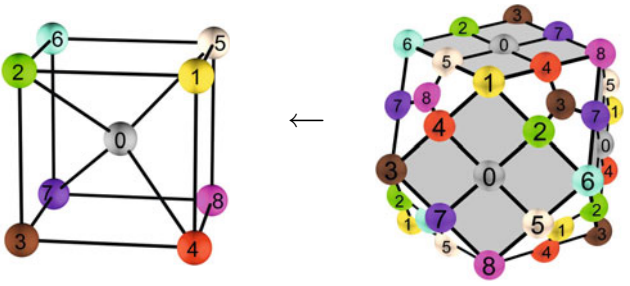


Fig. 4. A colour-coded 3D-rendering of a planar emulator patched on a polyhedral body (right) for the graph $K_{4,5} - 4K_2$ (left), taken from `http://vivaldi.ics.nara-wu.ac.jp/~yamasita/emulator/`

Firstly, we note that the properties of planar-coverability and planar-emulability are closed under taking minors (Proposition 3.1), and all 35 minor-minimal nonprojective graphs (*projective forbidden minors*, Fig. 3) are known [1]. If a connected graph G is projective, then G is planar-coverable (and hence also planar-emulable); otherwise, G contains one of the mentioned projective forbidden minors. Hence to prove Conjecture 1.1, only a seemingly simple task remains: we have to show that the known 32 connected projective forbidden minors have no planar covers. The following was established through a series of previous papers:

Theorem 2.3 (Archdeacon, Fellows, Hliněný, and Negami, 1988–98). *If the (complete four-partite) graph $K_{1,2,2,2}$ has no planar cover, then Conjecture 1.1 is true.*

One can naturally think about applying the same arguments to planar emulators, i.e. to Conjecture 1.2. The first partial results of Fellows [6]—see an overview in Section 3—were, in fact, encouraging. Yet, all the more sophisticated tools (of structural and discharging flavor) used to show the non-existence of planar covers in Theorem 2.3 fail on a rather technical level when applied to emulators. As these problems seemed to be more of technical than conceptual nature, Fellows' conjecture was always believed to be true until the following:

Theorem 2.4 (Rieck and Yamashita [19], 2008). *The graphs $K_{1,2,2,2}$ and $K_{4,5} - 4K_2$ do have planar emulators (cf. Fig. 4). Consequently, the class of planar-emulable graphs is strictly larger than the class of planar-coverable graphs, and Conjecture 1.2 is false.*

We remark that this is not merely an existence result, but the actual (and, surprisingly, not so large) emulators were published together with it. Both $K_{1,2,2,2}$ and $K_{4,5} - 4K_2$ are among the projective forbidden minors, and $K_{4,5} - 4K_2$ has already been proved not to have a planar cover.

One important new message of our paper is that Theorem 2.4 is not a rarity—quite the opposite, many other nonprojective graphs have planar emulators. In

particular we prove that, among the projective forbidden minors that have been in doubt since Fellows' [6], all except possibly $K_{4,4}-e$ do have planar emulators:

Theorem 2.5. *All of the graphs (Fig. 3)* $K_{4,5}-4K_2$, $K_{1,2,2,2}$, $\mathcal{B}_7$, $\mathcal{C}_3$, $\mathcal{C}_4$, $\mathcal{D}_2$, $\mathcal{E}_2$, *and also* K_7-C_4, $\mathcal{D}_3$, $\mathcal{E}_5$, $\mathcal{F}_1$ *have planar emulators.*

Consequently, the class of planar-emulable graphs is much larger than the class of planar-coverable ones. We refer to Section 4 for details.

3 Basic Properties of Emulators

In this section, we review the basic established properties of planar-emulable graphs. These are actually all the properties of planar-coverable graphs which are known to extend to planar emulators (though, the extensions of some of the proofs are not so straightforward).

The claims presented here, except for Theorem 3.4, were proved or sketched already in the manuscript [6] of Fellows. However, since [6] has never been published, we consider it appropriate to include their full statements with proof sketches here (while the complete formal proofs can be found also in [2]).

We begin with two crucial closure properties.

Proposition 3.1 (Fellows [6]). *The property of being planar-emulable is closed under taking* minors*; i.e., under taking subgraphs and edge contractions.*

Proposition 3.2 (Fellows [6]). *The property of being planar-emulable is closed under applying* $Y\Delta$-transformations*; i.e., the operations replacing (successively) any degree-*3 *vertex with a triangle on its three neighbors.*

Next, we identify some basic forbidden minors for planar-emulable graphs among the known list of projective forbidden minors (cf. Lemma 2.2 b). These again extend folklore knowledge about planar-coverable graphs, but the arguments are definitely not trivial this time. Actually, the following two theorems represent all the current knowledge about non-planar-emulable graphs (besides the trivial cases of K_7 and $K_{4,4}$, for which the nonexistence of planar emulators is immediate from Euler's formula).

Theorem 3.3 (Fellows [6]). *A planar-emulable graph cannot contain "two disjoint k-graphs" (see [2]). Consequently, each of the* 19 *graphs—projective forbidden minors—in the first three rows of Fig. 3 has no planar emulator.*

Theorem 3.4 (Fellows / Huneke [13]). *The graph* $K_{3,5}$ *has no planar emulator.*

In a remaining limited space we try to briefly outline the two important technical tools used to prove Theorems 3.3 and 3.4. Lemma 3.5 particularly implies both Proposition 3.2 and Theorem 3.4 with simple arguments.

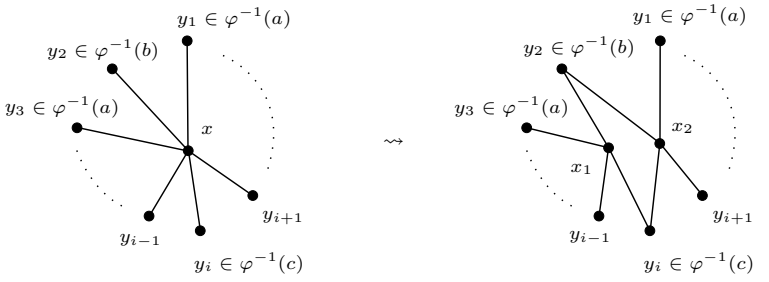


Fig. 5. Splitting vertex x with a cubic image in X into vertices of lower degree

Lemma 3.5 (Fellows [6]). *Let G be a planar-emulable graph and $X \subseteq V(G)$ an independent set of vertices of degree 3. Then there exists a planar emulator H of G with a projection $\varphi : V(H) \to V(G)$ such that every vertex $u \in \varphi^{-1}(v)$ over all $v \in X$ is of degree 3.*

Proof (sketch). Whenever F is an emulator of our graph G with a projection $\psi : V(F) \to V(G)$; let $Dg(F)$ (≥ 3) shortly denote the maximal F-degree of the vertices $u \in \psi^{-1}(v)$ over all $v \in X$. We choose H as a planar emulator of G with projection φ such that the value $Dg(H)$ is minimized.

If $Dg(H) > 3$, then we choose any vertex $x \in \varphi^{-1}(v)$ where $v \in X$ such that x is of H-degree $Dg(H) = d > 3$. Let a, b, c be the three neighbors of v in G. The neighbors of x in H naturally define a cyclic word over the alphabet $\{a, b, c\}$, and we analyze its structure in three easy cases, showing in each of them how the degree of x can be decreased (while touching only the neighbors of x). The most interesting case is a "split" illustrated in Fig. 5. The proof then proceeds inductively, and we skip the remaining details. □

On the other hand, Theorem 3.3 is implied by the next Lemma 3.6. For motivation we briefly explain that the property to "contain two disjoint k-graphs" roughly means that a graph has two minors, each isomorphic to nonplanar K_5 or $K_{3,3}$, that "overlap" one another in one vertex (which may be formed by the other graph). Validity of Theorem 3.3 then follows from a suitable local application of the following:

Lemma 3.6 (Fellows [6]). *In every planar emulator H of a nonplanar connected graph G with the projection $\varphi : V(H) \to V(G)$, the following holds: $|\varphi^{-1}(v)| \geq 2$ for each $v \in V(G)$.*

Proof (sketch). The claim is proved separately for $G = K_5$, $G = K_{3,3}$, and then it is routinely extended to all nonplanar graphs using Proposition 3.1. We illustrate here the first case $G = K_5$:

Suppose, for a contradiction, that $\varphi^{-1}(w) = \{x\}$ for some $w \in V(K_5)$ and $x \in V(H)$. Then $H - x$ is an emulator of $K_4 = K_5 - w$, and $H - x$ is outerplanar, i.e. all its vertices are incident with one face since they are all adjacent to the

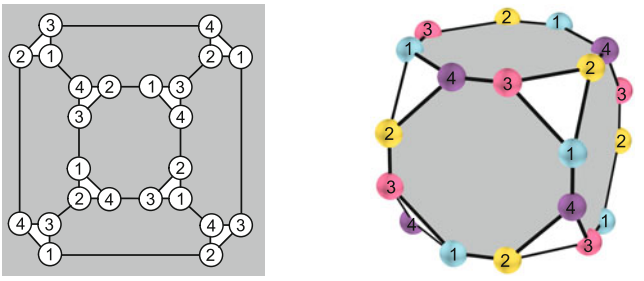


Fig. 6. A planar emulator (actually, a cover) for the complete graph K_4 with the rich faces depicted in gray colour. The same figure in a "polyhedral" manner on the right.

same vertex x in H. However, all degrees in $H - x$ are at least 3 while an outerplanar simple graph must contain a vertex of degree ≤ 2, a contradiction. □

4 Constructing New Planar Emulators

The central part of this paper deals with new constructions of planar emulators which consequently give the proof of Theorem 2.5. In this section we sketch the interesting (and in some sense central) emulators for the graphs $\mathcal{E}_2$ and $K_7 - C_4$ (Fig. 3), while a more detailed description together with emulators for the rest of the graphs discussed in Theorem 2.5 can be found in [2]. We remark that, to our best knowledge, no planar emulators of nonprojective graphs other than those mentioned in Theorem 2.4 have been studied or published prior to our paper. Moreover, using our systematic techniques we have succeeded in finding a much smaller emulator for $K_{1,2,2,2}$ than the one presented by Rieck and Yamashita in [19].

Planar Emulator for $\mathcal{E}_2$. In order to obtain an easily understandable description of an emulator for $\mathcal{E}_2$, we note the following: A graph isomorphic to $\mathcal{E}_2$ (in Fig. 3) can be constructed from the complete graph K_4 on $V(K_4) = \{1, 2, 3, 4\}$ by subdividing each edge once, calling the new vertices *bi-vertices*, and finally introducing a new vertex 0 adjacent to all the bi-vertices.

A similar sketch can be applied to a construction of a planar emulator for $\mathcal{E}_2$: If one can find a planar emulator for K_4 with the additional property that each edge is incident to at least one *rich* face—i.e., a face bordered by representatives of all edges of K_4, then a planar emulator for $\mathcal{E}_2$ can be easily derived from this. More precisely, if H_0 is such a special emulator of K_4, see an example in Fig. 6, then the following construction is applied. Each edge of H_0 is subdivided with a new vertex representing the corresponding bi-vertex of $\mathcal{E}_2$, and a new vertex representing the vertex 0 of $\mathcal{E}_2$ is added to every rich face of H_0 such that it is adjacent to all the subdividing vertices within this face. The resulting plane graph H clearly is an emulator for $\mathcal{E}_2$ (and this construction is reversible).

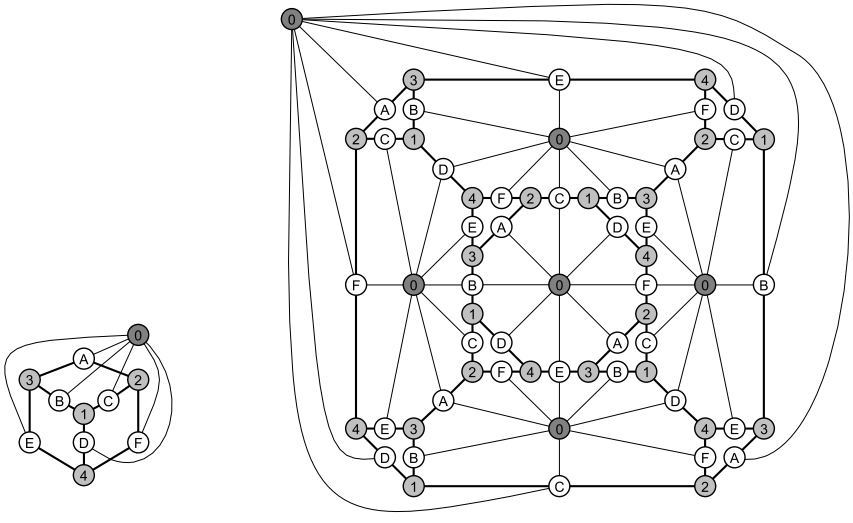


Fig. 7. A planar emulator for $\mathcal{E}_2$. The bi-vertices of the construction are in white and labeled with letters, while the numbered core vertices (cf. Fig. 6) are in gray.

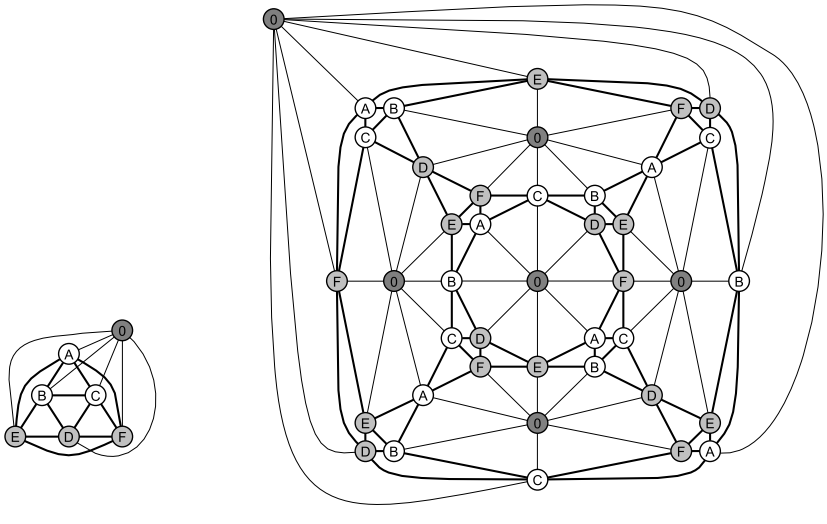


Fig. 8. A planar emulator for $K_{1,2,2,2}$; obtained by taking $Y\Delta$-transformations on the core vertices labeled $1, 2, 3, 4$ of the $\mathcal{E}_2$ emulator from Fig. 7.

Perhaps the simplest possible such an emulator for K_4 with rich faces is depicted in Fig. 6 (left). This leads to the nicely structured planar emulator for the graph $\mathcal{E}_2$ in Fig. 7. It is also worth to note that the same core ideas which helped us to find this emulator for $\mathcal{E}_2$, were actually used in [10] to prove the

nonexistence of a planar cover for $\mathcal{E}_2$. This indicates how different the coverability and emulability concepts are from each other, too.

More Emulators Derived from the $\mathcal{E}_2$ Case. By Proposition 3.2, the property of having a planar emulator is closed under taking $Y\Delta$-transformations. Moreover, the proof is constructive, and we may use it to mechanically produce new emulators from existing ones (this principle goes even slightly beyond straightforward $Y\Delta$-transformations, see Section 5). Therefore we can easily obtain an alternative emulator for $K_{1,2,2,2}$ (cf. Theorem 2.4) which is significantly smaller and simpler than the original one in [19]. The emulator is presented in Fig. 8.

Furthermore, in the same mechanical way, we can obtain planar emulators for other members of the "$K_{1,2,2,2}$-family"; namely for $\mathcal{B}_7$, $\mathcal{C}_3$, $\mathcal{D}_2$ in Fig. 3. On the other hand, finding a planar emulator for the last member, $\mathcal{C}_4$, seems to be a more complicated case—the smallest one currently has 338 vertices [2].

Planar Emulator for $K_7 - C_4$. Already the survey [11]—when commenting on the surprising Rieck–Yamashita construction—stressed the importance of deciding whether the graph $K_7 - C_4$ is planar-emulable. Its importance is tied with the structural search for all potential nonprojective planar-emulable graphs; see [12,3] and Section 5 for a detailed explanation. Briefly saying, $K_7 - C_4$ (and its "family" of $\mathcal{D}_3$, $\mathcal{E}_5$, $\mathcal{F}_1$; Fig. 3) are the only projective forbidden minors which have planar emulators and are not "internally 4-connected". In fact, for several reasons we believed that $K_7 - C_4$ cannot have a planar emulator, and so it came as another surprise when we have just recently discovered one.

In order to describe our planar emulator construction for $K_7 - C_4$, it is useful to divide the vertex set of $K_7 - C_4$ into three groups: the triple of *central vertices* (named $1, 2, 3$ in Fig. 10 left) adjacent to all other vertices, and the two vertex pairs (named A, B and C, D) each of which has connections only to its mate and to the central triple. This view allows us to identify a *skeleton* of the potential emulator as the subgraph induced on the vertices representing the central triple $1, 2, 3$ and place the remaining vertices representing A, B and C, D into the skeleton faces, provided certain additional requirements are met.

This simple idea leads to the introduction of basic building blocks (Fig. 9), each of which "almost" emulates the subgraph induced on 1,2,3,A,B and

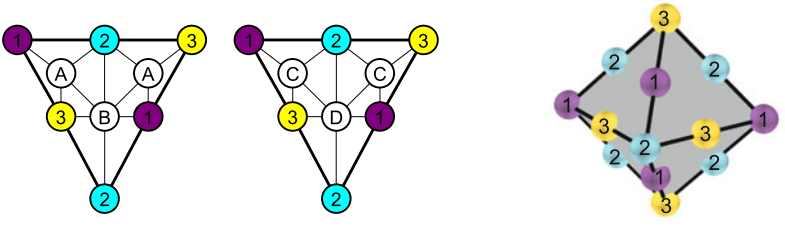


Fig. 9. Basic building blocks for our $K_7 - C_4$ planar emulator: On the left, only vertex 2 misses an A-neighbor and 1,3 miss a B-neighbor. Analogically on the right. The right-most picture shows the skeleton of the emulator in a "polyhedral" manner.

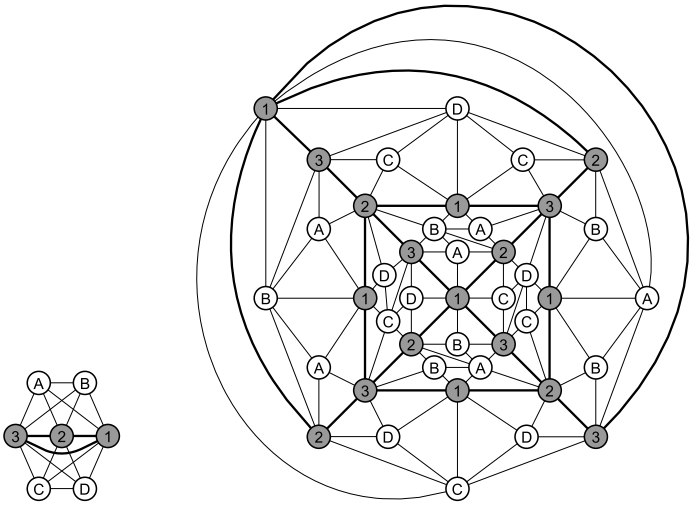


Fig. 10. A planar emulator for $K_7 - C_4$, constructed from the blocks in Fig. 9. The skeleton representing the central vertices is drawn in bold.

1,2,3,C,D, respectively. The crucial property of the blocks is that the vertices labeled A,B or C,D have all the required neighbors in place. Finally, four copies of each of the blocks can be arranged in the shape of an *octahedron* such that all missing requirements in the blocks are satisfied. The resulting planar emulator is in Figure 10.

Similar, though much more involved, procedures lead to constructions of planar emulators for the graphs $\mathcal{D}_3$, $\mathcal{E}_5$, $\mathcal{F}_1$ (which are $Y\Delta$-transformable to $K_7 - C_4$). Those emulators have 126, 138, and 142 vertices, respectively, and we refer readers to an illustration in Figure 11 and the full description in [2].

5 Structural Search: How Far Can We Go?

Until now, we have presented several newly discovered planar emulators of nonprojective graphs. Unfortunately, despite the systematic construction methods introduced in Section 4, we have got nowhere closer to a real understanding of the class of planar-emulable graphs. It is almost the other way round—the new planar emulators evince more and more clearly how complicated the problem is. Hence, we also need to consider a different approach.

The structural search method, on which we briefly report in this section, is directly inspired by previous [12]; we refer to [3,4] for closer details which cannot fit into this paper.

The general idea can be outlined as follows: If H is a mysterious nonprojective planar-emulable graph, then H must contain one of the projective forbidden minors, say F, while F cannot be among those forbidden minors not having

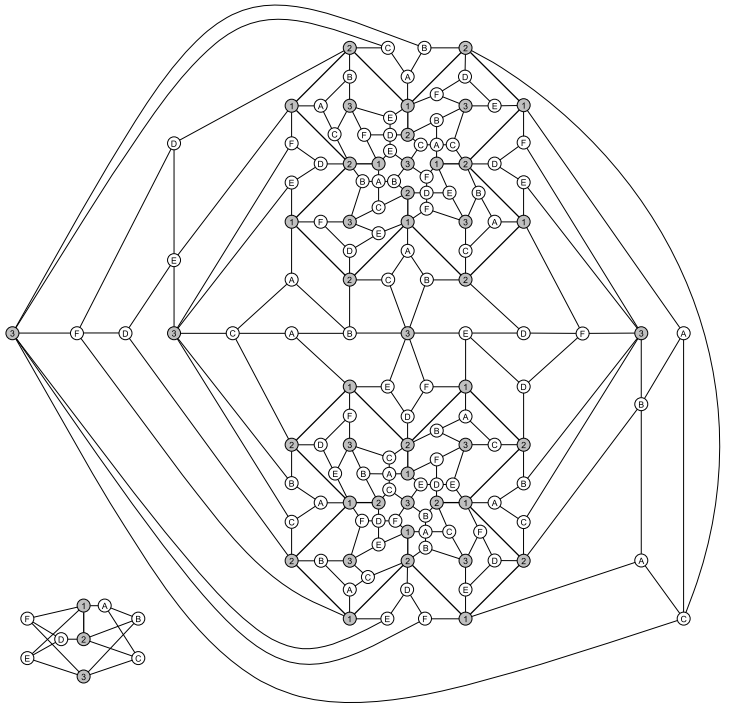


Fig. 11. A planar emulator for $\mathcal{F}_1$

planar emulators (Theorems 3.3, 3.4). Now there are basically three mutually exclusive possibilities:

i. H is a planar expansion of a smaller graph. A graph H is a *planar expansion* of G if it can be obtained by repeatedly substituting a vertex of degree ≤ 3 in G by a planar subgraph with the attachment vertices on the outer face.
ii. H contains a nonflat 3-separation. A separation in a graph is called *flat* if one of the sides has a plane drawing with all the boundary vertices on the outer face.
iii. H is *internally* 4*-connected*, i.e., it is 3-connected and each 3-separation in H has one side inducing the subgraph $K_{1,3}$ (informally, H is 4-connected up to possible degree-3 vertices with stable neighborhood).

We denote by $\langle K_7 - C_4 \rangle = \{K_7 - C_4, \mathcal{D}_3, \mathcal{E}_5, \mathcal{F}_1\}$ the *family of* $K_7 - C_4$. The underlying idea is that all the graphs in a family are $Y\Delta$-transformable to the family's base graph. Particularly the family of $K_7 - C_4$ comprises all the projective forbidden minors in question which are not internally 4-connected. See in Fig. 3.

In the case (i.) above, we simply pay attention to the smaller graph G. In the case (ii.), one can argue that either the projective forbidden minor F (in H) itself contains a nonflat 3-separation (so $F \in \langle K_7 - C_4 \rangle$), or F is internally 4-connected and H then is not planar-emulable (a contradiction). The former is left for further investigation. Finally, in the case (iii.) we may apply a so-called *splitter theorem* for internally 4-connected graphs [14], provided that F is also internally 4-connected. This leads to a straightforward computerized search which has a high chance to finish in finitely many steps, producing all such desired internally 4-connected graphs H.

Actually, when the aforementioned procedure was applied to the planar cover case in [12], the search was so efficient that the outcome could have been described by hand; giving all 16 specific graphs that potentially might be counterexamples to Conjectures 1.1. In our emulator case, we get the following:

Theorem 5.1 ([4]). *Let H be a nonprojective planar-emulable graph. Then, H is a planar expansion of one of specific* 175 *internally 4-connected graphs, or H contains a minor isomorphic to a member of $\{\mathcal{E}_2, K_{4,5} - 4K_2\} \cup \langle K_7 - C_4 \rangle$.*

Up to this point, we have not been successful in finishing the computations for the graphs $F = K_{4,5} - 4K_2$ and $\mathcal{E}_2$, due to the high complexity of the generated extensions. Yet, we strongly believe that it is possible to obtain finite results also for those cases, perhaps with the help of an improved generating procedure. On the other hand, the cases starting with $F \in \langle K_7 - C_4 \rangle$ will need an alternative procedure, e.g., using so-called "separation bridging". This is subject to future investigations.

6 Conclusion and Further Questions

While our paper presents new and surprising findings about planar-emulable graphs, the truth is that these findings are often negative in the sense that they bring *more intriguing questions than answers*. Of course, the fundamental open question in the area is to find a characterization of the class of planar-emulable graphs in terms of some other natural (and preferably topological) graph property. Even coming up with a plausible conjecture (cf. Conjecture 1.1) would be of high interest, but, with our current knowledge, already this seems to be out of reach yet.

Instead, we suggest to consider the following specific (sub)problems:

- Is there a planar emulator of the graph $K_{4,4} - e$? We think the answer is *no*, but are currently unable to find a proof, e.g. extending the arguments of [8].
- The emulators shown in Section 4 suggest that we can, in some mysterious way, reflect ΔY-transformations in emulator constructions (i.e., the converse direction of Proposition 3.2). Such a claim cannot be true in general since, e.g., a $Y\Delta$-transformation of the graph $\mathcal{D}_4$ (Fig. 3) leads to a strict subgraph of $\mathcal{B}_3$, which therefore has a two-fold planar cover while $\mathcal{D}_4$ is not planar-emulable by Theorem 3.3. But where is the precise breaking point?

– The two smallest projective forbidden minors are on 7 vertices, $K_7 - C_4$ (missing four edges of a cycle) and $K_{1,2,2,2}$ (missing three edges of a matching). Both of them, however, have planar emulators while their common supergraph K_7 does not. What is a minimal subgraph of K_7 not having a planar emulator? Can we, at least, find a short argument that the graph $K_7 - e$ has no planar emulator?
– Finally, Conjecture 1.1 can be reformulated in a way that a graph has a planar cover iff it has a two-fold planar cover. The results of [12] moreover imply that the minimal required fold number for planar-covers is bounded by a constant. Although, in the emulator case, the numbers of representatives for each vertex of the emulated graph differ, there is still a possibility of a fixed upper bound on them: Is there a constant K such that every planar-emulable graph H has a planar emulator with projection ψ such that $|\psi^{-1}(v)| \leq K$ for all $v \in V(H)$? A computerized search as in Section 5 would be of great help in this task.

References

1. Archdeacon, D.: A Kuratowski Theorem for the Projective Plane. J. Graph Theory 5, 243–246 (1981)
2. Chimani, M., Derka, M., Hliněný, P., Klusáček, M.: How Not to Characterize Planar-emulable Graphs. ArXiv e-prints 1107.0176, http://arxiv.org/abs/1107.0176
3. Derka, M.: Planar Graph Emulators: Fellows' Conjecture. Bc. Thesis, Masaryk University, Brno (2010), http://is.muni.cz/th/255724/fi_b/thesis.pdf
4. Derka, M.: Towards Finite Characterization of Planar-emulable Non-projective Graphs. Congressus Numerantium, 207–211 (submitted, 2011)
5. Fellows, M.: Encoding Graphs in Graphs. Ph.D. Dissertation, Univ. of California, San Diego (1985)
6. Fellows, M.: Planar Emulators and Planar Covers (1988) (unpublished manuscript)
7. Glover, H., Huneke, J.P., Wang, C.S.: 103 Graphs That Are Irreducible for the Projective Plane. J. of Comb. Theory Ser. B 27, 332–370 (1979)
8. Hliněný, P.: $K_{4,4} - e$ Has No Finite Planar Cover. J. Graph Theory 27, 51–60 (1998)
9. Hliněný, P.: Planar Covers of Graphs: Negami's Conjecture. Ph.D. Dissertation, Georgia Institute of Technology, Atlanta (1999)
10. Hliněný, P.: Another Two Graphs Having no Planar Covers. J. Graph Theory 37, 227–242 (2001)
11. Hliněný, P.: 20 Years of Negami's Planar Cover Conjecture. Graphs and Combinatorics 26, 525–536 (2010)
12. Hliněný, P., Thomas, R.: On possible counterexamples to Negami's planar cover conjecture. J. of Graph Theory 46, 183–206 (2004)
13. Huneke, J.P.: A Conjecture in Topological Graph Theory. In: Robertson, N., Seymour, P.D. (eds.) Graph Structure Theory. Contemporary Mathematics, Seattle, WA, vol. 147, pp. 387–389 (1991/1993)
14. Johnson, T., Thomas, R.: Generating Internally Four-Connected Graphs. J. Combin. Theory Ser. B 85, 21–58 (2002)

15. Klusáček, M.: Construction of planar emulators of graphs. Bc. Thesis, Masaryk University, Brno (2011), http://is.muni.cz/th/324101/fi_b/bc_thesis.pdf
16. Negami, S.: Enumeration of Projective-planar Embeddings of Graphs. Discrete Math. 62, 299–306 (1986)
17. Negami, S.: The Spherical Genus and Virtually Planar Graphs. Discrete Math. 70, 159–168 (1988)
18. Negami, S.: Graphs Which Have No Finite Planar Covering. Bull. of the Inst. of Math. Academia Sinica 16, 378–384 (1988)
19. Rieck, Y., Yamashita, Y.: Finite planar emulators for $K_{4,5} - 4K_2$ and $K_{1,2,2,2}$ and Fellows' Conjecture. European Journal of Combinatorics 31, 903–907 (2010)

Testing Monotone Read-Once Functions

Dmitry V. Chistikov

Faculty of Computational Mathematics and Cybernetics
Moscow State University, Russia
dch@cs.msu.ru

Abstract. A checking test for a monotone read-once function f depending essentially on all its n variables is a set of vectors M distinguishing f from all other monotone read-once functions of the same variables. We describe an inductive procedure for obtaining individual lower and upper bounds on the minimal number of vectors $T(f)$ in a checking test for any function f. The task of deriving the exact value of $T(f)$ is reduced to a combinatorial optimization problem related to graph connectivity. We show that for almost all functions f expressible by read-once conjunctive or disjunctive normal forms, $T(f) \sim n/\ln n$. For several classes of functions our results give the exact value of $T(f)$.

1 Introduction

A Boolean function of variables X is called *monotone read-once* iff it can be expressed by a formula over $\{\wedge, \vee\}$ without repetitions of variables (such formulae are also called read-once). By definition, we say that 0 and 1 are monotone read-once functions too. One can see that f depends essentially on a variable x_i iff x_i appears in a read-once formula for f.

Suppose that f is a monotone read-once function depending essentially on all variables from X; then a set M of input vectors is a *checking test* (or simply a *test*) for f iff for each monotone read-once function $f' \not\equiv f$ of variables X there exists a vector $\alpha \in M$ such that $f'(\alpha) \neq f(\alpha)$. In other words, M is a checking test for f iff values of f on vectors from M allow one to distinguish between f and all other monotone read-once functions of variables X.

The *length* of a test is the number of vectors contained in it. For a read-once function f, the minimal length of a checking test for f is denoted by $T(f)$. Any test for f having this length is called *optimal* or, equivalently, *minimal*.

The problem of *checking* for read-once functions and study of minimal test length were suggested by A. A. Voronenko, whose paper [9] investigated this problem for the basis of all binary Boolean functions (in this setting the definition of a read-once function is appropriately generalized; most further results are available in English; see, e. g., [11]). It was proved that every n-variable read-once function over this basis has a checking test of length less or equal to $4\binom{n}{2}$. This universal bound was subsequently lowered to match a trivial individual lower bound of $\binom{n}{2} + n + 1$ for an n-ary disjunction $x_1 \vee \ldots \vee x_n$ [6]. It is known, though, that there exist individual functions allowing checking tests of length

C.S. Iliopoulos and W.F. Smyth (Eds.): IWOCA 2011, LNCS 7056, pp. 121–134, 2011.
© Springer-Verlag Berlin Heidelberg 2011

$O(n)$ [8,12]. Generalizations of these results to the case of arbitrary Boolean bases are discussed in [11,13]. For the basis $\{\wedge, \vee, \neg\}$, a universal upper bound of $7n/2$ is proved in [10]. This bound has recently been improved to reach $2n + 1$ [2], whereas the highest known individual lower bound is equal to $n + 1$.

In this paper, we study minimal test length for individual monotone read-once functions (these functions are read-once over the basis $\{\wedge, \vee\}$). It is known [1] that for any such function f depending essentially on n variables it holds that $2\sqrt{n} \leq T(f) \leq n+1$. An example of a function requiring $n+1$ vectors in a checking test is an n-ary disjunction $x_1 \vee \ldots \vee x_n$, whereas the best known individual upper bound is $3\sqrt{n}-1$, for a special subsequence of *CNF-expressible* functions, i. e., those expressible by monotone read-once conjunctive normal forms (CNF). It is also known that all CNF-expressible functions f have $T(f) \geq 2\sqrt{2}\sqrt{n} - 1$.

We demonstrate that for any monotone read-once function f sets of *zeros* and *ones* (false and true points) of f in an optimal test can be chosen independently: for instance, one cannot reduce the number of zeros at the expense of adding several extra ones. If we denote by $T_0(f)$ and $T_1(f)$ the smallest possible number of zeros and ones, respectively, in a checking test for f, this result is stated as

$$T(f) = T_0(f) + T_1(f) \ .$$

For CNF-expressible functions f, it turns out that the value of $T(f)$ is primarily determined by $T_0(f)$. More precisely, for almost all such functions f depending essentially on n variables (regarded as mappings from $\{0,1\}^n$ to $\{0,1\}$) it holds that

$$T(f) \sim T_0(f) = l(f) \sim \frac{n}{\ln n} \ ,$$

where $l(f)$ is the number of clauses in the CNF expressing f, and the equality in the center holds for *all* CNF-expressible f. Interestingly, the task of deriving the exact value of $T_1(f)$ reveals a curious combinatorial optimization problem related to graph connectivity. Our bounds on the optimal solution to this problem show that for a CNF with $r_1 \geq \ldots \geq r_l$ variables in its clauses it holds that

$$T_1(f) \geq \max\left\{ r_1 + r_2 - 1,\ \log_2\left(\sum_{i=1}^{l} 2^{r_i - 1} - l + 1\right) + 1 \right\} \ ,$$

$$T_1(f) \leq \min\left\{ \sum_{i \neq 3k} (r_i - 1),\ 4\,(\max\{r_1, l\} - 1),\ \sum_{i=1}^{\lceil \log_2(l+1) \rceil} (r_i - 1) \right\} + 1 \ .$$

For the general case of monotone read-once functions, we show that the task of deriving the exact value of $T(f)$ for an individual function f can be performed by an inductive procedure traversing a read-once formula for f. Calculations on each step involve determining the optimal solution to the combinatorial problem mentioned above. For any function f our results allow one to easily obtain individual lower and upper bounds on the value of $T(f)$. The known universal upper bound of $n + 1$ can be deduced as a simple corollary. For one class of read-once functions, we present a simple way of computing exact values of $T(f)$ using read-once formulae for f.

Note that our definition of a checking test requires that all variables of f be essential, while alternatives f' may well have fictitious variables. This restriction is imposed so that the definition would be word-to-word identical to that for wider bases. Take, for instance, the basis $\{\wedge, \vee, \neg\}$. If the restriction on f is not imposed, then any checking test for $f(x_1, \ldots, x_n) \equiv 0$ must distinguish it from all conjunctions of n literals and, therefore, must contain all 2^n vectors. Such a setting should be considered degenerate.

For monotone read-once functions considered in this paper, however, such a restriction can be freely lifted. Indeed, if a monotone read-once function f depends essentially on variables $x_1, \ldots, x_n$, and g is obtained from f by adding fictitious variables $y_1, \ldots, y_m$, then a checking test for g can be obtained from a checking test M for f by extending all zeros of f in M with values $y_1 = \ldots = y_m = 1$, and all ones of f in M with values $y_1 = \ldots = y_m = 0$. Hence, the length of an optimal checking test for g is equal to that for f.

2 Combinatorial Reduction

Let f be a monotone read-once function of variables X. Denote by $R(f)$ a graph on vertices X such that an edge $\{x_i, x_k\}$ is present in $R(f)$ iff f has a projection equivalent to $x_i \wedge x_k$. A classic result of V. A. Gurvich [4] (see also [5]) states that all projections of f that depend essentially on exactly two variables x_i and x_k are equal to each other, equivalent either to $x_i \wedge x_k$ or to $x_i \vee x_k$, and that at least one such projection exists. Every monotone read-once function is uniquely determined by its graph $R(f)$. We shall use a widely known fact that the complement of any nontrivial connected induced subgraph of $R(f)$ is disconnected ($R(f)$ is a *cograph* [3]). We say that a formula $\mathcal{F}$ is *of type* $\wedge$ ($\vee$) iff it is either a variable or a conjunction (a disjunction) of two or more subformulae. A monotone read-once function f is *of type* $\circ \in \{\wedge, \vee\}$ iff it can be expressed by a read-once formula of type $\circ$ over $\{\wedge, \vee\}$. Clearly, f is of type $\wedge$ iff $R(f)$ is connected.

We need the following notation. An *integer partition* is a way of representing an integer as a sum of several positive integers. When referring to integer partitions, we write $m = t_1 + \cdots + t_p$, where $m, p, t_1, \ldots, t_p \geq 1$. We also use one operation on equivalence relations. Suppose that ϵ' and ϵ'' are equivalence relations on a set S; then by $\epsilon' \vee \epsilon''$ we denote the transitive closure of the union of ϵ' and ϵ''. Thus, $\epsilon' \vee \epsilon''$ is itself an equivalence relation ϵ such that $a \sim_\epsilon b$ iff there exists a sequence $c_0, \ldots, c_k \in S$ such that $a = c_0$, $b = c_k$ and for each $i = 1, \ldots, k$ either $c_{i-1} \sim_{\epsilon'} c_i$ or $c_{i-1} \sim_{\epsilon''} c_i$. In other words, $\epsilon' \vee \epsilon''$ is the finest equivalence relation on S that is coarser than both ϵ' and ϵ''. By **true** we denote the binary all-true relation on a set. Finally, we use symbols **0** and **1** to denote vectors consisting only of zeros and ones, respectively.

Suppose that $l \geq 1$ and $r_1, \ldots, r_l \geq 1$. Take arbitrary positive integers $t_{i,j}$ for $1 \leq i \leq l$ and $1 \leq j \leq r_i$. Denote by F the multiset of l integer partitions $m_i = t_{i,1} + \cdots + t_{i,r_i}$, for $i = 1, \ldots, l$. Define $L(F)$ as the smallest number t having the following property: there exist equivalence relations $\epsilon_1, \ldots, \epsilon_l$ on the

set $\{1,\dots,t\}$ such that each ϵ_i has r_i equivalence classes of cardinality greater or equal to $t_{i,1},\dots,t_{i,r_i}$, respectively, and $\epsilon_i \vee \epsilon_k = \mathbf{true}$ whenever $i \neq k$. If $t_{i,j} = 1$ for all possible i, j, the number $L(F)$ will also be referred to as $L(r_1,\dots,r_l)$.

Theorem 1. *Let $l \geq 2$ and $r_1,\dots,r_l \geq 1$. Suppose that $f_{i,j}$, $1 \leq j \leq r_i$, $1 \leq i \leq l$, are monotone read-once functions of type $\wedge$ depending on disjoint sets of variables, $f_i = f_{i,1} \vee \ldots \vee f_{i,r_i}$ for $i = 1,\dots,l$, $f = f_1 \wedge \ldots \wedge f_l$, and $f_{i,1}$ is a single variable whenever $r_i = 1$. Then*

$$T(f) = T_0(f) + T_1(f) \ ,$$

$$T_0(f) = \sum_{i=1}^{l} T_0(f_i) \ , \quad \text{and} \quad T_1(f) = L(F) \ ,$$

where F is the multiset of integer partitions $T_1(f_{i,1}) + \cdots + T_1(f_{i,r_i})$ for all $i = 1,\dots,l$.

Proof. First obtain the lower bounds. Suppose that a checking test M for f contains a vector α such that $f(\alpha) = 1$. Clearly, replacing α with any vector $\beta \leq \alpha$ such that $f(\beta) = 1$ yields a set of vectors that retains the property of being a checking test. Similarly, one can replace all vectors γ such that $f(\gamma) = 0$ with vectors $\delta \geq \gamma$ such that $f(\delta) = 0$. The obtained set M' will still constitute a checking test for f, and $|M'| \leq |M|$ (note that it may be the case that, e. g., different vectors α' and α'' can be replaced with a single vector β). Further on, we assume that all possible replacements have been performed, i. e., M' contains only *lower ones* and *upper zeros* of f.

Now take an arbitrary upper zero α of f. Let α be a concatenation of vectors $\alpha_1,\dots,\alpha_l$ such that each f_i depends essentially on the variables assigned by α_i. Since f is a read-once conjunction of all f_i, $1 \leq i \leq l$, this means that there exists a unique index i such that $f_i(\alpha_i) = 0$ and $f_k(\alpha_k) = 1$ for all $k \neq i$. Moreover, it follows that α_i is an upper zero of f_i and all $\alpha_k = \mathbf{1}$ for $k \neq i$. Denote by z_i the number of all vectors α in M' such that $f_i(\alpha_i) = 0$. Since M' is a checking test for f, it follows that $z_i \geq T_0(f_i)$. (Indeed, if $z_i < T_0(f_i)$, then there exists a monotone read-once function $f_i' \not\equiv f_i$ which depends on the same variables as f_i and agrees with it on all z_i vectors α_i. It then follows that f cannot be distinguished from the function f' obtained by substituting f_i' for f_i in the read-once formula for f.) Observe that no vector α can be counted more than once in $z_1,\dots,z_l$, for all non-constant monotone read-once functions take the value of 1 at $\mathbf{1}$. Thus, $T_0(f) \geq \sum_{i=1}^{l} z_i \geq \sum_{i=1}^{l} T_0(f_i)$.

In order to prove the lower bound on $T_1(f)$, consider the set $\{\alpha^{(1)},\dots,\alpha^{(t)}\}$ of all lower ones of f contained in M'. One can see that an arbitrary lower one α of f is a concatenation of $\alpha_1,\dots,\alpha_l$ such that $f_i(\alpha_i) = 1$ for all i. Moreover, each α_i must be a lower one of f_i. Hence, each α_i is a concatenation of $\alpha_{i,1},\dots,\alpha_{i,r_i}$, and there exists a unique index j such that $f_{i,j}(\alpha_{i,j}) = 1$ and all $\alpha_{i,s} = \mathbf{0}$ for $s \neq j$. For each $\alpha^{(p)}$, these indices will be denoted by $j_1(p),\dots,j_l(p)$.

For each $i = 1,\dots,l$ consider an equivalence relation ϵ_i on $\{1,\dots,t\}$ such that $p \sim_{\epsilon_i} q$ iff $j_i(p) = j_i(q)$. We claim that $\epsilon_i \vee \epsilon_k = \mathbf{true}$ if $i \neq k$. Assume

the converse, then there exists a non-empty proper subset S of $\{1,\ldots,t\}$ such that for all $p \in S$ and $q \in \{1,\ldots,t\} \setminus S$ it holds that $p \not\sim_{\epsilon_i} q$ and $p \not\sim_{\epsilon_k} q$. By definition, put $I = \{ j_i(p) \mid p \in S \}$ and $K = \{ j_k(p) \mid p \in S \}$. Construct a monotone read-once function f' by replacing the conjunction $f_i \wedge f_k$ in the read-once formula for f with a disjunction $(f_i' \wedge f_k') \vee (f_i'' \wedge f_k'')$, where

$$f_i' = \bigvee_{j \in I} f_{i,j} \ , \qquad f_k' = \bigvee_{j \in K} f_{k,j} \ ,$$
$$f_i'' = \bigvee_{j \notin I} f_{i,j} \ , \qquad f_k'' = \bigvee_{j \notin K} f_{k,j} \ .$$

We see that f is always greater or equal to f' and disagrees with it only on vectors α such that either $f_i'(\alpha_i) \wedge f_k''(\alpha_k) = 1$ or $f_i''(\alpha_i) \wedge f_k'(\alpha_k) = 1$. Since M' is a checking test, such a vector α must be present among $\alpha^{(1)},\ldots,\alpha^{(t)}$. Assume without loss of generality that $f_i'\left(\alpha_i^{(p)}\right) \wedge f_k''\left(\alpha_k^{(p)}\right) = 1$. By definition of f_i' and f_k'', it holds that $j_i(p) \in I$ and $j_k(p) \notin K$, so $p \in S$ and $p \notin S$, which is a contradiction.

In order to prove that $T_1(f) \geq L(F)$, we show that $j_i(p) = j$ for at least $T_1(f_{i,j})$ numbers $p \in \{1,\ldots,t\}$. Indeed, if this is not the case, then M' contains fewer than $T_1(f_{i,j})$ vectors α such that $\alpha_{i,j} \neq \mathbf{0}, \mathbf{1}$, $f_{i,j}(\alpha_{i,j}) = 1$ and $f(\alpha) = 1$. By definition of $T_1(f_{i,j})$, this means that these $\alpha_{i,j}$ do not allow one to distinguish between $f_{i,j}$ and a certain monotone read-once function $f_{i,j}'$. (Note that $\mathbf{1}$ is included in a minimal checking test for $f_{i,j}$ iff $T_1(f_{i,j}) = 1$, otherwise it is obviously of no use.) Substituting $f_{i,j}'$ for $f_{i,j}$ in the read-once formula for f yields a formula expressing a monotone read-once function $f' \not\equiv f$ such that f agrees with f' on all vectors from M'. This concludes the proof of the lower bounds.

We now prove the upper bounds and the equality $T(f) = T_0(f) + T_1(f)$. We use induction on the depth of the read-once formula for f. In the inductive step, we shall use only the fact that $T(f_i) = T_0(f_i) + T_1(f_i)$ as an inductive assumption. Therefore, we need to check this equality for single variables, conjunctions and disjunctions. It can easily be checked that for $n \geq 1$,

$$T_0(x_1 \vee \ldots \vee x_n) = 1 \ , \quad T_1(x_1 \vee \ldots \vee x_n) = n \ , \quad T(x_1 \vee \ldots \vee x_n) = n+1 \ ,$$
$$T_0(x_1 \wedge \ldots \wedge x_n) = n \ , \quad T_1(x_1 \wedge \ldots \wedge x_n) = 1 \ , \quad T(x_1 \wedge \ldots \wedge x_n) = n+1 \ ,$$

so we proceed to the main part of the proof.

Let $M_1,\ldots,M_l$ be optimal checking tests for $f_1,\ldots,f_l$, respectively, all consisting of upper zeros and lower ones of f_i. Let N consist of all vectors $\alpha = (\mathbf{1},\ldots,\mathbf{1},\alpha_i,\mathbf{1},\ldots,\mathbf{1})$ for all $\alpha_i \in M_i$ such that $f_i(\alpha_i) = 0$ and $i = 1,\ldots,l$. Now suppose that $L(F) = t$ and equivalence relations $\epsilon_1,\ldots,\epsilon_l$ on $\{1,\ldots,t\}$ satisfy the conditions of $L(F)$ definition. Assume that equivalence classes of each ϵ_i are numbered 1 through r_i so that jth class's cardinality is greater or equal to $T_1(f_{i,j})$. Now recall that every vector $\alpha_i \in M_i$ such that $f_i(\alpha_i) = 1$ is a lower one of f_i and thus has a special representation as a concatenation of $\alpha_{i,j}$, $1 \leq j \leq r_i$. Put $U_{i,j} = \{ \alpha_{i,j} \mid \alpha_i \in M_i,\ f_i(\alpha_i) = 1,\ \alpha_{i,j} \neq \mathbf{0} \}$. For each $p = 1,\ldots,t$ put

$\alpha^{(p)} = \left(\alpha_1^{(p)}, \ldots, \alpha_l^{(p)}\right)$, where $\alpha_i^{(p)} = \left(\alpha_{i,1}^{(p)}, \ldots, \alpha_{i,r_i}^{(p)}\right)$ such that $\alpha_{i,j}^{(p)} \in U_{i,j}$ if the number p belongs to the jth equivalence class of ϵ_i and $\alpha_{i,j}^{(p)} = \mathbf{0}$ otherwise. We also require that for each valid pair i, j the set of all non-$\mathbf{0}$ vectors $\alpha_{i,j}^{(p)}$ be equal to $U_{i,j}$. This is possible because the number of elements in jth equivalence class of ϵ_i is greater or equal to $|U_{i,j}| = T_1(f_{i,j})$ (this equality follows from the inductive assumption and our choice of M_i, for it is easily checked that $\sum_{j=1}^{r_i} |U_{i,j}| = T_1(f_i)$). We claim that $M = N \cup \{\alpha^{(1)}, \ldots, \alpha^{(t)}\}$ is a checking test for f.

We show that if a monotone read-once function f' coincides with f on all vectors from M, then $f' \equiv f$. This is sufficient both for proving the claimed upper bound and, as a corollary, for establishing the equality $T(f) = T_0(f) + T_1(f)$. First, observe that $f'(\alpha^{(p)}) = 1$ for all $p = 1, \ldots, t$. Since f' is monotone, it follows that $f'(\alpha) = 1$ for any α of type $\left(\mathbf{1}, \ldots, \mathbf{1}, \alpha_i^{(p)}, \mathbf{1}, \ldots, \mathbf{1}\right)$, where $p = 1, \ldots, t$. By our construction of M, all vectors from M_i are present in $\left\{ \alpha_i^{(p)} \mid \alpha^{(p)} \in M \right\}$. Since M contains all vectors from N, it follows that $f'(\alpha) = f(\alpha)$ for all $\alpha = (\mathbf{1}, \ldots, \mathbf{1}, \alpha_i, \mathbf{1}, \ldots, \mathbf{1})$, where $\alpha_i \in M_i$. One concludes, then, that for each $i = 1, \ldots, l$ the function f' has a projection equivalent to f_i.

Now we are going to reconstruct the graph $R(f') = R(f)$ using the values of f on vectors from M. Note that all subgraphs $R_i = R(f_i)$ are already known. Recall that all functions $f_{i,j}$ are of type $\wedge$, so all $R_{i,j}$ are connected. On the contrary, each R_i is either disconnected or a single vertex. It suffices to show that each subgraph $R_i \cup R_k$ of $R(f')$ (a subgraph of $R(f')$ induced by vertices of R_i and R_k) is connected if $i \neq k$, for then *all* edges between R_i and R_k must be present in it, otherwise any edge between R_i and R_k not present in $R_i \cup R_k$ would imply the connectivity of $R_i \cup R_k$'s complement, which contradicts the connectivity of $R_i \cup R_k$ (recall that $R(f')$ is a cograph). We now contract all vertices in each $R_{i,j}$ ($R_{k,j}$) to a single vertex j (j') and prove the connectivity of the obtained bipartite graph R' on vertices $\{1, \ldots, r_i\} \cup \{1', \ldots, r_k'\}$.

We first claim that the equality $f'(\alpha^{(p)}) = 1$ implies that the edge $\{j_i(p), j_k(p)\}$ is present in R'. Indeed, since $f'(\alpha^{(p)}) = 1$, it holds that $f'(\beta) = 1$, where β is obtained from $\alpha^{(p)}$ by changing all $\alpha_u^{(p)}$ to $\mathbf{1}$ for $u \neq i, k$. Suppose that γ is obtained from β by changing a 1 in α_i to 0. One now observes that replacing α_k with $\mathbf{1}$ in γ yields a vector γ' with a known value $f'(\gamma') = 0$, since every α_u, where $u \neq i$, is now replaced by $\mathbf{1}$, and α_i is a lower one of the known projection f_i. Monotonicity of f' implies $f'(\gamma) = 0$. Arguing as above, we see that $f'(\delta) = 0$, where δ is obtained from β by changing a 1 in α_k to 0. The values of f' on vectors β, γ, and δ are uniquely determined by its values on vectors from M and imply that f' has a projection of the type $x' \wedge x''$, where $f_{i,j_i(p)}$ and $f_{k,j_k(p)}$ depend on x' and x'', respectively. Thus, R' contains all edges $\{j_i(p), j_k(p)\}$ for $p = 1, \ldots, t$.

It remains to prove that these edges imply the connectivity of R'. Consider a graph G on $2t$ vertices $\{1, \ldots, t\} \cup \{1', \ldots, t'\}$. Let G contain all the edges $\{p, p'\}$ for $p = 1, \ldots, t$, edges $\{p, q\}$ whenever $p \sim_{\epsilon_i} q$, and $\{p', q'\}$ whenever $p' \sim_{\epsilon_k} q'$. Contracting all the edges within $\{1, \ldots, t\}$ and $\{1', \ldots, t'\}$ yields a graph H,

which is known to be isomorphic to a subgraph of R'. Clearly, H is connected if so is G. Contracting all the edges $\{p, p'\}$ in G yields a graph G' on vertices $\{1, \ldots, t\}$ such that $\{p, q\}$ is present in G' if and only if $p \sim_{\epsilon_i} q$ or $p \sim_{\epsilon_k} q$. Since $\epsilon_i \vee \epsilon_k = \mathbf{true}$, it follows that G' is connected, and so are G, H and R'. This concludes the proof.

Corollary 2. *For all non-constant monotone read-once functions f,*

$$T(f) = T_0(f) + T_1(f) \ .$$

Corollary 3. *For a monotone read-once function*

$$f = (x_{1,1} \vee \ldots \vee x_{1,r_1}) \wedge (x_{2,1} \vee \ldots \vee x_{2,r_2}) \wedge \ldots \wedge (x_{l,1} \vee \ldots \vee x_{l,r_l}) \ ,$$

where $l \geq 1$ and all $r_i \geq 1$, the following equality holds:

$$T(f) = l + L(r_1, \ldots, r_l) \ .$$

Remark 4. The statements of Theorem 1 and Corollary 3 hold true if all symbols $\wedge$ and $\vee$ are exchanged and so are Boolean constants 0 and 1. Provided that an algorithm for determining $L(F)$ is known, one can use a simple inductive procedure to determine the value of $T(f)$ for any arbitrary monotone read-once function f. The induction basis is given by the values of $T_0(f)$ and $T_1(f)$ for disjunctions and conjunctions of $n \geq 1$ variables. Since $L(F)$ is evidently monotonically non-decreasing in all parameters in F, one can substitute lower and upper bounds for the unknown parameters to obtain lower and upper bounds on $L(F)$, respectively.

3 Obtaining Bounds on $L(F)$

Proposition 5. $L(r_1, \ldots, r_l) \leq L(F) \leq L(r_1, \ldots, r_l) + \max_{1 \leq i \leq l}(m_i - r_i) - d$, *where $d = 1$ if there exist numbers $s \neq k$ such that s is a maximum point of $m_i - r_i$ and $r_k \geq 2$, and $d = 0$ otherwise.*

Proof. The lower bound is obvious and the upper bound follows from the observation that in all non-trivial cases any equivalence relation ϵ_i from the definition of $t = L(r_1, \ldots, r_l)$ has at least one equivalence class of size greater or equal to 2, so for each i such that $m_i > r_i$ one needs less or equal to $m_i - r_i - 1$ new elements beyond $1, \ldots, t$. The only exception is the case when $r_k = 1$ for all $k \neq i$.

When obtaining bounds on $L(F)$, we often use graph-theoretic terminology, as given by the following lemma. (Note that when we speak of graphs, we always mean undirected graphs without loops or multiple edges.)

Lemma 6. *The number $L(F)$ is the smallest number t having the following property: there exist graphs $G_1, \ldots G_l$ on vertices $\{1, \ldots, t\}$ such that each G_i has exactly r_i connected components of size greater or equal to $t_{i,1}, \ldots, t_{i,r_i}$, and all graphs $G_i \cup G_k$ are connected whenever $i \neq k$.*

The proof is trivial. Note that we can use arbitrary connected graphs as components of G_i. It is often convenient, though, to use only trees as these components; in this case all graphs G_i are *required* to be forests.

Theorem 7. *Suppose that F is a multiset of integer partitions $m_i = t_{i,1} + \cdots + t_{i,r_i}$ for $i = 1, \ldots, l$. Then the following inequality holds:*

$$L(F) \geq \max \left\{ \max_i m_i,\ \max_{i \neq k} r_i + r_k - 1,\ \log_2 \left(\sum_{i=1}^{l} 2^{r_i - 1} - l + 1 \right) + 1 \right\} .$$

Proof. Let t be the value of $L(F)$. One can easily see that $t \geq m_i$ for all $i = 1, \ldots, l$. Indeed, since G_i has to contain r_i connected components of size at least $t_{i,1}, \ldots, t_{i,r_i}$, it follows that $t \geq \sum_{j=1}^{r_i} t_{i,j} = m_i$.

Now suppose that $i \neq k$ and assume without loss of generality that both G_i and G_k are forests. Then the number of edges in these two graphs is $t - r_i$ and $t - r_k$, respectively. Since $G_i \cup G_k$ is connected, we see that $(t - r_i) + (t - r_k) \geq t - 1$ and $t \geq r_i + r_k - 1$.

Finally, consider partitions of $\{1, \ldots, t\}$ into two non-empty sets S' and S''. The number of such partitions is $2^{t-1} - 1$. For each graph G_i having r_i connected components there exist exactly $2^{r_i - 1} - 1$ such partitions without edges between S' and S'' in G_i. These sets of partitions must be disjoint for $i \neq k$, otherwise $G_i \cup G_k$ contains no edges between S' and S'' and, therefore, is disconnected. Hence, $2^{t-1} - 1 \geq \sum_{i=1}^{l} (2^{r_i - 1} - 1)$, which gives the desired.

In several cases the bounds of Theorem 7 turn out to be tight. Two next propositions show that all three expressions under max can give the exact value of $L(F)$ for some F.

Proposition 8. *If $l = 2$, then $L(F) = \max\{m_1,\ m_2,\ r_1 + r_2 - 1\}$.*

Proof. Let F consist of partitions $m_1 = t_1 + \cdots + t_p$ and $m_2 = s_1 + \cdots + s_q$. By Theorem 7, $L(F) \geq m$, where $m = \max\{m_1, m_2, p + q - 1\}$. Our goal is to prove an equal upper bound. Assume without loss of generality that $m_1 = m_2 \geq p + q - 1$. (If this is not the case, increase some of the numbers $t_1, \ldots, t_p$ and $s_1, \ldots, s_q$ appropriately so that m_1 and m_2 would reach m and observe that for the multiset F' of the two obtained partitions it holds that $L(F) \leq L(F')$.) We claim that for any multiset F satisfying this condition and any appropriate graph G_1 on $m = m_1$ vertices there exists a graph G_2 with the needed properties. The proof of this claim is by induction over $q \geq 1$. For $q = 1$, the desired is straightforward. Indeed, since $m = m_1$ and $m = m_2 = s_1$, one can take a complete graph on m vertices for G_2. Now suppose that $q \geq 2$. Assume that $t_1 \leq t_2 \leq \ldots \leq t_p$ and $s = \max_i s_i$. If $s = 1$, then $p = (p + q - 1) - (q - 1) \leq m - q + 1 = q - q + 1 = 1$. So, $p = 1$ and G_1 is a complete graph on m vertices, similarly to the case above. If $s \geq 2$, take connected components of size $t_1, \ldots, t_{s-1}$ and t_p in G_1, choose one vertex from each of these components and form a clique of size s on these vertices in G_2. If $p \leq s$, missing $s - p$ vertices are chosen arbitrarily and we have proved the desired without the inductive assumption. If $p > s$, the problem is reduced

to a simpler one, with $q' = q - 1$ (we use prime symbols for distinguishing a new instance of the problem from the old one), $p' = p - s + 1$, $p' + q' - 1 = (p + q - 1) - s$ and $m' = m'_1 = m'_2 = m - s$. Indeed, components of size $t_1, \ldots, t_{s-1}$ and t_p in G_1 are connected with a clique in G_2, whose s vertices are excluded from further consideration. The rest $t'_1 = (t_1 - 1) + \ldots + (t_{s-1} - 1) + (t_p - 1)$ vertices from these components (note that $t'_1 \geq 0 + \ldots + 0 + 1 = 1$) are considered to belong to the same component of G'_1 and so may be connected with, e. g., a clique in G'_1. Thus, q is decreased by 1 and the property $m' = m'_1 = m'_2 \geq p' + q' - 1$ still holds. This concludes the proof.

Proposition 9. $L(2, \ldots, 2) = \lceil \log_2(l + 1) \rceil + 1$.

Proof. The lower bound is given by Theorem 7. To prove the upper bound, take all possible graphs G_i on vertices $\{1, \ldots, t\}$ having exactly two connected components and all edges within each component (any G_i is a union of two cliques). Clearly, if G_i and G_k are two such graphs, then $G_i \cup G_k$ is connected if and only if $G_i \neq G_k$. So, if t is fixed, l can be chosen as large as $2^{t-1} - 1$. Hence, $t \leq \lceil \log_2(l + 1) \rceil + 1$.

We now present an example of using Theorem 1 for deriving exact values of $T(f)$. This example is directly related to the result of Proposition 8. We say that a formula $\mathcal{F}$ over $\{\wedge, \vee\}$ is *strictly alternating* iff it is either a variable or a disjunction (respectively, a conjunction) of exactly two strictly alternating formulae that have type $\wedge$ (respectively, $\vee$). The structure of strictly alternating formulae can be represented by binary trees with alternating levels of internal vertices labeled with $\wedge$ and $\vee$. By $\mathcal{F}^*$ we denote a formula obtained from $\mathcal{F}$ by exchanging all symbols $\wedge$ and $\vee$.

Proposition 10. *Let f be a monotone read-once function expressed by a strictly alternating read-once formula $\mathcal{F}$. By definition, put $x \star y = 3$ if $x = y = 2$ and $x \star y = \max\{x, y\}$ otherwise. Then $T(f) = E(\mathcal{F}) + E(\mathcal{F}^*)$, where $E(\mathcal{F})$ is the value of the formula obtained from $\mathcal{F}$ by changing all symbols $\vee$ to $+$, all symbols $\wedge$ to $\star$, and setting all variables' values to* 1.

Proof. Use the inductive procedure of Theorem 1. For conjunctions and disjunctions, there is nothing to prove. For all other cases, use Proposition 8. Observe that if $\max\{m_1, m_2\} < r_1 + r_2 - 1$, then $r_1 = r_2 = 2$, since each r_i is less or equal to $\min\{m_i, 2\}$. It follows that $m_1 = m_2 = 2$, which gives the desired.

Example 11. Suppose that formulae $\mathcal{F}_n$ are represented by *perfect* binary trees with $n = 2^h$ leaves. For the corresponding read-once functions f_n, if $h \geq 3$, then $T(f_n) = 3\sqrt{n}$ if h is even and $T(f_n) = 9\sqrt{2}\,/\,4 \cdot \sqrt{n}$ if h is odd.

Proof. Without loss of generality, denote by f_n a read-once function expressed by a formula $\mathcal{F}_n$ of type $\wedge$, and by f_n^* a read-once function expressed by $\mathcal{F}_n^*$. Clearly, $T_0(f_n) = T_1(f_n^*)$ and $T_1(f_n) = T_0(f_n^*)$. Straightforward application of Proposition 10 shows that

$$T(f_8) = 9 = 9\sqrt{2}\,/\,4 \cdot \sqrt{8}\ ,$$
$$T(f_{16}) = 12 = 3\sqrt{16}\ ,$$

which proves the induction basis. For $n \geq 8$, Proposition 10 reveals that

$$T_0(f_{4n}) = 2\,T_0(f^*_{2n}) = 2\,T_0(f_n) \ ,$$
$$T_1(f_{4n}) = T_1(f^*_{2n}) = 2\,T_1(f_n) \ ,$$

and it follows that

$$T(f_{4n}) = 2\,T(f_n) \ .$$

This completes the proof.

By $F_1 + F_2$ denote the sum of multisets F_1 and F_2, i. e., a multiset which contains each element of F_1 or F_2 as many times as F_1 and F_2 do together. Simple decomposition gives the following upper bound on $L(F_1 + F_2)$.

Lemma 12. $L(F_1 + F_2) \leq L(F_1) + L(F_2) - 1$.

Proof. Suppose that $t_1 = L(F_1)$ and $t_2 = L(F_2)$. Let $G^1_1, \ldots, G^1_{l_1}$ and $G^2_1, \ldots, G^2_{l_2}$ be graphs from the alternative definition of $L(F_1)$ and $L(F_2)$ given by Lemma 6. Assume without loss of generality that each G^s_i, where $s = 1, 2$, is a graph on vertices $\{(s,1), \ldots, (s,t_s)\}$. Take each G^s_i and extend it with a clique on vertices $\{(3-s,1), \ldots, (3-s,t_{3-s})\}$, and then identify vertices $(1,1)$ and $(2,1)$. The obtained graph H^s_i has $t_1 + t_2 - 1$ vertices and the same number of connected components as G^s_i. The number of vertices in each component is greater or equal to that in G^s_i. For all possible $i \neq k$, graphs $H^s_i \cup H^s_k$ are connected because so are $G^s_i \cup G^s_k$. Moreover, all $H^1_i \cup H^2_k$ for any possible i, k are connected too, for they all contain cliques on vertices $\{(1,1), \ldots, (1,t_1)\}$ and $\{(2,1), \ldots, (2,t_2)\}$, where $(1,1)$ and $(2,1)$ are one vertex. This means that $L(F_1 + F_2) \leq t_1 + t_2 - 1$.

The results of Theorem 1 and Lemma 12 lead to an inductive proof of the following known result:

Theorem 13. *For all monotone read-once functions f depending on n variables,*

$$T(f) \leq n + 1 \ .$$

Proof. Use induction on $n \geq 1$. For $n = 1$, the bound is clearly true. Suppose now that $n \geq 2$. Assume without loss of generality that $f = f_1 \wedge \ldots \wedge f_l$ and $f_i = f_{i,1} \vee \ldots \vee f_{i,r_i}$, where all $f_{i,j}$ are monotone read-once functions of type $\wedge$ depending on disjoint sets of variables, and $f_{i,1}$ is a single variable whenever $r_i = 1$. Let n_i be the number of variables of f_i. By the inductive assumption, $T(f_i) \leq n_i + 1$. By Theorem 1, $T_1(f) \leq L(F)$, where F is the multiset of integer partitions $T_1(f_{i,1}) + \cdots + T_1(f_{i,r_i})$ for $i = 1, \ldots, l$. By Lemma 12,

$$L(F) \leq \sum_{i=1}^{l} L(\{T_1(f_{i,1}) + \cdots + T_1(f_{i,r_i})\}) - l + 1 = \sum_{i=1}^{l} \sum_{j=1}^{r_i} T_1(f_{i,j}) - l + 1 \ .$$

Applying Theorem 1 three more times yields

$$T_1(f) \le \sum_{i=1}^{l} T_1(f_i) - l + 1 \ , \qquad T_0(f) = \sum_{i=1}^{l} T_0(f_i) \ ,$$

$$\text{and} \quad T(f) \le \sum_{i=1}^{l} (T(f_i) - 1) + 1 \le \sum_{i=1}^{l} n_i + 1 = n + 1 \ ,$$

which completes the proof.

To prove more accurate individual upper bounds, we need the following notation. If m is a positive integer, then $\mathbf{Z}_2^m = \{ (x_1, \ldots, x_m) \mid x_1, \ldots, x_m \in \mathbf{Z}_2 \}$ is a vector space over $\mathbf{Z}_2 = \{0, 1\}$. For an arbitrary vector $\mathbf{x} \in \mathbf{Z}_2^m$, put $\operatorname{supp} \mathbf{x} = \{ i \mid 1 \le i \le m,\ x_i = 1 \}$. The following lemma gives an alternative definition of $L(r_1, \ldots, r_l)$.

Lemma 14. *The number $L(r_1, \ldots, r_l) - 1$ is the smallest integer m having the following property: there exist linear subspaces $V_1, \ldots, V_l$ of $\mathbf{Z}_2^m$ such that $\dim V_i = r_i - 1$, $V_i \cap V_k = \{\mathbf{0}\}$ for $i \ne k$ and each V_i has a basis $\mathbf{e}_{i,1}, \ldots, \mathbf{e}_{i,r_i-1}$ with $\operatorname{supp} \mathbf{e}_{i,j'} \cap \operatorname{supp} \mathbf{e}_{i,j''} = \emptyset$ for $j' \ne j''$.*

The idea of the proof is that if $i = 1, \ldots, l$, then sets $\operatorname{supp} \mathbf{e}_{i,j}$ for $j = 1, \ldots, r_i - 1$ are equivalence classes of ϵ_i not containing $t = L(r_1, \ldots, r_l)$. Details are left to the reader. Note that we can also reformulate the definition of $L(F)$ in a way similar to that of Lemma 14. Such a definition, though, is out of our scope now. We now aim to obtain an efficient upper bound on $L(r_1, \ldots, r_l)$. For convenience, by $\mathcal{L}(\mathbf{e}_1, \ldots, \mathbf{e}_s)$ we denote the linear subspace spanned by vectors $\mathbf{e}_1, \ldots, \mathbf{e}_s$.

Proposition 15. *If $r_1 \ge r_2 \ge r_3$, then $L(r_1, r_2, r_3) = r_1 + r_2 - 1$.*

Proof. The lower bound follows from Theorem 7. To prove the upper bound, put $m = r_1 + r_2 - 2$. By $\mathbf{e}_j$ denote a vector from $\mathbf{Z}_2^m$ with $m - 1$ zeros and an only one in jth position. Consider subspaces

$$V_1 = \mathcal{L}(\mathbf{e}_1, \ldots, \mathbf{e}_{r_1-1}) \ , \qquad V_2 = \mathcal{L}(\mathbf{e}_{r_1}, \ldots, \mathbf{e}_m) \ ,$$
$$V_3 = \mathcal{L}(\mathbf{e}_1 + \mathbf{e}_{r_1},\ \mathbf{e}_2 + \mathbf{e}_{r_1+1},\ \ldots,\ \mathbf{e}_{r_3-1} + \mathbf{e}_{r_1+r_3-2}) \ .$$

Since r_3 is less or equal to both r_1 and r_2, the sets $V_i \setminus \{\mathbf{0}\}$ are pairwise disjoint. Each V_i is spanned by $r_i - 1$ linearly independent vectors without common ones, so, by Lemma 14, we get the needed upper bound.

In the next proposition, the proof of the upper bound follows from a construction of [1] due to Voronenko, and the lower bound is given by Theorem 7.

Proposition 16. *If $r_1 \ge \ldots \ge r_l$, then $L(r_1, \ldots, r_l) \le 2\,\widehat{p}(\max\{r_1, l\}) - 1 \le 4 \max\{r_1, l\} - 3$, where $\widehat{p}(k)$ is the smallest prime greater or equal to k. In particular, if $r_1 = \ldots = r_l = l$ and l is prime, then $L(l, \ldots, l) = 2l - 1$.*

Now we are ready to prove our main upper bounds.

Theorem 17. *If* $r_1 \geq r_2 \geq \ldots \geq r_l$, *then*

$$L(r_1, \ldots, r_l) \leq \min \left\{ \sum_{i \neq 3k} (r_i - 1),\ 4\,(\max\{r_1, l\} - 1),\ \sum_{i=1}^{\lceil \log_2(l+1) \rceil} (r_i - 1) \right\} + 1 \ .$$

Proof. First combine the results of Propositions 8 and 15 using Lemma 12. Put $l = 3s + d$, where $d \in \{1, 2, 3\}$, and observe that the difference $L(r_1, \ldots, r_l) - L(r_{3s+1}, \ldots, r_{3s+d})$ cannot be greater than

$$\sum_{i=0}^{s-1} L(r_{3i+1}, r_{3i+2}, r_{3i+3}) - s = \sum_{i=0}^{s-1} (r_{3i+1} + r_{3i+2} - 1) - s = \sum_{\substack{1 \leq i \leq 3s \\ i \neq 3k}} (r_i - 1) \ ,$$

which gives the first needed inequality. The second inequality follows from Proposition 16. To prove the third inequality, use Lemma 14 directly. Choose $d = \lceil \log_2(l+1) \rceil$ as the number of digits in a binary representation of l. For $i = 1, \ldots, l$ by $\alpha_s(i)$ denote sth least significant bit in a d-bit binary representation $\alpha(i) = (\alpha_d(i) \ldots \alpha_1(i))$ of i. Let π be any permutation on $\{1, \ldots, l\}$ such that $\pi(s) = 2^{s-1}$ for $s = 1, \ldots, d$. Put $u_{\pi(i)} = r_i - 1$ for $i = 1, \ldots, l$ and choose $m = \sum_{s=1}^{d} u_{\pi(s)} = \sum_{s=1}^{d} (r_s - 1)$. For $s = 1, \ldots, d$ and $j = 1, \ldots, u_{\pi(s)}$ by $\mathbf{f}_{s,j}$ denote a vector from $\mathbf{Z}_2^m$ with $m-1$ zeros and an only one in $(u_{\pi(1)} + \ldots + u_{\pi(s-1)} + j)$th position. Take

$$\mathbf{e}_{i,j} = \sum_{s=1}^{d} \alpha_s(i)\, \mathbf{f}_{s,j} \ , \qquad j = 1, \ldots, u_i, \quad i = 1, \ldots, l,$$

and consider subspaces $U_i = \mathcal{L}(\mathbf{e}_{i,1}, \ldots, \mathbf{e}_{i,u_i})$ for $i = 1, \ldots, l$. Clearly, $\dim U_i = u_i$, for $\sum_{j=1}^{u_i} \lambda_j \mathbf{e}_{i,j} = \mathbf{0}$ implies $\sum_{j=1}^{u_i} \sum_{s=1}^{d} \lambda_j \alpha_s(i)\, \mathbf{f}_{s,j} = \mathbf{0}$ and $\lambda_j = 0$ for all possible j, since vectors $\mathbf{f}_{s,j}$ are linearly independent. (Note the special case $i = 2^q$, where j can assume values greater than u_s, but it turns out that all $\alpha_s(i)$ equal 0 except for one s and we still see that all λ_j must be equal to 0.) It is easily observed that for each i sets $\operatorname{supp} \mathbf{e}_{i,j}$, where $j = 1, \ldots, u_i$, are disjoint. We claim that $U_i \cap U_k = \{\mathbf{0}\}$ whenever $i \neq k$, which is evidently sufficient for obtaining the desired bound.

Instead of using linear algebra in a straightforward way, we prove a special property of sets U_i. For each $\mathbf{x} \in \mathbf{Z}_2^m$, define $\operatorname{sig} \mathbf{x} = \{\, s \mid 1 \leq s \leq d,\ \exists\, j : \mathbf{x} \geq \mathbf{f}_{s,j}, 1 \leq j \leq u_s \,\}$. Take an arbitrary non-$\mathbf{0}$ vector $\mathbf{x} \in U_i$. Observe that

$$\mathbf{x} = \sum_{j=1}^{u_i} \lambda_j \mathbf{e}_{i,j} = \sum_{j=1}^{u_i} \lambda_j \sum_{s=1}^{d} \alpha_s(i)\, \mathbf{f}_{s,j} = \sum_{s=1}^{d} \alpha_s(i) \sum_{j=1}^{u_i} \lambda_j \mathbf{f}_{s,j} \ .$$

If $\alpha_s(i) = 1$, then j assumes the values $1, \ldots, u_i \leq u_{\pi(s)}$. Therefore, all vectors $\mathbf{f}_{s,j}$ have ones in different single components. It follows that $\operatorname{sig} \mathbf{x} = \operatorname{supp} \alpha(i)$, where $\alpha(i) = (\alpha_d(i) \ldots \alpha_1(i))$ is a d-bit binary representation of i. Hence, the sets $U_i \setminus \{\mathbf{0}\}$ are disjoint, which completes the proof.

In the next theorem, when we speak of *almost all* CNF- and DNF-expressible functions, these functions are regarded as mappings from $\{0,1\}^n$ to $\{0,1\}$. For each n, all *mappings* expressible by monotone read-once CNF or DNF are assigned equal non-zero probabilities. For instance, formulae $(x_1 \vee x_2) \wedge x_3$ and $(x_2 \vee x_1) \wedge x_3$ express the same function, different from one expressed by $(x_1 \vee x_3) \wedge x_2$.

Theorem 18. *For almost all monotone read-once CNF- and DNF-expressible functions f depending essentially on n variables,*

$$T(f) \sim \frac{n}{\ln n} \ .$$

Proof. Consider a monotone read-once CNF-expressible function f which depends essentially on n variables $x_1, \ldots, x_n$. One can easily see that there exists a one-to-one mapping ϕ between the set of all such functions and the set of all partitions of the set $\{1, \ldots, n\}$. For a function f expressed by a CNF $\mathcal{F}$, indices p and q belong to the same block in $\phi(f)$ iff they belong to the same clause in $\mathcal{F}$. Denote by l the number of clauses in $\mathcal{F}$. It is known that almost all partitions have asymptotically $n/\ln n$ blocks, all of which have size less or equal to $O(\ln n)$ (see, e. g., [7]). One observes then that the upper bound of Theorem 17 is almost always less or equal to

$$\left\lceil \log_2\left(\frac{n}{\ln n}\right) + o(1) \right\rceil \cdot O(\ln n) = O(\ln^2 n) = o\left(\frac{n}{\ln n}\right) \ ,$$

so the value of $T(f)$ given by Corollary 3 is asymptotically equivalent to $T_0(f) = l \sim n/\ln n$.

4 Conclusions

We reduced the problem of deriving the value of $T(f)$ to several instances of another combinatorial problem, that of determining the smallest number of vertices L allowing the construction of a set of graphs with special properties. Our results give several explicit bounds on L numbers and allow to deduce the implied bounds on $T(f)$ easily. For almost all CNF- and DNF-expressible functions, these bounds determine the asymptotic behaviour of $T(f)$. For arbitrary read-once functions f, one can use Theorem 1 repeatedly to obtain both lower and upper bounds on $T(f)$. For several classes of functions our results determine the exact value of $T(f)$. Finally, we remark that one can easily indicate a special class of read-once functions which shows that it is impossible to derive the exact values of $T(f)$ for *all* f without computing L.

Acknowledgements. The author wishes to thank Prof. Bruce M. Kapron from the University of Victoria, Canada, who kindly agreed to present this paper at IWOCA 2011. This research was supported by Russian Presidential grant MD–757.2011.9.

References

1. Bubnov, S.E., Voronenko, A.A., Chistikov, D.V.: Some test length bounds for non-repeating functions in the $\{\&, \vee\}$ basis. Computational Mathematics and Modeling 21(2), 196–205 (2010)
2. Chistikov, D.V.: Testing read-once functions over the elementary basis. Moscow University Computational Mathematics and Cybernetics (to appear)
3. Corneil, D.G., Lerchs, H., Stewart Burlingham, L.: Complement reducible graphs. Discrete Applied Mathematics 3(3), 163–174 (1981)
4. Gurvich, V.A.: On repetition-free Boolean functions. Uspehi Matematicheskih nauk 32(1), 183–184 (1977) (in Russian)
5. Karchmer, M., Linial, N., Newman, I., Saks, M., Widgerson, A.: Combinatorial characterization of read-once formulae. Discrete Mathematics 114(1-3), 275–282 (1993)
6. Ryabets, L.V.: Checking test complexity for read-once Boolean functions. Ser. Diskretnaya matematika i informatika, vol. 18. Izdatel'stvo Irkutskogo gosudarstvennogo pedagogicheskogo universiteta (2007) (in Russian)
7. Sachkov, V.N.: Probabilistic methods in combinatorial analysis. Encyclopedia of Mathematics and its Applications, vol. 56. Cambridge University Press (1997)
8. Voronenko, A.A.: Estimating the length of a diagnostic test for some nonrepeating functions. Computational Mathematics and Modeling 15(4), 377–386 (2004)
9. Voronenko, A.A.: On checking tests for read-once functions. In: Matematicheskie Voprosy Kibernetiki, Fizmatlit, Moscow, vol. 11, pp. 163–176 (2002) (in Russian)
10. Voronenko, A.A.: On the length of checking test for repetition-free functions in the basis $\{0, 1, \&, \vee, \neg\}$. Discrete Mathematics and Applications 15(3), 313–318 (2005)
11. Voronenko, A.A.: Recognizing the nonrepeating property in an arbitrary basis. Computational Mathematics and Modeling 18(1), 55–65 (2007)
12. Voronenko, A.A., Chistikov, D.V.: Learning read-once functions individually. Uchenye zapiski Kazanskogo universiteta. Ser. Fiziko-matematicheskie nauki 151(2), 36–44 (2009) (in Russian)
13. Voronenko, A.A., Chistikov, D.V.: On testing read-once Boolean functions in the basis B_5. In: Proceedings of the XVII International Workshop "Synthesis and complexity of control systems", pp. 24–30. Izdatel stvo Instituta matematiki, Novosibirsk (2008) (in Russian)

Complexity of Cycle Transverse Matching Problems

Ross Churchley[1], Jing Huang[2], and Xuding Zhu[3]

[1] Department of Mathematics and Statistics, University of Victoria, Victoria, B.C., Canada, V8W 3R4
rosschurchley@gmail.com
[2] Department of Mathematics and Statistics, University of Victoria, Victoria, B.C., Canada, V8W 3R4
huangj@uvic.ca
[3] Department of mathematics, Zhejiang Normal University, Jinhua, Zhejiang, People's Republic of China
xudingzhu@gmail.com

Abstract. The stable transversal problem for a fixed graph H asks whether a graph contains a stable set that meets every induced copy of H in the graph. Stable transversal problems generalize several vertex partition problems and have been studied for various classes of graphs. Following a result of Farrugia, the stable transversal problem for each C_ℓ with $\ell \geq 3$ is NP-complete. In this paper, we study an 'edge version' of these problems. Specifically, we investigate the problem of determining whether a graph contains a matching that meets every copy of H. We show that the problem for C_3 is polynomial and for each C_ℓ with $\ell \geq 4$ is NP-complete. Our results imply that the stable transversal problem for each C_ℓ with $\ell \geq 4$ remains NP-complete when it is restricted to line graphs. We show by contrast that the stable transversal problem for C_3, when restricted to line graphs, is polynomial.

Keywords: Stable transversal problem, transverse matching problem, algorithm, complexity.

1 Introduction

For a fixed graph H, an *H-transversal* of a graph G is a vertex set T that *meets* every induced copy of H in G; in other words, $G - T$ does not contain H as an induced subgraph. When such a set T is a stable set, it is called a *stable H-transversal* of G. While every graph has an H-transversal, stable H-transversals may not exist in a graph. The *stable H-transversal problem* $\mathrm{ST}(H)$ for a fixed H asks whether a graph has a stable H-transversal.

It is easy to see that a graph has a stable K_2-transversal if and only if it is bipartite and it has a stable $\overline{K_2}$-transversal if and only if it is a split graph (cf. [4]). As both bipartite and split graphs are recognizable in polynomial time, $\mathrm{ST}(H)$ is polynomial when H has at most two vertices. Farrugia [3] proved that $\mathrm{ST}(H)$ is NP-complete when H has at least three vertices and is connected. It

C.S. Iliopoulos and W.F. Smyth (Eds.): IWOCA 2011, LNCS 7056, pp. 135–143, 2011.
© Springer-Verlag Berlin Heidelberg 2011

follows that $\mathrm{ST}(C_\ell)$ is NP-complete when $\ell \geq 3$, where C_ℓ denotes the cycle with ℓ vertices. For graphs which have at least three vertices and are disconnected, there are polynomial as well as NP-complete cases, but the dichotomy of stable transversal problems has not been determined. Stable transversal problems have also been studied for various classes of graphs, cf. [7,9].

In this paper, we investigate an 'edge version' of stable transversal problems. The *H-transverse matching problem* $\mathrm{TM}(H)$ for a fixed graph H asks whether a graph G has a matching M which meets every (not necessarily induced) copy of H in G, i.e., H is not a subgraph of $G - M$. Such a matching M will be called an *H-transverse matching*. We shall prove the following:

Theorem 1. *TM(C_ℓ) is polynomial when $\ell = 3$ and NP-complete otherwise.*

Given a graph $G = (V, E)$, the *line graph* $L(G)$ of G is the graph with vertex set E and two vertices adjacent in $L(G)$ if and only if the two corresponding edges have a common endvertex in G. With the sole exception of $H = C_3$, the transverse matching problem $\mathrm{TM}(H)$ is equivalent to the stable transversal problem $\mathrm{ST}(L(H))$ for line graphs. Since $L(H) = C_\ell$ if and only if $H = C_\ell$ for each $\ell \geq 4$, Theorem 1 implies that $\mathrm{ST}(C_\ell)$ remains NP-complete for line graphs for each $\ell \geq 4$.

Due to the fact that C_3 is the line graph of two different graphs C_3 and $K_{1,3}$, $\mathrm{ST}(C_3)$ is not equivalent to $\mathrm{TM}(C_3)$ for line graphs. However, we show that determining whether a graph G has a matching M which meets every copy of C_3 and $K_{1,3}$ in G is polynomial. Hence the problem $\mathrm{ST}(C_3)$ is polynomial for line graphs and therefore we also obtain the following:

Theorem 2. *When restricted to line graphs, ST(C_ℓ) is polynomial for $\ell = 3$ and NP-complete otherwise.*

We remark that the polynomial case stated in Theorem 2 is in sharp contrast to the fact that $\mathrm{ST}(C_3)$ is NP-complete in general.

2 TM(C_3) and ST(C_3) for Line Graphs are Polynomial

A *paw* is the graph consisting of a triangle and an edge sharing a common vertex. Call a matching M *pawssible* if, for every paw consisting of triangle xyz and edge xw, $xw \in M$ implies $yz \in M$. The lemma below follows immediately from the fact that any C_3-transverse matching contains at least one of the three edges in each triangle and is pawssible.

Lemma 3. *If a graph G has a C_3-transverse matching, then at least one of the three edges in each triangle is contained in a pawssible matching.* □

In polynomial time we can check whether an edge e of G is contained in a pawssible matching and find one if it exists. This can be done by letting $F_e = \{e\}$ initially, then adding edges one by one to F_e which are 'forced by' paws, and finally checking if F_e is a matching. In the case when F_e is a matching, it is also a minimal pawssible matching containing e.

Lemma 4. *Suppose that M is a pawssible matching and e is an edge in G. If both $M \cup \{e\}$ and F_e are matchings, then $M \cup F_e$ is a pawssible matching where F_e is defined as above.*

Proof. From the definition of F_e, $M \cup F_e$ may be viewed as the set obtained from M by including e initially and then adding edges which are 'forced by' paws. The process of adding edges ensures that $M \cup F_e$ is pawssible. We claim that it produces a matching. Indeed, suppose that a paw consists of triangle xyz and edge xw such that $M \cup \{xw\}$ is a matching. Since M is pawssible, M does not contain any edge incident with y or z as otherwise M would contain xz or xy, contradicting the assumption that $M \cup \{xw\}$ is a matching. Hence $M \cup \{xw, yz\}$ is also a matching. This means that the process of adding edges always produces matchings. So $M \cup F_e$ is a matching. □

Theorem 5. *A graph G has a C_3-transverse matching if and only if every triangle has an edge that is contained in a pawssible matching.*

Proof. The necessity is treated in Lemma 3. For sufficiency, suppose that every triangle has an edge that is contained in a pawssible matching. The assumption implies in particular that every triangle has an edge e for which F_e is a pawssible matching.

We apply the following algorithm to construct a set M of edges in G: Initially, $M = \emptyset$. As long as $G - M$ contains a triangle, find an edge e which is contained in a triangle in $G - M$ and for which F_e is a matching and enlarge M to include F_e.

Clearly, the set M contains at least one edge from each triangle in G. Lemma 4 ensures that M obtained by the algorithm is a matching. Therefore M is a C_3-transverse matching in G. □

Theorem 5 and its proof suggest a polynomial time algorithm which determines if a graph G has a C_3-transverse matching and constructs one if it exists: We first compute F_e for each edge e of G as described above. If for some triangle xyz, none of F_{xy}, F_{yz}, F_{xz} is a matching, then G does not have a C_3-transverse matching according to Theorem 5. Otherwise, the algorithm in the proof of Theorem 5, which can be implemented in polynomial time, constructs a C_3-transverse matching.

Corollary 6. *TM(C_3) is polynomial.* □

Next we show how to determine, in polynomial time, whether a graph has a matching which meets every copy of C_3 and of $K_{1,3}$. It is well-known that a perfect matching (if exists) can be found in polynomial time (cf. [2,8]). We can apply such an algorithm to determine if a graph has a matching covering specified vertices.

Lemma 7. *Given a graph G and vertex set S, it can be determined in polynomial time whether G has a matching covering every vertex of S.*

Proof. We show how to convert the problem in the lemma into the problem of finding a perfect matching. Let G' be a (disjoint) copy of G with $f : G \to G'$ an isomorphism. Let G^* be obtained from $G \cup G'$ by adding the edge $vf(v)$ if $v \in V(G) - S$. If M is a matching of G covering every vertex of S, then $M \cup f(M) \cup \{vf(v) : v \text{ is not covered by } M\}$ is a perfect matching of G^*. Conversely, any perfect matching of G^* restricts to a desired matching of G. Thus G has a matching covering every vertex of S if and only if G^* has a perfect matching. □

Clearly, no graph of maximum degree ≥ 4 contains a matching which meets every copy of $K_{1,3}$. So we may restrict the search of a matching among graphs of maximum degree at most three (i.e., *subcubic* graphs).

Proposition 8. *There exists a polynomial time algorithm to determine whether a subcubic graph has a matching that meets every copy of C_3 and of $K_{1,3}$.*

Proof. We may assume without loss of generality that G is connected. We show how to reduce the problem in the proposition to the problem of determining whether a graph has a matching covering specified vertices, which can be solved in polynomial time according to Lemma 7. Specifically, we will construct a subcubic graph G' and a set $S \subseteq V(G')$ such that G has a desired matching if and only if G' has a matching covering S.

Initially, we let $G' = G$ and let S consist of all vertices of degree 3 in G'. If G' has no triangle, then G' together with S satisfy the desired properties. So assume that G' contains triangles. We modify G' and the set S recursively for every triangle. Each modification on a triangle results in a subcubic graph and maintains the property that S consists of all vertices of degree three. Moreover, any matching covering S in the resulting graph must meet the triangle.

Let abc be a triangle in G' and a', b', c' be the only other neighbours of a, b, c respectively, if they exist. If a', b', c' exist and are the same vertex, then the graph is K_4 which has a desired matching. If, without loss of generality, $a' = b'$ exist and are distinct from c' (if it exists), then we do nothing. Any matching covering a, b (which are in S) in the resulting graph meets the triangle abc. On the other hand, if a', b', c' exist and are distinct we contract the triangle, identifying the vertices a, b, c in G' and in S. Finally, if a', b', c' are distinct but do not all exist, we contract the triangle but also remove the identified vertex from S. Clearly, G has a desired matching if and only if the resulting graph has a desired matching.

By repeating the above reduction for every triangle in G', we either obtain K_4 (in which case G has a desired matching) or a graph G' and a set S such that G has a desired matching if and only if G' has a matching covering S. □

Corollary 9. *ST(C_3) for line graphs is polynomial.* □

3 TM(C_4) is NP-Complete

In this section, we show that TM(C_4) is NP-complete even when it is restricted to bipartite graphs.

Proposition 10. *TM(C_4) for bipartite graphs is NP-complete.*

Proof. We reduce from 4-SAT, which is well-known to be NP-complete [5]. Let $\mathcal{C} = \{c_1, \dots c_q\}$ be a set of clauses each consisting of four variables from $\mathcal{A} = \{a_1, a_2, \dots, a_p, \overline{a_1}, \dots, \overline{a_p}\}$. A valid truth assignment for $\mathcal{C}$ is a function $f : \mathcal{A} \to \{0, 1\}$ such that $f(a_i) \neq f(\overline{a_i})$ and $f^{-1}(1) \cap c_j \neq \emptyset$ for every $i = 1, \dots p$ and $j = 1, \dots, q$. In other words, $\overline{a_i}$ is the negation of a_i and every clause has at least one true variable. We construct in polynomial time a bipartite graph H which admits a C_4-transverse matching if and only if such a satisfiability function exists.

There are two main gadgets in our reduction: one corresponding to each variable and one corresponding to each clause. The variable gadgets (see Fig. 1, upper left) are constructed as follows. Associate each a_i, $1 \leq i \leq p$ with a path $w_0^i w_1^i w_2^i w_3^i w_4^i$; three vertices v_1^i, v_2^i, v_3^i adjacent to w_0^i and w_2^i, w_1^i and w_3^i, and w_2^i and w_4^i, respectively; three more vertices u_1^i, u_2^i, u_3^i; and nine vertices $v_{k,1}^i, v_{k,2}^i, v_{k,3}^i$ each adjacent to the respective u_k^i and v_k^i. It is easy to check that the gadget admits C_4-transverse matchings containing $w_0^i w_1^i$ and $w_2^i w_3^i$, or containing $w_1^i w_2^i$ and $w_3^i w_4^i$, and that any such matching contains one of those pairs.

The clause gadgets (see Fig. 1), on the other hand, are somewhat simpler than the variable gadgets. Each clause $c_j \in \mathcal{C}$ is associated with a $C_8 : s_1^j t_1^j s_2^j t_2^j s_3^j t_3^j s_4^j t_4^j$ and two additional vertices r_1^j, r_2^j adjacent to t_1^j, t_2^j, t_3^j and t_2^j, t_3^j, t_4^j, respectively. It turns out that this gadget has a C_4-transverse leaving any three of $s_1^i, s_2^i, s_3^i, s_4^i$ uncovered, but no C_4-transverse matching leaving all four uncovered.

We complete the construction of H by connecting the clause gadgets to the variable gadgets. Let $c_j = \{a_{j1}, a_{j2}, a_{j3}, a_{j4}\}$. If $a_{jk} = a_i$, add two new vertices x_k^j, y_k^j and edges $x_k^j y_k^j, s_k^j y_k^j, w_2^j y_k^j, w_1^i x_k^j$ and $w_3^i s_k^j$ (thus $w_1^i w_2^i w_3^i s_2^j y_k^j x_k^j$ is a C_6: see Fig. 1). On the other hand, if $a_{jk} = \overline{a_i}$, do the same thing with the roles of w_1^i and w_3^i reversed.

Suppose the resulting graph H has a C_4-transverse matching M. We interpret M as a truth assignment f according to its intersection with each variable gadget, setting

$$f(a_i) = \begin{cases} 0 & \text{if } w_1^i w_2^i, w_3^i w_4^i \in M \\ 1 & \text{if } w_0^i w_1^i, w_2^i w_3^i \in M \end{cases}$$

and $f(\overline{a_i}) \neq f(a_i)$. We must show that $f^{-1}(1) \cap c_j \neq \emptyset$ for all $j = 1, \dots q$.

The above discussion of the clause gadgets implies that, for a fixed j, the matching M contains at most three of the four edges $s_k^j y_k^j$, $k = 1, 2, 3, 4$; as M must cover at least one s_k^j with an edge from the clause gadget associated with c_j. Suppose that $a_i = a_{jk} \in c_j$. The above discussion of the variable gadgets implies that neither $w_2^i y_k^j$ nor $w_3^i s_k^j$ is in M, and since M covers the $C_4 : w_2^i w_3^i s_k^j y_k^j$ it must be the case that $w_2^i w_3^i \in M$. Consequently, $a_i \in f^{-1}(1) \cap c_j$. A similar argument applies when $\overline{a_i} = a_{jk} \in c_j$, showing that $\overline{a_i} \in f^{-1}(1) \cap c_j$. It follows that f is a satisfiability function.

Conversely, suppose that f is a satisfiability function for the clauses $\mathcal{C}$. Then, as mentioned above, we can extend

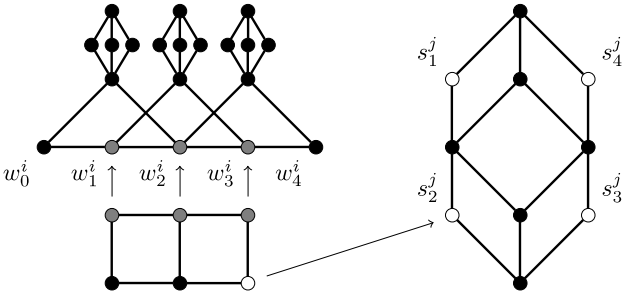


Fig. 1. Connecting the gadget corresponding to clause $c_j = \{a_{j1}, a_{j2}, a_{j3}, a_{j4}\}$ (right) with the gadget corresponding to the variable $a_i = a_{j2}$ (upper left)

$$\begin{aligned} M' = &\{w_1^i w_2^i, w_3^i w_4^i : f(a_i) = 0\} \\ &\cup \{w_0^i w_1^i, w_2^i w_3^i : f(a^i) = 1\} \\ &\cup \{y_k^j s_k^j : f \text{ maps the } k\text{th variable in clause } c_j \text{ to } 0\} \end{aligned}$$

to a C_k-transverse matching of H, using the fact that M' leaves at least one s_k^j uncovered for every j, and the clause gadget has a C_k-transverse matching leaving the other three uncovered.

Finally, H has bipartition $(X, V(H) \setminus X)$, where

$$\begin{aligned} X = &\{w_0^i, w_2^i, w_4^i, v_{k,1}^i, v_{k,2}^i, v_{k,3}^i : i = 1, \ldots, p; k = 1, 2, 3\} \\ &\cup \{x_k^j, s_k^j : j = 1, \ldots, q; k = 1, 2, 3, 4\}. \end{aligned}$$

The construction of H takes $O(p+q)$ time, so this reduction is polynomial. Therefore, the C_4-transverse matching problem for bipartite graphs is NP-complete.

□

4 TM(C_ℓ) is NP-Complete for $\ell \geq 5$

A broadly similar approach can be taken to show that TM(C_ℓ) is NP-complete for each $\ell \geq 5$. As the variable gadgets are slightly different depending on whether ℓ is odd or even, we treat these cases separately. However, both cases depend on the following useful structure.

A *u-v accordion path* consists of edge-disjoint u-v paths $ux_1w_1x_2 \ldots w_kx_{k+1}v$, $uy_1w_1y_2 \ldots w_{k-1}y_kv$, and $uy_1w_1y_2 \ldots w_ky_{k+1}v$, each having the same (even) length and sharing every second vertex. An example is given in Fig. 2. The *length* of an accordion path is the length (i.e. number of edges) of any of the paths comprising it.

The importance of accordion paths comes from the following observation: any matching in a u-v accordion path fails to cover some u-v path of the same length k. More importantly, if a graph contains such an accordion path and an additional u-v path P of length $\ell - k$, then any C_ℓ-transverse matching contains at least one of the edges of P. We will find this property very useful when constructing our gadgets.

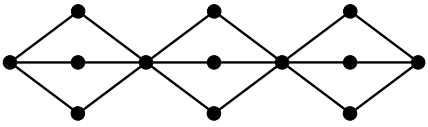


Fig. 2. An accordion path of length 6

Proposition 11. *TM(C_ℓ) for bipartite graphs is NP-complete when $\ell \geq 6$ is even.*

Proof. The reduction is from $\frac{\ell}{2}$-SAT, using a similar strategy as the one in the previous section. This time, the clause gadget corresponding to each c_j is simply a $C_\ell : s_1^j t_1^j \dots s_{\ell/2}^j t_{\ell/2}^j$. By definition, any C_ℓ-transverse matching of the resulting graph contains at least one edge of this cycle and hence covers at least one s_k^j. The variable gadget corresponding to a_i consists of a path $w_0^i w_1^i w_2^i w_3^i w_4^i$ and three disjoint accordion paths of length $\ell - 2$ from w_0^i to w_2^i, from w_1^i to w_3^i, and from w_2^i to w_4^i. The only minimal C_ℓ-transverse matchings of this gadget are $\{w_0^i w_1^i, w_2^i w_3^i\}$ and $\{w_1^i w_2^i, w_3^i w_4^i\}$.

We connect the variable gadgets to the clause gadgets as follows. If $c_j = \{a_{j1}, a_{j2}, a_{j3}, a_{j4}\}$ and $a_{jk} = a_i$, add three new vertices x_k^j, y_k^j, z_k^j, edges $x_k^j y_k^j$, $s_k^j y_k^j$, $w_0^i x_k^j$, $w_1^i z_k^j$, $w_3^i s_k^j$, and an accordion path of length $\ell - 4$ from z_k^j to y_k^j. Likewise, if $a_{jk} = \overline{a_i}$, do the same except adding edges $w_1^i s_k^j$ and $w_3^i x_k^j$ instead of $w_3^i s_k^j$ and $w_1^i x_k^j$. This process is illustrated in Fig. 3.

As in the previous section, we can interpret a C_4-transverse matching M of the resulting bipartite graph H as a truth assignment f by

$$f(a_i) = \begin{cases} 0 & w_1^i w_2^i, w_3^i w_4^i \in M \\ 1 & w_0^i w_1^i, w_2^i w_3^i \in M \end{cases}$$

and $f(\overline{a_i}) \neq f(a_i)$. A similar argument to that in the proof of Proposition 10 shows that f is a satisfiability function for $\mathcal{C}$. Likewise, a satisfiability function can be used to construct a C_ℓ-transverse matching for H. The graph H is easily seen to be bipartite. □

When ℓ is odd, we can use a nearly identical reduction and proof to show that TM(C_ℓ) is NP-complete. The graph produced by the reduction is not bipartite in this case, for any matching in a bipartite graph is C_ℓ transverse for odd ℓ.

Proposition 12. *TM(C_ℓ) is NP-complete when $\ell \geq 5$ is odd.*

Proof. We only describe the construction of H from an instance $\mathcal{C}$ of $\frac{(k+1)}{2}$-SAT. As in the proof of Proposition 11, the gadget corresponding to each clause c_j is a cycle $C_\ell : s_1^j t_1^j s_2^j t_2^j \dots s_{(k+1)/2}^j$. Each variable a_i corresponds to a path $w_0^i w_1^i w_2^i w_3^i w_4^i$ in such a way that any C_ℓ-transverse matching contains either $\{w_0^i w_1^i, w_2^i w_3^i\}$ or $\{w_1^i w_2^i, w_3^i w_4^i\}$; to ensure this property we add vertices u^i, v^i,

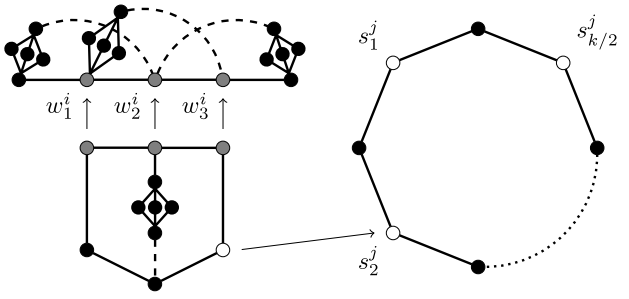


Fig. 3. Connecting the cycle corresponding to clause c_j to the variable gadget corresponding to a_i. Dashed lines indicate accordion paths.

edges $u^i w_2^i, u^i w_3^i, u^i w_4^i$ and $u^i v^i$, an accordion path of length $\ell - 1$ from u^i to v^i, and accordion paths of length $\ell - 3$ from w_0^i, w_1^i, and w_2^i to u^i. (See Fig. 4). As desired, the minimal C_ℓ-transverse matchings of this gadget are $\{w_0^i w_1^i, w_2^i w_3^i, u^i v^i\}$ and $\{w_1^i w_2^i, w_3^i w_4^i, u^i v^i\}$.

The clause gadgets are connected with the appropriate variable gadgets in nearly the same way as in Proposition 11. If $c_j = \{a_{j1}, a_{j2}, a_{j3}, a_{j4}\}$ and $a_{jk} = a_i$, add two new vertices x_k^j, y_k^j, edges $x_k^j y_k^j, s_k^j y_k^j, w_0^i x_k^j, w_3^i s_k^j$, and an accordion path of length $\ell - 3$ from w_1^i to y_k^j; if a_i is negated in c_j, replace the edges $w_3^i s_k^j$ and $w_1^i x_k^j$ with $w_1^i s_k^j$ and $w_3^i x_k^j$.

The same argument of Propositions 10 and 11 shows that the resulting graph H has a C_ℓ-transverse matching if and only if the original instance of $\frac{(\ell+1)}{2}$-SAT has a satisfiability function. When $\ell \geq 5$, $\frac{(\ell+1)}{2}$-SAT is NP-complete. Therefore, the C_ℓ-transverse matching problem is NP-complete. □

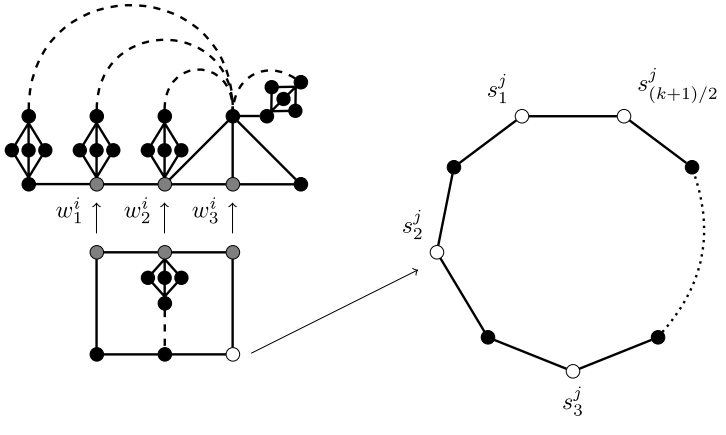


Fig. 4. Connecting the cycle corresponding to c_j to the variable gadget for a_i. Dashed lines in the left-hand gadgets indicate accordion paths.

5 Conclusion and Further Remarks

Theorem 1 follows from Corollary 6 and Propositions 10, 11, and 12. Theorem 2 follows from Theorem 1, Corollary 9, and the fact that $\mathrm{ST}(C_\ell)$ for line graphs is equivalent to $\mathrm{TM}(C_\ell)$ for each $\ell \geq 4$.

We have classified in this paper the complexity of the transverse matching problems for cycles of fixed lengths. Brandstädt [1] proved that determining whether a graph has a stable set which meets every cycle in the graph is an NP-complete problem even when it is restricted to bipartite graphs. We remark that an edge version of this problem is also NP-complete.

Consider the class $\mathcal{C}$ of subcubic planar graphs having exactly two two vertices of degree two. It is an NP-complete problem to determine whether a graph in $\mathcal{C}$ contains a Hamiltonian path joining the two vertices of degree two [6]. Observe that such a path exists if and only if there is a matching that meets every cycle in the graph. Hence determining whether a graph in $\mathcal{C}$ contains a matching that meets every cycle in the graph is an NP-complete problem.

Acknowledgements. The first author would like to thank NSERC for the support through a Canadian Graduate Scholarship award. The second author would also like to thank NSERC for the support and is grateful to the hospitality from Department of Mathematics, Zhejiang Normal University - part of research was carried out during his visit to the department. The research of the third author is supported under grant ZJNSF NO. Z6110786. Finally, all three authors would like to thank Dr. Frank Ruskey for originally bringing the transverse matching problems to their attention.

References

1. Brandstädt, A.: Partitions of graphs into one or two independent sets and cliques. Discrete Math. 152, 47–54 (1986)
2. Edmonds, J.: Paths, trees, and flowers. Canad. J. Math. 17, 449–467 (1965)
3. Farrugia, A.: Vertex-partitioning into fixed additive induced-hereditary properties is NP-hard. Electron. J. Combin. 11 (2004)
4. Foldes, S., Hammer, P.L.: Split graphs. In: Proc. 8th South-Eastern Conf. on Combinatorics, Graph Theory and Computing, pp. 311–315 (1977)
5. Garey, M.R., Johnson, D.S.: Computers and Intractability. W.H. Freeman and Company, San Francisco (1979)
6. Garey, M.R., Johnson, D.S., Tarjan, R.E.: The planar Hamiltonian circuit problem is NP-complete. SIAM J. Comput. 5, 704–714 (1976)
7. Hoáng, C.T., Le, V.C.: On P_4-transversals of perfect graphs. Discrete Math. 216, 195–210 (2000)
8. Micali, S., Vazirani, V.: An $O(\sqrt{|V|} \cdot |E|)$ algorithm for finding maximum matching in general graphs. In: Proc. 21st IEEE Symp. Foundations of Computer Science, pp. 17–27 (1980)
9. Stacho, J.: On P_4-transversals of chordal graphs. Discrete Math. 308, 5548–5554 (2008)

Efficient Conditional Expectation Algorithms for Constructing Hash Families

Charles J. Colbourn

Computing, Informatics, and Decision Systems Engineering
Arizona State University
PO Box 878809
Tempe, AZ 85287-8809, U.S.A.
Charles.Colbourn@asu.edu

Abstract. Greedy methods for solving set cover problems provide a guarantee on how close the solution is to optimal. Consequently they have been widely explored to solve set cover problems arising in the construction of various combinatorial arrays, such as covering arrays and hash families. In these applications, however, a naive set cover formulation lists a number of candidate sets that is exponential in the size of the array to be produced. Worse yet, even if candidate sets are not listed, finding the 'best' candidate set is NP-hard. In this paper, it is observed that one does not need a best candidate set to obtain the guarantee — an average candidate set will do. Finding an average candidate set can be accomplished using a technique employing the method of conditional expectations for a wide range of set cover problems arising in the construction of hash families. This yields a technique for constructing hash families, with a wide variety of properties, in time polynomial in the size of the array produced.

1 Introduction

Let X be a finite set of size n, and let $\mathcal{B}$ be a collection of subsets of X. A *set cover* for the set system $(X, \mathcal{B})$ is a collection $\mathcal{B}' \subseteq \mathcal{B}$ so that $\cup_{B \in \mathcal{B}'} B = X$. Finding the smallest set cover is NP-hard [15], and hence approximation and heuristic techniques have been developed. Stein [20], Lovász [16], and Johnson [14] (see also [5]) analyze a greedy algorithm and establish a useful upper bound on the sizes of set covers that it produces; we review the Stein-Lovász-Johnson theorem and its constructive proof in Section 2. In terms of the sizes of X and $\mathcal{B}$, the greedy method that they employ requires only polynomial run time.

The Stein-Lovász-Johnson method has been applied in establishing upper bounds on the sizes of numerous combinatorial arrays. In these contexts, however, the admissible sets $\mathcal{B}$ are often known implicitly rather than presented explicitly. Then the size of the input is simply $|X|$, and the run time of the greedy algorithm may be exponential in $|X|$, because $|\mathcal{B}|$ may be exponentially large with respect to $|X|$. This has limited the practical uses of such greedy methods for the actual construction of the set covers needed to produce the corresponding

C.S. Iliopoulos and W.F. Smyth (Eds.): IWOCA 2011, LNCS 7056, pp. 144–155, 2011.
© Springer-Verlag Berlin Heidelberg 2011

combinatorial arrays. Despite this, the Stein-Lovász-Johnson paradigm has been used to develop methods whose run time is polynomial in $|X|$. One method is to list only a small subset of the sets [7]. Another method employs an implicit representation of all of the sets; we outline two instances next.

A first example arises with covering arrays. Let N, k, t, and v be positive integers. Let C be an $N \times k$ array with entries from an alphabet Σ of size v; we typically take $\Sigma = \{0, \ldots, v-1\}$. When $(\nu_1, \ldots, \nu_t)$ is a t-tuple with $\nu_i \in \Sigma$ for $1 \le i \le t$, $(c_1, \ldots, c_t)$ is a tuple of t column indices ($c_i \in \{1, \ldots, k\}$), and $c_i \ne c_j$ whenever $\nu_i \ne \nu_j$, the t-tuple $\{(c_i, \nu_i) : 1 \le i \le t\}$ is a *t-way interaction*. The array *covers* the t-way interaction $\{(c_i, \nu_i) : 1 \le i \le t\}$ if, in at least one row ρ of C, the entry in row ρ and column c_i is ν_i for $1 \le i \le t$. Array C is a *covering array* $\mathsf{CA}(N; t, k, v)$ of *strength* t when every t-way interaction is covered. The goal here is to minimize N for given values of t, k, and v.

Existence of a $\mathsf{CA}(N; t, k, v)$ can be formulated as a set cover problem as follows. Let K be a set of k column indices, let Σ be a set of size v, and let $X = \binom{K}{t} \times \Sigma^t$. Then $|X| = \binom{k}{t} v^t$. Form a set $\mathcal{B}$ of v^k sets as follows. The set corresponding to the k-tuple $(x_1, \ldots, x_k) \in \Sigma^k$ contains element $\{\gamma_1, \ldots, \gamma_t\} \times (\nu_1, \ldots, \nu_t)$ (where $\gamma_i < \gamma_{i+1}$ for $1 \le i < t$) exactly when $x_{\gamma_i} = \nu_i$ for $1 \le i \le t$. Every element of X appears in exactly v^{k-t} sets in $\mathcal{B}$, and every set in $\mathcal{B}$ has size $\binom{k}{t}$. By the Stein-Lovász-Johnson theorem, there exists a $\mathsf{CA}(N; t, k, v)$ with $N \le \frac{v^k}{v^{k-t}}(1 + \ln \binom{k}{t}) = v^t(1 + \ln \binom{k}{t})$. A similar result was established in [6] without recourse to the Stein-Lovász-Johnson theorem. An explicit presentation of the set system involves listing $v^k \binom{k}{t}$ elements, but X contains 'only' $\binom{k}{t} v^t$ elements.

When v and t are fixed, the size of X is a polynomial in k. Still the number of sets in $\mathcal{B}$ remains exponential in k. In the specific case of covering arrays, Cohen *et al.* [6] exploit the fact that we need not list all sets in $\mathcal{B}$. Rather we can generate sets one at a time, as needed. After some sets have been generated (each corresponding to rows of a partial covering array), certain elements are covered by sets already selected and the remaining elements are uncovered. Following the Stein-Lovász-Johnson paradigm, the next set to be selected should be one that covers the largest number of previously uncovered elements. There may be no need to list all possible sets explicitly in order to select one that covers this maximum number.

Unfortunately, given a specific set of uncovered elements, it is not clear how to find a set that covers the maximum number efficiently (in time polynomial in the number of elements). Indeed, in the covering array situation, this problem is itself NP-hard [3], even when $t = 2$. For covering arrays, Bryce and Colbourn [3,4] nevertheless developed an efficient (time polynomial in k for fixed v and t) algorithm for generating covering arrays with a number of rows meeting that of the Stein-Lovász-Johnson theorem. To do this, they made two improvements. First they showed that the same bound is obtained when one chooses a set that covers the *average* number of uncovered elements at each stage, rather than the *maximum*. Then they developed a 'density' method (a form of the 'method of

conditional expectations', to be described in Section 5) to produce a set that covers at least the average number at each step, and to do so efficiently.

A second example also involves an implicit representation of the sets. A *hash family* $\mathsf{HF}(N; k, v)$, $\mathsf{A} = (a_{ij})$, is an $N \times k$ array; each cell contains one symbol from a set Σ of v symbols. A *perfect hash family* $\mathsf{PHF}(N; k, v, t)$ is an $\mathsf{HF}(N; k, \mathbf{v})$, in which in for every set C of at most t columns, there exists a row ρ for which $|\{a_{\rho c} : c \in C\}| = |C|$. These have been explored for numerous applications (see [12,21], for example). Although perfect hash families were first studied for applications in hashing [17], our reasons for interest in them are quite different (see [9] for the principal applications in which we are interested). Hence this paper does not discuss the use of such hash families for hashing, despite their name.

Take K to be a set of k column indices and $X = \binom{K}{t}$. Form a set $\mathcal{B}$ of v^k sets, one for each k-tuple in Σ^k. The set corresponding to the k-tuple $(x_1, \ldots, x_k) \in \Sigma^k$ contains element $\{\gamma_1, \ldots, \gamma_t\}$ exactly when $|\{x_{\gamma_i} : 1 \le i \le t\}| = t$. Then every element belongs to $v^{k-t} \cdot v \cdot (v-1) \cdot \cdots \cdot (v-t+1)$ sets. Every set covers at most $\binom{k}{t}$ elements, and hence by the Stein-Lovász-Johnson theorem we find that $N \le 1 + \frac{v^t}{v(v-1)\cdots(v-t+1)} \ln \binom{k}{t}$. Once again this yields an exponential time method. But again by showing that it suffices to find a set that covers an *average* number of uncovered elements and developing a method of conditional expectations to find such a set, Colbourn [8] developed an efficient (time polynomial in k for fixed t) algorithm for for the construction of perfect hash families with a number of rows meeting the given bound.

Unfortunately, in each case the analysis is specific to the problem at hand: Different proofs are employed both to show that it suffices to select a set for inclusion that covers at least the average number of uncovered elements, and to show that such a set can be found in time polynomial in $|X|$. Here we establish a substantial generalization of both methods.

In Section 2, we review the statement and proof of the Stein-Lovász-Johnson theorem. Then we demonstrate that it always suffices to choose a set that covers the average number of elements in order to achieve the bound in the Stein-Lovász-Johnson theorem. This observation, while quite easy, seems not to have been explicitly stated or used in the literature. In Section 3, we describe numerous variants of hash families. Then in Section 4 develop a greedy construction method for these many variants. In Section 5, we apply the method of conditional expectations to make the construction of hash families efficient.

2 The Stein-Lovász-Johnson Paradigm

A greedy strategy for set cover problems has been widely used, starting with work of Stein [20], Lovász [16], and Johnson [14]. We give (one version of) the algorithm in Figure 1.

The size of the set cover can be viewed either as the smallest value of i for which we encounter $Y_i = 0$, or it can be viewed as $\sum_{i=0}^{\alpha} \ell_i$, taking $\ell_i = |\mathcal{M}_i|$. We consider the latter expression. First, $n_{i-1} = n_i - i\ell_i$, and hence $\ell_i = (n_i - n_{i-1})/i$.

GREEDY_SET_COVER$(X, \mathcal{B})$: $(|X| = n;\ |\mathcal{B}| = c)$
Set $r(x) = |\{B : x \in B \in \mathcal{B}\}|$ for $x \in X$
Set $\alpha = \max\{|B| : B \in \mathcal{B}\}$ and $r = \min\{r(x) : x \in X\}$
Set $\mathcal{M}_j = \emptyset$ for $0 \le j \le \alpha$
Set $\mathcal{D}_0 = \mathcal{B}$ and $Y_0 = X$
Set $n_\alpha = |X|$
Set $\mathcal{L} = \emptyset$ and $i = 0$
while $Y_i \neq \emptyset$ **do**
Set $\gamma_i = |\mathcal{D}_i|$; $\rho_i = |Y_i|$; and $\alpha_i = \max\{|B| : B \in \mathcal{D}_i\}$
If $i > 0$ and $\alpha_i < \alpha_{i-1}$ set $n_{\alpha_i} = |Y_i|$
Choose a set $D_i \in \mathcal{D}_i$ for which $|D_i| \ge \alpha_i$
Set $\mathcal{L} = \mathcal{L} \cup \{D_i\}$
Set $\mathcal{M}_{\alpha_i} = \mathcal{M}_{\alpha_i} \cup \{D_i\}$
Set $Y_{i+1} = Y_i \setminus D_i$
Set $\mathcal{D}_{i+1} = \{B \setminus D_i : B \in \mathcal{D}_i, B \not\subseteq D_i\}$
Set $i = i + 1$
Set $n_0 = 0$
return $\mathcal{L}$

Fig. 1. The Greedy Algorithm

Because in each set system $(X_i, \mathcal{B}_i)$, every element appears in at least r sets and every set has size at most i, $rn_i \le ic$. Moreover,

$$\begin{aligned}\ell &= \textstyle\sum_{i=1}^{\alpha} \ell_i = \sum_{i=1}^{a} \frac{n_i - n_{i-1}}{i} \\ &= \tfrac{n_\alpha}{\alpha} + \tfrac{n_{\alpha-1}}{\alpha(\alpha-1)} + \tfrac{n_{\alpha-2}}{(\alpha-1)(\alpha-2)} + \cdots + \tfrac{n_1}{2\cdot 1} - n_0\end{aligned}$$

Combining these, we obtain $\ell \le \frac{n}{\alpha} + \frac{c}{r}\left(\sum_{i=2}^{\alpha} \frac{1}{i}\right)$, which yields the bound in the theorem of Stein [20], Lovász [16], and Johnson [14]:

Theorem 1. [Stein-Lovász-Johnson] *Let $(X, \mathcal{B})$ be an set system with $|X| = n$ and $|\mathcal{B}| = c$ so that $|B| \le \alpha$ for every $B \in \mathcal{B}$ and $|\{B : x \in B \in \mathcal{B}\}| \ge r$ for every $x \in X$. Then there is a collection $\mathcal{B}' \subseteq \mathcal{B}$ forming a set cover with ℓ sets for some $\ell \le \frac{n}{\alpha} + \frac{c}{r} \ln \alpha \le \frac{c}{r}(1 + \ln \alpha)$.*

A second analysis of GREEDY_SET_COVER uses the fact that it terminates when $Y_i = \emptyset$, or equivalently when $\rho_i < 1$. An element x_j of Y_i is not a member of $\cup_{j=0}^{i-1} D_j$, and hence $r(x_j)$ is unchanged by the deletion of the elements already covered. Because $r(x_j) \ge r$ for all $x_j \in Y_i$, and $\gamma_i \le c$, the size of D_i is at least $\frac{r\rho_i}{c}$. Then $\rho_{i+1} \le \rho_i - \frac{r\rho_i}{c} = \rho_i \frac{c-r}{c}$. Because this holds for every $i > 0$, we have that $\rho_i \le n\left(\frac{c-r}{c}\right)^i$. We determine the smallest value of i for which $n\left(\frac{c-r}{c}\right)^i < 1$. Equivalently, $n < \left(\frac{c}{c-r}\right)^i$. Taking logarithms base $c/(c-r)$ of both sides, we have that $\log_{c/(c-r)} n < i$.

This establishes:

Theorem 2. *Let $(X, \mathcal{B})$ be an set system with $|X| = n$ and $|\mathcal{B}| = c$ so that $r(x) \ge r$ for every $x \in X$. Then there is a collection $\mathcal{B}' \subseteq \mathcal{B}$ forming a set cover with ℓ sets for some $\ell \le 1 + \frac{\ln n}{\ln c/(c-r)}$.*

This improves the constant in the bound over that of the Stein-Lovász-Johnson theorem when $\ln \frac{c}{c-r} \ln \alpha \geq \frac{c-r}{c} \ln n$, but yields a weaker bound in other cases. This apparent discrepancy is an artifact of the analyses, not the algorithms.

When the set system $(X, \mathcal{B})$ is provided as an explicit listing of the elements and sets, the running time of either method is a polynomial in the size of the input. Careful examination shows that the only operations that consider all of the sets in $\mathcal{D}_i$ are the ones to select D_i, and to remove all elements of D_i to form $\mathcal{D}_{i+1}$. To obtain an algorithm whose running time is polynomial in $|X|$, we cannot hope to examine (or even list) all sets in $\mathcal{B}$. Our first task, then, is to simplify the selection of the set D_i. In fact, we show that selecting a set of *average* size yields the same results. At the same time, we equip $\mathcal{B}$ with a probability distribution, so that $\Pr[B]$ is the probability that set $B \in \mathcal{B}$ is selected.

AVERAGE_SET_COVER$(X, \mathcal{B})$: $(|X| = n;\ |\mathcal{B}| = c)$
  Set $r(x) = c \sum_{\{B \in \mathcal{B}: x \in B\}} \Pr[B]$ for $x \in X$
  Set $r = \min\{r(x) : x \in X\}$ and $\alpha = \left\lceil \frac{\sum_{x \in X} r(x)}{c} \right\rceil$
  Set $\mathcal{M}_j = \emptyset$ for $0 \leq j \leq \alpha$
  Set $\mathcal{D}_0 = \mathcal{B}$ and $Y_0 = X$
  Set $n_\alpha = |X|$
  Set $\mathcal{L} = \emptyset$ and $i = 0$
  while $Y_i \neq \emptyset$ **do**
    Set $\gamma_i = |\mathcal{D}_i|$; $\rho_i = |Y_i|$; and $\alpha_i = \left\lceil \frac{\sum_{x \in Y_i} r(x)}{c} \right\rceil$
    If $i > 0$ and $\alpha_i < \alpha_{i-1}$ set $n_{\alpha_i} = |Y_i|$
    Choose a set $D_i \in \mathcal{D}_i$ for which $|D_i| \geq \alpha_i$ and $\Pr[D_i] > 0$
    Set $\mathcal{L} = \mathcal{L} \cup \{D_i\}$
    Set $\mathcal{M}_{\alpha_i} = \mathcal{M}_{\alpha_i} \cup \{D_i\}$
    Set $Y_{i+1} = Y_i \setminus D_i$
    Set $\mathcal{D}_{i+1} = \{B \setminus D_i : B \in \mathcal{D}_i, B \not\subseteq D_i\}$
    Set $i = i + 1$
  Set $n_0 = 0$
  return $\mathcal{L}$

Fig. 2. The Average Algorithm

AVERAGE_SET_COVER, shown in Figure 2, is essentially the same method – with one important difference. Each set selected is only required to cover the average number of as yet uncovered elements of X rather than the maximum. Moreover, this average is weighted by the initial probability distribution selected on $\mathcal{B}$.

The analyses of GREEDY_SET_COVER carry through for the average method as well. For the first, we employed the fact that $n_{i-1} = n_i - i\ell_i$, and hence $\ell_i = (n_i - n_{i-1})/i$; and the fact that $rn_i \leq ic$. For the average method, $n_{i-1} \leq n_i - i\ell_i$, and hence $\ell_i \leq (n_i - n_{i-1})/i$; and $rn_i \leq ic$ because $i = \lceil \frac{rn_i}{c} \rceil$. For the second, we employed only the fact that $|D_i| \geq \frac{r\rho_i}{c}$, which holds for the average method as well.

Hence we have shown that

Theorem 3. *Let $(X, \mathcal{B})$ be a set system with $|X| = n$ and $|\mathcal{B}| = c$, for which $Pr[B]$ is the probability that $B \in \mathcal{B}$ is selected. Let $r(x) = c\sum_{\{B \in \mathcal{B}: x \in B\}} Pr[B]$ and $r = \min\{r(x) : x \in X\}$. Let $\beta = \left\lceil \frac{\sum_{x \in X} r(x)}{c} \right\rceil$. Then* AVERAGE_SET_COVER *produces a set cover $\mathcal{B}' \subseteq \mathcal{B}$ with ℓ sets, with $\ell \leq \min\left(\frac{c}{r}(1 + \ln \beta), 1 + \frac{\ln n}{\ln c/(c-r)}\right)$.*

When the probability distribution is uniform (and in many other cases), $\beta \leq \alpha$, and hence Theorem 3 improves on Theorem 1. This may come as a surprise, because the original Stein-Lovász-Johnson method selects a largest set while the average method may select a smaller one. As we have noted, the discrepancy arises from the algorithm analysis, not from the algorithm itself. In practice, it is quite possible that selecting the maximum coverage yields a better result in the end that does selecting the average coverage. Nevertheless, Theorem 3 shows that the *conclusion* of Theorem 1 can be obtained by selecting the average.

Whether the sets are listed explicitly or not, one potential benefit of selecting a set with average rather than maximum coverage is that the average can be often be easily computed or bounded, and then finding any set with at least that average coverage suffices. We explore this next, for a broad class of combinatorial construction problems.

3 Variants of Hash Families

Let $\mathbf{v} = (v_1, \ldots, v_N)$ be a tuple of positive integers. A *hash family* $\mathsf{HF}(N; k, \mathbf{v})$, $\mathsf{A} = (a_{ij})$, is an $N \times k$ array; each cell contains one symbol, and for $1 \leq \rho \leq N$, $|\{a_{\rho j} : 1 \leq j \leq k\}| \leq v_\rho$. When $v_1 = \cdots = v_N = v$ (i.e., the array is *homogeneous*), the result can be treated as an $N \times k$ array on v symbols; we employ the notation $\mathsf{HF}(N; k, v)$. We permit *heterogeneity* here (i.e. that v_i and v_j are not necessarily equal when $i \neq j$), for two reasons. First, heterogeneity is useful in certain applications of hash families in so-called column replacement techniques [9,10,11]. Secondly, and more importantly, when we build a hash family one row at a time by the Stein-Lovász-Johnson strategy, it turns out to be an easy matter to change the number of permitted symbols in each row (by using a different choice of Σ for each row). Naturally this renders the determination of the number of rows needed much more complicated, but it in no way complicates the algorithm for the construction of the hash family.

Many variants of perfect hash families have been studied before. We mention a few, in order to indicate the types of requirements placed on the hash family. An $\mathsf{HF}(N; k, \mathbf{v})$, $\mathsf{A} = (a_{ij})$, is

***perfect* of *strength* t:** denoted $\mathsf{PHF}(N; k, \mathbf{v}, t)$, when for every set C of at most t columns, there exists a row ρ for which $|\{a_{\rho c} : c \in C\}| = |C|$ (see [1,21], for example);

$(w_1, w_2, \ldots, w_s)$-*separating*: denoted $\mathsf{SHF}(N; k, \mathbf{v}, (w_1, w_2, \ldots, w_s))$, if whenever C is a set of $\sum_{i=1}^{s} w_i$ columns and $C_1, C_2, \ldots, C_s$ is a partition of C with $|C_i| = w_i$ for $1 \leq i \leq s$, there exists a row ρ for which $a_{\rho x} \neq a_{\rho y}$ whenever $x \in C_i$, $y \in C_j$ and $i \neq j$ (see [2,22], for example).

$\mathcal{W}$-***separating***: denoted $\mathsf{SHF}(N; k, \mathbf{v}, \mathcal{W})$ for $\mathcal{W}$ a set of tuples of nonnegative integers of the form $(w_1, w_2, \ldots, w_s)$, if whenever $(w_1, w_2, \ldots, w_s) \in \mathcal{W}$, C is a set of $\sum_{i=1}^{s} w_i$ columns, and $C_1, C_2, \ldots, C_s$ is a partition of C with $|C_i| = w_i$ for $1 \leq i \leq s$, there exists a row ρ for which $a_{\rho x} \neq a_{\rho y}$ whenever $x \in C_i$, $y \in C_j$ and $i \neq j$ (see [22], for example).

(t, s)-***distributing***: denoted $\mathsf{DHF}(N; k, \mathbf{v}, t, s)$, if it is $\mathcal{W}$-separating with $\mathcal{W}$ containing every tuple $(w_1, \ldots, w_s)$ of nonnegative integers with $\sum_{i=1}^{s} w_i = t$.

The definition of $\mathcal{W}$-separating encompasses the remaining three, so we can treat this general situation. On occasion, we wish to further restrict the choice of rows that can be employed to provide the desired separation. There are at least two natural ways to do this.

Let $\mathbf{d} = (d_1, \ldots, d_N)$ and $\mathbf{m} = (m_1, \ldots, m_N)$ be tuples of positive integers. An $\mathsf{HF}(N; k, \mathbf{v})$, $\mathsf{A} = (a_{ij})$, is

$\mathcal{W}$-***separating* and d-*scattering***: if whenever $(w_1, w_2, \ldots, w_s) \in \mathcal{W}$, C is a set of $\sum_{i=1}^{s} w_i$ columns, and $C_1, C_2, \ldots, C_s$ is a partition of C with $|C_i| = w_i$ for $1 \leq i \leq s$, there exists a row ρ for which $a_{\rho x} \neq a_{\rho y}$ whenever $x \in C_i$, $y \in C_j$ and $i \neq j$ *and* the multiset $\{a_{\rho x} : x \in C\}$ contains no symbol more than d_ρ times.

$\mathcal{W}$-***separating* and m-*strengthening***: if whenever $(w_1, w_2, \ldots, w_s) \in \mathcal{W}$, C is a set of $\sum_{i=1}^{s} w_i$ columns, and $C_1, C_2, \ldots, C_s$ is a partition of C with $|C_i| = w_i$ for $1 \leq i \leq s$, there exists a row ρ for which $a_{\rho x} \neq a_{\rho y}$ whenever $x \in C_i$, $y \in C_j$ and $i \neq j$ *and* the multiset $\{a_{\rho x} : x \in C\}$ contains no more than m_ρ different symbols.

A justification of the need for scattering or strengthening hash families is beyond the scope of this paper. Suffice it to say that O'Brien [19] proposed the scattering requirement and that strengthening generalizes the notion of (t, s)-partitioning hash families (see [9]). There is evidently a wide variety of possible conditions that might be imposed on the hash family to be constructed, and we have surely not exhausted them here. To treat these and other variants, we proceed as follows. Let $C = \{\gamma_1, \ldots, \gamma_t\}$ be a set of columns. In constructing the ρth row, an alphabet Σ_ρ of size v_ρ is available. A *requirement* is a partition of C into sets $C_1, \ldots, C_s$. An *assignment* $A \in \Sigma_\rho^t$ for C is a determination of a value for each column in C. A *constraint* for requirement R is a logical predicate $P_\rho(A) : \Sigma_\rho^t \mapsto \{\mathbf{true}, \mathbf{false}\}$. In essence, a constraint specifies which selections of symbols on the columns of C meet the requirement. A constraint could specify, for example, that a certain $(w_1, \ldots, w_s)$-separation is accomplished, that no more than m_i symbols are used on these t columns, or that no symbol occurs more than d_i times. Indeed it could involve any combination of these, and a variety of other properties. The requirements in constructing a hash family are fixed at the outset, but the constraints on meeting each requirement may vary depending on which row is being selected. Hence we employ constraints $P_\rho(A)$ for every assignment A to every requirement R, for $1 \leq \rho \leq N$, in forming the hash family.

From Theorem 3, one can immediately deduce bounds on the sizes of hash families in a general setting. We suppose that the candidate rows are selected

uniformly at random, and that the hash family needed is homogeneous with v symbols. Take $c = v^k$ and $r = \mu v^k$ in Theorem 3 to establish:

Theorem 4. *An* $\mathsf{HF}(N; k, v)$ *satisfying* q *requirements exists whenever*

$$N \geq \min\left(\frac{1}{\mu}(1 + \ln \beta), 1 + \frac{\ln q}{\ln 1/(1-\mu)}\right),$$

taking δ_R *to be the ratio of the number of assignments* A *to* R *that satisfy the constraint for* R *to the total number of assignments to* R*,* μ *to be the minimum of* δ_R *over all requirements* R*, and* β *to be the integer ceiling of the sum of* δ_R *over all requirements* R*.*

In Theorem 4, it may be puzzling that the bound does not appear to involve k. However, when requirements are placed on all of the k columns, the number q of requirements must be a function of k, and β is a function of q. The quantities in Theorem 4 can often easily be calculated; we give one example. Suppose that $v_i = v = 6$, $d_i = 3$, and $m_i = 3$ for $1 \leq i \leq N$. Suppose that $\mathcal{W} = \{(1,4),(2,3)\}$, and our objective is to produce a $\mathcal{W}$-separating, **d**-scattering, and **m**-strengthening $\mathsf{HF}(N; k, 6)$. Write $K = k(k-1)(k-2)(k-3)(k-4)$. There are $\binom{k}{1}\binom{k-1}{4} = \frac{1}{24}K$ requirements for the (1,4) separation, and $\binom{k}{2}\binom{k-2}{3} = \frac{1}{12}K$ for the (2,3) separation. Each has $6^5 = 7776$ assignments. A (1,4) separation R has 840 separations that meet the constraint, so $\delta_R = \frac{35}{324}$. A (2,3) separation R has 510 separations that meet the constraint, so $\delta_R = \frac{85}{1296}$. Then $\mu = \frac{85}{1296}$, and $\beta = \frac{155}{15552}K$. Then the hash family exists provided that $N \geq \min\left(\frac{1296}{85}(1 + \ln \frac{155}{15552}), 1 + \frac{\ln \frac{K}{8}}{\ln \frac{1296}{1211}}\right)$.

4 Constructing a Hash Family

Following the paradigm of Average_Set_Cover, we proceed as follows. The set X is the set of all requirements, and N is (an upper bound on) the number of rows permitted. Then Average_Hash_Family$(X, N, \{P_\rho(A, R)\})$, given in Figure 3, produces the desired hash family, or may fail if N is too small.

Average_Hash_Family requires that we repeatedly select a next row for inclusion. For a requirement $R \in X_\rho$, which is a set $C = \{\gamma_1, \ldots, \gamma_t\}$ of columns and a partition of C into sets $C_1, \ldots, C_s$, and a candidate row $\mathbf{x} = (x_1, \ldots, x_k) \in \Sigma_\rho^k$, R is covered by the row exactly when $P_\rho(A_{\mathbf{x},R})$ holds for the assignment $A_{\mathbf{x},R}$ in which column γ_i contains symbol x_{γ_i} for $1 \leq i \leq t$. Select_Average_Row must find a row $\mathbf{x} \in \Sigma_\rho^k$ for which $\Pr[\mathbf{x}] > 0$ and $|R \in X_\rho : P_\rho(A_{\mathbf{x},R})\}|$ is at least the average over all choices of row $\mathbf{y}$.

Suppose that we simply selected the row $\mathbf{x}$ at random (according to the probability distribution) from Σ_ρ^k. Then $\Pr[\mathbf{x}] > 0$, and the expectation of $|R \in X_\rho : P_\rho(A_{\mathbf{x},R})\}|$ is precisely the desired average,

$$\sum_{\mathbf{x} \in \Sigma_\rho^k} Pr[\mathbf{x}] \cdot \left(\sum_{R \in X_\rho} P_\rho(A_{\mathbf{x},R})\right) = \sum_{R \in X_\rho} \left(\sum_{\mathbf{x} \in \Sigma_\rho^k} Pr[\mathbf{x}] \cdot P_\rho(A_{\mathbf{x},R})\right),$$

treating $P_\rho(A_{\mathbf{x},R})$ as a 0,1-indicator variable. This yields a randomized algorithm for producing hash families, but in some cases we can do better.

AVERAGE_HASH_FAMILY$(X, N, \{P_\rho(A, R)\})$
// $\{P_\rho(A, R)\}$ provides a predicate for each row ρ, each $R \in X$,
// and each assignment A to R
Set $X_1 = X$ and $\mathcal{L} = \emptyset$
for ρ from 1 to N do
$\mathbf{y} =$ SELECT_AVERAGE_ROW$(\rho, X_\rho, \{P_\rho(A, R)\})$
Set $\mathcal{L} = \mathcal{L} \cup \{\mathbf{y}\}$
Set $X_{\rho+1} = X_\rho \setminus \{R \in X_\rho : P_\rho(A_{\mathbf{y},R})\}$
if $X_{N+1} = \emptyset$ **return** $\mathcal{L}$ **else return** fail

Fig. 3. The Average Algorithm for Hash Families

5 The Method of Conditional Expectations

It suffices to calculate the expectation of $P_\rho(A_{\mathbf{x},R})$ for each $R \in X_\rho$ in order to determine the average sought. Nevertheless, we must also find a row $\mathbf{x} \in \Sigma_\rho^k$ that yields at least this average. To do this, we start with a row in which no entries have been chosen, and repeatedly choose one coordinate whose entry is unspecified in which to choose an entry. Our objective is to ensure that at each stage the expectation of finding a row that covers at least the average does not decrease. In other words, we want the conditional expectation, based on the selection of the entries already made, never to decrease. Hence we employ the fundamental idea in the *method of conditional expectations* [13,18].

We must deal with rows in which only some of the entries have been chosen. Suppose that $(x_1, \ldots, x_k) \in (\Sigma_\rho \cup \{\star\})^k$. We interpret an entry in Σ_ρ to mean that the entry has been chosen, while the entry $\star$ means that the entry has not yet been chosen. A row $(y_1, \ldots, y_k) \in \Sigma_\rho^k$ is a *completion* of $(x_1, \ldots, x_k)$ if $x_i = y_i$ or $x_i = \star$ for $1 \le i \le k$. A row with s $\star$ entries has v_ρ^s completions, and these are denoted by $\mathcal{B}_{\mathbf{x}}$. For two rows $\mathbf{x}$ and $\mathbf{y}$, the probability that $\mathbf{y}$ occurs given that $\mathbf{x}$ occurs, $\Pr[\mathbf{y}|\mathbf{x}]$, is 0 whenever $y_i \neq x_i$ but $x_i \in \Sigma_\rho$; otherwise it is $(\sum_{\mathbf{z} \in \mathcal{B}_{\mathbf{y}}} \Pr[\mathbf{z}])/(\sum_{\mathbf{z} \in \mathcal{B}_{\mathbf{x}}} \Pr[\mathbf{z}])$. The *expected coverage* $ec(\mathbf{x})$ for a row $\mathbf{x} = (x_1, \ldots, x_k) \in (\Sigma_\rho \cup \{\star\})^k$ is $\sum_{\mathbf{z} \in \mathcal{B}_{\mathbf{x}}} \Pr[\mathbf{z}|\mathbf{x}] \cdot (\sum_{R \in X_\rho} P_\rho(A_{\mathbf{z},R}))$.

A row $(y_1, \ldots, y_k) \in (\Sigma_\rho \cup \{\star\})^k$ is a *j-successor* of $(x_1, \ldots, x_k)$ if, for some $1 \le j \le k$, it holds that $x_j = \star$ and $y_j \in \Sigma_\rho$, and that $x_i = y_i$ for $1 \le i \le k$ when $i \neq j$. A row having a $\star$ entry in the jth position has exactly v_ρ j-successors.

Letting $\chi(R, \mathbf{x})$ be the probability that a completion $\mathbf{z}$ of $\mathbf{x}$ satisfies the predicate $P_\rho(A_{\mathbf{z},R})$, we have $ec(\mathbf{x}) = \sum_{R \in X_\rho} \chi(R, \mathbf{x})$. The selection of a row is accomplished by SELECT_AVERAGE_ROW in Figure 4, when furnished with a routine EXPECTED_COMPLETIONS that calculates $\chi(R, \mathbf{x})$.

When $\mathbf{r}^{(i-1)}$ has $y_\gamma = \star$, and $\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(v_\rho)}$ are its γ-successors, $ec(\mathbf{r}^{(i-1)}) = \sum_{i=1}^{v_\rho} \Pr[\mathbf{y}^{(i)}|\mathbf{x}] \cdot ec(\mathbf{y}^{(i)})$. Hence $ec(\mathbf{r}^{(i-1)}) \le ec(\mathbf{r}^{(i)})$ for $1 \le i \le k$. Now $ec(\mathbf{r}^{(0)})$ is the expected number of elements of X covered by a row selected at random from Σ_ρ^k according to the probability distribution. Moreover, $ec(\mathbf{r}^{(k)})$ is the actual number of elements of X covered by the row $\mathbf{r}^{(k)}$, which therefore covers at least the expected number.

```
SELECT_AVERAGE_ROW(ρ, X_ρ, {P_ρ(A, R)})
    // {P_ρ(A, R)} provides a predicate for each R ∈ X and assignment A to R
    Set r^(0) = {⋆}^k
    for i from 1 to k do
        Choose an coordinate γ for which r_γ^(i−1) = ⋆
        Set maxcov = 0 and choice = ∅
        for σ ∈ Σ_ρ
            Let z be the γ-successor of r^(i−1) with z_γ = σ
            if Pr[z|r^(i−1)] > 0
                Set cov = 0
                for R ∈ X_ρ
                    cov = cov + EXPECTED_COMPLETIONS(ρ, R, z)
                if cov ≥ maxcov {maxcov = cov; choice = z}
        Set r^(i) = choice
    return r^(k)
```

Fig. 4. The Average Algorithm for Hash Families: Selecting a Row

It remains to compute $\chi(R, \mathbf{x})$ by EXPECTED_COMPLETIONS for each requirement R and an arbitrary $\mathbf{x} \in (\Sigma_\rho \cup \{\star\})^k$. While this can be carried out for some different probability distributions, in Figure 5 we treat only the case when the probability distribution is uniform.

```
EXPECTED_COMPLETIONS(ρ, R, x)
    // for the uniform distribution
    // R is the set C = {γ_1, ..., γ_t} of columns and the partition C_1, ..., C_s
    Let F = {γ ∈ C : x_γ = ⋆} and F̄ = C \ F
    Set count = 0
    for each assignment A = {a_γi : 1 ≤ i ≤ t} with a_γi = x_γi for γ_i ∈ F̄
            and a_γi ∈ Σ_ρ for γ_i ∈ F
        if P_ρ(A, R) then count = count + 1
    return count · (v_ρ)^−|F|
```

Fig. 5. The Average Algorithm for Hash Families: Expected Completions

EXPECTED_COMPLETIONS relies on the fact whether or not a completion $\mathbf{z}$ of $\mathbf{x}$ satisfies predicate $P_\rho(A_{\mathbf{z},R})$ depends only on the assignment to the coordinates C specified by R. Because every completion is equally probable, once the assignment to coordinates of C is specified, either every completion satisfies the predicate or none does. Therefore we can just treat each assignment to the coordinates of C.

The routines in Figures 3, 4, and 5 implement the method of Figure 2 for a wide variety of hash families, producing a hash family of size no larger than that produced by applying the average algorithm directly. The improvement is that, by using a method of conditional expectations, the algorithm has running time polynomial in the number of requirements rather than the number of sets, when the (maximum) number of symbols v and the strength t are fixed.

To see this, EXPECTED_COMPLETIONS takes time $O(v^t) = O(1)$. Then when there are r requirements, SELECT_AVERAGE_ROW takes time $O(k \cdot v \cdot r)$, but r is bounded by $(kv)^t$, so the time is $O(k^{t+1})$. When N rows are produced, AVERAGE_HASH_FAMILY takes time $O(N \cdot k^{t+1})$. By Theorem 4, N is $O(\log k)$ when v and t are fixed, and hence the running time is indeed a polynomial in k. (Surprisingly, the large constants suppressed in this analysis do not render the method impractical, as evidenced by [4,8]. But that is a story for another day.)

6 Conclusion

Being less greedy in solving set cover problems does not negate the guarantee on the size of the set cover obtained. Indeed, for general set cover problems, at each stage selecting a set that covers at least the average number of uncovered elements suffices. When all sets are listed explicitly, and all can be examined, finding a set with average coverage is not substantially easier than finding one with maximum coverage. However, if candidate sets are generated from a known probability distribution, finding – with high probability – a set with average coverage is an easy task. Focussing on set cover problems arising in the construction of hash families, we have shown that when the probability distribution is uniform, finding a set with average coverage admits an algorithm whose running time is polynomial in the size of the set cover produced.

The method developed provides not only useful bounds on the sizes of hash families, but also an efficient algorithm for their construction. We expect that this method will prove to be a practical one, in view of the similar but simpler method previously developed for perfect hash families [8].

Acknowledgments. Thanks to Daniel Horsley, Chris McLean, Peyman Nayeri, Devon O'Brien, and Violet Syrotiuk for helpful discussions. Thanks particularly to Daniel for suggesting the use of general probability distributions.

References

1. Alon, N.: Explicit construction of exponential sized families of k-independent sets. Discrete Math. 58, 191–193 (1986)
2. Blackburn, S.R., Etzion, T., Stinson, D.R., Zaverucha, G.M.: A bound on the size of separating hash families. J. Combin. Theory Ser. A 115(7), 1246–1256 (2008)
3. Bryce, R.C., Colbourn, C.J.: The density algorithm for pairwise interaction testing. Software Testing, Verification, and Reliability 17, 159–182 (2007)
4. Bryce, R.C., Colbourn, C.J.: A density-based greedy algorithm for higher strength covering arrays. Software Testing, Verification, and Reliability 19, 37–53 (2009)
5. Chvátal, V.: A greedy heuristic for the set-covering problem. Math. Oper. Res. 4(3), 233–235 (1979)
6. Cohen, D.M., Dalal, S.R., Fredman, M.L., Patton, G.C.: The AETG system: An approach to testing based on combinatorial design. IEEE Transactions on Software Engineering 23, 437–444 (1997)
7. Cohen, G., Litsyn, S., Zémor, G.: On greedy algorithms in coding theory. IEEE Trans. Inform. Theory 42, 2053–2057 (1996)

8. Colbourn, C.J.: Constructing perfect hash families using a greedy algorithm. In: Li, Y., Zhang, S., Ling, S., Wang, H., Xing, C., Niederreiter, H. (eds.) Coding and Cryptology. World Scientific, Singapore (2008)
9. Colbourn, C.J.: Covering arrays and hash families. In: Information Security and Related Combinatorics. NATO Peace and Information Security, pp. 99–136. IOS Press (2011)
10. Colbourn, C.J., Torres-Jiménez, J.: Heterogeneous hash families and covering arrays. Contemporary Mathematics 523, 3–15 (2010)
11. Colbourn, C.J., Zhou, J.: Improving two recursive constructions for covering arrays. Journal of Statistical Theory and Practice (2011)
12. Czech, Z.J., Havas, G., Majewski, B.S.: Perfect hashing. Theoret. Comput. Sci. 182, 1–143 (1997)
13. Erdős, P., Selfridge, J.L.: On a combinatorial game. J. Combinatorial Theory Ser. A 14, 298–301 (1973)
14. Johnson, D.S.: Approximation algorithms for combinatorial problems. J. Comput. System Sci. 9, 256–278 (1974)
15. Karp, R.M.: Reducibility among combinatorial problems. In: Complexity of Computer Computations, pp. 85–103. Plenum, New York (1972)
16. Lovász, L.: On the ratio of optimal integral and fractional covers. Discrete Math. 13(4), 383–390 (1975)
17. Mehlhorn, K.: Data Structures and Algorithms 1: Sorting and Searching. Springer, Berlin (1984)
18. Motwani, R., Raghavan, P.: Randomized algorithms. Cambridge University Press, Cambridge (1995)
19. O'Brien, D.J.: Exploring hash families and their applications to broadcast encryption. Master's thesis, Arizona State University (2011)
20. Stein, S.K.: Two combinatorial covering theorems. J. Combinatorial Theory Ser. A 16, 391–397 (1974)
21. Stinson, D.R., Tran Van Trung, Wei, R.: Secure frameproof codes, key distribution patterns, group testing algorithms and related structures. J. Statist. Plann. Infer. 86, 595–617 (2000)
22. Stinson, D.R., Wei, R., Chen, K.: On generalized separating hash families. J. Combinat. Theory (A) 115, 105–120 (2008)

2-Layer Right Angle Crossing Drawings*

Emilio Di Giacomo[1], Walter Didimo[1], Peter Eades[2], and Giuseppe Liotta[1]

[1] Università di Perugia, Italy
{digiacomo,didimo,liotta}@diei.unipg.it
[2] University of Sydney, Australia
peter@cs.usyd.edu.au

Abstract. A 2-layer drawing represents a bipartite graph so that the vertices of each partition set are points of a distinct horizontal line (called a *layer*) and the edges are straight-line segments. In this paper we study 2-layer drawings where all edge crossings form right angles. We characterize which graphs admit this type of drawing, provide linear-time testing and embedding algorithms, and present a polynomial-time crossing minimization technique. Also, for a given graph G and a constant k, we prove that it is $\mathcal{NP}$-complete to decide whether G contains a subgraph of at least k edges having a 2-layer drawing with right angle crossings.

1 Introduction

The study of drawings of graphs where any two crossing edges form crossing angles that are not too small is among the emerging topics in Graph Drawing. This interest is motivated by recent experiments of Huang *et al.* [15,16], who show that crossing angles guarantee good readability properties only if they are "large enough" (approximatively larger than $\frac{\pi}{3}$). These experiments therefore imply that non-planar drawings of graphs should not only be optimized in terms of classical parameters such as the number of edge crossings and the number of bends along the edges, but also in terms of the minimum angle formed by any two crossing edges.

We study straight-line *Right Angle Crossing drawings* (or *RAC drawings* for short). In a RAC drawing any two crossing edges form $\frac{\pi}{2}$ crossing angles. RAC drawings have been first introduced in [4], where it is proved that straight-line RAC drawings with n vertices have at most $4n-10$ edges, which is a tight bound. Straight-line RAC drawings are also studied by Dujmović *et al.* [7], who give an alternative proof of the $4n-10$ bound. The relationship between straight-line RAC drawings with $4n-10$ edges and 1-planar graphs is studied in [10]. Angelini et al. [2] investigate straight-line upward RAC drawings of digraphs. Van Kreveld [21] studies how much better a straight-line RAC drawing of a planar graph can be than any straight-line planar drawing of the same graph. Complete bipartite straight-line RAC drawable graphs are studied in [5].

Despite the growing literature about straight-line RAC drawings, no algorithms for computing such drawings have been described so far. Existing papers either establish combinatorial properties of RAC drawings (they typically address Túran-type questions) or compute RAC drawings with bends along the edges. Also, deciding whether a

* Work supported in part by MIUR of Italy under project AlgoDEEP prot. 2008TFBWL4.

C.S. Iliopoulos and W.F. Smyth (Eds.): IWOCA 2011, LNCS 7056, pp. 156–169, 2011.
© Springer-Verlag Berlin Heidelberg 2011

graph admits a straight-line RAC drawing is $\mathcal{NP}$-hard in the general case [3] and it is not even known if this problem is in $\mathcal{NP}$.

In this paper we present the first efficient algorithms for straight-line RAC drawings. We focus on bipartite graphs and both consider the problem of deciding whether a bipartite graph admits a straight-line RAC drawing and the problem of computing one in the positive case. We also study how to efficiently compute straight-line RAC drawings of bipartite graphs with the minimum number of edge crossings. We adopt the widely accepted 2-*layer drawing* paradigm, in which the vertices of each partition set lie on a distinct horizontal layer. A limited list of papers on 2-layer drawings of bipartite graphs includes [9,13,17,20]; for more references see also [18]. A 2-*layer RAC drawing* is a 2-layer straight-line drawing with right angle crossings. An overview of our results is given below (n denotes the number of vertices of the input graph).

- We characterize 2-layer RAC drawable graphs (Theorem 6). This is the counterpart for RAC drawings of the characterization of 2-layer planar drawings (see, e.g., [11,14,19]). Our characterization implies that 2-layer RAC drawings have at most $1.5n - 2$ edges, which is a tight bound (Corollary 1); we also give an $O(n)$-time algorithm that tests whether a graph has a 2-layer RAC drawing and, if so, it computes one.
- We show an $O(n^2 \log n)$-time algorithm to compute a 2-layer RAC drawing with the minimum number of edge crossings (Theorem 7). The algorithm models the optimization problem as the one of computing a flow of minimum cost on a suitable network. We recall that computing a 2-layer drawing of a graph with the minimum number of crossings is $\mathcal{NP}$-hard and that heuristics [9], approximation algorithms [13], FPT algorithms [6,8], and exact methods [17,20] for this problem have been described.
- Finally, we study the complexity of computing the maximum 2-layer RAC drawable subgraph. We prove that for a given bipartite graph G and for a given k, it is $\mathcal{NP}$-complete to decide whether G has a 2-layer RAC drawable subgraph with at least k edges (Theorem 8). This extends to RAC drawings the $\mathcal{NP}$-completeness result for the maximum 2-layer planar subgraph problem [12].

2 Geometry and Combinatorics of 2-Layer RAC Drawings

Let $G = (V_1, V_2, E)$ be a bipartite graph. A 2-layer drawing of G has a *fan crossing* if there exist two adjacent edges that are both crossed by a third edge. For a given 2-layer drawing of G, denote by ℓ_i the horizontal line on which the vertices of V_i are drawn ($i = 1, 2$). We always assume that ℓ_1 is above ℓ_2. Two 2-layer drawings of G are *equivalent* if they have the same left-to-right order π_i of the vertices of V_i ($i = 1, 2$) along ℓ_i. A 2-*layer embedding* is an equivalence class of 2-layer drawings and it is described by a pair of linear orderings (i.e., permutations) $\gamma = (\pi_1, \pi_2)$ of the vertices in V_1 and V_2, respectively. If Γ is a drawing within class γ, we also say that γ is the *embedding of* Γ. Let Γ_1 and Γ_2 be 2-layer drawings of G with the same embedding γ. Two edges e and e' cross in Γ_1 if and only if they cross in Γ_2. We say that embedding γ *has a crossing* at edges e and e'. Also, three edges e, e' and e'' form a fan crossing in Γ_1

if and only if they form a fan crossing in Γ_2. Correspondingly, we say that embedding γ *has a fan crossing* at edges e, e', and e''. Let $\gamma = (\pi_1, \pi_2)$ be a 2-layer embedding of a bipartite graph G. The first (last) vertex of π_1 and the first (last) vertex of π_2 are the *leftmost vertices* (*rightmost vertices*) of γ.

The following result proves that the problem of computing a 2-layer RAC drawing of a bipartite graph $G = (V_1, V_2, E)$ can be studied in purely combinatorial terms as the one of choosing a suitable pair (π_1, π_2) of permutations of the vertices in V_1 and V_2, disregarding details about the exact coordinates of the vertices.

Theorem 1. *Let G be a connected graph with n vertices. G is 2-layer RAC drawable if and only if it has a 2-layer embedding without fan crossings. Also, if γ is a 2-layer embedding of G without fan crossings, there exists an $O(n)$-time algorithm that computes a 2-layer RAC drawing of G with embedding γ.*

Sketch of Proof: The proof is by construction. The drawing algorithm assigns real coordinates to the vertices of G in such a way that the vertex ordering defined by γ on the two layers is preserved and each crossing edge has either 45-degree or -45-degree slope. Crossing edges have opposite slopes, so to form orthogonal crossings. □

A 2-layer embedding without fan crossings is a *2-layer RAC embedding*. Based on Theorem 1, the problem of characterizing 2-layer RAC drawable graphs is equivalent to characterizing which graphs have a 2-layer RAC embedding. Note that, a graph is 2-layer RAC drawable if and only if its connected components are 2-layer RAC drawable. Hence, from now on we assume to work on connected graphs. Given two vertices a and b of a path, we will denote by $d(a, b)$ the *distance* between a and b on the path, that is, the number of edges from a to b along the path. Given a vertex v, the *degree* of v is the number of edges incident to v and is denoted as $deg(v)$.

3 Characterization and Testing Algorithms

We start by giving an intuition of the characterization of 2-layer RAC drawable graphs. In the more general case, a 2-layer RAC drawable graph consists of a set of non-trivial biconnected components[1] and a set of tree components that are ordered along the two layers. Such an order is not, in fact, a total order because there can be some overlap between different tree components. For example, Fig. 1(b) shows a 2-layer RAC drawing of the graph G depicted in Fig. 1(a). G consists of two non-trivial biconnected components and five tree components (highlighted in Fig. 1(c)) that are ordered left-to-right along the two layers in the drawing; as shown in Fig. 1(d), the left-to-right order is not a total ordering because in some cases the tree components overlap as it happens for components T_1 and T_2 and for components T_4 and T_5 in the drawing of Fig. 1(b).

In what follows we first characterize biconnected 2-layer RAC drawable graphs (Section 3.1), and then 2-layer RAC drawable trees (Section 3.2). These two characterizations will then be combined to characterize 2-layer RAC drawable graphs. The combination, however, is not straightforward because the drawings of the different components of G must satisfy additional properties in order to be assembled together.

[1] A trivial biconnected component consists of a single edge.

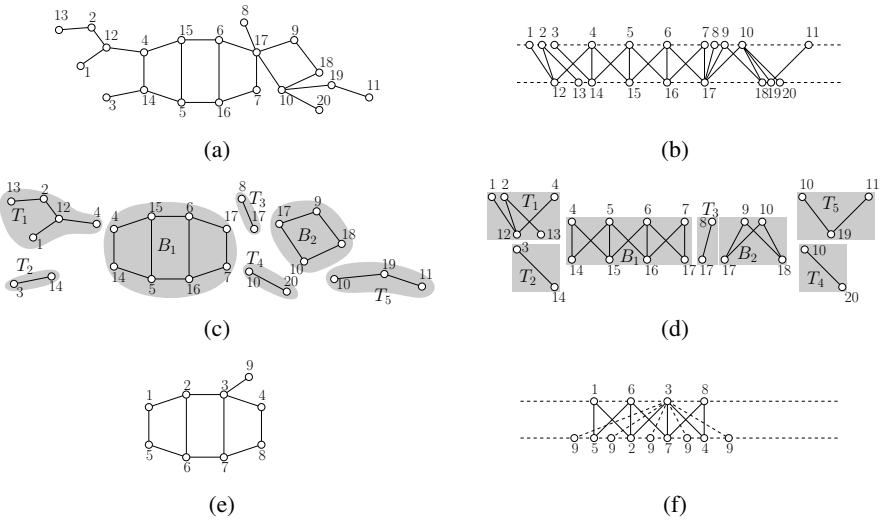


Fig. 1. (a) A graph G. (b) A 2-layer RAC drawing Γ of G. (c) The biconnected components and the tree components of G. (d) The biconnected components and the tree components of G in Γ. (e) A graph G'. (f) G' does not admit a 2-layer RAC drawing.

In particular, it must be possible to order the components along the two layers in such a way that: (i) Each component shares vertices only with the components that immediately precede and follow it and with the components that overlap with it; (ii) The vertices shared by two consecutive components C_1 and C_2 (in this left-to-right order) must be the rightmost for C_1 and the leftmost for C_2. For example, the graph G' of Fig. 1(e) has a biconnected component (isomorphic to B_1 in Fig. 1(c)) and a tree component (consisting of edge $(3, 9)$). Both components are 2-layer RAC drawable, but the biconnected component does not admit a 2-layer RAC drawing with the vertex 3 as leftmost or rightmost vertex; as a consequence whatever the position of vertex 9 will be, it is not possible to obtain a 2-layer RAC drawing of G' (see Fig. 1(f)). For this reason, Sections 3.1 and 3.2 contain: (i) the characterizations of biconnected 2-layer RAC drawable graphs and 2-layer RAC drawable trees; (ii) the characterization of biconnected graphs that are 2-layer RAC drawable in such a way that two specified edges are the leftmost and the rightmost ones; (iii) the characterization of trees that are 2-layer RAC drawable so that two specified leaves are the leftmost and the rightmost ones.

3.1 Characterization of Biconnected Graphs

Let G be a biconnected bipartite graph with at least two edges, and let e and e' be two independent edges of G. If there exists a 2-layer RAC embedding γ of G such that the end-vertices of e are the leftmost vertices of γ and the end-vertices of e' are the rightmost vertices of γ, we say that γ is a 2-*layer RAC embedding of* G *with respect to* e *and* e'. Clearly, edges e and e' cannot cross any edge in γ. A 2-layer RAC drawing with embedding γ is a 2-*layer RAC drawing with respect to* e *and* e'. If G admits such a

drawing, we say that G is 2-*layer RAC drawable with respect to* e *and* e' (see Fig. 2(b)) A biconnected bipartite graph is a *ladder* if it consists of two paths of the same length $\langle u_1, u_2, \ldots, u_{\frac{n}{2}} \rangle$ and $\langle v_1, v_2, \ldots, v_{\frac{n}{2}} \rangle$ plus the edges (u_i, v_i) $(i = 1, 2, \ldots, \frac{n}{2})$ (see Fig. 2(a)). The edges $e = (u_1, v_1)$ and $e' = (u_{\frac{n}{2}}, v_{\frac{n}{2}})$ are the *extremal edges* of the ladder. Theorem 2 characterizes those biconnected graphs that are 2-layer RAC drawable with respect to two independent edges. Theorem 3 is the counterpart of Theorem 2 without any fixed extremal edges.

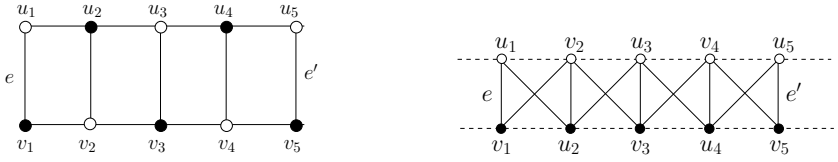


Fig. 2. (a) A ladder G. (b) A 2-layer RAC drawing of G with respect to e and e'.

Theorem 2. *Let G be a biconnected graph and let e and e' be two independent edges of G. G is 2-layer RAC drawable with respect to e and e' if and only if it is a spanning subgraph of a ladder with extremal edges e and e'. Also, if G has n vertices, there exists an $O(n)$-time algorithm that tests whether G admits a 2-layer RAC drawing with respect to e and e' and, if so, it computes such a drawing.*

Sketch of Proof: We prove that every 2-layer RAC embedding of G with respect to edges e and e' is such that the edges of the external cycle are interlaced like shown in Fig. 2(b). These edges are the only edges that can cross in a 2-layer RAC drawing. □

Theorem 3. *Let G be a biconnected graph. G is 2-layer RAC drawable if and only if it is a spanning subgraph of a ladder. Also, if G has n vertices, there exists an $O(n)$-time algorithm that tests whether G is 2-layer RAC drawable and, if so, it computes a 2-layer RAC drawing of G.*

3.2 Characterization of Trees

Roughly speaking, a 2-layer RAC drawing of a tree consists of a monotone "zig-zag" path between the two layers with some suitable sub-structures attached to its vertices (see Fig. 3(b)). In order to define the different types of sub-structures we define a sort of simplified version of the tree, called its *weighted contraction*. The characterization for trees relies on this concept. As explained at the beginning of this section, along with the characterization of 2-layer RAC drawable trees (Theorem 5), we give the characterization of 2-layer RAC drawable trees where two given vertices u and v are required to be a leftmost vertex and a rightmost vertex, respectively (Theorem 4). The latter characterization will be used in Subsection 3.3 to characterize 2-layer RAC drawable graphs. In Theorem 4 we focus on the case when u and v are two leaves for two reasons: (i) This is

the only case needed for the characterization of 2-layer RAC drawable graphs that contain both non-trivial biconnected components and tree components; (ii) As Lemma 1 claims, any 2-layer RAC drawable tree admits a 2-layer RAC drawing where a leftmost vertex and a rightmost vertex are leaves.

Let T be a tree and let u and v be two leaves of T. If there exists a 2-layer RAC embedding γ where u and v are a leftmost vertex and a rightmost vertex of γ, respectively, we say that γ is a 2-*layer RAC embedding with respect to* u *and* v. A 2-layer RAC drawing with embedding γ is a 2-*layer RAC drawing with respect to* u *and* v. If T admits such a drawing, we say that T is 2-*layer RAC drawable with respect to* u *and* v.

Lemma 1. *Let T be a 2-layer RAC drawable tree. There exists two leaves u and v such that T is 2-layer RAC drawable with respect to u and v.*

A *weighted contraction* of T is a weighted tree obtained from T by replacing each chain of length $k > 1$ with a single edge of weight k. We denote by $wc(T)$ the weighted contraction of T and by $\omega(a, b)$ the weight of an edge (a, b) of $wc(T)$. An edge of T that also belongs to $wc(T)$ has weight 1. The vertices of $wc(T)$ are a subset of the vertices of T and every vertex of $wc(T)$ has the same degree in T and in $wc(T)$. A path between two leaves u, v of T is a *spine* of T and is denoted by $s_{uv}(T)$. The path between u and v in $wc(T)$, denoted by $sc_{uv}(T)$, is a *spine* of $wc(T)$. The vertices of $sc_{uv}(T)$ are the *spine vertices*; the others are *non-spine vertices* (see also Fig. 3).

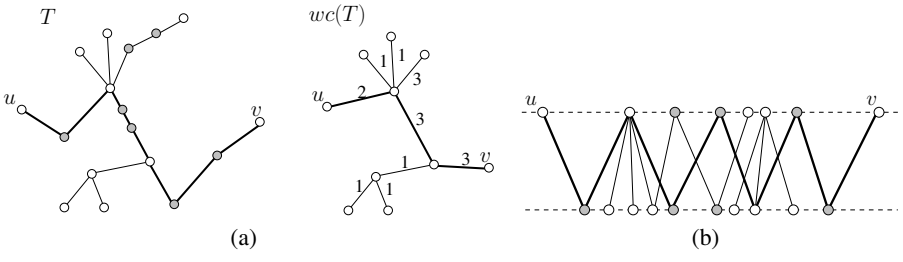


Fig. 3. (a) A tree T and its weighted contraction $wc(T)$; the gray vertices are internal vertices of chains and disappear in $wc(T)$; the path between u and v in T (edges in bold) is a spine. (b) A 2-layer RAC embedding of T with respect to u and v.

Our characterization of a 2-layer RAC drawable tree is expressed in terms of properties of its weighted contraction. In order to do that, we look at what type of subtrees are "attached" to the vertices of the spine. For a given spine $sc_{uv}(T)$ of $wc(T)$, we define three kinds of subtrees in $wc(T)$, called *k-fence*, *y-tree*, and *star-tree*, respectively. Also, for a subtree T' of any of the types above, we give the definition of *feasibility* of T', which expresses the possibility of representing the "non-contracted" version of T' in a 2-layer RAC embedding with respect to the end-vertices, u and v, of the spine. We will prove that a tree T is 2-layer RAC drawable with respect to u and v if and only if every subtree "attached" to $sc_{uv}(T)$ is either a feasible *k-fence*, or a feasible *y-tree*, or a feasible *star-tree*. The three kinds of subtrees are defined as follows:

- k**-fence** (refer to Fig. 4(a) and 4(c)). Let $\langle z_0, z_1, z_2, \ldots, z_k, z_{k+1} \rangle$ ($k \geq 2$) be a maximal sequence of spine vertices such that: $deg(z_i) = 3$ ($i \in \{1, \ldots, k\}$); $\omega(z_i, z_{i+1}) = 1$ ($i \in \{1, \ldots, k-1\}$); $deg(w_i) = 1$, where w_i is the non-spine vertex adjacent to z_i ($i \in \{1, \ldots, k\}$). The weighted subtree T' induced by the vertices of $\{z_i \mid i = 1, \ldots, k\} \cup \{w_i \mid i = 1, \ldots, k\}$ is a *k-fence* of $wc(T)$. The subsequence $z_1, z_2, \ldots, z_k$ is the *chain root* of T', each w_i is a *leaf* of T'.
  - A 2-fence is *feasible* if either $\omega(z_1, w_1) \leq \omega(z_0, z_1) + 1$ and $\omega(z_2, w_2) \leq \omega(z_2, z_3) + 1$ or $\omega(z_2, w_2) \leq \omega(z_0, z_1)$ and $\omega(z_1, w_1) \leq \omega(z_2, z_3)$.
  - A 3-fence is *feasible* if one of the following conditions holds:
    - the subtree induced by z_1, z_2, w_1, w_2 is a feasible 2-fence and $\omega(z_3, w_3) \leq \omega(z_3, z_4) + 1$;
    - the subtree induced by z_2, z_3, w_2, w_3 is a feasible 2-fence and $\omega(z_1, w_1) \leq \omega(z_0, z_1) + 1$;
  - A k-fence ($k > 3$) is *feasible* if the following conditions holds:
    - the subtree induced by z_1, z_2, w_1, and w_2 is a feasible 2-fence;
    - the subtree induced by z_{k-1}, z_k, w_{k-1}, and w_k is a feasible 2-fence; (iii) If $k \geq 5$, then $\omega(z_i, w_i) \leq 2$ ($i \in \{3, \ldots, k-2\}$).
- y**-tree** (refer to Fig. 4(d)). Let z be a degree-3 spine vertex of $wc(T)$ that does not belong to any k-fence and such that: (i) $deg(w) = 3$, where w is the non-spine vertex adjacent to z: and (ii) the vertices $a, b \neq z$ adjacent to w have degree one. The weighted subtree T' induced by z, w, a, b is a *y-tree* of $wc(T)$. Vertex z is the *root*, w is the *internal vertex*, and a, b are the *leaves* of T'. Denote by z_1, z_2 the spine vertices adjacent to z. Tree T' is *feasible* if: (i) $\omega(z, w) = 1$; (ii) either $\omega(w, a) \leq \omega(z_1, z)$ and $\omega(w, b) \leq \omega(z, z_2)$ or $\omega(w, b) \leq \omega(z_1, z)$ and $\omega(w, a) \leq \omega(z, z_2)$.
- **star-tree** (refer to Fig. 4(e)). Let z be a spine vertex of $wc(T)$ that does not belong to either a k-fence or a y-tree and such that: (i) $deg(z) \geq 3$; (ii) every non-spine vertex w_i adjacent to z has degree one ($i = 1, \ldots, deg(z)$). The weighted subtree T' induced by z and all vertices w_i is a *star-tree* of $wc(T)$. Vertex z is the *root* of T'. Denote by z_1, z_2 the spine vertices adjacent to z. Tree T' is *feasible* if: (i) There exist at most two vertices w_j, w_h such that $1 < \omega(z, w_j) \leq \omega(z_1, z) + 1$ and $1 < \omega(z, w_h) \leq \omega(z, z_2) + 1$; (ii) for every vertex $w_i \notin \{w_j, w_h\}$, $\omega(z, w_i) = 1$.

A spine $s_{uv}(T)$ is *feasible* if for every vertex $z \in sc_{uv}(T)$ distinct from u and v one of the three conditions holds: (i) z belongs to the chain root of a k-fence; (ii) z is the root of exactly one feasible y-tree; (iii) z is the root of exactly one feasible star-tree.

Theorem 4. *Let T be a tree and let u, v be two leaves of T. Tree T is 2-layer RAC drawable with respect to u and v if and only if $s_{uv}(T)$ is a feasible spine. Also, if T has n vertices, there exists an $O(n)$-time algorithm that tests whether T admits a 2-layer RAC drawing with respect to u and v and, if so, it computes such a drawing.*

Sketch of Proof: The sufficiency is by construction: The spine is drawn as a monotone "zig-zag" path; the non-contracted version of each feasible substructure of the weighted contraction is embedded as shown in Fig. 5(a)– 5(f). The necessity is proved by case analysis on the vertices of degree larger than two in the weighted contraction of T. □

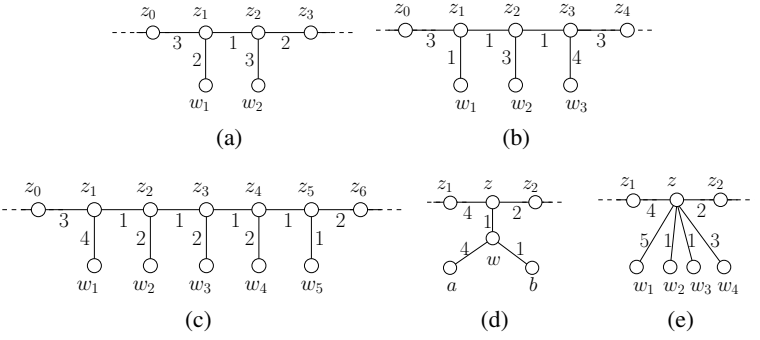


Fig. 4. Feasible (a) 2-fence, (b) 3-fence, (c) 5-fence, (d) y-tree, and (e) star-tree

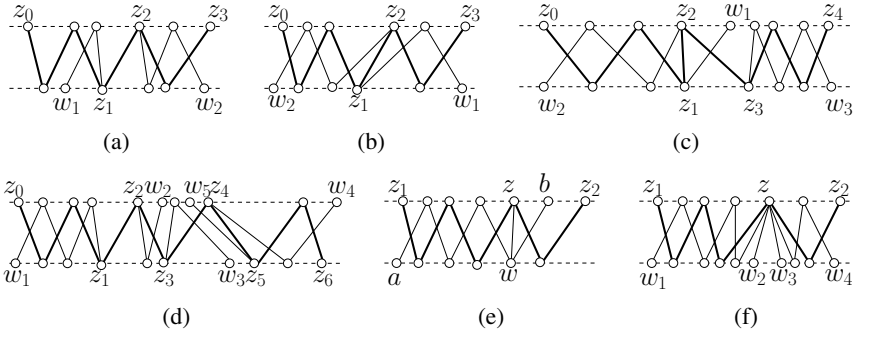


Fig. 5. 2-layer RAC embeddings for the non-contracted version of the feasible substructures of Fig. 4; (f)-(g) 2-fence, (h) 3-fence, (i) 5-fence, (j) y-tree, and (k) star-tree

Theorem 5. *A tree T is 2-layer RAC drawable if and only if it admits a feasible spine. Also, if T has n vertices, there exists an $O(n)$-time algorithm that tests whether T is 2-layer RAC drawable and, if so, it computes a 2-layer RAC drawing of T.*

Sketch of Proof: The characterization is a consequence of Lemma 1 and Theorem 4. The existence of a linear time testing algorithm is proved by showing that a constant number of pairs of leaves as end-vertices of a feasible spine can be considered. □

3.3 Characterization of 2-Layer RAC Graphs

Intuitively, a general 2-layer RAC drawable graph is a chain of non-trivial biconnected components (each of them being the spanning subgraph of a ladder) alternated with trees having a feasible spine. Additionally, some other simple types of trees can be attached to the vertices of the extremal edges of a biconnected component (we call each of such trees an *addendum*). Fig. 6(a) shows a 2-layer RAC drawable graph and Fig. 6(d) depicts a 2-layer RAC embedding of G. More formally, a connected graph

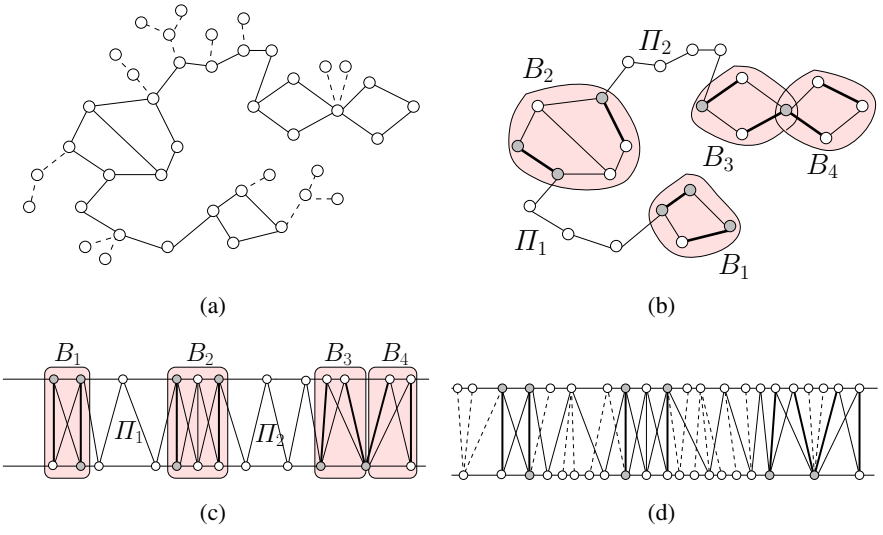
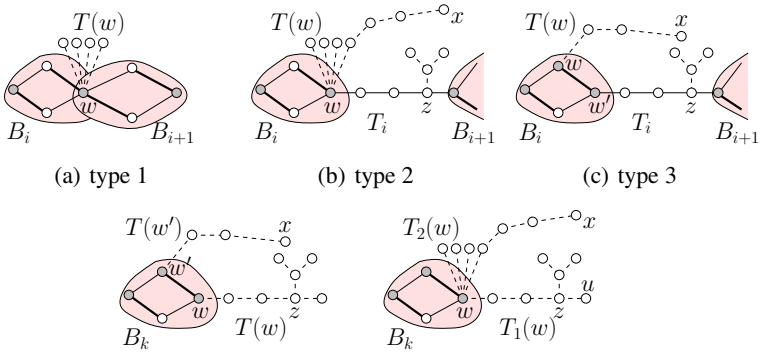


Fig. 6. (a) A graph G. Solid edges form $skel(G)$. (b) $skel(G)$ is feasible; for each component B_i, the gray vertices are the cut-vertices of G in B_i, and the bold edges are two edges e, e' that cover all the cut-vertices in B_i and such that B_i is 2-layer RAC drawable with respect to e, e'. (c) A 2-layer RAC embedding of $skel(G)$. (d) A 2-layer RAC embedding of G.

G is *outerplanarly biconnectible* if it has two vertices $\{s, t\}$ such that $G \cup (s, t)$ is outerplanar and biconnected. Let G be a bipartite graph that is neither biconnected nor a tree. The *skeleton of* G, denoted as $skel(G)$, is the subgraph of G obtained by repeatedly removing the vertices of degree one. We denote by $\{B_1, \ldots, B_k\}$ the non-trivial biconnected components of $skel(G)$. We say that $skel(G)$ is *feasible* if $skel(G)$ is outerplanarly biconnectible and each B_i $(i = 1, \ldots, k)$ contains two independent edges e and e' such that: (a) Each cut-vertex of G in B_i is an end-vertex of e or e'; (b) B_i is 2-layer RAC drawable with respect to e and e'. Fig. 6(a)-6(c) show a graph G, its (feasible) skeleton $skel(G)$, and a 2-layer RAC embedding of $skel(G)$. Observe that, if $skel(G)$ is feasible then each B_i has at most four cut-vertices of G. Also, the components $\{B_1, \ldots, B_k\}$ form a sequence such that B_i and B_{i+1} are connected by a path Π_i from a vertex u_i of B_i to a vertex v_i of B_{i+1} $(i = 1, \ldots, k-1)$, where u_i may coincide with v_i. For each path Π_i, we denote by T_i the tree consisting of Π_i and all the subtrees of G rooted at each internal vertex of Π_i. A path Π_i is a *bridge* of $skel(G)$ and tree T_i is the *tree of* Π_i. If Π_i is a single vertex then T_i coincides with Π_i, and Π_i is a *degenerate bridge*.

It is easy to see that G is 2-layer RAC drawable only if $skel(G)$ is feasible and each Π_i is a feasible spine of T_i. However, for the characterization, we need additional conditions. Let T be a tree that is 2-layer RAC drawable with respect to two leaves u and v. We denote by $\nu(u, T)$ the vertex closest to u along $s_{uv}(T)$ such that $deg(\nu(u, T)) \geq 3$ in T. If such a vertex does not exist then $\nu(u, T)$ coincides with v. We can analogously define $\nu(v, T)$. Let w be a cut-vertex of G that belongs to B_i $(1 \leq i \leq k)$. Denote by

$T(w)$ the subtree of G rooted at w. Clearly G is the union of all B_i, all T_i, and all $T(w)$, for each cut-vertex w of G that belongs to some B_i. Each tree $T(w)$ is an *addendum* of $skel(G)$. We now classify each addendum $T(w)$ of $skel(G)$ and correspondingly define the *partner of* $T(w)$ (the partner is the tree that can overlap with $T(w)$). Also, we define the properties that each type of addendum must satisfy to be embedded correctly in a 2-layer RAC embedding of G whose skeleton is feasible (refer to Fig. 7):



(a) type 1 (b) type 2 (c) type 3

(d) type 4 (e) type 5

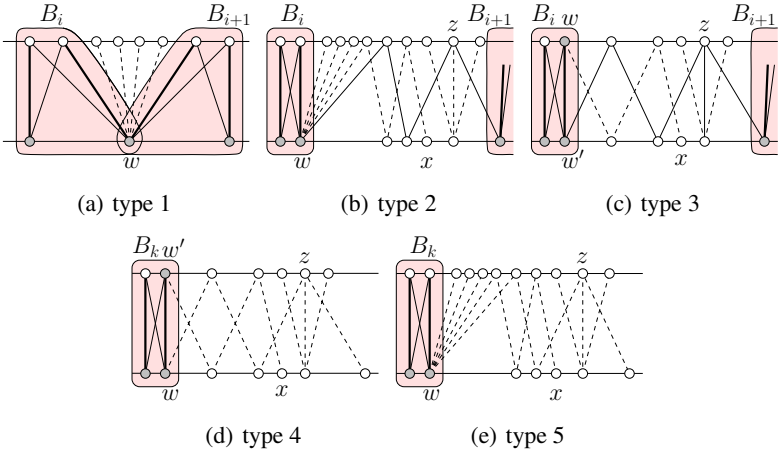
Fig. 7. (a)-(e): Illustration of the different types of addendum. Each addendum in this picture is feasible

- If w coincides with the single vertex of a degenerate bridge, $T(w)$ is a tree of *type 1* and it has no partner. $T(w)$ is *feasible* if it is a star centered at w.
- If w coincides with an end-vertex of a non-degenerate bridge Π_i, $T(w)$ is a tree of *type 2* and its partner is T_i. $T(w)$ is *feasible* if it consists of a star centered at w plus at most one path from w to a leaf x such that $d(w, x) \leq d(w, z) + 1$, where $z = \nu(w, T_i)$.
- If w is adjacent to an end-vertex w' of a non-degenerate bridge Π_i, $T(w)$ is a tree of *type 3* and its partner is T_i. $T(w)$ is *feasible* if it consists of a path from w to a leaf x such that $d(w, x) \leq d(w', z)$, where $z = \nu(w', T_i)$.
- Otherwise, w is a cut-vertex of $B \in \{B_1, B_k\}$ and one of the following cases applies:
  - w is adjacent to another cut-vertex w' of G that belongs to B and that does not belong to any Π_i, in which case $T(w)$ and $T(w')$ are trees of *type 4* and each of them is the partner of the other. $T(w)$ and $T(w')$ are *feasible* if (i) w is a leaf of $T(w)$ and there is another leaf u of $T(w)$ such that $T(w)$ is 2-layer RAC drawable with respect to u and w; (ii) $T(w')$ is a path from w to a leaf x such that $d(w', x) \leq d(w, z)$, where $z = \nu(w, T(w))$.
  - w is not adjacent to a cut-vertex, in which case $T(w)$ is of *type 5* and has no partner. $T(w)$ is *feasible* if it can be decomposed into subtrees $T_1(w)$ and $T_2(w)$ such that: (i) $T_1(w) \cap T_2(w) = \{w\}$ ($T_2(w)$ may consist of w only); (ii) w is a leaf of $T_1(w)$ and there is another leaf u of $T_1(w)$ such that $T_1(w)$

is 2-layer RAC drawable with respect to u and w; (iii) $T_2(w)$ is a star centered at w plus at most one path from w to a leaf x, such that $d(w, x) \leq d(w, z) + 1$, where $z = \nu(w, T_1(w))$.

Theorem 6. *A graph G is 2-layer RAC drawable if and only if one of the following cases holds: (i) G is biconnected and it is a spanning subgraph of a ladder. (ii) G is a tree that admits a feasible spine. (iii) $skel(G)$ is feasible, each path of $skel(G)$ is a feasible spine of its tree, and each addendum of $skel(G)$ is also feasible. Furthermore, if G has n vertices, there exists an $O(n)$-time algorithm that tests whether G is 2-layer RAC drawable and, if so, it computes a 2-layer RAC drawing of G.*

Sketch of Proof: The proof of the theorem combines the arguments used to prove Theorems 2-5. Fig. 8 shows how to embed each type of addendum. □



(a) type 1 (b) type 2 (c) type 3

(d) type 4 (e) type 5

Fig. 8. (a)-(e) 2-layer RAC embedding for the different types of addendum

Since by Theorem 6, ladders are the densest graphs admitting a 2-layer RAC drawing, it is immediate to derive the following bound.

Corollary 1. *A 2-layer RAC drawable graph with n vertices has at most $1.5n-2$ edges, which is a tight bound.*

4 Optimization Problems

In this section we consider two optimization problems that are naturally raised by the results in the previous sections. Namely, we consider both the problem of computing a 2-layer RAC drawing with the minimum number of edge crossings and the problem of extracting the maximum 2-layer RAC drawable subgraph of a given bipartite graph.

We first show that if G is a 2-layer RAC drawable graph then a 2-layer RAC drawing of G with the minimum number of crossings can be computed in polynomial time. If

G is a biconnected graph the problem is easy, because all crossings in any 2-layer RAC drawing of G are formed by the edges of the external cycle. Hence, every 2-layer RAC drawing of a biconnected component G with $n > 2$ vertices has $(n-2)/2$ crossings. For reasons of space, we only show how to model the crossing minimization problem for a tree T, assuming that no k-fences are present in the weighted contraction of T.

Suppose that T is a 2-layer RAC drawable tree with respect to two leaves u and v and let $sc_{uv}(T) = \langle u = z_0, z_1, z_2, \ldots, z_h, z_{h+1} = v \rangle$ such that each z_i is the root of a subtree T' that is either a y-tree or a star-tree of $wc(T)$. In both cases, the non-contracted version of T' contains at most two paths $\Pi_{i,1}$ and $\Pi_{i,2}$, going from z_i to two leaves $x_{i,1}$ and $x_{i,2}$, respectively, such that $d(z_i, x_{i,1}) > 1$ and $d(z_i, x_{i,2}) > 1$. Note that, if T' is a y-tree, $\Pi_{i,1}$ and $\Pi_{i,2}$ share the first edge. In a 2-layer RAC embedding γ of T, $\Pi_{i,1}$ and $\Pi_{i,2}$ are embedded one to the left and one to the right of z_i, as explained in the previous sections (see Fig. 5(e), where $z_i = z$, $x_{i,1} = a$, $x_{i,2} = b$, and Fig. 5(f), where $z_i = z$, $x_{i,1} = w_1$, $x_{i,2} = w_4$). If $d(z_{i-1}, z_i)$ and $d(z_i, z_{i+1})$ are both greater than or equal to $\max\{d(z_i, x_{i,1}) - 1, d(z_i, x_{i,2}) - 1\}$, then we can arbitrarily decide which path $\Pi_{i,j}$ goes to the left of z_i and which one goes to the right of z_i $(j = 1, 2)$; in this case, we say that $\Pi_{i,1}$ and $\Pi_{i,2}$ are *free paths*, otherwise we say that they are *constrained paths*. The number of crossings that $\Pi_{i,j}$ forms with the spine $s_{uv}(T)$ is the same for each of the two choices, and it is equal to $d(z_i, x_{i,j}) - 1$. However, the path $\Pi_{i,j}$ embedded to the left of z_i may form some crossings with a path $\Pi_{i-1,l}$ $(l = 1, 2)$ embedded to the right of z_{i-1}, if such a path exists; if so, the number of crossings between the two paths is at least $d(z_i, x_{i,j}) + d(z_{i-1}, x_{i-1,l}) - d(z_{i-1}, z_i) - 3$, and the embedding constructed with our technique matches this number. Hence, our crossing minimization problem reduces to the problem of embedding each path $\Pi_{i,j}$ to the left or to the right of z_i so that the total number of crossings between all paths is minimized. In the following we assume that for each vertex z_i $(i = 1, \ldots, h)$ there are exactly two paths $\Pi_{i,1}$ and $\Pi_{i,2}$. If not, we can attach to z_i dummy paths of length 1, which can always be embedded without crossings. If γ is a 2-layer RAC embedding where $\Pi_{i,j}$ is to the left of z_i and $\Pi_{i-1,l}$ is to the right of z_{i-1} we say that $\Pi_{i,j}$ and $\Pi_{i-1,l}$ are *matched*, and the *cost* of their matching equals $\max\{0, d(z_i, x_{i,j}) + d(z_{i-1}, x_{i-1,l}) - d(z_{i-1}, z_i) - 3\}$. Each path $\Pi_{i,j}$ is matched with some other path in γ, but the path to the left of z_1 and the path to the right of z_h (these paths do not cross any other path). Thus, if γ is a 2-layer RAC embedding with the minimum number of crossings, γ corresponds to a maximum matching among $2h - 2$ paths with minimum total cost. Also, the path to the left of z_1 in γ is the longest between $\Pi_{1,1}$ and $\Pi_{1,2}$, unless $\Pi_{1,1}$ and $\Pi_{1,2}$ are constrained to be embedded in the other way. Similarly, the path to the right of z_h is the longest between $\Pi_{h,1}$ and $\Pi_{h,2}$, unless $\Pi_{h,1}$ and $\Pi_{h,2}$ are constrained to be embedded in the other way.

To compute a 2-layer RAC embedding γ with the minimum number of crossings, we compute the corresponding optimal matching by solving a minimum cost flow problem on a suitable capacitated network $\mathcal{N}$, having a single source and a single sink. Since $\mathcal{N}$ has $O(n)$ vertices and edges, and since the value of the maximum flow is $O(n)$, a maximum flow of minimum cost can be computed in $O(n^2 \log n)$ time [1].

Theorem 7. *Let G be a 2-layer RAC drawable graph. A 2-layer RAC embedding of G with the minimum number of crossings can be computed in $O(n^2 \log n)$ time.*

As for the computation of the maximum 2-layer RAC drawable subgraph, we prove the $\mathcal{NP}$-completeness of its associated decision problem, namely the following:

MAXIMUM 2-LAYER RAC SUBGRAPH (M2LS): Given a bipartite graph G and a positive integer k, does G admit a 2-layer RAC drawable subgraph with k edges?

Guessing an ordering of the vertices along the two layers proves that M2LS is in $\mathcal{NP}$. The hardness is proved by reducing HP3 to M2LS, where HP3 is the problem of deciding whether a cubic graph has a Hamiltonian path.

Theorem 8. M2LS *is* $\mathcal{NP}$*-complete*

5 Open Problems

We conclude the paper by listing some open problems that we consider worth to investigate.

1. Design heuristic methods or approximation algorithms for the Maximum 2-layer RAC Drawable Subgraph problem.
2. Study the problem of computing 2-layer drawings where the number of crossings that form $\frac{\pi}{2}$ angles is maximized.
3. Study the problem of computing 2-layer drawings of a bipartite graphs where the minimum crossing angle is at least a given α such that $0 \leq \alpha < \frac{\pi}{2}$.

References

1. Ahuja, R.K., Magnanti, T.L., Orlin, J.B.: Network Flows: Theory, Algorithms, and Applications. Prentice-Hall (1993)
2. Angelini, P., Cittadini, L., Di Battista, G., Didimo, W., Frati, F., Kaufmann, M., Symvonis, A.: On the perspectives opened by right angle crossing drawings. Journal of Graph Algorithms and Applications 15(1), 53–78 (2011)
3. Argyriou, E.N., Bekos, M.A., Symvonis, A.: The Straight-Line RAC Drawing Problem is NP-Hard. In: Černá, I., Gyimóthy, T., Hromkovič, J., Jefferey, K., Králović, R., Vukolić, M., Wolf, S. (eds.) SOFSEM 2011. LNCS, vol. 6543, pp. 74–85. Springer, Heidelberg (2011)
4. Didimo, W., Eades, P., Liotta, G.: Drawing Graphs with Right Angle Crossings. In: Dehne, F., Gavrilova, M., Sack, J.-R., Tóth, C.D. (eds.) WADS 2009. LNCS, vol. 5664, pp. 206–217. Springer, Heidelberg (2009)
5. Didimo, W., Eades, P., Liotta, G.: A characterization of complete bipartite RAC graphs. Inf. Process. Lett. 110(16), 687–691 (2010)
6. Dujmović, V., Fellows, M.R., Hallett, M.T., Kitching, M., Liotta, G., McCartin, C., Nishimura, N., Ragde, P., Rosamond, F.A., Suderman, M., Whitesides, S., Wood, D.R.: A fixed-parameter approach to 2-layer planarization. Algorithmica 45(2), 159–182 (2006)
7. Dujmović, V., Gudmundsson, J., Morin, P., Wolle, T.: Notes on large angle crossing graphs. In: Proceedings of the Sixteenth Symposium on Computing: the Australasian Theory, CATS 2010, vol. 109, pp. 19–24. Australian Computer Society, Inc. (2010)
8. Dujmović, V., Whitesides, S.: An efficient fixed parameter tractable algorithm for 1-sided crossing minimization. Algorithmica 40(1), 15–31 (2004)
9. Eades, P., Kelly, D.: Heuristics for drawing 2-layered networks. Ars Comb. 21, 89–98 (1986)
10. Eades, P., Liotta, G.: Right angle crossing graphs and 1-planarity. In: EuroCG (2011)

11. Eades, P., McKay, B., Wormald, N.: On an edge crossing problem. In: Proc. of 9th Australian Computer Science Conference, pp. 327–334 (1986)
12. Eades, P., Whitesides, S.: Drawing graphs in two layers. Theoretical Computer Science 131(2), 361–374 (1994)
13. Eades, P., Wormald, N.C.: Edge crossings in drawings of bipartite graphs. Algorithmica 11(4), 379–403 (1994)
14. Harary, F., Schwenk, A.: A new crossing number for bipartite graphs. Utilitas Mathematica 1, 203–209 (1972)
15. Huang, W.: Using eye tracking to investigate graph layout effects. In: APVIS, pp. 97–100 (2007)
16. Huang, W., Hong, S.-H., Eades, P.: Effects of crossing angles. In: PacificVis, pp. 41–46 (2008)
17. Jünger, M., Mutzel, P.: 2-layer straightline crossing minimization: Performance of exact and heuristic algorithms. J. Graph Algorithms Appl. 1 (1997)
18. Mutzel, P.: An alternative method to crossing minimization on hierarchical graphs. SIAM J. on Optimization 11(4), 1065–1080 (2001)
19. Tomii, N., Kambayashi, Y., Yajima, S.: On planarization algorithms of 2-level graphs. Technical Report EC77-38, Inst. of Elect. and Comm. Eng. Japan (1977)
20. Valls, V., Martí, R., Lino, P.: A branch and bound algorithm for minimizing the number of crossing arcs in bipartite graphs. Europ. J. of Oper. Res. 90(2), 303–319 (1996)
21. van Kreveld, M.: The Quality Ratio of RAC Drawings and Planar Drawings of Planar Graphs. In: Brandes, U., Cornelsen, S. (eds.) GD 2010. LNCS, vol. 6502, pp. 371–376. Springer, Heidelberg (2011)

Hamiltonian Orthogeodesic Alternating Paths

Emilio Di Giacomo[1], Luca Grilli[1], Marcus Krug[2],
Giuseppe Liotta[1], and Ignaz Rutter[2]

[1] Università di Perugia, Italy
{digiacomo,grilli,liotta}@diei.unipg.it
[2] Faculty of Informatics, Karlsruhe Institute of Technology (KIT), Germany
{marcus.krug,rutter}@kit.edu

Abstract. Given a set of red and blue points, an orthogeodesic alternating path is a path such that each edge is a geodesic orthogonal chain connecting points of different colour and no two edges cross. We consider the problem of deciding whether there exists a *Hamiltonian* orthogeodesic alternating path, i.e., an orthogeodesic alternating path visiting all points. We provide an $O(n \log^2 n)$-time algorithm for finding such a path if no two points are horizontally or vertically aligned. We show that the problem is NP-hard if bends must be at grid points. Nevertheless, we can approximate the maximum number of vertices of an orthogeodesic alternating path on the grid by roughly a factor of 3. Finally, we consider the problem of finding orthogeodesic alternating matchings, cycles, and trees.

1 Introduction

Let R and B be two disjoint point-sets in the plane with $|R| \leq |B|$. We shall refer to points of R and B as *red points* and *blue points*, respectively. We say that the set $P = R \cup B$ is *equitable* if $|B| - |R| \leq 1$ and it is *balanced* if $|B| = |R|$. An *alternating path* on a set of red and blue points P is a sequence of points $p_1, \ldots, p_h$ alternatingly red and blue, such that p_i and p_{i+1} ($i = 1, \ldots, h-1$) are connected by a straight-line segment and no two segments cross.

The problem of computing an alternating path on a given equitable set of points in general position is a classical subject of investigation in the computational geometry field. Several papers are devoted to alternating paths containing all points of P; this type of alternating paths are called *Hamiltonian*. Akiyama and Urrutia [3] studied Hamiltonian alternating paths on equitable point-sets in convex positions. They show that it is not always possible to compute a Hamiltonian alternating path on a given equitable point-set and give an $O(n^2)$-time algorithm that, given an equitable point-set, computes a Hamiltonian alternating path if it exists. Abellanas et al. [2] studied the case when points are not restricted to be in convex position; they prove that if either the convex hull of P consists of all the red points and no blue points or the two point-sets are linearly separable (i.e., there exists a straight line that separates the red from the blue points), then a Hamiltonian alternating path always exists. Kaneko, Kano, and Suzuki [7] studied the values of n for which every equitable set of n points admits a Hamiltonian alternating path and proved that this happens only for $n \leq 12$ and $n = 14$; for any other value of n, there exist equitable point-sets that do not admit a Hamiltonian

C.S. Iliopoulos and W.F. Smyth (Eds.): IWOCA 2011, LNCS 7056, pp. 170–181, 2011.
© Springer-Verlag Berlin Heidelberg 2011

alternating path. Cibulka et al. [5] described arbitrarily large equitable point-sets that admit a Hamiltonian alternating path for any colouring of the points. Non-Hamiltonian alternating paths have also been considered in the literature. In particular, the following problem has been investigated: given a set of n red and blue points P in the plane, what is the length $\ell(n)$ of the longest alternating path that can be defined on P? Abellanas et al. [1] and Kynčl et al. [11] studied this problem on special cases of points in convex position and proved upper and lower bounds on the value of $\ell(n)$.

Similar problems on red-blue points with graph families other than paths have also been studied. Abellanas et al. [2] investigate *alternating spanning trees*, i.e., spanning trees of red and blue point-sets such that each edge is a straight-line segment connecting points of different colours and no two edges cross, and prove that every point-set $P = R \cup B$ admits an alternating spanning tree whose maximum vertex degree is $O(\frac{|B|}{|R|} + \log |R|)$. Kaneko, Kano, and Yoshimoto [8] consider *Hamiltonian alternating cycles*, but they allow edge crossings. They proved that at most $n - 1$ crossings are sufficient to compute a Hamiltonian alternating cycle and that this is the best possible in some cases.

In this paper we study *orthogeodesic alternating paths*, i.e., crossing-free alternating paths where the edges are drawn as geodesic orthogonal chains instead of straight-line segments. A *geodesic orthogonal chain* is an orthogonal chain (i.e., a polygonal chain of horizontal and vertical segments) whose length is equal to the Manhattan distance of its endvertices. Since a geodesic orthogonal chain is a connection between two points that has the shortest length in the L_1 metric, orthogeodesic alternating paths can be regarded as the counterpart in the L_1 metric of alternating paths in the L_2 metric. Kano [9] has recently studied equitable point-sets such that no two points are horizontally and vertically aligned. He shows that any of such point-sets admits a perfect matching connecting the red points to the blue ones such that every edge is "L-shaped", that is it consists of exactly one horizontal and exactly one vertical segment. While it is easy to construct an equitable point-set for which a Hamiltonian orthogeodesic alternating path whose edges are all L-shaped does not exist, one may wonder whether every equitable point-set admits a Hamiltonian orthogeodesic alternating path in the L_1 metric.

Contribution. In this paper we describe an $O(n \log^2 n)$-time algorithm that computes a Hamiltonian orthogeodesic alternating path on an equitable set of red and blue points P such that no two points are horizontally or vertically aligned. The computed path has at most two bends per edge which is worst-case optimal. However, the bends along the edges may not have integer coordinates. For a contrast, we show that deciding whether a set of red and blue grid points P admits a Hamiltonian orthogeodesic alternating path with bends at grid points is NP-complete. We also consider several related questions. Namely, we prove that there exist point-sets that do not admit a Hamiltonian orthogeodesic alternating cycle and point-sets such that every alternating spanning tree is, in fact, a path; we describe a $O(n \log^2 n)$-time algorithm that computes an orthogeodesic alternating path of length $(|P| + 2)/3$ with bends at grid points; finally, we show that if points of P are allowed to be horizontally or vertically aligned then it is NP-complete to decide whether a point-set $P = R \cup B$ with $|B| = |R|$ has a perfect orthogeodesic alternating matching. This contrasts a recent paper by Kano stating that such a

matching always exists if we are not allowed to place more than one point per horizontal or vertical line.

For reasons of space some proofs are sketched or omitted.

2 Preliminaries

An *orthogonal chain* is a polygonal chain of horizontal and vertical segments. A *geodesic chain* is an orthogonal chain whose length is equal to the Manhattan distance of its endvertices. A *crossing* between two geodesic chains is an intersection that occurs at an interior point of at least one of the two chains.

Let $P = R \cup B$ be a set of red and blue points; we use $c(p)$, $x(p)$ and $y(p)$ to denote the colour, the x-coordinate and the y-coordinate of point $p \in P$, respectively. An *orthogeodesic path (cycle)* is a drawing of a path (cycle) such that each edge is represented by a geodesic chain and edges intersect only at common endvertices. An *orthogeodesic alternating path (cycle)* on P is an orthogeodesic path (cycle) whose vertices are the points of P and each edge connects points of distinct colours. An orthogeodesic alternating path (cycle) is *Hamiltonian* if it contains all points of P. Clearly, for an orthogeodesic alternating path (cycle) to be Hamiltonian it is necessary that P is equitable (balanced). An *orthogeodesic alternating spanning tree* on a point-set $P = R \cup B$ is a spanning tree of P such that each edge is a geodesic chain connecting points of different colours and no two edges cross.

Given a point-set $P' \subseteq P$, the *bounding box of* P', denoted as $\mathcal{B}(P')$, is the smallest axis-parallel rectangle enclosing P'. Let p and q be two points such that $\mathcal{B}(\{p, q\})$ is a non-degenerate rectangle. A *horizontal chain* (*vertical chain*) is a two-bend geodesic chain such that the first and the last segment are horizontal (vertical). Notice that a horizontal chain (vertical chain) is uniquely determined when the x-coordinate (y-coordinate) of its vertical (horizontal) segment is specified.

A point-set $P = R \cup B$ is a *butterfly* if it has the following properties: (i) for every two blue points p and q of P, $x(p) < x(q)$ implies $y(p) < y(q)$; (ii) for every two red points p and q of P, $x(p) < x(q)$ implies $y(p) < y(q)$; (iii) for every pair consisting of a blue point p and a red point q of P, $x(p) > x(q)$ and $y(p) < y(q)$ (see Figure 2(a)). When printed in black and white, the darker dots in our figures represent blue points while the light gray ones represent red points.

3 Hamiltonian Orthogeodesic Alternating Paths

We describe now an algorithm to compute a Hamiltonian orthogeodesic alternating path on an equitable set of red and blue points such that no two points are horizontally or vertically aligned. We assume that points are not vertically or horizontally aligned in order to avoid straightforward counterexamples. Namely, it is easy to find point-sets that contain vertically or horizontally aligned points and that do not admit a Hamiltonian orthogeodesic alternating path. Consider, for example, a set of three points on the same vertical/horizontal line with two consecutive points of the same colour.

The algorithm is a recursive algorithm that incrementally constructs the path. Consider a generic call of the algorithm. When such a call is activated, a connected subpath

Π has already been constructed on a subset U of the input point-set P. Given a point p in U and a point q in the plane (not necessarily a point of P), we say that q is *left-connectible* to p if $y(p) = y(q)$ and $\overline{pq} \cap \Pi = \{p\}$. The input of the recursive call is a balanced point-set $P' \subseteq P \setminus U$ and a point q_l (not necessarily belonging to P) to the left of $\mathcal{B}(P')$. The point q_l is called the *enter point*. The point-set P' and the enter point q_l are such that the intersection between $\mathcal{B}(P' \cup \{q_l\})$ and the path Π (if any) is completely contained in the left side of $\mathcal{B}(P' \cup \{q_l\})$. Also, q_l is either an endvertex of Π or it is left-connectible to an endvertex of Π. The output of the recursive call is a Hamiltonian orthogeodesic alternating path Π' on $P' \cup \{q_l\}$ with the following properties: **(P1)** q_l is an endpoint of Π'; also, Π' is completely contained in $\mathcal{B}(P' \cup \{q_l\})$ and the intersection between Π' and the left side of $\mathcal{B}(P' \cup \{q_l\})$ is q_l; **(P2)** there exists a point q_r, called *exit point*, on the right side of $\mathcal{B}(P')$ such that $c(q_r) = c(q_l)$; also, q_r is either an endvertex of Π' or it is left-connectible to an endvertex of Π'. **(P3)** each geodesic chain in Π is drawn with at most two bends;

Before describing the algorithm it is worth clarifying what is the input of the main call of the algorithm. If P is balanced, we extend P with a point q_l to the left of $\mathcal{B}(P)$ arbitrarily coloured red or blue. This point will be the enter point of the first recursive call. If P is unbalanced, we add a point r to the left of $\mathcal{B}(P)$ in order to make it balanced; the colour of r is therefore red (recall that we are assuming $|R| \leq |B|$). Then we can proceed as in the previous case, i.e., we add an enter point q_l to the left of r arbitrarily coloured red or blue. The input is then q_l and $P \cup \{r\}$. Clearly, at the end of the algorithm the added points and the chains incident to them will be removed.

We are now ready to describe our recursive algorithm. Without loss of generality we assume that $c(q_l)$ is blue. If $|P'| = 2$ then we compute a drawing as follows (see Figure 1(a)). Let p' be the point of the left side of $\mathcal{B}(P')$ that has the same y-coordinate of the red point p of P' (p and p' may coincide). Connect point p' to p with zero bends and to q_l with a horizontal chain whose vertical segment has x-coordinate $(x(q_l) + x(p'))/2$. Also, connect p to the blue point r of P' with a vertical chain whose horizontal segment has y-coordinate $(y(p) + y(r))/2$.

Assume now that $|P'| > 2$ and let p_t, p_b, p_l, and p_r be the points on the top, bottom, left, and right side of $\mathcal{B}(P')$, respectively. Notice that some of these points may coincide. We distinguish the following cases, depending on the colour of p_l:
Case 1: $c(p_l)$ **is blue.** We further distinguish the following sub-cases: **Case 1.a:** $c(p_t)$ **is red.** Notice that, in this case $p_t \neq p_l$. Let p'_t be the top-left corner of $\mathcal{B}(P')$. Connect point p'_t to p_t with zero bends and to q_l with a horizontal chain whose vertical segment has x-coordinate $(x(q_l) + x(p'_t))/2$. Let r be the point of P' immediately below p_t. Connect p_t and p_l with a vertical chain whose horizontal segment has y-coordinate $(y(p_t) + y(r))/2$. Recursively apply the algorithm with input the point p_l and the set $P' \setminus \{p_l, p_t\}$. See Figure 1(b) for an illustration. **Case 1.b:** $c(p_b)$ **is red.** Symmetric to Case 1.a. The drawing is obtained from Case 1.a by a vertical reflection. **Case 1.c:** $c(p_t) = c(p_b)$ **is blue.** There are two sub-cases: **Case 1.c.1:** $c(p_r)$ **is red.** First recursively apply the algorithm with input the point q_l and the set $P' \setminus \{p_t, p_r\}$. Let Π' be the sub-path computed by the recursive call and let q'_r be its exit point. Connect q'_r to its corresponding extremal point with a horizontal segment if necessary and connect p_r to q'_r with a horizontal chain whose vertical segment has x-coordinate $(x(p_r) + x(q'_r))/2$.

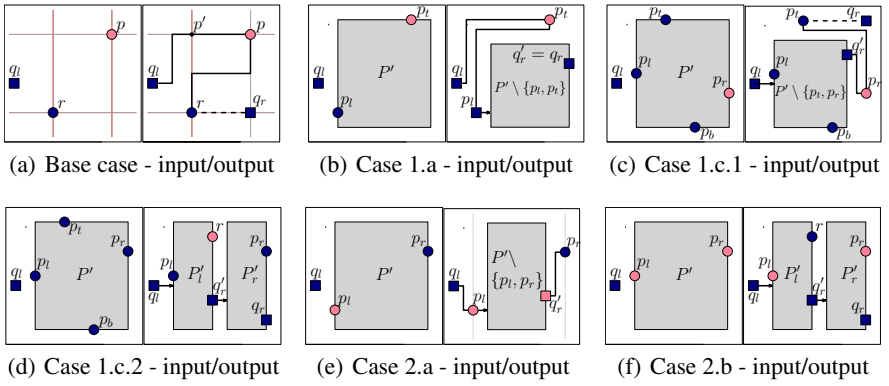


Fig. 1. The different cases of the algorithm. Case 1.b is a vertical reflection of Case 1.a. The squares in the pictures represents the enter points and the exit points of the different recursive calls.

Let r be the point of P' immediately below p_t. Connect p_r and p_t with a vertical chain whose horizontal segment has y-coordinate $(y(p_t) + y(r))/2$. See Figure 1(c) for an illustration. **Case 1.c.2:** $c(p_r)$ **is blue.** Let r be the leftmost red vertex such that the set $P'_l = \{p \mid p \in P' \ \wedge \ x(p) \leq x(r)\}$ is balanced and let $P'_r = P' \setminus P'_l$. Since point r cannot coincide with p_r, both P'_l and P'_r are not empty. Also, P'_r is balanced because so are P' and P'_l. Recursively apply the algorithm with input the point q_l and the set P'_l. Let Π' be the sub-path computed by the recursive call and let q'_r be its exit point. Recursively apply the algorithm with input the point q'_r and the set P'_r. See Figure 1(d) for an illustration.

Case 2: $c(p_l)$ **is red.** Based on the colour of p_r, we distinguish the following sub-cases: **Case 2.a:** $c(p_r)$ **is blue.** Connect q_l to p_l with a horizontal chain whose vertical segment has x-coordinate $(x(q_l)+x(p_l))/2$. Recursively apply the algorithm with input the point p_l and set $P' \setminus \{p_l, p_r\}$. Let Π' be the sub-path computed by the recursive call and let q'_r be its exit point. Connect q'_r to its corresponding extremal point with a horizontal segment if necessary and connect p_r to q'_r with a horizontal chain whose vertical segment has x-coordinate $(x(p_r)+x(q'_r))/2$. See Figure 1(e) for an illustration. **Case 2.b:** $c(p_r)$ **is red.** Let r be the leftmost blue vertex such that the set $P'_l = \{p \mid p \in P' \ \wedge \ x(p) \leq x(r)\}$ is balanced and let $P'_r = P' \setminus P'_l$. Since point r cannot coincide with p_r, both P'_l and P'_r are not empty. Also, P'_r is balanced because so are P' and P'_l. Recursively apply the algorithm with input the point q_l and the set P'_l. Let Π' be the sub-path computed by the recursive call and let q'_r be its exit point. Recursively apply the algorithm with input the point q'_r and the set P'_r. See Figure 1(f) for an illustration.

Theorem 1. *Every equitable set of n red and blue points such that no two points are horizontally or vertically aligned admits a Hamiltonian orthogeodesic alternating path. Also, there exists an $O(n \log^2 n)$-time algorithm to compute such a path. Furthermore, the computed path has at most two bends per edge, which is worst-case optimal. Finally, if the input point-sets consists of grid points all bends are at half-integer grid points.*

Sketch of Proof: For reasons of space the proof that the algorithm described above correctly computes a Hamiltonian orthogeodesic alternating path on P is omitted. In this sketch of proof we only show that two bends per edge are worst-case optimal and that the algorithm can be implemented to run in $O(n \log^2 n)$ time, where $n := |P|$.

We start by proving that it is not always possible to obtain a Hamiltonian orthogeodesic alternating path such that every edge has at most one bend. Consider a butterfly P with at least four points and let Π be any Hamiltonian orthogeodesic alternating path on P. Let p be a blue point that is not an endpoint of Π (notice that at most one blue point is an endpoint of Π because $|R| = |B|$). Point p is connected to two red points q_1 and q_2 by two geodesic chains χ_1 and χ_2, respectively, as depicted in Figure 2(b). One of the two chains, say χ_1, must have a horizontal segment incident to p while the other chain, that is χ_2, must have a vertical segment incident to p. If we use 1-bend geodesic chains, then χ_1 has a vertical segment incident to q_1 while χ_2 has a horizontal segment incident to q_2. Since all red points are above and to the left of all the blue points, and since the two chains cannot cross, then $x(q_1) < x(q_2)$. At least one of q_1 and q_2 must be connected to a blue point p' distinct from p (recall that at most one red point is an endpoint of Π because $|R| = |B|$). If $x(p') < x(p)$ then the geodesic chain connecting q_1 or q_2 to p' would cross chain χ_1; If $x(p') > x(p)$ then the geodesic chain connecting q_1 or q_2 to p' would cross chain χ_2.

We now prove that the algorithm can be implemented to run in $O(n \log^2 n)$ time. We sort the points with respect to their x- and y-coordinates and maintain respective arrays, such that each point p can be addressed by two integers $h(p)$ and $v(p)$ denoting the index of p in the horizontal and vertical array, respectively. Further, we maintain two spatial data structures with $O(n \log n)$ initialization time and $O(\log n)$ query time for orthogonal range queries [4] for the blue and red points, respectively.

We assume that we are given the bounding box R of the instance P' in the form of at most four points p_l, p_r, p_t and p_b on the bounding box of the instance, each of which is specified by two integers pointing to the position of the points in the horizonal and vertical array, respectively. First we consider all cases, except for Case 1.c.2 and Case 2.b. In these cases we compute the geodesic chain from two points p_1 and p_2 on the boundary of $\mathcal{B}(P')$ and the sub-path computed for $P'' := P' \setminus \{p_1, p_2\}$. In order to recurse on P'' we need to compute the extremal points of P'', given an axis-aligned rectangle $R \supset P''$ with $R \cap \{p_1, p_2\} = \emptyset$ that can easily be obtained from the extremal points of P' and the horizontal and vertical arrays. Suppose that R is given by two intervals $[i, j]$ and $[k, l]$ where i, j, k and l are integers pointing to the horizontal and vertical arrays. First we determine the number of points m in $R \cap P$ using the spatial data structures. For each horizontal and vertical boundary of R that is not yet covered by a point, we perform a binary search on the horizontal and vertical arrays $[i, j]$ and $[k, l]$, respectively, in order to locate the extremal point in P'' along the axis orthogonal to the boundary line. In each iteration of the binary search we query the spatial data structures with the corresponding rectangle R to determine the number of points in R. If this number is equal to m and the boundary is defined by a point in P'', then we have found an extremal point. Since the number of steps is at most $\log n$ and each step can be performed in $O(\log n)$, we can find the extremal points in $O(\log^2 n)$ time.

Fig. 2. (a) A butterfly. (b) Illustration for the proof of Theorem 1. If point p is connected to q_1 and q_2 with two 1-bend geodesic chains, then q_1 and q_2 cannot be connected to any other blue point by a geodesic chain without introducing a crossing.

Next, we describe how to split the point sets as in Cases 1.c.2 and 2.b. In both cases, we would like to split the current instance vertically into two balanced subsets. Since both cases can be treated similarly, we only describe Case 1.c.2. In this case, we would like to split the instance vertically such that the rightmost point in the left sub-instance is red. Given an integer i, we can count the number of red and blue points in the rectangle R_i defined by the horizontal interval $[h(p_l), i]$ and the vertical interval $[v(p_b), v(p_t)]$ in time $O(\log n)$. Let $f(i)$ be defined as the number of blue points in R_i minus the number of red points in R_i. Clearly, $f(h(p_l)) = 1$ and $f(h(p_r) - 1) = -1$ since both $c(p_l)$ and $c(p_r)$ are blue and the instance is balanced. Since there is at most one point in each column, we have $|f(i) - f(i+1)| \leq 1$ for all i in the range. This implies that f must attain the value 0 in the interval (i, j) whenever $f(i) > 0$ and $f(j) < 0$ or vice versa. Using this observation, we can find an index i such that $f(i) = 0$ in the interval $(h(p_l), h(p_r))$ in $O(\log^2 n)$ using binary search. Whenever we encounter an index i such that $f(i) = 0$ and the rightmost point on the left sub-instance is a blue point, we continue with the binary search by considering the interval $(h(p_l), i - 1)$. Note that $f(i-1) = -1$ in this case. Having found an index i with the desired properties, we can find all points on the boundary of the bounding boxes of the two sub-instances in time $O(\log^2 n)$ similar to the cases described above.

Finally, note that each operation of the algorithm can be implemented to run in $O(\log^2 n)$ time and is executed at most n times. Hence, the running time of the algorithm is in $O(n \log^2 n)$. □

4 Hamiltonian Orthogeodesic Alternating Paths on the Grid

In the previous section, we have seen that any equitable set of red and blue points such that no two points are on a common horizontal or vertical line allows for a Hamiltonian orthogeodesic alternating path. A common requirement when computing orthogonal drawings is that the endpoints of each segment be represented by grid points, i.e., points with integer coordinates. A Hamiltonian orthogeodesic alternating path satisfying this requirement is said to be *on the grid*. Clearly, such a path can exist only if the points of P are grid points. The algorithm described above may not produce Hamiltonian orthogeodesic alternating paths on the grid even if the points of P are grid points. Namely, our algorithm sometimes needs to draw a horizontal chain connecting two points with

consecutive x-coordinates. In this case the bends of the chain have a x-coordinate that is half-way between the x-coordinates of the two points. Clearly if the two points have consecutive integer x-coordinate, the bends will have a non-integer x-coordinate. The same happens with vertical chains between points having consecutive y-coordinates.

One may wonder whether it is always possible to compute Hamiltonian orthogeodesic alternating paths on the grid. The following theorem shows that this is not possible.

Theorem 2. *For every $n \geq 5$, there exists an equitable set of red and blue grid points of size n that does not admit a Hamiltonian orthogeodesic alternating path on the grid.*

Motivated by Theorem 2, we study the HAMILTONIAN ORTHOGEODESIC ALTERNATING PATH ON THE GRID problem, i.e., the problem of deciding whether an equitable set of grid points such that no two points are horizontally or vertically aligned, admits a Hamiltonian orthogeodesic alternating path on the grid. We show that this problem is NP-complete. Similar techniques can be used to show that it is NP-complete to decide whether there is a Hamiltonian orthogeodesic alternating cycle. If we are allowed to place more than one point on a horizontal or vertical line, we can show that it is NP-complete to decide whether there exists an orthogeodesic alternating perfect matching. This contrasts a result by Kano [9] stating that such a matching always exists if we are not allowed to place more than one point per horizontal or vertical line.

Theorem 3. HAMILTONIAN ORTHOGEODESIC ALTERNATING PATH ON THE GRID *is NP-complete.*

Sketch of Proof: We show that the problem is in NP by showing that it suffices to guess a linear number of bends for each of the edges. For reasons of space we omit the proof. We show NP-hardness by reduction from 3-PARTITION.

Similar techniques have been used, for example, in [10]. An instance of 3-PARTITION consists of a multiset $A = \{a_1, \ldots, a_{3m}\}$ of $3m$ positive integers, each in the range $(B/4, B/2)$, where $B = (\sum_{i=1}^{3m} a_i)/m$, and the question is whether there exists a partition of A into m subsets $A_1, \ldots, A_m$ of A, each of cardinality 3, such that the sum of the numbers in each subset is B. Since 3-PARTITION is *strongly* NP-hard [6], we may assume that B is bounded by a polynomial in m.

Given an instance A of 3-PARTITION, we construct a corresponding instance $P = R \cup B$ of the HAMILTONIAN ORTHOGEODESIC ALTERNATING PATH ON THE GRID problem such that P allows for a Hamiltonian orthogeodesic alternating path if and only if there exists a partition of A with the desired properties as follows.

A sequence of diagonally aligned grid points is called *k-spaced* if the Euclidean distance between subsequent points is exactly $k\sqrt{2}$. The point-set P consists of four different types of points, called *hinge points*, *element points*, *mask points* and *partition points*, and is aligned on a regular sawtooth-pattern with $3m + 2$ teeth, numbered $T_0, \ldots, T_{3m+1}$ from left to right. The pointset, as well as the sawtooth-pattern and the teeth are illustrated in Figure 3.

Let L be some integer to be specified later. Each tooth T_i consists of a diagonal segment with slope 1 of length $L\sqrt{2}$, denoted by S_i, and a diagonal segment with slope -1 of length $(2L+1)\sqrt{2}$. Hence, the tips of the teeth are aligned along a line with negative slope such that the tip of T_i is below the lowest point of S_{i-1} for $1 \leq i \leq 3m + 1$.

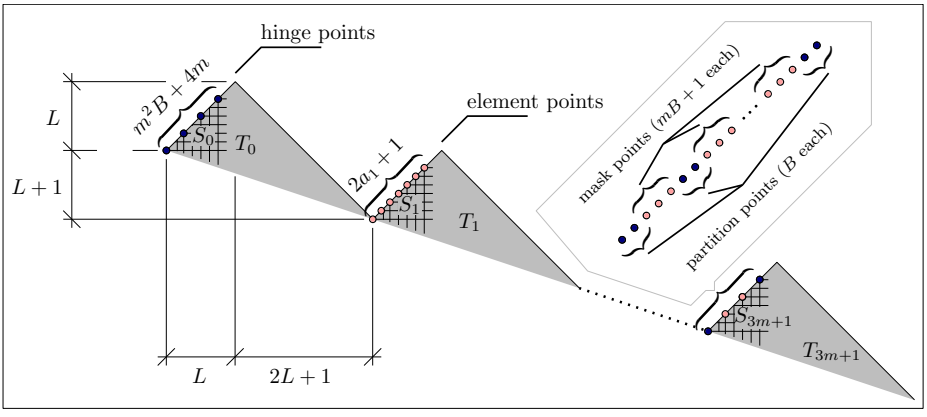


Fig. 3. Pointset used in the reduction. Each shaded triangle constitutes a tooth T_i. All points are arranged on the ascending slope S_i of T_i.

Along S_0, we align $m^2B + 4m$ 2-spaced blue hinge points starting at the leftmost point of S_0. For each element a_i we align $2a_i + 1$ 1-spaced red element points along S_i. Further, we align m sets of B 2-spaced blue partition points along S_{3m+1}, each acting as a partition. These partitions are separated by $m - 1$ sequences of $mB + 1$ 2-spaced red mask points which will act as a sort of "dot mask" separating the partitions. The maximal sequences of blue points along S_{3m+1} are called *partitions* and the maximal sequences of red points along S_{3m+1} are called *masks*. By construction P contains $m^2B + mB + 4m - 1$ red and $m^2B + mB + 4m$ blue points and, thus, is equitable with one more blue point. Hence, any alternating path must start and end with a blue point and all red points must be interior points. This implies that every red point must be connected to exactly two blue points.

We now show that there is a partition of A if and only if the point-set contains a Hamiltonian orthogeodesic alternating path. Assume that we are given such a path. See Figure 4 for a high-level illustration. In each mask there must be one mask point that is connected to a blue hinge point on S_0. To see this, note that there are $mB + 1$ red mask points in each of the $m - 1$ masks, but only mB blue points in total on S_{3m+1}. Hence, at least one of the red mask points in each mask must be connected to a blue hinge point. Each edge between a mask point and a hinge point is called a *partitioner*.

Since the element points corresponding to a single element are 1-spaced, no partitioner can pass between them. Hence, the partitioners will partition the element points according to the element sizes, such that all element points corresponding to a single element are contained in the same partition.

Next, consider the element points. Let D_i be the diagonal line through S_i and let H_i^+ and H_i^- denote the upper and lower halfplanes defined by D_i, respectively. We claim that each group of $2a_i + 1$ element points corresponding to element a_i can have at most $2a_i + 2$ blue incidences in H_i^+. Each of these incidences is a geodesic chain starting either with a horizontal segment to the left or with a vertical segment towards the top.

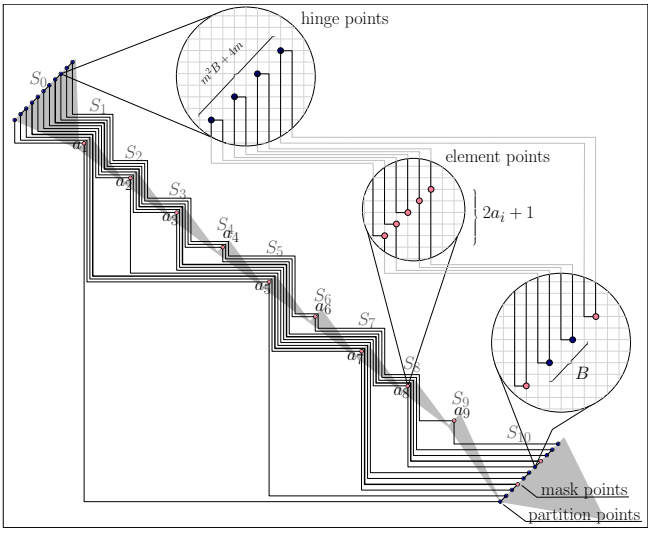


Fig. 4. A high-level illustration of an examplary reduction from 3-PARTITION to HAMILTONIAN ORTHOGEODESIC ALTERNATING PATH ON THE GRID using the instance
$A_1 = \{a_1, a_5, a_7\}$,
$A_2 = \{a_2, a_3, a_8\}$,
$A_3 = \{a_4, a_6, a_9\}$
(not to scale). Details are depicted in the circles.

These segments can be covered by gridpoints adjacent to the element points. As there are only $2a_i + 2$ such gridpoints, the claim holds.

Since the group of element points corresponding to a_i must have $4a_i + 2$ incidences in total, it must therefore have at least $2a_i$ blue incidences in H_i^-. Hence all element points must have $2mB$ blue incidences on S_{3m+1} in total. On the other hand, there are only mB blue points on S_{3m+1}, each of which must have two red incidences. This implies that element a_i has exactly $2a_i$ incidences in H_i^- and that the blue partition points are connected only to the element points. Since there are B blue points in each of the partitions, the number of element points must add up to $2B$, i.e., the corresponding elements add up to B and thus yield a valid 3-partition of A.

Conversely, suppose that we are given a valid partition of A. Then we can find a Hamiltonian orthogeodesic alternating path as follows. Each geodesic chain is drawn as the bottommost geodesic that runs across all geodesics drawn so far as illustrated in Figure 4. We start with the leftmost element a_i that is inserted in the first partition. We draw an alternating path starting at the leftmost hinge point using the first a_i partition points, the first $2a_i$ element-points corresponding to a_i as well as the leftmost hinge points on S_0. After that, we connect the last element point only to the two next hinge points. Then we proceed in the same manner with the second and third elements that are inserted in the first partition. After that, we draw the partition starting at the current hinge point and going back and forth between the hinge points and mask points until we have connected all mask points of the first mask. The rest of the path is connected in a

similar fashion. Since we are given a valid partition, the drawing will be a Hamiltonian path and by choosing L large enough, e.g., $L = |P| + 1$, we can make sure that there is enough space between the element points for all geodesic chains, so that the resulting drawing is planar. □

The following corollary can be obtained by similar techniques.

Corollary 1. *It is NP-complete to decide whether a given balanced set of red and blue grid points such that no two points are on a common horizontal or vertical line allows for a Hamiltonian orthogeodesic alternating cycle if bends are only allowed at grid points.*

Due to Kano [9] every balanced set of red and blue points such that no two points are on a common horizontal or vertical line contains a perfect orthogeodesic alternating matching consisting of L-shaped orthogonal chains. Hence such a matching is completely on the grid whenever the points are grid points. If we are given an arbitrary balanced set of red and blue grid points allowing points to be aligned on a common horizontal or vertical line, the problem becomes NP-complete. The proof is similar to the proof of Theorem 3 and can be found in the appendix.

Theorem 4. *Given an arbitrary balanced set of red and blue grid points, it is NP-complete to decide whether there is a perfect orthogeodesic alternating matching on the grid.*

5 Additional Results

In this section we consider some questions naturally related with Theorems 1 and 3. First, we investigate whether the result of Theorem 1 can be extended to other families of graphs, such as cycles and trees having vertex degree larger than two. The following two theorems present counterexamples using equitable point-sets that are butterflies.

Theorem 5. *For every even $n \geq 4$, there exists a balanced set of red and blue points of size n that does not admit a Hamiltonian orthogeodesic alternating cycle.*

Theorem 6. *For every $n \geq 4$, there exists a set of n red and blue points such that any orthogeodesic alternating spanning tree has maximum vertex degree 2.*

Motivated by Theorem 3, we consider the following optimization problem: Given an equitable set of red and blue grid points P such that no two points are on a common horizontal or vertical line, we wish to find a subset $P' \subseteq P$ of maximum size such that there is a Hamiltonian orthogeodesic alternating path for P' on the grid. The following theorems show lower and upper bounds on the maximum size of P'.

Theorem 7. *Let P be an equitable set of grid points. There is an $O(n \log^2 n)$-time algorithm that computes an equitable set $P' \subseteq P$ with $|P'| \geq (|P| + 2)/3$ that admits a Hamiltonian orthogeodesic alternating path on the grid.*

Theorem 8. *The maximum number of points on any orthogeodesic alternating path on the grid for the butterfly with $2n$ points is at most $n + 1$.*

6 Open Problems

The results of this paper suggest some problems that can be further investigated:

- Based on the results of Theorems 2, 5, and 6, it may be worth characterizing those point-sets that admit an alternating path on the grid, an alternating cycle, and an alternating spanning tree, respectively.
- Also, it would be interesting to close the gap between Theorem 7 and Theorem 8.
- Finally, the concept of Hamiltonian orthogeodesic alternating path can be extended to the case that the point-set P is partitioned into k colour classes. In this case a point of one colour can be connected to any point of one of the other $k-1$ colours. Is it always possible to compute a Hamiltonian orthogeodesic alternating path in this multi-coloured setting? We recall here that the (straight-line) alternating paths on multi-coloured point-sets have been studied by Merino, Salazar, and Urrutia [12].

References

1. Abellanas, M., García, A., Hurtado, F., Tejel, J.: Caminos alternantes (in Spanish). In: X Encuentros de Geometría Computacional, Sevilla, Spanish, pp. 7–12 (2003)
2. Abellanas, M., Garcia-Lopez, J., Hernández-Peñalver, G., Noy, M., Ramos, P.A.: Bipartite embeddings of trees in the plane. Discrete Applied Mathematics 93(2-3), 141–148 (1999)
3. Akiyama, J., Urrutia, J.: Simple alternating path problem. Discr. Math. 84(1), 101–103 (1990)
4. Chazelle, B.: Functional approach to data structures and its use in multidimensional searching. SIAM J. Comput. 17, 427–462 (1988)
5. Cibulka, J., Kynčl, J., Mészáros, V., Stolař, R., Valtr, P.: Hamiltonian Alternating Paths on Bicolored Double-Chains. In: Tollis, I.G., Patrignani, M. (eds.) GD 2008. LNCS, vol. 5417, pp. 181–192. Springer, Heidelberg (2009)
6. Garey, M.R., Johnson, D.S.: Computers and Intractability. A Guide to the Theory of NP-Completeness. W.H. Freeman and Company (1979)
7. Kaneko, A., Kano, M., Suzuki, K.: Path coverings of two sets of points in the plane. In: Pach, J. (ed.) Towards a Theory of Geometric Graphs, vol. 342, pp. 99–111. American Mathematical Society (2004)
8. Kaneko, A., Kano, M., Yoshimoto, K.: Alternating Hamilton cycles with minimum number of crossings in the plane. Int. J. Comput. Geometry Appl. 10(1), 73–78 (2000)
9. Kano, M.: Discrete geometry on red and blue points on the plane lattice. In: Proc. of JCCGG 2009, pp. 30–33 (2009)
10. Katz, B., Krug, M., Rutter, I., Wolff, A.: Manhattan-Geodesic Embedding of Planar Graphs. In: Eppstein, D., Gansner, E.R. (eds.) GD 2009. LNCS, vol. 5849, pp. 207–218. Springer, Heidelberg (2010)
11. Kynčl, J., Pach, J., Tóth, G.: Long alternating paths in bicolored point sets. Discrete Mathematics 308(19), 4315–4321 (2008)
12. Merino, C., Salazar, G., Urrutia, J.: On the length of longest alternating paths for multicoloured point sets in convex position. Discrete Mathematics 306(15), 1791–1797 (2006)

Ranking and Loopless Generation of k-ary Dyck Words in Cool-lex Order

Stephane Durocher[1], Pak Ching Li[1], Debajyoti Mondal[1], and Aaron Williams[2]

[1] Department of Computer Science, University of Manitoba
[2] Department of Mathematics and Statistics, Carleton University
{durocher,lipakc,jyoti}@cs.umanitoba.ca, haron@uvic.ca

Abstract. A binary string B of length $n = kt$ is a k-ary Dyck word if it contains t copies of 1, and the number of 0s in every prefix of B is at most $k-1$ times the number of 1s. We provide two loopless algorithms for generating k-ary Dyck words in cool-lex order: (1) The first requires two index variables and assumes k is a constant; (2) The second requires t index variables and works for any k. We also efficiently rank k-ary Dyck words in cool-lex order. Our results generalize the "coolCat" algorithm by Ruskey and Williams (*Generating balanced parentheses and binary trees by prefix shifts* in CATS 2008) and provide the first loopless and ranking applications of the general cool-lex Gray code by Ruskey, Sawada, and Williams (*Binary bubble languages and cool-lex order* under review).

1 Background

1.1 k-ary Dyck Words

Let $\mathbb{B}(n,t)$ be the set of binary strings of length n containing t copies of 1. A string $B \in \mathbb{B}(kt,t)$ is a *k-ary Dyck word* if the number of 0s in each prefix is at most $k-1$ times the number of 1s. Let $\mathbb{D}_k(t)$ be the set of k-ary Dyck words of length kt. For example, the k-ary Dyck words with $k = t = 3$ are given below

$$\mathbb{D}_3(3) = \{111000000, 110100000, 101100000, 110010000, 101010000, 100110000, 110001000, 101001000, 100101000, 110000100, 101000100, 100100100\}.$$

The k-ary Dyck words of length kt have simple bijections with a number of combinatorial objects including k-ary trees with t internal nodes [2,3]. The 2-ary Dyck words are known as *balanced parentheses* when 1 and 0 are replaced by '(' and ')' respectively, and the cardinality of $\mathbb{D}_2(t)$ is the tth Catalan number.

A simple property of k-ary Dyck words is that they can be "separated" according to the following remark. We let $\alpha\beta$ denote the concatenation of the binary strings α and β, and we say that α and β have the same *content* if they have equal length and an equal number of 1s.

Remark 1. If $\alpha\beta, \gamma\delta \in \mathbb{D}_k(t)$ and α and γ have the same content, then $\alpha\delta, \beta\gamma \in \mathbb{D}_k(t)$. In other words, prefixes (or suffixes) of k-ary Dyck words with the same content can be separated and recombined.

C.S. Iliopoulos and W.F. Smyth (Eds.): IWOCA 2011, LNCS 7056, pp. 182–194, 2011.
© Springer-Verlag Berlin Heidelberg 2011

1.2 Combinatorial Generation

Many computational problems require iterating through combinatorial objects of a given type and size without duplication. Generation algorithms store one object in a data structure, and create successive objects by modifying its contents. *Constant-amortized time (CAT)* and *loopless* algorithms create successive objects in amortized O(1)-time and worst-case O(1)-time, respectively. Memory for input parameters and the aforementioned data structure are fixed expenses, and the algorithm's remaining variables are *additional variables. Index variables* have values in $\{1, 2, \ldots, n\}$ when generating combinatorial objects of size $O(n)$.

Successive objects created by loopless algorithms differ by a constant amount (in the chosen data structure) and the resulting order of objects is a *Gray code*. If successive objects differ by operation 'x', then the order is an *'x' Gray code*; in a *2-'x' Gray code* successive objects differ by at most two applications of 'x'. In a *cyclic* Gray code the last object differs from the first object in same way.

Suppose $B = B_1 B_2 \cdots B_n$ is a binary string of length n and $1 \leq i \leq j \leq n$. Informally, $\mathsf{swap}(B, i, j)$ exchanges the ith and jth bits of B, and $\mathsf{shift}(B, j, i)$ moves the jth bit of B leftwards into the ith position by moving the intermediate bits to the right. Formally, the *swap* and *shift* operations are defined as follows:

- $\mathsf{swap}(B, i, j) = B_1 \cdots B_{i-1} B_j B_{i+1} \cdots B_{j-1} B_i B_{j+1} \cdots B_n$, and
- $\mathsf{shift}(B, j, i) = B_1 \cdots B_{i-1} B_j B_i B_{i+1} \cdots B_{j-1} B_{j+1} \cdots B_n$.

When appropriate we shorten $\mathsf{swap}(B, i, j)$ to $\mathsf{swap}(i, j)$, and $\mathsf{shift}(B, j, i)$ to $\mathsf{shift}(j, i)$. Swaps are also known as *transpositions* with special cases including

- *adjacent-transpositions*: $\mathsf{swap}(i, i{+}1)$,
- *two-close-transpositions*: $\mathsf{swap}(i, i{+}1)$ or $\mathsf{swap}(i, i{+}2)$, and
- *homogeneous-transpositions*: $\mathsf{swap}(B, i, j)$ where $B_i = B_{i+1} = \cdots = B_{j-1}$.

Prefix-shifts are usually defined as operations of the form $\mathsf{shift}(j, 1)$. Swaps and prefix-shifts are efficient operations for binary strings stored in arrays and computer words, respectively.

Given an order of combinatorial objects, the *rank* of an object is its position in the order. *Ranking* determines the rank of a particular object in a given order, and *unranking* determines the object with a particular rank in a given order.

1.3 CoolCat Order

Balanced parentheses are among the most studied objects in combinatorial generation [3] but fewer results exist for k-ary Dyck words. Generation of $\mathbb{D}_k(t)$ was first discussed by Zaks [10]. A general result by Pruesse and Ruskey implies that $\mathbb{D}_k(t)$ has a 2-adjacent-transposition Gray code [4] and a result by Canfield and Williamson [1] proves that $\mathbb{D}_k(t)$ can be generated by a loopless algorithm[1]. More recently, Vajnovszki and Walsh [9] found a two-close transposition Gray

[1] Both results use that strings in $\mathbb{D}_k(t)$ correspond to linear-extensions of a poset with cover relations $a_1 \prec \cdots \prec a_t$, $b_1 \prec \cdots \prec b_{(k-1)t}$, and $a_i \prec b_{(k-1)(i-1)+1}$ for $1 \leq i \leq t$.

code and created a loopless algorithm that requires twelve if-statements and $O(n)$ additional variables stored in three additional arrays e, s, and p. Results on k-ary trees date back to Ruskey [5] and Trojanowski [8].

There are no prefix-shift Gray codes for $\mathbb{D}_k(t)$ (except when $k, t \leq 2$). However, the first bit of every k-ary Dyck word is 1, so we can instead define a *prefix-shift* as $\mathsf{shift}(i, 2)$ with the understanding that the redundant bit could be omitted from a computer word representation. Using this definition Ruskey and Williams [7] discovered an ordering of $\mathbb{D}_2(t)$ with the following properties:

- it is both a cyclic prefix-shift Gray code, and a cyclic 2-swap Gray code that uses at most one adjacent-transposition and one homogeneous -transposition,
- it can be generated by a loopless algorithm using only two if-statements and two additional index variables, and
- the ordering has an efficient ranking algorithm.

Furthermore, the Gray code can be created by the "successor rules" in Table 1. More specifically, every string in $\mathbb{D}_2(t)$ has a prefix that matches a unique rule in (1a)-(1d) which describes how the prefix is changed to obtain the next string. Table 1 uses exponentiation for symbol repetition, and the order for $\mathbb{D}_2(4)$ is:

$$10111000, 11011000, 11101000, 10110100, 11010100, 10101100, 11001100,$$
$$11100100, 10110010, 11010010, 10101010, 11001010, 11100010, 11110000.$$

For example, the matched prefix for 11001100 is 1^i0^j11 with $i = 2$ and $j = 2$. By (1a), $\mathsf{shift}(i{+}j{+}1, 2)$ (or $\mathsf{swap}(i{+}1, i{+}j{+}1)$) creates the next string 11100100. Similarly, the matched prefix for 11100100 is 1^i0^j10 with $i = 3$ and $j = 2$. By (1c), $\mathsf{shift}(i{+}j{+}2, 2)$ (or $\mathsf{swap}(2, i{+}1)$ $\mathsf{swap}(i{+}j{+}1, i{+}j{+}2)$) creates 10110010.

Table 1. Rules for generating balanced parentheses $\mathbb{D}_2(t)$ from [7]. Prefixes change according to (1a)-(1d) by the specified shift or the equivalent swap(s). $^\dagger j > 0$.

	Current Prefix†	Next Prefix	Shift	Swap(s)
(1a)	1^i0^j11	$1^{i+1}0^j1$	$(i{+}j{+}1, 2)$	$(i{+}1, i{+}j{+}1)$
(1b)	1^i0^j10 for $i = j$	$1^{i+1}0^{j+1}$	$(i{+}j{+}1, 2)$	$(i{+}1, i{+}j{+}1)$
(1c)	1^i0^j10 for $i > j$	$101^{i-1}0^j1$	$(i{+}j{+}2, 2)$	$(2, i{+}1)$ $(i{+}j{+}1, i{+}j{+}2)$
(1d)	1^i0^j for $i = j = t$	$101^{i-1}0^{j-1}$	$(i{+}j, 2)$	$(2, i{+}1)$

Rules (1a) and (1b) can be combined (see [7]) since they perform the same operation and Rule (1d) simply transforms the 'last' string in the cyclic Gray code into the 'first' string. The Gray code is also interesting because it generates $\mathbb{D}_k(t)$ according to a cyclic Gray code for $\mathbb{B}(kt, t)$ known as *cool-lex order*. That is, if $\alpha \in \mathbb{D}_k(t)$ comes before $\beta \in \mathbb{D}_k(t)$ in the cool-lex order of $\mathbb{B}(kt, t)$, then α comes before β in the Gray code defined by Table 1. The order and algorithm are named "CoolCat" after cool-lex order and the Catalan numbers.

Theorem 1 ([7]). *The balanced parentheses of length* $2t$ *in* $\mathbb{D}_2(t)$ *are generated in cool-lex order by the prefix-shift (or equivalent swap(s)) in Table 1.*

1.4 Bubble Languages and Cool-lex Order

A *bubble language*[2] is a set of binary strings $\mathbb{L} \subseteq \mathbb{B}(n,t)$ with the following property: If $B \in \mathcal{L}$ where $B = 1^i0^j01\gamma$ for some $j \geq 0$, then $1^i0^j10 \in \mathbb{L}$. In other words, they are sets of binary strings with the same content in which the leftmost 01 of any string can be replaced by 10 to give another string in the set. This definition comes from Ruskey, Sawada, and Williams who showed that bubble languages generalize many combinatorial objects including binary necklaces and solutions to knapsack problems [6]. They substantially generalized Theorem 1 by proving that cool-lex order provides a cyclic Gray code for any bubble language. In particular, the successor rules in Table 2 generate all of these Gray codes.

Lemma 1 ([6]). *The k-ary Dyck words in $\mathbb{D}_k(t)$ are a bubble language. Furthermore, the k-ary Dyck prefixes in $\mathbb{D}_k(t,s)$ (see Section 4) are a bubble language.*

Proof. Replacing 01 by 10 cannot decrease the number of 1s in a string's prefix. □

Table 2. Rules for generating a bubble language $\mathbb{L}$ from [6]. Strings change according to (2a)-(2e) by the specified shift or equivalent swap(s). $^\dagger j > 0$. $^\ddagger h$ is the minimum value such that $1^h01^{i-h}0^j1\gamma \in \mathbb{L}$ and g is the minimum value such that $1^g01^{i-g}0^{j-1} \in \mathbb{L}$.

	Current String†	Next String‡	Shift	Swap(s)
(2a)	$1^i0^j11\gamma$	$1^{i+1}0^j1\gamma$	$(i{+}j{+}1, 1)$	$(i{+}1, i{+}j{+}1)$
(2b)	$1^i0^j10\gamma$ for $1^i0^{j+1}1\gamma \notin \mathbb{L}$	$1^{i+1}0^{i+1}\gamma$	$(i{+}j{+}1, 1)$	$(i{+}1, i{+}j{+}1)$
(2c)	$1^i0^j10\gamma$ for $1^i0^{j+1}1\gamma \in \mathbb{L}$	$1^h01^{i-h}0^j1\gamma$	$(i{+}j{+}2, h{+}1)$	$(h{+}1, i{+}1)$ $(i{+}j{+}1, i{+}j{+}2)$
(2d)	1^i0^j	$1^g01^{i-g}0^{j-1}$	$(i{+}j, g{+}1)$	$(g{+}1, i{+}1)$
(2e)	1^i0^j1	$1^{i+1}0^j$	$(i{+}j{+}1, 1)$	$(i{+}1, i{+}j{+}1)$

Theorem 2 ([6]). *The strings in any bubble language are generated in cool-lex order by the shift (or equivalent swap(s)) in Table 2.*

We will examine how this result applies to k-ary Dyck words later in this article. In the meantime, observe that the rules in Table 2 refer to entire strings, and not just specific prefixes as in Table 1. This is due to the fact that bubble languages do not necessarily have the separability property mentioned in Remark 1. Also note that Table 2 produces a shift Gray code that is not necessarily a prefix-shift Gray code. On the other hand, the Gray code is still a 2-swap Gray code using at most one adjacent-transposition and one homogeneous-transposition.

1.5 New Results

We apply Theorem 2 to obtain a simple set of successor rules that generate a cyclic prefix-shift Gray code of k-ary Dyck words in Section 2. Then we use the Gray code as the basis for two loopless generation algorithms that store the

[2] These are called "binary fixed-density bubble languages" in [6].

current string in an array in Section 3. The first algorithm works for constant k and requires only two additional index variables. The second algorithm works for arbitrary k and requires four if-statements and one array of $O(n)$ additional index variables. In Section 4 we show how the Gray code can be efficiently ranked and unranked. With respect to the existing literature these results include

- the first prefix-shift Gray code for k-ary Dyck words [6],
- the first loopless algorithm for generating k-ary Dyck words that uses O(1) additional index variables (when k is constant),
- a simpler loopless algorithm for generating k-ary Dyck words using 1/3 the if-statements and additional arrays as [9] (when k is arbitrary), and
- the first order of k-ary Dyck words that has a loopless generation algorithm as well as efficient ranking and unranking algorithms.

Our results also include the first application of bubble languages to loopless generation and efficient ranking and unranking. Due to the generalization from "CoolCat" to k-ary Dyck words, we name the order and algorithms "CoolKat".

2 CoolKat Order

In this section we specialize the cool-lex Gray code for bubble languages to the special case of k-ary Dyck words of length kt. In particular, Theorem 3 will prove that k-ary Dyck words can be generated cyclically using the rules in Table 3. The resulting "CoolKat" order appears below for $\mathbb{D}_3(3)$

$$101100000, 110100000, 101010000, 100110000, 110010000, 101001000,$$
$$100101000, 110001000, 101000100, 100100100, 110000100, 111000000.$$

As in Table 1 for balanced parentheses, the rules in Table 3 refer to string prefixes and the stated shifts are prefix-shifts. Also, the rule (3d) refers only to the 'last' string $1^t0^{(k-1)t}$. In the second half of this section we optimize the swap rules in Table 3 for the array-based loopless algorithms in Section 3.

Table 3. New rules for generating k-ary Dyck words $\mathbb{D}_k(t)$ in cool-lex order. These rules generalize those in Table 1 and specialize those in Table 2. $^\dagger j > 0$.

	Current Prefix†	Next Prefix	Shift	Swap(s)
(3a)	1^i0^j11	$1^{i+1}0^j1$	$(i{+}j{+}1, 2)$	$(i{+}1, i{+}j{+}1)$
(3b)	1^i0^j10 for $(k{-}1)i = j$	$1^{i+1}0^{j+1}$	$(i{+}j{+}1, 2)$	$(i{+}1, i{+}j{+}1)$
(3c)	1^i0^j10 for $(k{-}1)i > j$	$101^{i-1}0^j1$	$(i{+}j{+}2, 2)$	$(2, i{+}1)$ $(i{+}j{+}1, i{+}j{+}2)$
(3d)	1^i0^j for $i = t$, $j = (k-1)t$	$101^{i-1}0^{j-1}$	$(i{+}j, 2)$	$(2, i{+}1)$

Theorem 3. *The k-ary Dyck words of length kt are generated in cool-lex order by the rules in Table 3.*

Proof. Since $\mathbb{L} = \mathbb{D}_k(t)$ is a bubble language [6], Theorem 2 implies that its strings are generated by Table 2. We now compare each rule in Table 2 to its proposed specialization in Table 3. In the comparison, recall that (2a)-(2e) refer to entire strings, whereas (3a)-(3d) refer to prefixes, and that $j > 0$ is always assumed in 1^i0^j1.

	Current	Next	Shift	Swap
(2a)	$1^i0^j11\gamma$	$1^{i+1}0^j1\gamma$	$(i{+}j{+}1, 1)$	$(i{+}1, i{+}j{+}1)$
(3a)	1^i0^j11	$1^{i+1}0^j1$	$(i{+}j{+}1, 2)$	$(i{+}1, i{+}j{+}1)$

If a k-ary Dyck word has prefix 1^i0^j11 and $j > 0$, then it must be that $i > 0$. Therefore, $\mathsf{shift}(i{+}j{+}1, 2)$ in (3a) is the special case of $\mathsf{shift}(i{+}j{+}1, 1)$ in (2a).

	Current	Next	Shift	Swaps
(2b)	$1^i0^j10\gamma$ for $1^i0^{j+1}1\gamma \notin \mathbb{L}$	$1^{i+1}0^{i+1}\gamma$	$(i{+}j{+}1, 1)$	$(i{+}1, i{+}j{+}1)$
(3b)	1^i0^j10 for $(k{-}1)i = j$	$1^{i+1}0^{j+1}$	$(i{+}j{+}1, 2)$	$(i{+}1, i{+}j{+}1)$

Suppose $1^i0^j10\gamma$ is a k-ary Dyck word. Remark 1 implies that $1^i0^{j+1}1\gamma$ is not k-ary Dyck word if and only if $(k{-}1)i = j$. Therefore, the condition "for $(k{-}1)i = j$" in (3b) is a special case of the condition "for $1^i0^{j+1}1\gamma \notin \mathbb{L}$" in (2b). Next observe that $i > 0$ since k-ary Dyck words must begin with the symbol 1. Therefore, $\mathsf{shift}(i{+}j{+}1, 2)$ in (3b) is the special case of $\mathsf{shift}(i{+}j{+}1, 1)$ in (2b).

	Current	Next[‡]	Shift	Swaps
(2c)	$1^i0^j10\gamma$ for $1^i0^{j+1}1\gamma \in \mathbb{L}$	$1^h01^{i-h}0^j1\gamma$	$(i{+}j{+}2, h{+}1)$	$(i{+}j{+}1, i{+}j{+}2)$ $(h{+}1, i{+}1)$
(3c)	1^i0^j10 for $(k{-}1)i > j$	$101^{i-1}0^j1$	$(i{+}j{+}2, 2)$	$(i{+}j{+}1, i{+}j{+}2)$ $(2, i{+}1)$

[‡] h is the minimum value such that $1^h01^{i-h}0^j1\gamma \in \mathbb{L}$.

Suppose $1^i0^j10\gamma$ is a k-ary Dyck word. Remark 1 implies that $1^i0^{j+1}1\gamma$ is a k-ary Dyck word if and only if $(k{-}1)i > j$. Therefore, the condition "for $(k{-}1)i > j$" in (3c) is a special case of the condition "for $1^i0^{j+1}1\gamma \in \mathbb{L}$" in (2c). Next observe that Remark 1 implies that $h = 1$ is the minimum value such that $1^h01^{i-h}0^j1\gamma \in \mathbb{L}$. Therefore, the shifts and swaps in (3c) are special cases of those in (2c).

	Current	Next[‡]	Shift	Swaps
(2d)	1^i0^j	$1^g01^{i-g}0^{j-1}$	$(i{+}j, g{+}1)$	$(g{+}1, i{+}1)$
(3d)	1^i0^j for $i = t$, $j = (k-1)t$	$101^{i-1}0^{j-1}$	$(i{+}j, 2)$	$(2, i{+}1)$

[‡] g is the minimum value such that $1^g01^{i-g}0^{j-1} \in \mathbb{L}$.

By similar reasoning as above, $g = 1$ is the minimum value such that $1^g01^{i-g}0^{j-1}$ is a k-ary Dyck word.

	Current	Next	Shift	Swaps
(2e)	1^i0^j1	$1^{i+1}0^j$	$(i{+}j{+}1, 1)$	$(i{+}1, i{+}j{+}1)$

This general rule for bubble languages does not apply to k-ary Dyck words because k-ary Dyck words cannot have 1 as the last symbol. □

2.1 Optimized Swap Rules

Table 4 gives swap rules that are equivalent to those in Table 3. In these rules, $\mathsf{swap}(i{+}1, i{+}j{+}1)$ is performed when creating the successor of every string (except $1^t0^{(k-1)t}$). This allows more compact array-based algorithms in Section 3.

Table 4. Equivalent swap rules for generating k-ary Dyck words. These swap rules differ slightly from those in Table 3 and allow for a more efficient algorithm.$^\dagger j > 0$.

	Current Prefix†	Next Prefix	Swap(s)
(4a)	1^i0^j11	$1^{i+1}0^j1$	$(i{+}1, i{+}j{+}1)$
(4b)	1^i0^j10 for $(k{-}1)i = j$	$1^{i+1}0^{j+1}$	$(i{+}1, i{+}j{+}1)$
(4c)	1^i0^j10 for $(k{-}1)i > j$	$101^{i-1}0^j1$	$(i{+}1, i{+}j{+}1)$ $(2, i{+}j{+}2)$
(4d)	1^i0^j for $i = t$, $j = (k-1)t$	$101^{i-1}0^{j-1}$	$(2, i{+}1)$

Corollary 1. *$\mathbb{D}_k(t)$ is generated in cool-lex order by the rules in Table 4.*

Proof. The swap(s) are identical to those in Table 3 except for (4c) below.

	Current Prefix	Next Prefix	Swap(s)
(3c)	1^i0^j10 for $(k{-}1)i > j$	$101^{i-1}0^j1$	$(i{+}j{+}1, i{+}j{+}2)$ $(2, i{+}1)$
(4c)	1^i0^j10 for $(k{-}1)i > j$	$101^{i-1}0^j1$	$(i{+}1, i{+}j{+}1)$ $(2, i{+}j{+}2)$

When $B \in \mathbb{D}_k(t)$ has prefix 1^i0^j10 with $j > 0$, then the relevant bit values are

$$B[2] = \begin{cases} 1 & \text{if } i > 1 \\ 0 & \text{if } i = 1, \end{cases} \quad B[i{+}1] = 0,\ B[i{+}j{+}1] = 1,\ \text{and } B[i{+}j{+}2] = 0.$$

If $i > 1$, then (3c) and (4c) both change the prefix to $101^{i-1}0^j1$. If $i = 1$, then (3c) and (4c) both change the prefix to $1^i0^j01 = 10^j1^{i-1}01$ via $\mathsf{swap}(i{+}j{+}1, i{+}j{+}2)$. □

3 Loopless Algorithms

In this section we provide two loopless algorithms for generating k-ary Dyck words in cool-lex order: coolkat (for "small" k) and coolKat (for "large" k).

3.1 Algorithm for Constant k

We begin with coolkat, which is a simple algorithm that uses only two additional index variables. We prove its correctness in Theorem 4 and then prove that it is loopless for constant k in Theorem 5.

Theorem 4. *Algorithm* $\mathsf{coolkat}(k, t)$ *generates each successive k-ary Dyck word of length kt in cool-lex order.*

```
Procedure coolkat(k, t)
 1:
 2: B ← array(1^t 0^((k−1)t))
 3: x ← t
 4: y ← t
 5: visit()
 6: while x < k(t − 1) + 1
 7:    B[x] ← 0
 8:    B[y] ← 1
 9:    x ← x + 1
10:    y ← y + 1
11:    if B[x] = 0 then
12:       if x − 2 = k(y − 2) then
13:
14:          while B[x] = 0
15:             x ← x + 1
16:          end
17:
18:       else
19:          B[x] ← 1
20:          B[2] ← 0
21:          if y > 3 then
22:             x ← 3
23:          end
24:          y ← 2
25:       end
26:    end
27:    visit()
28: end
```

```
Procedure coolKat(k, t)
 1: A ← array(0^(t−2))
 2: B ← array(1^t 0^((k−1)t))
 3: x ← t
 4: y ← t
 5: visit()
 6: while x < k(t − 1) + 1
 7:    B[x] ← 0
 8:    B[y] ← 1
 9:    x ← x + 1
10:    y ← y + 1
11:    if B[x] = 0 then
12:       if x − 2 = k(y − 2) then
13:          if B[x + 1] = 1 then
14:             A[y − 2] ← 0
15:          end
16:          A[y − 2] ← A[y − 2] + 1
17:          x ← x + A[y − 2]
18:       else
19:          B[x] ← 1
20:          B[2] ← 0
21:          if y > 3 then
22:             x ← 3
23:          end
24:          y ← 2
25:       end
26:    end
27:    visit()
28: end
```

Algorithms 1: coolkat(k,t) and coolKat(k,t) generate k-ary Dyck words of length kt in cool-lex order for any $k,t \geq 1$ (with $1^t0^{(k-1)t}$ visited first).

Proof. We prove that the "main loop" on lines 6-28 always modifies B according to Table 4 by induction on the number of iterations. The first iteration visits $1^t0^{(k-1)t}$ and the second iteration begins with $y = 2$, $x = 3$, and $B = 101^{t-1}0^{(k-1)t-1}$, which is correct by (4c). The second iteration provides a base case for the following main-loop invariant:

If B has prefix 1^i0^j1 for $j > 0$ on line 6, then $y = i+1$ and $x = i+j+1$.

Inductively suppose this invariant holds for the mth iteration and consider the next iteration. Lines 7-8 apply swap$(i{+}1, i{+}j{+}1)$, which is the first swap listed in each of (4a)-(4c). Lines 9-10 increment the additional variables to $y = i + 2$ and $x = i + j + 2$. Now consider the possible paths through the algorithm.

- If $B[x] = 1$ on line 11, then the mth string in cool-lex order had prefix 1^i0^j11. By (4a) the successor has already been obtained by swap$(i{+}1, i{+}j{+}1)$. Furthermore, $y = i + 2$ and $x = i + j + 2$ correctly satisfy the invariant.

– If $B[x] = 0$ on line 11, then the mth string in cool-lex order had prefix 1^i0^j10.
  - If $x - 2 = k(y - 2)$ on line 12, then $j = (k - 1)i$ by simple algebra. By (4b) the successor has already been obtained by $\mathsf{swap}(i{+}1, i{+}j{+}1)$. Furthermore, $y = i + 2$ is correct. Since B now has prefix $1^{i+1}0^{j+1}$, x is greater than its current value of $i + j + 2$. The loop on line 14 scans the remainder of the B to determine the correct value of x.
  - If $x - 2 < k(y - 2)$ on line 12, then $j < (k - 1)i$. Lines 19-20 correctly apply $\mathsf{swap}(2, i + j + 2)$ by (4c) and change the prefix of B to $101^{i-1}0^j1$.
    * If $y > 3$ on line 21, then $i > 1$ and $x = 2$ is correctly set by line 22.
    * If $y = 3$ on line 21, then $i = 1$ and the current value of $x = i + j + 2$ is already correct.

    Finally, $y = 2$ is correctly set by line 24.

This induction continues until $B = 1^{t-1}0^{(k-1)(t-1)}10^{k-1}$ since this is the only string in $\mathbb{D}_k(t)$ for which $x \geq k(t-1)+1$ by the loop-invariant $x = i + j + 1$. By (4b) the successor of $1^{t-1}0^{(k-1)(t-1)}10^{k-1}$ is $1^t0^{(k-1)t}$, which was the first string visited. Therefore, $\mathsf{coolkat}(k, t)$ visits every string in $\mathbb{D}_k(t)$. □

Now we analyze coolkat. We need to show that the loop on line 14 runs a constant number of times when generating k-ary Dyck words for constant k. Towards this goal we present the following lemma.

Lemma 2. *If $1^i0^j10\gamma$ is a k-ary Dyck word and $j = (k - 1)i$, then γ does not have 0^{k-1} as a prefix.*

Proof. A k-ary Dyck word cannot have $1^i0^{(k-1)i}10^k$ as a prefix. □

Theorem 5. *Algorithm $\mathsf{coolkat}(k, t)$ uses two additional variables, and when k is a constant each successive string is created in worst-case $O(1)$-time.*

Proof. The algorithm uses the input values k and t, and stores the current k-ary Dyck word in the array B. Otherwise, the only additional variables are x and y. Therefore, the stated memory requirements are correct.

Next consider the run-time of creating each successive string in B. Notice that the only loop inside of the main loop on lines 6–28 is on 14. This loop is run when the current string stored in B at line 6 has a prefix equal to $1^i0^{(k-1)i}10$. By Lemma 2, the next k bits in B cannot all be 0. Therefore, the line 14 runs at most k times. If k is treated as a constant, then this loop can be replaced by a constant number of nested if-statements. Therefore, when k is a constant, successive strings are created in worst-case O(1)-time. □

3.2 Algorithm for Arbitrary k

To obtain a loopless algorithm for arbitrary k we perform the loop on line 14 with in O(1)-time by introducing an additional array of index variables A.

Theorem 6. $\mathsf{coolKat}(k, t)$ *is a loopless algorithm that generates each successive k-ary Dyck word of length kt in cool-lex order and uses only t index variables.*

Proof. Observe that coolkat and coolKat differ only in line 1 and lines 13-17. These lines are executed in coolKat when B begins the main-loop with a prefix of the form $1^i0^{(k-1)i}10$. By line 16, the A array is updated so that $A[i]$ contains the number of 0s that follow the prefix of the form $1^i0^{(k-1)i}1$. A formal proof of correctness requires an understanding of the recursive formulation of cool-lex order presented in Section 4 and is omitted. □

4 Ranking and Unranking

In this section we generalize k-ary Dyck words, discuss cool-lex order recursively, and then efficiently rank and unrank k-ary Dyck words in cool-lex order.

A string $B \in \mathbb{B}(s{+}t, t)$ is a *k-ary Dyck prefix* if the number of 0s in each prefix is at most $k{-}1$ times the number of 1s. Notice that k-ary Dyck prefixes with t $1s$ can have $s \leq (k{-}1)t$ $0s$, whereas k-ary Dyck words with t $1s$ must have $s = (k{-}1)t$ $0s$. Let $\mathbb{D}_k(t, s)$ be the k-ary Dyck prefixes in $\mathbb{B}(s{+}t, t)$. Thus,

$$\mathbb{D}_k(t, s) = \{B \in \mathbb{B}(s + t, t) \mid B0^{(k-1)t-s} \in \mathbb{D}_k(t)\}.$$

Let $\mathsf{N}_k(t, s)$ be the cardinality of $\mathbb{D}_k(t, s)$. Also let $v = (k{-}1)(t{-}1)$ in this section. The significance of this value is that every $B \in \mathbb{D}_k(t, s)$ has suffix 0^{s-v} if $s > v$.

Lemma 3. *$\mathsf{N}_k(t, s) = 0$ if $t = 0$, $\mathsf{N}_k(t, s) = 1$ if $t > 0$ and $s = 0$, and otherwise*

$$\mathsf{N}_k(t, s) = \begin{cases} \mathsf{N}_k(t{-}1, s) + \mathsf{N}_k(t, s{-}1) & \text{if } 1 \leq s \leq v; \\ \frac{1}{kt+1}\binom{kt+1}{t} & \text{if } v < s \leq (k-1)t. \end{cases}$$

Proof. $\mathbb{D}_k(0, s) = \emptyset$ and $\mathbb{D}_k(t, 0) = \{1^t\}$ if $t > 0$. If $1 \leq s \leq v$, then $B1 \in \mathbb{D}_k(t, s) \iff B \in \mathbb{D}_k(t{-}1, s)$ and $B0 \in \mathbb{D}_k(t, s) \iff B \in \mathbb{D}_k(t, s{-}1)$. Thus, $\mathsf{N}_k(t, s) = \mathsf{N}_k(t{-}1, s){+}\mathsf{N}_k(t, s{-}1)$. If $v < s \leq (k{-}1)t$, then all strings in $\mathbb{D}_k(t, s)$ end in 0 and $B \in \mathbb{D}_k(t, s) \iff B0^{(k-1)t-s} \in \mathbb{D}_k(t)$. Thus, $\mathsf{N}_k(t, s) = \frac{1}{kt+1}\binom{kt+1}{t}$ by the bijection between $\mathbb{D}_k(t)$ and k-ary trees with t internal nodes [3,10]. □

Ruskey, Sawada, Williams [6] prove that the following recursive formula gives the cool-lex order of any bubble language $\mathbb{L}$. The formula is explained below.

$$\mathcal{C}(t, s, \gamma) = \begin{cases} \mathcal{C}(t{-}1, 1, 10^{s-1}\gamma), \ldots, \mathcal{C}(t{-}1, s{-}j, 10^j\gamma), 1^t0^s\gamma & \text{if } t > 0; \quad \text{(1a)} \\ 0^s\gamma & \text{if } t = 0. \quad \text{(1b)} \end{cases}$$

If $1^t0^s\gamma \in \mathbb{L}$ and γ doesn't begin with 0, then $\mathcal{C}(t, s, \gamma)$ is the cool-lex order for the strings in $\mathbb{L}$ with suffix γ. The "fixed-suffix" γ is extended in turn in (1) to $10^{s-1}\gamma, 10^{s-2}\gamma, \ldots, 10^j\gamma$ where j is the minimum value such that $10^j\gamma$ is the suffix of a string in $\mathbb{L}$. Notice that γ is extended by 10^i for decreasing i with one exception: The single string resulting from $i = s$ (namely, $1^t0^s\gamma = 1^{t-1}\underline{10^s\gamma} = \mathcal{C}(t{-}1, 0, \underline{10^s\gamma})$) is last instead of first. In fact, this is the only difference between cool-lex order and conventional "co-lex order" (see [3] for lexicographic orders). The entire cool-lex order for some $\mathbb{L}$ with $1^t0^s \in \mathbb{L}$ is $\mathcal{C}(t, s, \epsilon)$. Now we specialize cool-lex order to k-ary Dyck prefixes. Let the *coolKat order* for $\mathbb{L} = \mathbb{D}_k(t, s)$ be denoted $\mathcal{D}_k(t, s, \epsilon) = \mathcal{C}(t, s, \epsilon)$.

Lemma 4. *CoolKat order is* $\mathcal{D}_k(t,s,\gamma)=\epsilon$ *if* $t=0$*, and otherwise*

$$\mathcal{D}_k(t,s,\gamma)=\begin{cases}\mathcal{D}_k(t{-}1,1,10^{s-1}\gamma),\ldots,\mathcal{D}_k(t{-}1,s,1\gamma),1^t0^s & \text{if } s\le v;\\ \mathcal{D}_k(t{-}1,1,10^{s-1}\gamma),\ldots,\mathcal{D}_k(t{-}1,v,10^{s-v}\gamma),1^t0^s & \text{if } v<s\le(k{-}1)t.\end{cases}$$

Proof. $\mathbb{L}=\mathbb{D}_k(t,s)$ is a bubble language, so $\mathcal{D}_k(t,s,\gamma)$ follows from (1) by giving the minimum j such that 10^j is the suffix of a string in $\mathbb{L}$. If $s\le v$, then $j=0$ by $1^{t-1}0^s1\in\mathbb{L}$. If $v<s\le(k-1)t$, then $j=s-v$ by $1^{t-1}0^s10^{s-v}\in\mathbb{L}$. □

Now we efficiently rank and unrank k-ary Dyck prefixes with examples after Theorems 7 and 8. With respect to an ordered set of strings $\mathcal{L}=B_1,B_2,\ldots,B_m$, the *rank* of B_i is $\mathrm{rank}(B_i,\mathcal{L})=i{-}1$, and $\mathrm{unrank}(i{-}1,\mathcal{L})=B_i$ for $1\le i\le m$. For convenience let $\mathsf{R}(B,\mathcal{L})=\mathrm{rank}(B,\mathcal{L})+1$. Also let $\mathcal{D}_k(t,s)$ denote $\mathcal{D}_k(t,s,\epsilon)$.

Theorem 7. *If* $B=\alpha10^m\in\mathcal{D}_k(t,s)$ *for possibly empty* α *and* $m\ge0$*, then*

$$\mathsf{R}(B,\mathcal{D}_k(t,s))=\begin{cases}\mathsf{N}_k(t,s) & \text{if } B=1^t0^s;\\ \mathsf{R}(\alpha,\mathcal{D}_k(t{-}1,s{-}m))+\sum_{i=1}^{s-m-1}\mathsf{N}_k(t{-}1,i) & \text{if } B\neq1^t0^s \text{ and } s\le v;\\ \mathsf{R}(\beta,\mathcal{D}_k(t,v)) & \text{otherwise,}\end{cases}$$

where β *is the first* $t+v$ *bits of* B*.*

Proof. If $B=1^t0^s$, then $\mathsf{R}(B,\mathcal{D}_k(t,s))=\mathsf{N}_k(t,s)$ since B is last in $\mathcal{D}_k(t,s)$ by Lemma 4.

If $B\neq1^t0^s$ and $0\le s\le v$, then $\mathcal{D}_k(t{-}1,i)$ appears before B in $\mathcal{D}_k(t,s)$ for $1\le i\le s{-}m{-}1$ by Lemma 4.

If $s>v$, then by Lemma 4 each string of $\mathcal{D}_k(t,v)$ appears as a prefix of the corresponding string in $\mathcal{D}_k(t,s)$, i.e., $\mathcal{D}_k(t,s)=\mathcal{D}_k(t,v,0^{s-v})$. Therefore, $\mathsf{R}(B,\mathcal{D}_k(t,s))=\mathsf{R}(\beta,\mathcal{D}_k(t,v))$. □

With respect to an ordered set of strings $\mathcal{L}$, let $\mathsf{U}(x,\mathcal{L})=\mathrm{unrank}(x{-}1)$.

Theorem 8

$$\mathsf{U}(x,\mathcal{D}_k(t,s))=\begin{cases}1^t0^s & \text{if } x=\mathsf{N}_k(t,s);\\ \mathsf{U}(x-\sum_{i=1}^{y}\mathsf{N}_k(t{-}1,i),\mathcal{D}_k(t{-}1,y{+}1))10^{s-y-1} & \text{if } x<\mathsf{N}_k(t,s) \text{ and } s\le v;\\ \mathsf{U}(x,\mathcal{D}_k(t,v))0^{s-v} & \text{otherwise,}\end{cases}$$

where y *is the largest integer such that* $x>\sum_{i=1}^{y}\mathsf{N}_k(t{-}1,i)$*.*

Proof. If $x=\mathsf{N}_k(t,s)$, then $\mathsf{U}(x,\mathcal{D}_k(t,s))$ is the last string in $\mathcal{D}_k(t,s)$ and by Lemma 4, $\mathsf{U}(x,\mathcal{D}_k(t,s))=1^t0^s$.

We now consider the case when $x<\mathsf{N}_k(t,s)$ and $0\le s\le v$. Let p be an integer, such that $\mathsf{U}(x,\mathcal{D}_k(t,s))$ is in $\mathcal{D}_k(t,p,10^{s-p})$. By Lemma 4, $x>\sum_{i=1}^{p-1}\mathsf{N}_k(t{-}1,i)$. It is now straightforward to observe that $y=p-1$. Therefore, $\mathsf{U}(x,\mathcal{D}_k(t,s))=\mathsf{U}(x-\sum_{i=1}^{y}\mathsf{N}_k(t-1,i),\mathcal{D}_k(t{-}1,y+1))10^{s-y-1}$.

The remaining case is $x<\mathsf{N}_k(t,s)$ and $s>v$. By Lemma 4, each string of $\mathcal{D}_k(t,v)$ appears as a prefix of the corresponding string in $\mathcal{D}_k(t,s)$, i.e., $\mathcal{D}_k(t,s)=\mathcal{D}_k(t,v,0^{s-v})$. Therefore, $\mathsf{U}(x,\mathcal{D}_k(t,s))=\mathsf{U}(x,\mathcal{D}_k(t,v))0^{s-v}$. □

We precompute and store the values of $\mathsf{N}_k(t,s)$ in a table so that for any value of k,t,s, we can obtain $\mathsf{N}_k(t,s)$ in $O(1)$ time. As a result we obtain $O(t+s)$-time ranking and unranking algorithms for k-ary Dyck words using Theorems 7 and 8, respectively. For example, the following table illustrates the first few values of $\mathsf{N}_k(t,s)$ for $k=5$. In the ranking and unranking process we assume that such tables for small fixed values of k are computed in advance. Thus for the corresponding precomputed values, we can obtain $\mathsf{N}_k(t,s)$ in $O(1)$ time.

$\mathsf{N}_5(t,s)$	s=0	s=1	s=2	s=3	s=4	s=5	s=6	s=7	s=8	s=9	s=10	s=11	s=12
$t=1$	1	1	1	1	1								
$t=2$	1	2	3	4	5	5	5	5	5				
$t=3$	1	3	6	10	15	20	25	30	35	35	35	35	35

We now compute $\mathsf{R}(100100010, \mathcal{D}_5(3,6))$ and $\mathsf{U}(16, \mathcal{D}_5(3,6))$ as follows:

$$\begin{aligned}
\mathsf{R}(100100010, \mathcal{D}_5(3,6)) &= \mathsf{R}(1001000, \mathcal{D}_5(2,5)) + \textstyle\sum_{i=1}^{6-1-1} \mathsf{N}_5(2,i) \\
&= \mathsf{R}(100, \mathcal{D}_5(1,2)) + \textstyle\sum_{i=1}^{5-3-1} \mathsf{N}_5(1,i) + \sum_{i=1}^{4} \mathsf{N}_5(2,i) \\
&= \mathsf{N}_5(1,2) + \textstyle\sum_{i=1}^{1} \mathsf{N}_5(1,i) + \sum_{i=1}^{4} \mathsf{N}_5(2,i) \\
&= 16. \\
\mathsf{U}(16, \mathcal{D}_5(3,6)) &= \mathsf{U}(16 - \textstyle\sum_{i=1}^{4} \mathsf{N}_5(2,i), \mathcal{D}_5(2,5))10^{6-4-1} \\
&= \mathsf{U}(2, \mathcal{D}_5(2,5))10 \\
&= \mathsf{U}(2, \mathcal{D}_5(2,4))0^{5-(2-1)(5-1)}10 \\
&= \mathsf{U}(2 - \textstyle\sum_{i=1}^{1} \mathsf{N}_5(1,i), \mathcal{D}_5(1,2))10^{4-1-1}010 \\
&= \mathsf{U}(1, \mathcal{D}_5(1,2))100010 \\
&= 100100010.
\end{aligned}$$

Acknowledgements. We thank Frank Ruskey for helpful conversations.

References

1. Canfield, E., Williamson, S.: A loop-free algorithm for generating the linear extensions of a poset. Order 12, 57–75 (1995)
2. Heubach, S., Li, N.Y., Mansour, T.: A garden of k-Catalan structures (2008), `http://www.scientificcommons.org/43469719`
3. Knuth, D.E.: The Art of Computer Programming: Generating all Trees and History of Combinatorial Generation, vol. 4. Addison-Wesley (2006)
4. Pruesse, G., Ruskey, F.: Generating the linear extensions of certain posets by transpositions. SIAM Journal on Discrete Mathematics 4(3), 413–422 (1991)
5. Ruskey, F.: Generating t-ary trees lexicographically. SIAM Journal on Computing 7(4), 424–439 (1978)
6. Ruskey, F., Sawada, J., Williams, A.: Binary bubble languages and cool-lex order, 13 pages (2010) (under review)

7. Ruskey, F., Williams, A.: Generating balanced parentheses and binary trees by prefix shifts. In: Proceedings of the 14th Computing: The Australasian Theory Symposium (CATS 2008), NSW, Australia, January 22-25, vol. 77, pp. 107–115 (2008)
8. Trojanowski, A.E.: Ranking and listing algorithms for k-ary trees. SIAM Journal on Computing 7(4), 492–509 (1978)
9. Vajnovszki, V., Walsh, T.: A loop-free two-close Gray-code algorithm for listing k-ary Dyck words. Journal of Discrete Algorithms 4(4), 633–648 (2006)
10. Zaks, S.: Generation and ranking of k-ary trees. Information Processing Letters 14(1), 44–48 (1982)

Two Constant-Factor-Optimal Realizations of Adaptive Heapsort

Stefan Edelkamp[1,⋆], Amr Elmasry[2], and Jyrki Katajainen[2,⋆⋆]

[1] TZI, Universität Bremen, Germany
[2] Department of Computer Science, University of Copenhagen, Denmark

Abstract. In this paper we introduce two efficient priority queues. For both, *insert* requires $O(1)$ amortized time and *extract-min* $O(\lg n)$ worst-case time including at most $\lg n + O(1)$ element comparisons, where n is the number of elements stored. One priority queue is based on a weak heap (array-based) and the other on a weak queue (pointer-based). In both, the main idea is to temporarily store the inserted elements in a buffer, and once it is full to move its elements to the main queue using an efficient bulk-insertion procedure. By employing the new priority queues in adaptive heapsort, we guarantee, for several measures of disorder, that the formula expressing the number of element comparisons performed by the algorithm is optimal up to the constant factor of the high-order term. We denote such performance as constant-factor optimality. Unlike some previous constant-factor-optimal adaptive sorting algorithms, adaptive heapsort relying on the developed priority queues is practically workable.

Keywords: Priority queues, weak heaps, weak queues, adaptive sorting, adaptive heapsort, constant-factor optimality.

1 Introduction

A sorting algorithm is *adaptive* if it changes its performance according to the presortedness within the input. When sorting n elements, the running time of such algorithm is $O(n)$ for sequences that are sorted or almost sorted, and $O(n \lg n)$ for sequences that have a high degree of disorder. It is important to note that such algorithms do not know in advance about the amount of existing disorder.

In the literature, several *measures of disorder* that characterize the input sequence have been considered [18]. An adaptive sorting algorithm is said to be *asymptotically optimal*, or simply *optimal*, if its running time asymptotically matches the lower bound derived using the number of input elements and the amount of disorder as parameters. In this paper we focus on a stronger form of optimality. We call an asymptotically-optimal algorithm *constant-factor-optimal*, if the number of element comparisons performed matches the information-theoretic

⋆ Partially supported by DFG grant ED 74/8-1.
⋆⋆ Partially supported by the Danish Natural Science Research Council under contract 09-060411 (project "Generic programming—algorithms and tools").

C.S. Iliopoulos and W.F. Smyth (Eds.): IWOCA 2011, LNCS 7056, pp. 195–208, 2011.
© Springer-Verlag Berlin Heidelberg 2011

lower bound up to the constant factor hidden behind the big-Oh notation, i.e. the constant factor multiplied by the high-order term.

Some of the known measures of disorder are the number of oscillations *Osc* [15], the number of inversions *Inv* [14], the number of runs *Runs* [14], the number of blocks *Block* [3], and the measures *Max*, *Exc* and *Rem* [3]. Some measures dominate the others: every *Osc*-optimal algorithm is *Inv* optimal and *Runs* optimal; every *Inv*-optimal algorithm is *Max* optimal [15]; and every *Block*-optimal algorithm is *Exc* optimal and *Rem* optimal [3]. Natural mergesort, described by Knuth [14, Section 5.2.4], is an example of an adaptive sorting algorithm that is constant-factor-optimal; this is with respect to the measure *Runs*.

For a sequence $X = \langle x_1, x_2, \ldots, x_n \rangle$, the number of inversions is the number of pairs of elements that are in the wrong order, i.e. $Inv(X) = |\{(i,j) \mid 1 \leq i < j \leq n \text{ and } x_i > x_j\}|$ [14, Section 5.1.1]. An optimal algorithm with respect to the measure *Inv* sorts a sequence X in $\Theta(n \lg \big(Inv(X)/n\big) + n)$ time. The optimality is implied by the information-theoretic lower bound $\Omega(n \lg \big(Inv(X)/n\big) + n)$ known for the number of element comparisons performed by any sorting algorithm with respect to the parameters n and *Inv* [12]. A constant-factor-optimal algorithm should perform at most $n \lg \big(Inv(X)/n\big) + O(n)$ element comparisons.

Several adaptive sorting algorithms are known to be asymptotically optimal with respect to the measure *Inv*. The known approaches are inspired by insertionsort [7,8,12,19,20,23], quicksort [16], mergesort [8,21], or heapsort [6,15]. However, only a few of the known algorithms are constant-factor-optimal; these are the insertionsort-based and mergesort-based algorithms of Elmasry and Fredman [8], and the heapsort-based algorithm of Levcopoulos and Petersson [15] when implemented with the multipartite priority queue of Elmasry et al. [10]. Most of these algorithms are complicated and have never been implemented.

Adaptive heapsort [15] is optimal with respect to all the aforementioned measures of disorder. We recall the description and analysis of adaptive heapsort in Section 2. In this paper we present two new realizations of adaptive heapsort. Our main motivation is to use, instead of a worst-case efficient priority queue [10], a simpler priority queue that can support *insert* in $O(1)$ amortized time and *extract-min* in $O(\lg n)$ worst-case time including at most $\lg n + O(1)$ element comparisons. From these bounds and the analysis given in [15], the constant-factor optimality follows for the following measures of disorder: *Osc*, *Inv*, *Runs*, and *Max*. Our main contribution is to present two priority queues with the required performance guarantees. The first priority queue improves over a weak heap (an array-based priority queue described in the context of sorting by Dutton [4] and further analysed by Edelkamp and Wegener [5]). We modify and analyse this priority queue in Section 3. The second priority queue improves over a weak queue (a binomial queue implemented using binary trees as suggested by Vuillemin [24]). We modify and analyse this priority queue in Section 4. The simple—but powerful—tool we used in both data structures is a buffer, into which the new elements are inserted. When the buffer becomes full, all its elements are moved to the main queue. In accordance, for both priority queues, we give an efficient bulk-insertion procedure.

We demonstrate the effectiveness of our approach by comparing the new realizations to the best implementations of known efficient sorting algorithms (splaysort [20] and introsort [22]). Our experimental settings, measurements, and outcomes are discussed in Section 5.

2 Adaptive Heapsort

In this section we describe the basic version of adaptive heapsort [15] and summarize the analysis of its performance.

The algorithm begins by building the *Cartesian tree* [25] for the input $X = \langle x_1, \ldots, x_n \rangle$. The root of the Cartesian tree stores $x_k = \min\{x_1, \ldots, x_n\}$, the left subtree of the root is the Cartesian tree for $\langle x_1, \ldots, x_{k-1} \rangle$, and the right subtree of the root is the Cartesian tree for $\langle x_{k+1}, \ldots, x_n \rangle$. Such a tree can be built in $O(n)$ time [11] by scanning the input in order and inserting each element x_i into the existing tree as follows. The nodes along the *right spine* of the tree (the path from the root to the rightmost leaf) are traversed bottom up, until a node with an element x_j that is not larger than x_i is found. In such case, the right subtree of the node of x_j is made the left subtree of the node of x_i, and the node of x_i is made the right child of the node of x_j. If x_i is smaller than all the elements on the right spine, the whole tree is made the left subtree of the node of x_i. This procedure requires at most $2n - 3$ element comparisons [15].

The algorithm proceeds by moving the smallest element at the root of the Cartesian tree into a priority queue. The algorithm then continues by repeatedly outputting and deleting the minimum from the priority queue. After each deletion, the elements at the children of the Cartesian-tree node corresponding to the deleted element are inserted into the priority queue. As for the priority-queue operations, n *insert* and n *extract-min* operations are performed. But, the heap will be small if the input sequence has a high amount of existing order.

The following improvement to the algorithm [15] is both theoretically and practically effective; even though, in this paper, it only affects the constant in the linear term. Since at least $\lfloor n/2 \rfloor$ of the *extract-min* operations are immediately followed by an *insert* operation (deleting a node that is not a leaf of the Cartesian tree must be followed by an insertion), every such *extract-min* can be combined with the following *insert*. This can be implemented by replacing the minimum of the priority queue with the new element and thereafter reestablishing the heap properties. Accordingly, the cost for half of the insertions will be saved.

The worst-case running time of the algorithm is $O(n \lg \big(Osc(X)/n\big) + n) = O(n \lg \big(Inv(X)/n\big) + n)$ [15]. For a constant β, the number of element comparisons performed is $\beta n \lg \big(Osc(X)/n\big) + O(n) = \beta n \lg \big(Inv(X)/n\big) + O(n)$. Levcopolous and Petersson suggested using a binary heap [26], which results in $\beta = 3$ (can be improved to $\beta = 2.5$ by combining *extract-min* and *insert* whenever possible). By using a binomial queue [24], we get $\beta = 2$. By using a weak heap [4], we get $\beta = 2$ (can be improved to $\beta = 1.5$ by combining *extract-min* and *insert*). By using the complicated multipartite priority queue [10], we indeed get the optimal $\beta = 1$. The question then arises whether we can achieve the constant-factor optimality, i.e. $\beta = 1$, and in the meantime ensure practicality!

In addition to the priority queue, the storage required by the algorithm is $2n$ extra pointers for the Cartesian tree. (We need not keep parent pointers since, during the construction, on the right spine of the tree we can temporarily revert each right-child pointer to point to the parent.) We also need to store the n elements inside the nodes of the Cartesian tree, either directly or indirectly.

3 Weak Heaps with Bulk Insertions

A *weak heap* [4] is a binary tree, where each node stores an element. A weak heap is obtained by relaxing the requirements of a binary heap [26]. The root has no left child, and the leaves are found at the last two levels only. The height of a weak heap that has n elements is therefore $\lceil \lg n \rceil + 1$. The *weak-heap property* enforces that the element stored at a node is not larger than all the elements stored in the right subtree of that node. In our implementation, illustrated in Fig. 1, besides the element array a an array r of *reverse bits* is used, i.e. $r_i \in \{0, 1\}$ for $i \in \{0, \ldots, n-1\}$. We use a_i to refer to either the element at index i of array a or to a node in the corresponding tree structure. A weak heap is laid out such that, for a_i, the index of its left child is $2i + r_i$, the index of its right child is $2i + 1 - r_i$, and (assuming $i \neq 0$) the index of its parent is $\lfloor i/2 \rfloor$. Using the fact that the indices of the left and the right children of a_i are exchanged when flipping r_i, subtrees can be swapped in constant time by setting $r_i \leftarrow 1 - r_i$.

The *distinguished ancestor* of a_i, $i \neq 0$, is the parent of a_i if a_i is a right child, and the distinguished ancestor of the parent of a_i if a_i is a left child. We use *d-ancestor*(i) to denote the index of such ancestor. The weak-heap property enforces that no element is smaller than that at its distinguished ancestor.

To *insert* a new element e, we first add e to the next available array entry, making it a leaf in the heap. To reestablish the weak-heap property, as long as e is smaller than the element at its distinguished ancestor, we swap the two elements and repeat this for the new location of e. It follows that *insert* requires $O(\lg n)$ time and involves at most $\lceil \lg n \rceil$ element comparisons.

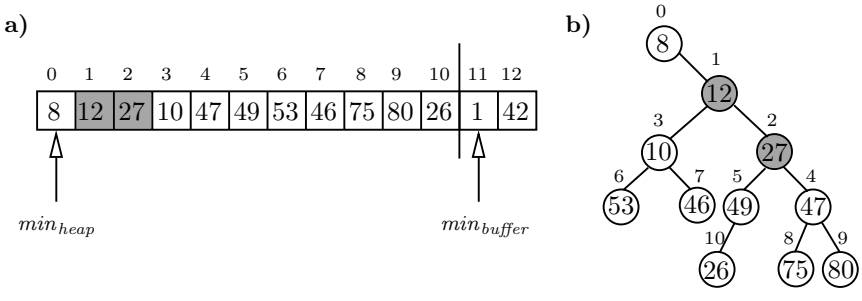


Fig. 1. A weak heap of size 11 and a buffer of size 2: **a)** the array representation and **b)** the corresponding tree representation of the weak heap; the nodes, for which the reverse bits are set, are highlighted

The subroutine *link* combines two weak heaps into one weak heap conditioned on the following settings. Let a_i and a_j be two elements in a weak heap, such that a_i is not larger than all the elements in the left subtree of a_j. Conceptually, a_j and its right subtree form a weak heap, while a_i and the left subtree of a_j form another weak heap. (Note that a_i could be at any location of the array.) If $a_j < a_i$, the subroutine *link* swaps the two elements and flips r_j; otherwise it does nothing. As a result, a_j will not be larger than any of the elements in its right subtree, and a_i will not be larger than any of the elements in the subtree rooted at a_j. All in all, *link* requires $O(1)$ time and involves one element comparison.

To perform *extract-min*, the element stored at the root of the weak heap is swapped with that stored at the last occupied array entry. To restore the weak-heap property, repeated *link* operations are performed that involve the current root of the weak heap; the details follow. The last node on the *left spine* (the path from a node to the leftmost leaf) of the right child of the root is identified. Starting from the child of the root, this is done by repeatedly traversing left children until reaching a node that has no left child. The path from this node to the child of the root is traversed upwards, and *link* operations are repeatedly performed between the root of the weak heap and the nodes along this path. The correctness of the *extract-min* operation follows from the fact that, after each *link*, the element at the root of the heap is not larger than all the elements in the left subtree of the node to be considered in the next *link*. Thus, *extract-min* requires $O(\lg n)$ time and involves at most $\lceil \lg n \rceil$ element comparisons.

The cost of *insert* can be improved to an amortized constant. The key idea is to use a *buffer* that supports constant-time insertion. The buffer can be implemented as a resizable array. Additionally, a pointer to the minimum element in the buffer is maintained. The maximum size of the buffer is set to $\lceil \lg n \rceil$, where n is the total number of elements stored. A new element is inserted into the buffer as long as its size is below the threshold. Once the threshold is reached, a *bulk insertion* is performed by moving all the elements of the buffer to the weak heap. For the *extract-min* operation, the minimum of the buffer is compared with the minimum of the weak heap, and accordingly the operation is performed either in the buffer or in the weak heap. Deleting the minimum of the buffer is done by removing the minimum and scanning the buffer to determine the new minimum. Almost matching the bounds for the weak heap, deleting the minimum of the buffer requires $O(\lg n)$ time and involves at most $\lceil \lg n \rceil - 2$ element comparisons. Thus, *extract-min* involves at most $\lceil \lg n \rceil + 1$ element comparisons.

Let us now consider how to perform a bulk insertion in $O(\lg n)$ time (see Fig. 2). First, we move the elements of the buffer to the next available entries of the array that stores the weak heap. The main idea is to reestablish the weak-heap property bottom-up level-by-level. Starting with the inserted nodes, for each node we *link* its distinguished ancestor to it. We then consider the parents of these nodes on the next upper level, and for each parent we *link* its distinguished ancestor to it, restoring the weak-heap property up to this level. This is repeated until the number of nodes that we need to deal with at a level is two (or less). At this point, we switch to a more efficient strategy. For each of

```
input: a: array of elements, r: array of bits, buffer: array of elements
right ← size(a) + size(buffer) − 1
left ← max{size(a), ⌊right/2⌋}
while size(buffer) > 0
    size(a)++
    a[size(a) − 1] ← buffer[size(buffer) − 1]
    size(buffer)−−
    size(r)++
    r[size(r) − 1] ← 0
while right > left + 1
    for j ∈ {right, right − 1, . . . , left}
        i ← d-ancestor(j)
        if a[j] < a[i]
            swap(a[i], a[j])
            r[j] ← 1 − r[j]
    left ← ⌊left/2⌋
    right ← ⌊right/2⌋
for j ∈ {left, right}
    while j ≠ 0
        i ← d-ancestor(j)
        if a[j] < a[i]
            swap(a[i], a[j])
            r[j] ← 1 − r[j]
        j ← i
```

Fig. 2. The pseudo-code for bulk insertion in a weak heap

these two nodes, we reestablish the weak-heap property by traversing the path from such node towards the root. If the value of the current node x is smaller than that at its distinguished ancestor, we *link* the distinguished ancestor to x. We then repeat after setting x to be its old distinguished ancestor.

The correctness of the bulk-insertion procedure follows since, before considering the ith level, the value at any node below level i is not smaller than that at its distinguished ancestor. Hence, the value at the distinguished ancestor of a node x at level i is guaranteed not to be larger than the value at any node of the left subtree of x; this ensures the validity of the *link* operations to be performed at level i. Once we reach the root, the weak-heap property is valid for all nodes.

Let k be the number of elements moved from the buffer to the weak heap by the bulk-insertion procedure. The number of element comparisons performed at the ith iteration equals the number of *link* operations at the ith last level of the weak heap, which is at most $\lfloor (k-2)/2^{i-1} \rfloor + 2$. Here, we use the fact that the number of parents of a contiguous block of b elements in the array of a weak heap is at most $\lfloor (b-2)/2 \rfloor + 2$. Since the number of iterations is at most $\lceil \lg n \rceil$, the total number of element comparisons is less than $\sum_{i=1}^{\lceil \lg n \rceil} (1/2^{i-1} \cdot k + 2) < 2k + 2\lceil \lg n \rceil$. When $k = \lceil \lg n \rceil$, the number of element comparisons is less than $4\lceil \lg n \rceil$; this accounts for four comparisons per element in the amortized sense. Due to the

bulk insertion and the check for whether the minimum of the buffer is up to date or not, *insert* involves amortized five element comparisons in total.

The running time of the bulk insertion is dominated by the localization of the distinguished ancestors of the involved nodes. To find the distinguished ancestor of a node, we repeatedly go to the parent and check whether the current node is its right child or not. We call such an operation an *ancestor check*. We separately consider two parts of the procedure. The first part comprises the process of finding the distinguished ancestors for the levels with more than two involved nodes. Recall that the total number of those nodes at the ith last level is $k/2^{i-1} + O(1)$, for a total of $2k + o(k)$. Among the nodes involved, at most $(2k + o(k))/2^j$ need j ancestor checks to get to the distinguished ancestor, where $j \geq 1$. This accounts for at most $\sum_{j\geq 1} j/2^{j-1} \cdot (k + o(k)) < 4(k + o(k)) = 4\lceil \lg n \rceil + o(\lg n)$ ancestor checks. The second part comprises two path traversals towards the root, which involve at most $2\lceil \lg n \rceil$ ancestor checks in total. We conclude that the amortized cost accounted per element is a constant.

4 Weak Queues with Bulk Insertions

In this section we resort to a binomial queue that is implemented using binary trees [24]; we call this variant a *weak queue*. This data structure is a collection of perfect weak heaps (binomial trees in the binary-tree form), where the size of each tree is a power of two. The binary representation of n specifies the sizes of the perfect weak heaps that are present. A 1-bit at position r indicates that a perfect weak heap of size 2^r is present. The *rank* of a perfect weak heap of size 2^r is r. In our implementation, illustrated in Fig. 3, every node stores a pointer to its left child, a pointer to its right child, and (a pointer to) an element.

Two perfect weak heaps of rank r can be *linked* to form a perfect weak heap of rank $r + 1$, by making the root that has the smaller element the root of the resulting weak heap, the other root the right child of the new root, and the previous right child of the new root the left child of the other root.

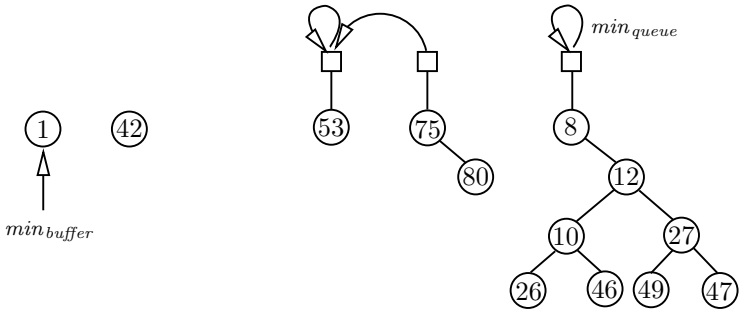


Fig. 3. A buffer of size 2 and a weak queue of size 11 (1011 in binary)

To *insert* a node into a weak queue, we let the new node form a single-node tree. This may trigger a sequence of *link* operations until no two trees of the same rank exist. Still, the amortized cost per *insert* is a constant [24].

To perform *extract-min*, we scan the roots of the perfect weak heaps to find the minimum. We then borrow the root of the smallest tree; let that root be x. In accordance, every node on the left spine of x's right subtree becomes the root of a perfect weak heap, and these heaps are added to the collection. Hereafter, we detach the root with the minimum value; let that root be y. Now every node on the left spine of y's right subtree is the root of a perfect weak heap. Using repeated *link* operations, the node x is combined with the roots of these heaps to create a perfect weak heap that has the same size as the heap rooted at y before the deletion. (A *link* operation is performed between x and the root of the smallest such heap, and the resulting heap is repeatedly linked with the next larger remaining heap, and so on.) It follows that *extract-min* requires $O(\lg n)$ time and involves at most $2\lceil \lg n \rceil - 2$ element comparisons.

To speed things up, we maintain prefix-minimum pointers for the roots of the perfect weak heaps (for the origin of this idea, consult [10] and the references therein). The prefix-minimum pointer of the root of a heap of size 2^r points to the root with the smallest value among the heaps of size 2^j for $j \leq r$. The overall minimum can be located by following the prefix-minimum pointer of the root of the largest heap. Now we have to borrow a node such that the prefix-minimum pointers need not be updated. As before, the borrowed node is repeatedly linked with the roots of the heaps resulting from detaching the minimum node. This requires r element comparisons if the rank of the deleted node is r. We still have to update the prefix-minimum pointers. The key idea is that we only need $\lceil \lg n \rceil - r$ element comparisons to update the prefix-minimum pointers of the larger heaps. Hence, *extract-min* involves at most $\lceil \lg n \rceil$ element comparisons.

If we implement *insert* in the normal way, we then have to update the prefix-minimum pointers; this would require a logarithmic number of element comparisons. Our way out is again to rely on bulk insertions (see Fig. 4). We collect at most $\lceil \lg n \rceil$ elements into a buffer, where n is the total number of elements stored. The buffer is implemented as a circular singly-linked list, having its minimum first. When the buffer becomes full, we clear it by repeatedly inserting its elements into the weak queue in the normal way, without updating the prefix-minimum pointers. After finishing these insertions, the prefix-minimum pointers are updated once. For the bulk insertion, an amortized analysis accounts for a constant amortized cost per element, involving amortized two element comparisons. Since it is necessary to maintain the minimum of the buffer, an insertion into the buffer involves one element comparison. This together with the bulk insertion accounts for three element comparisons per *insert*.

We have to implement borrowing carefully so that it does not invalidate the prefix-minimum pointers. While performing no element comparisons, it takes $O(\lg n)$ time. If the buffer is non-empty, a node is borrowed from there. Otherwise, a node is borrowed from the main queue. If the size of the smallest heap is larger than one, the last node from the left spine of the right subtree of its

```
input: Q: queue of perfect weak heaps, buffer: list of nodes
while size(buffer) > 0
    x ← pop(buffer)
    insert(Q, x)
update-prefix-minimum-pointers(Q)
```

Fig. 4. The pseudo-code for bulk insertion in a weak queue. For a list, subroutine *pop* removes and returns its last node. For a queue, subroutine *insert* adds the given node to the queue and makes the necessary linkings leaving at most one heap per rank. Subroutine *update-prefix-minimum-pointers* updates all the prefix-minimum pointers as they may not be up to date after *insert* operations.

root is borrowed. The root and the other nodes on the left spine are added as new roots to the main structure, and the prefix-minimum pointers associated with each of them is set to point to the old root (rooting a heap of size one now). Otherwise, the smallest heap is a singleton. This singleton is borrowed if the prefix-minimum pointer of the second smallest heap does not point to it. Otherwise, these two roots are swapped and the current singleton is borrowed.

For the *extract-min* operation, the minimum of the buffer is compared with the minimum of the weak queue, and accordingly the operation is performed either in the buffer or in the weak queue. After these modifications, *extract-min* involves at most $\lceil \lg n \rceil + 1$ element comparisons.

5 Experimental Findings

We implemented two versions of adaptive heapsort, one using a weak heap and another using a weak queue. In this section we discuss the settings and outcomes of our performance tests. In these tests we measured the actual running time of the programs and the number of element comparisons performed. The main purpose for carrying out these experiments was to validate our theoretical results.

Our implementation of adaptive heapsort using a weak heap was array-based. Each entry of the array representing the Cartesian tree stored a copy of an element and two references to other entries in the tree. The arrays representing the weak heap and the buffer stored references to the Cartesian tree, and a separate array was used for the reverse bits. In total, the space usage per element was three references, a copy of the element, and one bit. Dynamic memory allocation was avoided by preallocating all arrays from the stack. Users should be aware that, due to the large space requirements, the algorithm has a restricted utility depending on the amount of memory available.

In our implementation of adaptive heapsort using a weak queue, some non-trivial enhancements were made. First, we used two pointers per node: one pointing to the left child and another to the right child. As advised by Vuillemin [24], because of the lack of parent pointers, we reverted the left-child pointers to temporarily point to the parents while performing repeated linkings in *extract-min*. Second, we used an array of pointers to access the roots of the trees. This array

also infers the ranks of these roots; the nodes themselves did not store any rank information. Third, we used the same nodes to store the pointers needed by the Cartesian tree and the buffer. Fourth, all memory was preallocated from the stack. In total, the space usage per node was four pointers and a copy of an element; another $O(\lg n)$ space was used by the array of root pointers and the array of prefix-minimum pointers. Accordingly, this implementation used even more memory than the version employing a weak heap.

To select suitable competitors for our implementations, we consulted some earlier research papers concerning the practical performance of inversion-optimal sorting algorithms [9,20,23]. Based on this survey, we concluded that splaysort performs well in practice. In addition, the implementation of Moffat et al. [20] is highly tuned, practically efficient, and publicly available. Consequently, we selected their implementation of splaysort as our primary competitor. In the aforementioned experimental papers, splaysort has been reported to perform better than other tree-based algorithms (e.g. AVL-sort [7]), cache-oblivious algorithms (e.g. greedysort [1]), and partition-based algorithms (e.g. splitsort [16]).

When considering comparison-based sorting, one should not ignore quicksort [13]. Introsort [22] is a highly tuned variant of quicksort that is known to be fast in practice. It is based on half-recursive median-of-three quicksort, it coarsens the base case by leaving small subproblems unsorted, it calls insertionsort to finalize the sorting process, and it calls heapsort if the recursion depth becomes too large. Using the middle element as a candidate for the pivot, and using insertionsort at the back end, make introsort adaptive with respect to the number of element comparisons (though not optimally adaptive with respect to any known measure of disorder). Quicksort and its variants are also known to be optimally adaptive with respect to the number of element swaps performed [2]. For these reasons, we selected the standard-library implementation of introsort shipped with our C++ compiler as our secondary competitor.

In the experiments, the results of which are discussed here (see Figs. 5–8), we used 4-byte integers as input data. The results were similar for different input sizes; for the reported experiments the number of elements was fixed to 10^7 and 10^8. We ensured that all the input elements were distinct. Integer data was sufficient to back up our theoretical analysis. However, for other types of input data, the number of element comparisons performed and the number of cache misses incurred may have more significant influence on the running time.

We performed the experiments on one core of a desktop computer (model Intel i/7 CPU 2.67 GHz) running Ubuntu 10.10 (Linux kernel 2.6.32-23-generic). This computer had 32 KB L1 cache memory, 256 KB L2 cache memory, 8 MB (shared) L3 cache memory, and 12 GB main memory. With such memory capacity, there was no need to use virtual memory. We compiled all programs using GNU C++ compiler (`gcc` version 4.4.3 with option `-O3`).

To generate the input data, we used two types of generators:

Repeated Swapping. We started with a sorted sequence of the integers from 1 to n, and repeatedly performed random transpositions of two consecutive elements. This generator was used to produce data with few inversions.

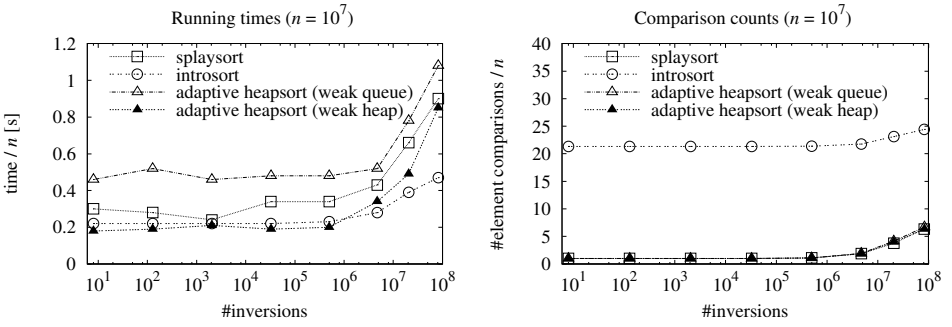


Fig. 5. Repeated swapping, $n = 10^7$: CPU time used and the number of element comparisons performed by different sorting algorithms

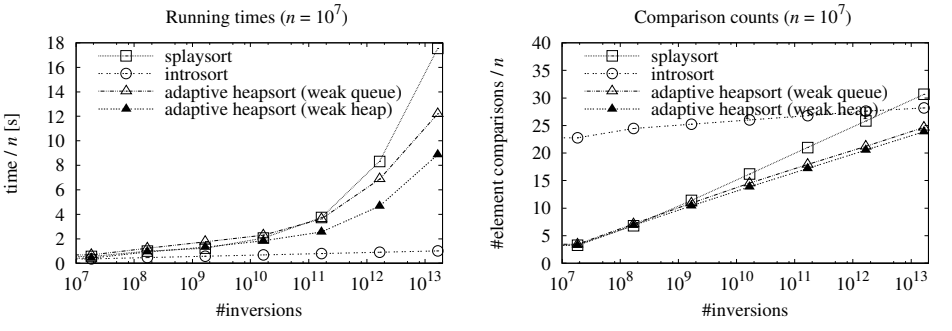


Fig. 6. Controlled shuffling, $n = 10^7$: CPU time used and the number of element comparisons performed by different sorting algorithms

Controlled Shuffling [9]**.** We started with a sorted sequence of the integers from 1 to n, and performed two types of perturbations; we call the sequences resulting from these two phases *local* and *global shuffles*. For local shuffles, the sorted sequence was broken into $\lceil n/m \rceil$ consecutive blocks each containing m elements (except possibly the last block), and the elements of each block were randomly permuted. For global shuffles, the sequence produced by the first phase was broken into m consecutive blocks each containing $\lceil n/m \rceil$ elements (except possibly the last block). From each block one element was selected at random, and these elements were randomly permuted. A small value of m means that the sequence is sorted or almost sorted, and a large value of m means that the sequence is random. Given a parameter m, this shuffling results in a sequence with expected $\Theta(n \cdot m)$ inversions.

Since in both cases the resulting sequence is a permutation of the integers from 1 to n, the number of inversions could be easily calculated as $\sum_{i=1}^{n} |x_i - i|/2$.

The experiments showed that our realizations of adaptive heapsort perform a low number of element comparisons. For both versions, the number of element comparisons was about the same, as already verified analytically. When

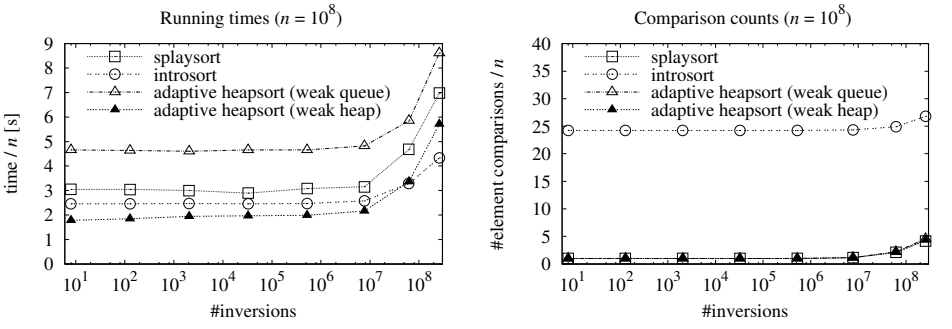


Fig. 7. Repeated swapping, $n = 10^8$: CPU time used and the number of element comparisons performed by different sorting algorithms

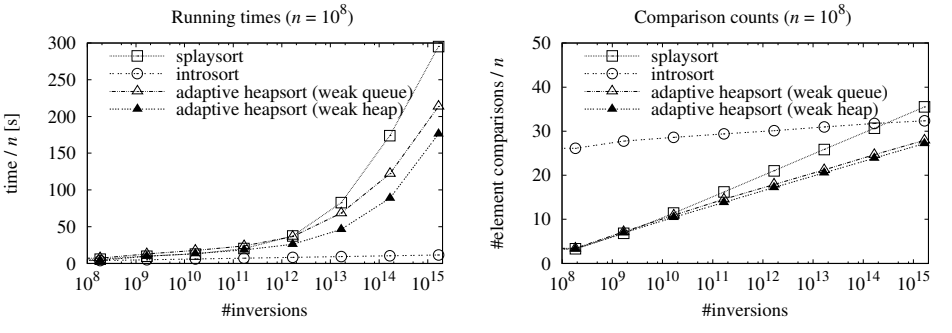


Fig. 8. Controlled shuffling, $n = 10^8$: CPU time used and the number of element comparisons performed by different sorting algorithms

the number of inversions was small, splaysort performed about the same number of element comparisons as the two realizations of adaptive heapsort. When the number of inversions was large, splaysort performed a few more element comparisons than the two realizations of adaptive heapsort. In all our experiments, introsort was a bad performer with respect to the number of element comparisons; it showed very little adaptivity and came last in the competition.

As to the running time, the weak-heap version of adaptive heapsort was faster than the weak-queue version; about 60% faster when the number of inversions was small and about 20% faster when the number of inversions was large. The running times of splaysort were larger than the ones of the weak-heap version for almost all experiments. For random data, splaysort performed worst, and adaptive heapsort could be up to a factor of 15 slower than introsort. (In our supplementary experiments, for random data, normal heapsort was only a factor of 2–6 slower than introsort depending on the input size.) In most experiments, introsort was the fastest sorting method; it was only beaten by the weak-heap version when the number of inversions was very small (less than n).

6 Conclusions

We studied the optimality and practicality of adaptive heapsort. We introduced two new realizations for it, which are theoretically optimal and practically workable. Even though our realizations outperformed the state-of-the-art implementation of splaysort, the C++ standard-library introsort was faster for most inputs, at least on integer data. Despite decades of research, there is still a gap between the theory of adaptive sorting and the actual computing practice.

In spite of the optimality with respect to several measures of disorder, the high number of cache misses is not on our side. Compared to earlier implementations of adaptive heapsort, a buffer increased the locality of memory references and thus reduced the number of cache misses incurred. Still, introsort has considerably better cache behaviour. Earlier research has pointed out [9] that existing cache-efficient adaptive sorting algorithms are not competitive. The question arises whether constant-factor optimality with respect to the number of element comparisons can be achieved side by side to cache efficiency.

Another drawback of adaptive heapsort is the extra space required by the Cartesian tree. In introsort the elements are kept in the input array, and sorting is carried out in-place. Overheads attributable to pointer manipulations, and a high memory footprint in general, deteriorate the performance of any implementation of adaptive heapsort. This is in particular true when the amount of disorder is high. As to the memory requirements, we used about $4n$ extra words of storage for pointers and n extra space for copies of elements. An in-place algorithm that is optimal with respect to the measure *Inv* exists [17], but it is not practical. The question arises whether the memory efficiency of adaptive heapsort can be improved without sacrificing the optimal adaptivity.

In another extension, one should carry out experiments on data types for which element comparisons are more expensive than other operations. We are not far away from $n \lg n$ element comparisons when the amount of disorder is high. (Our best bound on the number of element comparisons is $n \lg \big(1 + Osc(X)/n\big) + 5.5n$.) The question arises whether the constant factor for the linear term in the number of element comparisons can be improved; that is, how close we can get to the information-theoretic lower bound up to low-order terms.

Source Code

The programs used in the experiments are available via the home page of the CPH STL (`http://cphstl.dk/`) in the form of a PDF document and a `tar` file.

References

1. Brodal, G.S., Fagerberg, R., Moruz, G.: Cache-Aware and Cache-Oblivious Adaptive Sorting. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 576–588. Springer, Heidelberg (2005)
2. Brodal, G.S., Fagerberg, R., Moruz, G.: On the Adaptiveness of Quicksort. ACM J. Exp. Algorithmics 12, Article 3.2 (2008)

3. Carlsson, S., Levcopoulos, C., Petersson, O.: Sublinear Merging and Natural Mergesort. Algorithmica 9(6), 629–648 (1993)
4. Dutton, R.D.: Weak-heap Sort. BIT 33(3), 372–381 (1993)
5. Edelkamp, S., Wegener, I.: On the Performance of Weak-Heapsort. In: Reichel, H., Tison, S. (eds.) STACS 2000. LNCS, vol. 1770, pp. 254–266. Springer, Heidelberg (2000)
6. Elmasry, A.: Priority Queues, Pairing and Adaptive Sorting. In: Widmayer, P., Triguero, F., Morales, R., Hennessy, M., Eidenbenz, S., Conejo, R. (eds.) ICALP 2002. LNCS, vol. 2380, pp. 183–194. Springer, Heidelberg (2002)
7. Elmasry, A.: Adaptive Sorting with AVL Trees. In: IFIP Int. Fed. Inf. Process., vol. 155, pp. 315–324. Springer, New York (2004)
8. Elmasry, A., Fredman, M.L.: Adaptive Sorting: An Information Theoretic Perspective. Acta. Inform. 45(1), 33–42 (2008)
9. Elmasry, A., Hammad, A.: Inversion-sensitive Sorting Algorithms in Practice. ACM J. Exp. Algorithmics 13, Article 1.11 (2009)
10. Elmasry, A., Jensen, C., Katajainen, J.: Multipartite Priority Queues. ACM Trans. Algorithms 5(1), Article 14 (2008)
11. Gabow, H.N., Bentley, J.L., Tarjan, R.E.: Scaling and Related Techniques for Geometry Problems. In: 16th Annual ACM Symposium on Theory of Computing, pp. 135–143. ACM, New York (1984)
12. Guibas, L.J., McCreight, E.M., Plass, M.F., Roberts, J.R.: A New Representation for Linear Lists. In: 9th Annual ACM Symposium on Theory of Computing, pp. 49–60. ACM, New York (1977)
13. Hoare, C.A.R.: Quicksort. Comput. J. 5(1), 10–16 (1962)
14. Knuth, D.E.: Sorting and Searching, The Art of Computer Programming, 2nd edn., vol. 3. Addison Wesley Longman, Reading (1998)
15. Levcopoulos, C., Petersson, O.: Adaptive Heapsort. J. Algorithms 14(3), 395–413 (1993)
16. Levcopoulos, C., Petersson, O.: Splitsort: An Adaptive Sorting Algorithm. Inform. Process. Lett. 39(4), 205–211 (1991)
17. Levcopoulos, C., Petersson, O.: Exploiting Few Inversions When Sorting: Sequential and Parallel Algorithms. Theoret. Comput. Sci. 163(1-2), 211–238 (1996)
18. Mannila, H.: Measures of Presortedness and Optimal Sorting Algorithms. IEEE Trans. Comput. C-34(4), 318–325 (1985)
19. Mehlhorn, K.: Sorting Presorted Files. In: Weihrauch, K. (ed.) GI-TCS 1979. LNCS, vol. 67, pp. 199–212. Springer, Heidelberg (1979)
20. Moffat, A., Eddy, G., Petersson, O.: Splaysort: Fast, Versatile, Practical. Software Pract. Exper. 126(7), 781–797 (1996)
21. Moffat, A., Petersson, O., Wormald, N.C.: A Tree-based Mergesort. Acta Inform. 35(9), 775–793 (1998)
22. Musser, D.R.: Introspective Sorting and Selection Algorithms. Software Pract. Exper. 27(8), 983–993 (1997)
23. Saikkonen, R., Soisalon-Soininen, E.: Bulk-Insertion Sort: Towards Composite Measures of Presortedness. In: Vahrenhold, J. (ed.) SEA 2009. LNCS, vol. 5526, pp. 269–280. Springer, Heidelberg (2009)
24. Vuillemin, J.: A Data Structure for Manipulating Priority Queues. Commun. ACM 21(4), 309–315 (1978)
25. Vuillemin, J.: A Unifying Look at Data Structures. Commun. ACM 23(4), 229–239 (1980)
26. Williams, J.W.J.: Algorithm 232: Heapsort. Commun. ACM 7(6), 347–348 (1964)

A Unifying Property for Distribution-Sensitive Priority Queues

Amr Elmasry[1,⋆], Arash Farzan[2], and John Iacono[3,⋆⋆]

[1] Computer Science Department, University of Copenhagen, Denmark
elmasry@diku.dk
[2] Max-Planck-Institut für Informatik, Saarbrücken, Germany
afarzan@mpi-inf.mpg.de
[3] Polytechnic Institute of New York Univerity, Brooklyn, New York, USA
jiacono@poly.edu

Abstract. We present a priority queue that supports the operations: *insert* in worst-case constant time, and *delete*, *delete-min*, *find-min* and *decrease-key* on an element x in worst-case $O(\lg(\min\{w_x, q_x\}+2))$ time, where w_x (respectively, q_x) is the number of elements that were accessed after (respectively, before) the last access of x and are still in the priority queue at the time when the corresponding operation is performed. Our priority queue then has both the working-set and the queueish properties; and, more strongly, it satisfies these properties in the worst-case sense. We also argue that these bounds are the best possible with respect to the considered measures. Moreover, we modify our priority queue to satisfy a new unifying property — the time-finger property — which encapsulates both the working-set and the queueish properties.

In addition, we prove that the working-set bound is asymptotically equivalent to the unified bound (which is the minimum per operation among the static-finger, static-optimality, and working-set bounds). This latter result is of tremendous interest by itself as it had gone unnoticed since the introduction of such bounds by Sleater and Tarjan [10].

Together, these results indicate that our priority queue also satisfies the static-finger, the static-optimality and the unified bounds.

1 Introduction

Distribution-sensitive data structures are those structures for which the time bounds to perform operations vary depending on the sequence of operations performed [8]. These data structures typically perform as well as their distribution-insensitive counterparts on a random sequence of operations in the amortized sense. Yet, when the sequence of operations follows some particular distributions (for example, having temporal or spatial locality), the distribution-sensitive data structures perform significantly better.

⋆ Supported by fellowships from Alexander von Humboldt and VELUX foundations.
⋆⋆ Research partially supported by NSF grants CCF-0430849, CCF-1018370, and an Alfred P. Sloan fellowship, and by MADALGO—Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation, Aarhus University.

C.S. Iliopoulos and W.F. Smyth (Eds.): IWOCA 2011, LNCS 7056, pp. 209–222, 2011.
© Springer-Verlag Berlin Heidelberg 2011

The quintessential distribution-sensitive data structure is the splay tree [10]. Splay trees seem to perform very efficiently (much faster than $O(\lg n)$ search time on a set of n elements) over several natural sequences of operations. There still exists no single comprehensive distribution-sensitive analysis of splay trees; instead, there are theorems and conjectures that characterize their distribution-sensitive properties. These properties include: static finger, static optimality, sequential access, working set, unified bound, dynamic finger, and unified conjecture [1,3,4,10,11]. As defined in [10], the "unified bound" is the per-operation minimum of the static-optimality, static-finger, and working-set bounds. By the "unified conjecture", we follow the definition given in [1], which subsumes both the dynamic-finger and working-set bounds (The reason for such name is that the splay trees are conjectured to attain such bound.). We refer the reader to [1] and also [10] for a thorough definition and discussion of these properties.

There are implication relationships between the distribution-sensitive properties, as illustrated in Figure 1. Implications depicted by "$\longrightarrow$" were known in previous work, and the ones depicted by "$\Longrightarrow$" are contribution of this paper. In particular, we prove in Section 2 the implication of the unified bound from the working-set bound. This indicates that the working-set bound and the unified bound are asymptotically equivalent. In the course of the proof, we show that the sum of the working-set bounds on two sequences is asymptotically the same as the working-set bound of any *interleaving sequence* of these two sequences; this result is of independent interest. A sequence X is an interleaving sequence of two sequences Y and Z if and only if X is composed of all the elements of Y and Z in the same order they appear in their respective sequences. We defer the full discussion of the previously-known implications shown in Figure 1 to a full version of this paper.

Distribution-sensitive data structures are not limited to search trees. Priority queues have also been designed and analyzed in the context of distribution-sensitivity [2,5,7,8,9]. In the comparison model, it is easy to observe that a priority queue with constant insertion time cannot have the sequential-access property (and hence cannot as well have the dynamic-finger property). For otherwise, a sequence of insertions followed by a sequence of minimum-deletions lists the elements in sorted order in linear time. Alternatively, the working-set property has been of main interest for priority queues. Informally, the working-set property states that elements that have been recently operated on are faster to operate on subsequently compared to the elements that have not been accessed in the recent past. Iacono [7] proved that pairing-heaps [6] satisfy the working-set property as follows; in a heap of maximum size n, it takes $O\left(\lg(\min\{n_x, n\}+2)\right)$ amortized time to delete the minimum element x, where n_x is the number of operations performed since x's insertion. Funnel-heaps are I/O-efficient heaps for which it takes $O\left(\lg(\min\{i_x, n\}+2)\right)$ to delete the minimum element x, where i_x is the number of insertions made since x's insertion. Elmasry [5] gave a priority queue supporting the deletion of the minimum element x in $O\left(\lg(w_x+2)\right)$ worst-case time, where w_x is the number of elements inserted after the insertion of x and are still present in the priority queue when x is deleted (note that

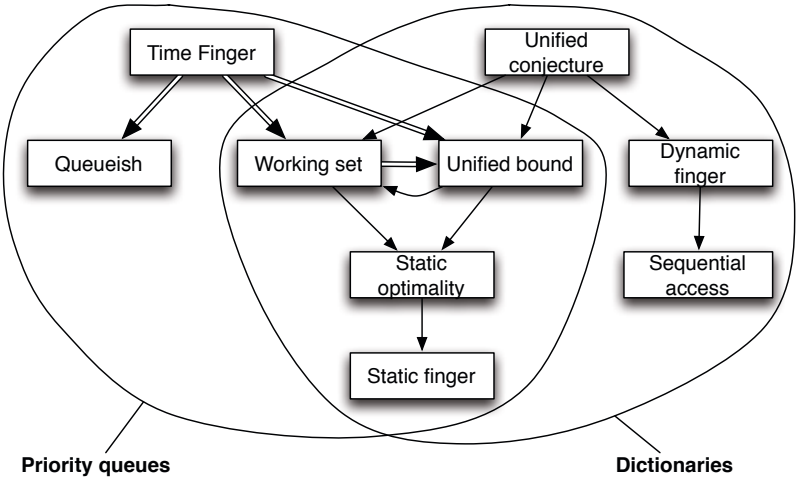


Fig. 1. The implication relationships between various distribution-sensitive properties. The implications depicted by "$\Longrightarrow$" are contribution of this paper.

$w_x \leq i_x \leq n_x$). We briefly review this priority queue in Section 3. None of the aforementioned priority queues supports *delete* within the working-set bound. In Section 4, we present a priority queue that supports *insert* in worst-case constant time, and supports *delete*, *delete-min*, and *decrease-key* in $O(\lg(w_x + 2))$ worst-case time.

One natural sequence of operations is the first-in-first-out type. Data structures sensitive to these sequences must operate fast on elements that have been least recently accessed. This distribution-sensitive property is referred to as the "queueish" property in [9]. In the context of priority queues, such property states that the time to perform *delete* or *delete-min* on an element x is $O(\lg(q_x + 2))$, where q_x is the number of elements inserted before x and are still present in the priority queue when x is deleted. Note that $q_x = n - w_x$, where n is the number of elements currently present in the priority queue. It is shown in [9] that no binary search tree can be sensitive to this property. However, a priority queue with the queueish property is presented in the same paper [9].

It remained open whether there exists a priority queue sensitive to both the working-set and the queueish properties. We resolve the question affirmatively by presenting such a priority queue in Section 5. In Section 6, we introduce the time-finger property for priority queues. This property encapsulates the queueish and the working-set properties, and thus also captures the unified-bound, static-optimality, and static-finger properties. As these properties are the complete list of the distribution-sensitive properties known for priority queues, we refer to the time-finger property as the "unifying property for priority queues". In consequence, we modify our priority queue to satisfy this unifying property.

2 From the Working-Set Bound to the Unified Bound

Consider a sufficiently long sequence of access operations $X = x_1, x_2 \ldots, x_m$ performed on a given data set of elements.

The *static-finger* property [10] indicates that, for any fixed item f (the finger), the amortized time to perform x_i is $O(\lg(d_X(i, f) + 2))$, where $d_X(i, f)$ is the difference in order within the data set between the item corresponding to x_i and the finger f. More specifically, for an access sequence X, the total access time for a structure with the static-finger property is $O(\sum_{i=1}^{m} \lg(d_X(i, f) + 2))$.

The *static-optimality* property (entropy bound) [10] indicates that, for an access sequence X, where the item corresponding to x_i is accessed $h_X(i)$ times, the total access time is $O(\sum_{i=1}^{m} \lg(\frac{m}{h_X(i)} + 1))$.

The *working-set* size $w_X(i)$, for an operation x_i in sequence X, is defined as the number of distinct elements accessed since the last access to the item corresponding to x_i, or from the beginning of the sequence if this is the first access to x_i. The working-set property [10] indicates that the total access time for a sequence X is $O(\sum_{i=1}^{m} \lg(w_X(i) + 2))$.

Iacono [8] observed that the working-set property implies the static-optimality and static-finger properties. Therefore, the working-set property is the strongest of the three properties. However, the *unified bound* [10] indicates an apparently (but not indeed) stronger property. The unified bound states that the total access time for a sequence X and any fixed finger f is

$$O\left(\sum_{i=1}^{m} \lg \min\left\{d_X(i, f) + 2,\ \frac{m}{h_X(i)} + 1,\ w_X(i) + 2\right\}\right). \tag{1}$$

For the rest of this section, we show that the working-set bound is asymptotically equivalent to the unified bound. First, we show that the sum of the working-set bounds of two sequences is asymptotically the same as the bound for the sequence that results when those two sequences are arbitrarily interleaved. This result will be needed to prove the main claim of this section, and is interesting in its own right.

Theorem 1. *Let X be an access sequence and let Y and Z be two subsequences that partition X. Stated another way, X is an interleaving of Y and Z. Then,*

$$\sum_{i=1}^{|X|} \lg(w_X(i) + 2) = \Theta\left(\sum_{i=1}^{|Y|} \lg(w_Y(i) + 2) + \sum_{i=1}^{|Z|} \lg(w_Z(i) + 2)\right).$$

Proof. The $\Omega(.)$ direction is immediate, and thus we focus on the $O(.)$ direction.

We use η_i to map the indices of Y to X: y_i is the operation that corresponds to x_{η_i}. The function β_i is analogously defined to map the indices of Z to X.

Let ℓ be the largest index such that $\ell < i$ and x_i and x_ℓ access the same element. That is, x_ℓ is the previous access to the same element as that accessed by x_i. Let $W_X(i)$ be the set of indices r such that x_r is the first access to that element in the range $x_{\ell+1} \,..\, x_{i-1}$. Note that $W_X(i)$ is the set envisioned by the concept of the "working set" of x_i and is constructed so that $|W_X(i)| = w_X(i)$.

Observe that the t-th largest index in $W_X(i)$ has a working-set size at least $w_X(i) - t$. Let W'_X be the set of the $\lceil w_X(i)/2 \rceil$ largest indices of W_X. Therefore, the working-set size for each index in W'_X is at least $\lfloor w_X(i)/2 \rfloor$.

Let A_i be formally defined as:

$$A_i = \left\{ j \mid w_X(\eta_j) > w_Y^2(j) \text{ and } i = \lfloor \lg(w_X(\eta_j) + 2) \rfloor \right\}.$$

That is, the set A_i consists of the indices of the access operations in Y for which the logarithm of the working-set size of the corresponding access in X is more than double that value for the access in Y, and in addition the working-set size in X is in the range $[2^i \,..\, 2^{i+1})$. All sets A_i are thus disjoint, and all indices j in $[1 \,..\, |Y|]$ are in some set A_i unless the logarithm of the working-set size for y_j at most doubles as a result of the merge with Z.

Now, pick some index $j \in A_i$. Consider the indices in $W_X(\eta_j)$ and $W'_X(\eta_j)$. Some of these indices come from Y, and some from Z. However, the vast majority come from Z, since the total number is at least the number from Y squared. Very conservatively, at least half of the indices in $W'_X(\eta_j)$ correspond to the Z sequence. We say that these indices of Z are covered by the element j at level i; this set is represented by $C_i(j)$ and is defined as:

$$C_i(j) = \{k \mid \beta_k \in W'_X(\eta_j)\}.$$

We can bound $|C_i(j)|$ from above as:

$$|C_i(j)| \le |W_X(\eta_j)| = w_X(\eta_j) < 2^{i+1},$$

and it can be bounded from below as:

$$|C_i(j)| \ge \frac{1}{2}|W'_X(\eta_j)| \ge \frac{1}{4}w_X(\eta_j) \ge \frac{1}{4}2^i = 2^{i-2}.$$

By construction, for any fixed value k, there are less than 2^{i+1} indices j such that $k \in C_i(j)$. Let C_i be the covered set of the union of all $C_i(j)$, for all $j \in A_i$. It follows that $|C_i| \ge (2^{i-2} \cdot |A_i|)/2^{i+1} = |A_i|/8$.

Also, following the fact that $|C_i(j)| = \Omega(2^i)$, each index $k \in C_i(j)$ has a working-set size of $w_Z(k) = \Omega(2^i)$. Hence,

$$\sum_{i=1}^{\infty} |C_i| \cdot i = O(\sum_{k=1}^{|Z|} \lg(w_Z(k) + 2)).$$

Putting this information together gives

$$\begin{aligned}\sum_{j=1}^{|Y|} \lg(w_X(\eta_j)+2) - \sum_{j=1}^{|Y|} 2\lg(w_Y(j)+2) &= O(|Y| + \sum_{i=1}^{\infty} |A_i| \cdot i) \\ &= O(|Y| + \sum_{i=1}^{\infty} |C_i| \cdot i) \\ &= O\left(|Y| + \sum_{k=1}^{|Z|} \lg(w_Z(k)+2)\right).\end{aligned}$$

Analogously, we have

$$\sum_{k=1}^{|Z|} \lg(w_X(\beta_k)+2) - \sum_{k=1}^{|Z|} 2\lg(w_Z(k)+2) = O\left(|Z| + \sum_{j=1}^{|Y|} \lg(w_Y(j)+2)\right).$$

Adding the last two equations, we get

$$\sum_{j=1}^{|Y|} \lg(w_X(\eta_j)+2) + \sum_{k=1}^{|Z|} \lg(w_X(\beta_k)+2) = O\left(\sum_{j=1}^{|Y|} \lg(w_Y(j)+2) + \sum_{k=1}^{|Z|} \lg(w_Z(k)+2)\right).$$

The theorem follows because

$$\sum_{j=1}^{|Y|} \lg(w_X(\eta_j)+2) + \sum_{k=1}^{|Z|} \lg(w_X(\beta_k)+2) = \sum_{i=1}^{|X|} \lg(w_X(i)+2). \qquad \square$$

Now, we prove the main result of the section.

Theorem 2. *The working-set bound is asymptotically equivalent to the unified bound.*

Proof. Clearly, the unified bound implies the working-set bound as

$$\sum_{i=1}^{m} \lg \min\left\{d_X(i,f)+2,\ \frac{m}{h_X(i)}+1,\ w_X(i)+2\right\} \le \sum_{i=1}^{m} \lg(w_X(i)+2).$$

The other direction requires more effort. We begin by partitioning X into two subsequences Y and Z, by placing in Y all operations x_i where $w_X(i)+2 = \min\left\{d_X(i,f)+2,\ \frac{m}{h_X(i)}+1,\ w_X(i)+2\right\}$ and the remainder in Z. We use η_i to map the indices of Y to X: y_i is the element that came from x_{η_i}. The function β_i is analogously defined to map the indices of Z to X. Then:

$$\sum_{i=1}^{m} \lg \min \left\{ d_X(i,f)+2,\ \frac{m}{h_X(i)}+1,\ w_X(i)+2 \right\} \tag{2}$$

$$= \sum_{i=1}^{|Y|} \lg(w_X(\eta_i)+2) + \sum_{i=1}^{|Z|} \lg \min \left\{ d_X(\beta_i,f)+2,\ \frac{m}{h_X(\beta_i)}+1 \right\} \tag{3}$$

$$= \Omega\left(\sum_{i=1}^{|Y|} \lg(w_Y(i)+2) + \sum_{i=1}^{|Z|} \left(1 + \lg \frac{1}{\max\left\{ \frac{1}{d_X(\beta_i,f)+1}, \frac{h_X(\beta_i)}{m} \right\}} \right) \right) \tag{4}$$

$$= \Omega\left(\sum_{i=1}^{|Y|} \lg(w_Y(i)+2) + \sum_{i=1}^{|Z|} \left(1 + \lg \frac{1}{\max\left\{ \frac{1}{(d_X(\beta_i,f)+1)^2}, \frac{h_X(\beta_i)}{m} \right\}} \right) \right) \tag{5}$$

$$= \Omega\left(\sum_{i=1}^{|Y|} \lg(w_Y(i)+2) + \sum_{i=1}^{|Z|} \lg\left(\frac{|Z|}{h_Z(i)}+1 \right) \right) \tag{6}$$

$$= \Omega\left(\sum_{i=1}^{|Y|} \lg(w_Y(i)+2) + \sum_{i=1}^{|Z|} \lg(w_Z(i)+2) \right) \tag{7}$$

$$= \Omega\left(\sum_{i=1}^{m} \lg(w_X(i)+2) \right) \tag{8}$$

Equation 3 splits one sum into two using the partitioning of X into Y and Z. The left term of Equation 4 is obtained by replacing w_X with w_Y, which can only cause a decrease. The right sum of Equation 4 follows from the fact that $\min(x,y) = \frac{1}{\max(1/x,1/y)}$. The only change in Equation 5 is the square in the denominator; because this is inside a logarithm it makes no asymptotic difference. The formula $1/\max\left\{ \frac{1}{(d_X(\beta_i,f)+1)^2}, \frac{h_X(\beta_i)}{m} \right\}$ now has two nice properties: First, it depends solely on the element corresponding to x_{β_i} (not on the value of i). Second, summed over all distinct elements, the sum is $O(1)$. Hence, this is a static-optimality type weighting scheme. It follows from information theory that the second sum of Equation 6 is asymptotically bounded from below by the entropy of the access frequencies of the elements, and this yields Equation 6. To get from Equation 6 to Equation 7, the fact that the static-optimality bound is big-Omega of the working-set bound is used; this is Theorem 10 of [8]. Moving from Equation 7 to Equation 8 requires the observation that the sum of the working-set bounds of two sequences is asymptotically the same as the working-set bound of an interleaving sequence of both; this is Theorem 1. □

3 A Priority Queue with the Working-Set Property

Our priority queue builds on the priority queue in [5], which supports *insert* in constant time and *delete-min* within the working-set bound. The advantage

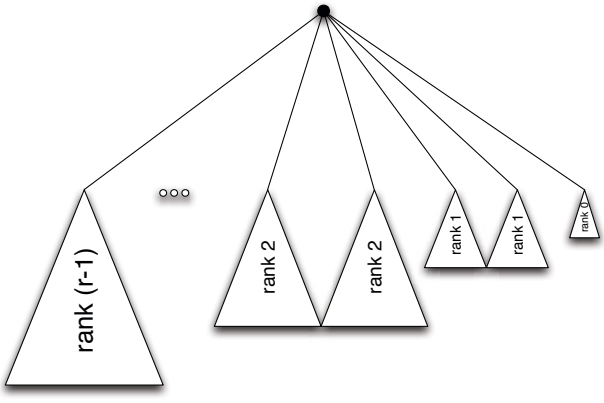


Fig. 2. The recursive structure of a $(2,3)$ binomial tree of rank r: Subtrees rooted at the children of the root are $(2,3)$ binomial trees. The sequence of ranks of such children forms a non-decreasing sequence from right to left such that each value from $0, 1, \ldots, r-1$ occurs either once or twice.

of the priority queue in [5] over those in [2,6,7] is that it satisfies the stronger working-set property, in which elements that are deleted do not count towards the working set. Next, we outline the structure of this priority queue.

The priority queue in [5] comprises heap-ordered $(2,3)$ binomial trees. As defined in [5], the subtrees of the root of a $(2,3)$ binomial tree of rank r are $(2,3)$ binomial trees; there are one or two children having ranks $0, 1, \ldots, r-1$, ordered in a non-decreasing rank order from right to left. It is easy to verify that the rank of any $(2,3)$ binomial tree is $\Theta(\lg n)$, where n is the number of nodes. Figure 2 illustrates the recursive structure of a $(2,3)$ binomial tree.

The ranks of the $(2,3)$ binomial trees of the priority queue are as well non-decreasing from right to left. For the amortized solution, there are at most two (possibly zero) trees per rank. For the worst-case solution, the number of trees per rank obey an extended-regular number system that imposes stronger regularity constraints, which implies that the ranks of any two adjacent trees differ by at most 2 (see [5] for the details of the number system). The root of every $(2,3)$ binomial tree has a pointer to the root with the minimum value among the roots to its left, including itself. Such *prefix-minimum* pointers allow for finding the overall minimum element in constant time, with the ability to maintain such pointers after deleting the minimum in time proportional to the rank of the deleted node. Figure 3 illustrates the structure of our priority queue.

Two primitive operations are *split* and *join*. A $(2,3)$ tree of rank r is split to two or three trees of rank $r-1$; this is done by detaching the one or two children of the root having rank $r-1$. On the other hand, two or three $(2,3)$ trees of rank $r-1$ can be joined to form a $(2,3)$ tree of rank r; this is done by making the root(s) with the larger value the leftmost child(ren) of the one with the smallest value. To join a tree of rank $r-1$ and a tree of rank $r-2$, we split the first tree then join all the resulting trees; the outcome is a tree whose rank is either $r-1$

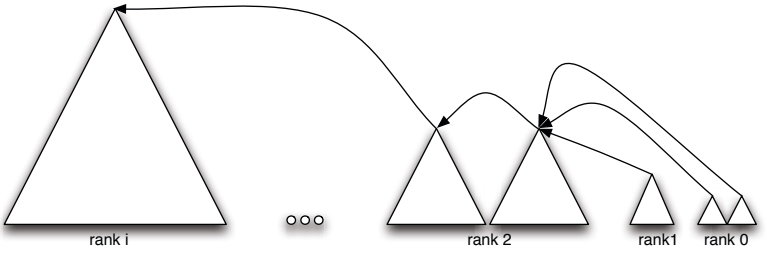


Fig. 3. $(2,3)$ binomial trees comprise the priority queue: The rank of the trees are non-decreasing from right to left. Prefix-minimum pointers are maintained at the roots of the trees. Each tree root points to the minimum root to the left of it, including itself.

or r. With these operations in hand, it is possible to detach the root of a $(2,3)$ binomial tree of rank r and reconstruct the tree again as a $(2,3)$ binomial tree with rank $r-1$ or r; this is done by repeated joins and splits starting from the rightmost subtrees of the deleted root to the leftmost (see [5] for the details).

A total order is maintained indicating the time the elements were inserted. We impose that if a binomial tree T_1 is to the right of another T_2, then all elements in T_1 must have been inserted after those in T_2. Furthermore, within an individual binomial tree, we aim that the preorder traversal of elements with a right-to-left precedence to subtrees would indicate the insertion time of these elements. However, when performing join operations, we occasionally disobey this ordering by possibly reversing the order of two entire subtrees. To get the right ordering, it is enough to maintain a *reverse bit* with every node x; such reverse bit indicates whether the elements in x's subtree were inserted before or after the elements in x's parent plus those in the descendants of the right siblings of x. When a join is performed, the corresponding reverse bit is properly set.

To perform an *insert* operation, a new single node is added as the rightmost tree in the priority queue. This may give rise to several joins once there are three trees with the same rank; the number of such joins is amortized constant, resulting in a constant amortized cost per insertion. After performing the joins, the prefix-minimum pointer of the surviving root is updated. For the worst-case solution, the underlying number system guarantees at most two joins per *insert*.

To perform a *delete-min* operation, the tree T of the minimum root is identified via the prefix-minimum pointer of the rightmost root, the tree T is reconstructed as a $(2,3)$ binomial tree after detaching its root. This may be followed by a split and a join if T has rank one less that its original rank. Finally, the prefix-minimum pointers are updated. For the amortized solution, several splits of T may follow the delete-min operation. Starting with T, we repeatedly split the rightmost tree resulting from previous splits until such tree and its right neighbor (the right neighbor of T before the *delete-min*) have consecutive ranks; this splitting is unnecessary for the worst-case solution. It is not hard to conclude that the cost of *delete-min* is $O(r)$, where r is the rank of the deleted node. In the worst-case solution, the rank of the deleted node x is $O(\lg(w_x+2))$. This follows from the fact that there are $O(w_x)$ elements in the trees to the right

of T, and hence the number of such trees is $O(\lg(w_x + 2))$. For the amortized solution, an extra lemma (see [5]) proves the same bound in the amortized sense.

4 Supporting *delete*, *find-min* and *decrease-key*

The existing distribution-sensitive priority queues [2,5,7,8,9] do not support *delete* within the working-set bound. In this section, we modify the priority queue outlined in Section 3 to support *delete* within the working-set bound. Including *delete* in the repertoire of operations is not hard but should be done carefully. The major challenge is to maintain the chronological order of the elements.

We start by traversing upwards via the parent pointers from the node x to-be-deleted until the root of the tree of x is reached. We use two stacks; a right stack and a left stack. Starting at this root, the current subtree is repeatedly split into two or three trees, one or two of them are pushed to one of the stacks (depending on the reverse bits) while continuing to split the tree that contains x, until we end up with a tree whose root is x. At this stage, we delete x analogously to the *delete-min* operation; the node x is detached and the subtrees resulting from removing x are incrementally joined from right to left, while possibly performing one split before each join (similar to the *delete-min* operation).

We now have to work our way up to the root of the tree and merge all subtrees which we have introduced by splits on the way down from the root. The one or two trees that have the same rank are repeatedly popped from the stacks and joined with the current tree, while possibly performing one split before each join (as required for performing a join operation). Once the two stacks are empty, a split and a join may be performed if the resulting tree has rank one less that its original rank (again, similar to the *delete-min* operation).

The total order is correctly maintained by noting that the only operations employed are the split and join, which are guaranteed to set the reverse bits correctly. Since the height of a $(2, 3)$ binomial tree is one plus its rank, the time bound for *delete* is $O(r)$, where r is the rank of the tree that contains the deleted node. This establishes the same time bound as that for *delete-min* in both the amortized and worst-case solutions.

Using the prefix-minimum pointers, the *find-min* operation is easily performed in constant time. However, this operation is not considered as an access to the minimum element. As otherwise, a following *delete* operation would have to be supported in constant time by the working-set property, which is impossible. The version of *find-min* that is considered as an access to the minimum element can not be supported in time asymptotically less than that for the *delete-min* operation. It is straightforward to implement such an operation in asymptotically optimal time by executing a *delete-min* followed by a re-insertion.

Also, the *decrease-key* operation that is considered as an access cannot be performed in asymptotically less time than a *delete* operation. As otherwise, a *delete* operation can be performed by executing a *decrease-key* operation with decrease value of zero, which essentially brings the element to the front of the working set, followed by a *delete* operation which now runs in constant time as

the element has just been accessed. A *decrease-key* operation can be made to run in $O(\lg(w_x + 2))$ time by simply executing a *delete* followed by a re-insertion.

Finally, the time optimality of *delete-min* and *delete* operations directly follows from the logarithmic lower bound of these operations for (non distribution-sensitive) priority queues in the comparison model.

Theorem 3. *The priority queue presented in this section performs insert in constant time, and delete, delete-min, find-min and decrease-key of an element x in asymptotically optimal time of $O(\lg(w_x + 2))$, where w_x is the number of elements accessed after the last access of x and are still present in the priority queue at the time of the current operation.*

5 Incorporating the Queueish Property

The queueish property for priority queues states that the time to perform *delete* or *delete-min* on an element x is $O(\lg(n - w_x + 2))$, where n is the number of elements currently present in the priority queue, and w_x is the number of elements accessed after the last access of x and are still present in the priority queue. In other words, the queueish property states that the time to perform *delete* or *delete-min* on an element x is $O(\lg(q_x + 2))$, where $q_x = n - w_x$ is the number of elements last accessed prior to x and are still present. Queaps [9] are queueish priority queues that support *insert* in amortized constant time and support *delete-min* of an element x in amortized $O(\lg(q_x + 2))$ time.

We extend our priority queue, in addition to supporting the working-set bound, to also support the operations within the queueish bound. Accordingly, the priority queue simultaneously satisfies both the working-set and the queueish properties. Instead of having the ranks of the trees of the queue non-decreasing from right to left, we split the queue in two sides, a right queue and a left queue, forming a two-sided priority queue. The ranks of the trees of the right queue are monotonically non-decreasing from right to left (as in the previous section), and those of the left queue are monotonically non-decreasing from left to right. We also impose the constraint that the difference in rank between the largest tree on each side is at most one. Figure 4 depicts the new priority queue.

The prefix-minimum pointers in the left and right queues are kept independently. In the right queue, the root of each tree maintains a pointer to the root with the minimum value among those in the right queue to the left of it. Conversely, in the left queue, the root of each tree maintains a pointer to the root with the minimum value among those in the left queue to the right of it. To find the overall minimum value, both the left and right queues are probed.

Insertions are performed exactly as before in the right queue. The *delete-min* operation is performed in the left or right queue depending on where the minimum lies. Deletions are also performed as mentioned in the previous section.

However, we must maintain the invariant that the difference in rank between the largest tree in the left and right sides is at most one. Since the total chronological order is maintained among our trees, this invariant guarantees that the rank of the tree of an element x is $O(\lg(\min\{w_x, q_x\} + 2))$. As a result of an

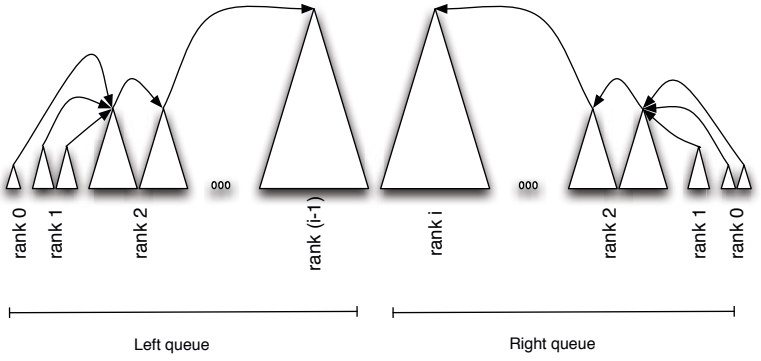


Fig. 4. A priority queue satisfying the queueish property: the priority queue comprises two queues one of which has tree ranks increasing from right to left (right queue) and one increasing from left to right (left queue). The ranks of the largest trees on the two sides must differ by at most one. The prefix-min pointers are separate for each side.

insertion or a deletion, the difference in such ranks may become two. Once the largest rank on one side is two more than that on the other side, the trees with such largest rank are split each in two or three trees, and the appropriate tree among the resulting ones is moved to the other side, increasing the largest rank on the second side by one. As a result of those splits, the number of trees of the same rank on the first side may now exceed the limit, and hence one or two joins would be needed to satisfy the constraints. Once a tree is moved from one side to the other, the prefix-minimum pointers of the priority queues on both sides need to be updated. Because such action happens only after a lot of operations (a fraction of the current number of elements), updating the prefix-minimum pointers only accounts for an extra constant in the amortized cost per operation. If we want to guarantee the costs in the worst case, updating those prefix-minimum pointers is to be done incrementally with the upcoming operations.

A deletion of a node x in a tree of rank r would still cost $O(r)$ time, but now $r = O(\lg(\min\{w_x, q_x\} + 2))$ in the amortized sense (for the amortized solution), or in the worst-case sense (for the worst-case solution).

Theorem 4. *The priority queue presented in this section performs insert in constant time, and delete, delete-min, find-min and decrease-key of an element x in asymptotically optimal time of $O(\lg(\min\{w_x, q_x\} + 2))$, where w_x and q_x are the number of elements accessed after, respectively before, the last access to x and are still present in the priority queue at the time of the current operation.*

6 Supporting Multiple Time Fingers

We define time fingers $t_1, t_2, \ldots, t_c$ as time instances within the sequence of operations, which are freely set by the implementer. We define the working-set of an element x with respect to time finger t_i, $w_x(t_i)$, as the number of elements that have been last accessed in the window of time between the last

access of x and t_i and are still present in the priority queue. We say that a priority queue satisfies the multiple-time-finger property if the time to access x is $O(\lg(\min_{i=1}^{c}\{w_x(t_i)\}+2))$. It is not hard to see that the working-set property is equivalent to having a single time finger $t_1 = +\infty$, and the queueish property is equivalent to having a single time finger $t_1 = 0$. The priority queue presented in Section 5, which supports both the working-set and the queueish properties, has two time fingers $t_1 = 0, t_2 = +\infty$. In this section, we present a priority queue that satisfies the property for a constant number of time fingers.

The structure consists of multiple two-sided priority queues, as those designed in Section 5. We start with a single two-sided priority queue PQ_0, and at each point when a new time finger is introduced we finalize the priority queue and start a new one. Therefore, corresponding to c time-fingers $t_1 = 0, \ldots, t_c = \infty$, we have $c-1$ two-sided priority queues $PQ_1, \ldots, PQ_{c-1}$.

Insertions are performed in the last (at the time when the insertion is performed) priority queue, and, following Theorem 4, takes constant time each. For *delete* operations, we are given a reference to an element x to delete. We determine to which priority queue PQ_j the element belongs and delete it. This requires $O(\lg(\min\{w_x(t_j), w_x(t_{j+1})\}+2))$ time, as indicated by Theorem 4. Since x belongs to PQ_j, for any $i < j$, $w_x(t_j) \leq w_x(t_i)$, and for any $i > j+1$, $w_x(t_{j+1}) \leq w_x(t_i)$. It follows that $\lg(\min\{w_x(t_j), w_x(t_{j+1})\}+2) = \lg(\min_{i=1}^{c}\{w_x(t_i)\}+2)$. For the *delete-min* operation, it suffices to note that finding the minimum element per queue takes constant time. Therefore, we can determine in constant time the priority queue containing the minimum, and perform the operation there. The running-time argument is the same as that for the *delete* operation.

Theorem 5. *Given a constant number of time fingers $t_1, t_2, \ldots, t_c$, the priority queue presented in this section performs insert in constant time, and delete, delete-min, find-min, and decrease-key of an element x in $O(\lg(\min_{i=1}^{c}\{w_x(t_i)\}+2))$ time, where $w_x(t_i)$ is the number of elements that have been last accessed in the window of time between the last access of x and time t_i and are still present in the priority queue at the time of the current operation.*

7 Conclusion

We gave a hierarchy of distribution-sensitive properties in Figure 1. We proved that the working-set and the unified bounds are asymptotically equivalent.

Our focus was on distribution-sensitive priority queues. Provably, priority queues cannot satisfy the sequential-access property, and in accordance neither the dynamic-finger nor the unified conjecture. We therefore considered other distribution-sensitive properties, namely: the working-set and the queueish properties. We presented a priority queue that satisfies both properties. Our priority queue builds on the priority queue of [5], which supports *insert* in constant time and *delete-min* in the working-set time bound. We showed that the same structure can also support *delete* operations within the working-set bound. We then

modified the structure to satisfy the queueish property as well. It is worthy to note that the priority queue designed supports the stronger definition of the working-set and the queueish properties in which the elements deleted are not accounted for in the time bounds.

Our result about the asymptotic equivalence of the working-set bound and the unified bound then implies that our priority queue also satisfies the unified, static-optimality and static-finger bounds. We defined the notion of time fingers, which encapsulates the working-set and the queueish properties. The priority queue described thus far has two time fingers. We extended our priority queue to possibly support any constant number of time fingers.

The bounds mentioned are amortized. However, we showed that the time bounds for the working-set and queueish properties can also be made to work in the worst case. More generally, the multiple time-finger bounds can be made to work in the worst case. However, the time bounds for other properties: unified bound, static optimality, and static finger, naturally remain amortized.

References

1. Bdoiu, M., Cole, R., Demaine, E.D., Iacono, J.: A Unified Access Bound on Comparison-based Dynamic Dictionaries. Theoretical Computer Science 382(2), 86–96 (2007)
2. Brodal, G.S., Fagerberg, R.: Funnel Heap - a Cache Oblivious Priority Queue. In: Bose, P., Morin, P. (eds.) ISAAC 2002. LNCS, vol. 2518, pp. 219–228. Springer, Heidelberg (2002)
3. Cole, R.: On the Dynamic Finger Conjecture for Splay Trees. Part II: Finger Searching. SIAM Journal on Computing 30, 44–85 (2000)
4. Elmasry, A.: On the Sequential Access Theorem and Dequeue Conjecture for Splay Trees. Theoretical Computer Science 314(3), 459–466 (2004)
5. Elmasry, A.: A Priority Queue with the Working-set Property. International Journal of Foundation of Computer Science 17(6), 1455–1466 (2006)
6. Fredman, M.L., Sedgewick, R., Sleator, D.D., Tarjan, R.E.: The Pairing Heap: a New Form of Self-adjusting Heap. Algorithmica 1(1), 111–129 (1986)
7. Lacono, J.: Improved Upper Bounds for Pairing Heaps. In: Halldórsson, M.M. (ed.) SWAT 2000. LNCS, vol. 1851, pp. 32–45. Springer, Heidelberg (2000)
8. Iacono, J.: Distribution-sensitive Data Structures. Ph.D. thesis, Rutgers, The state University of New Jersey, New Brunswick, New Jersey (2001)
9. Iacono, J., Langerman, S.: Queaps. Algorithmica 42(1), 49–56 (2005)
10. Sleator, D.D., Tarjan, R.E.: Self-adjusting Binary Search Trees. Journal of the ACM 32(3), 652–686 (1985)
11. Tarjan, R.E.: Sequential Access in Splay Trees Takes Linear Time. Combinatorica 5(4), 367–378 (1985)

Enumerating Tatami Mat Arrangements of Square Grids

Alejandro Erickson[1] and Mark Schurch[2]

[1] Department of Computer Science, University of Victoria, V8W 3P6, Canada
[2] Mathematics and Statistics, University of Victoria, V8W 3R4, Canada
{ate,mschurch}@uvic.ca

Abstract. We prove that the number of monomer-dimer tilings of an $n \times n$ square grid, with $m < n$ monomers in which no four tiles meet at any point is $m2^m + (m+1)2^{m+1}$, when m and n have the same parity. In addition, we present a new proof of the result that there are $n2^{n-1}$ such tilings with n monomers, which divides the tilings into n classes of size 2^{n-1}. The sum of these over all $m \leq n$ has the closed form $2^{n-1}(3n-4)+2$ and, curiously, this is equal to the sum of the squares of all parts in all compositions of n.

1 Introduction

Tatami mats are a traditional Japanese floor covering, whose outside is made of soft woven rush straw, and whose core is stuffed with rice straw. They have been in use by Japanese aristocracy since the 12th century, but modern versions of them are now available to the average consumer to be used as floor mats, in lieu of carpeting, or as essential furnishing in a tatami room. The mats are integral to Japanese culture, and are often used to describe the square footage of a room. A standard full mat measures $6' \times 3'$, and a half mat is $3' \times 3'$.

An arrangement of the mats in which no four meet at any point is often preferred because it is said to be auspicious. We call such arrangements monomer-dimer *tatami tilings*. The monomer-dimer tiling in Fig. 1 violates the tatami condition, and the one in Fig. 2 does not.

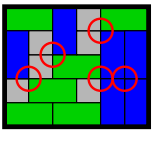


Fig. 1. The tatami condition is that no four tiles may meet at a point. Such violations are circled.

Tilings with polyominoes are well studied, and they appear in physical models, the theory of regular languages, and of course, combinatorics (see [2, 12, 4, 8, 5]).

C.S. Iliopoulos and W.F. Smyth (Eds.): IWOCA 2011, LNCS 7056, pp. 223–235, 2011.
© Springer-Verlag Berlin Heidelberg 2011

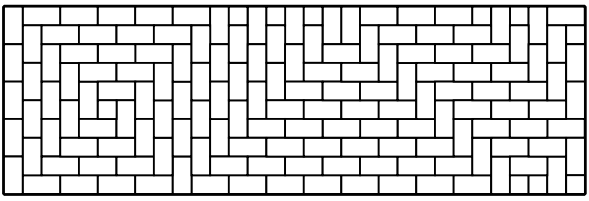


Fig. 2. A tatami tiling of the 10×31 grid with 10 monomers, 64 vertical dimers and 86 horizontal dimers. Compare this with Fig. 6.

Considerable credit is owed to these and other authors for the popularity of monomer-dimer tilings, particularly of their enumeration.

This begs the question, given a room's size, how many different auspicious arrangements of mats are there? The dimer-only version of the problem appears in the fourth volume of The Art of Computer Programming ([7]), a series named by American Scientist as one of the 100 most influential over a century of science ([9]). Motivated by the exercise posed by Prof. Knuth, our co-authors from [3] published the ordinary generating functions for fixed height, dimer-only tatami tilings in [11], and their solution appears in [7].

In Fall 2009 our research group discovered the structure which is published in [3], and described in the next section. Problems such as the enumeration of tatami tilings with r rows, c columns, and m monomers, have since become within reach. This paper completes such an enumeration for square grids.

In the same year, Prof. Ruskey shared a tatami flavoured New Year's greeting with Prof. Knuth, among others. He filled the letters of "Happy New Year" with tatami tilings that illustrate the structure we had discovered (see Fig. 3), and closed by proposing the problem:

> ... perhaps you will have fun proving that the number of such tilings of an $n \times n$ square that maximize the number of monomers is $n2^{n-1}$.

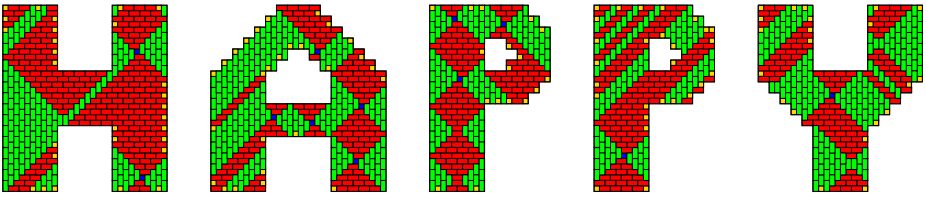


Fig. 3. Part of Prof. Ruskey's new year's greeting for 2010

Before the new decade came about, Ruskey received a reply from Knuth which read:

> Dear Frank,
> I resisted the challenge in your New Year's card (about $2^{n-1}n$) for more than four weeks, but finally realized that I couldn't live any longer without trying to find out what was going on with those tatami tilings.
> I budgeted half a day to explore the problem; and finally figured out enough of the structure to declare victory after two days; but my derivation is not at all simple. Certainly I have no way to group the solutions into, say, n classes of size 2^{n-1} (although I do have lots of classes of solutions of size 2^{n-2}).

Our solution in [3] does not divide the tilings into such classes either, so we remedy the lack of symmetry by presenting a new proof which does. This time we count the tilings directly, rather than recursively.

Alhazov et al. followed up on the aforementioned dimer-only research with a treatment of odd-area tatami tilings which include a single monomer. They closed [1] with this remark:

> However, the variety of tilings with arbitrary number of monominoes is quite "wild" in sense that such tilings cannot be easily decomposed, see Figure 11; therefore, most results presented here do not generalize to arbitrary number of monominoes, the techniques used here are not applicable, and it is expected that any characterization or enumeration of them would be much more complicated.

The structure we found, however, reveals the opposite; the tilings with an arbitrary number of monomers *are* easily decomposed. The decomposition has a satisfying symmetry, it is ammenable to inductive arguments, and it shows that the complexity of a tatami tiling is linear in the dimensions of the grid (compare Fig. 2 with Fig. 6). We use it extensively to prove our main result. Let $T(n, m)$ be the number of $n \times n$ tatami tilings with exactly m monomers.

Theorem 1. *If n and m have the same parity, and $m < n$, then $T(n, m) = m2^m + (m + 1)2^{m+1}$.*

This confirms Conjecture 3 in [3] for $d = 0$ (as was promised there), and completes the enumeration of tatami tilings of square grids, since $T(n, m) = 0$ when $m > n$.

Nice round numbers such as these tend to be easily identified in output generated by exhaustively listing all tilings on a given size of grid. Both our research group and Knuth saw the sequence long before it was proven. Knuth wrote to Ruskey:

> I did happen to notice that even more is true — although I won't have time to prove it. I just have overwhelming empirical evidence for the following theorem, at least up to 12×12 (and I'll soon have much more data because the computer programs are nowhere near any breaking points): The number of monomer-dimer tatami tilings of an $n \times n$ square, in which there are m monomers, where $m < n$ and $m + n$ is even, is

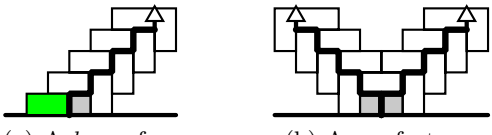
exactly $2^m(3m+2)$. For example, the generating function when $n = 11$ is $10z + 88z^3 + 544z^5 + 2944z^7 + 14848z^9 + 11264z^{11}$, and when $n = 12$ it is $2 + 32z^2 + 224z^4 + 1280z^6 + 6656z^8 + 32768z^{10} + 24576z^{12}$.

Evaluating Knuth's generating functions at $z = 1$ and simplifying reveals that there are $2^{n-1}(3n-4)+2$ tatami tilings of the $n \times n$ grid. As a curious afterthought, we note that this is equal to the sum of the squares of all parts in all compositions of n.

1.1 Structure

Tatami tilings of rectangular grids have an underlying structure, called the *T-diagram*, which is an arrangement of four possible types of *features*, shown in Figures 4-5, up to rotation and reflection (see [3] for a complete explanation). Each feature consists of a *source*, shown in colour, which forces the placement of up to four *rays*. Rays propagate to the boundary of the grid and do not intersect each other.

A valid placement in the grid of a non-zero number of features determines a tiling uniquely. Otherwise, the trivial T-diagram corresponds to the four possible running bond patterns (brick laying pattern).



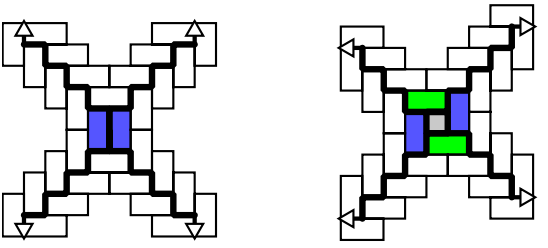
(a) A *loner* feature. (b) A *vee* feature.

Fig. 4. These two types of features must have their coloured tiles on a boundary, as shown, up to rotation and reflection

Monomers occur on the boundary of the grid, and inside vortices, but nowhere else. The T-diagram is also a partition of the tiles into blocks of horizontal and vertical running bond. Wherever horizontal bond meets a vertical boundary of the grid, dimers alternate with monomers, forming a section of *jagged boundary*, and similarly when vertical bond meets horizontal boundary. The tiling in Fig. 6 uses all four types of features, displays this partition into horizontal and vertical running bond, and indicates all instances of jagged boundary. Its T-diagram is shown on its own in Fig. 7.

Throughout this paper we consider tilings on the $n \times n$ integer grid, with the origin at the bottom left of the tiling. Let the coordinate of a grid square be the point at its bottom left corner as well.

A *diagonal* is a rotation of the following: a monomer at $(x_0, 0)$, and (vertical) dimers covering the pairs of grid squares $(x_0 + 1 + k, k)$ and $(x_0 + 1 + k, k + 1)$, for each non-negative k such that $x_0 + 1 + k \leq n - 1$. A diagonal can be *flipped*



(a) A vertical *bidimer* feature.

(b) A counterclockwise *vortex* feature.

Fig. 5. These two types of features may appear anywhere in a tiling provided that the coloured tiles are within the boundaries of the grid

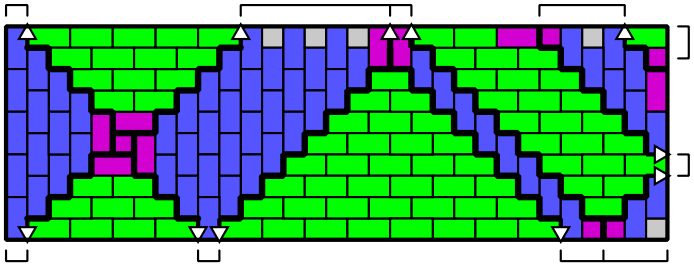


Fig. 6. A tiling showing all four types of sources. Coloured in magenta, from left to right they are, a clockwise vortex, a vertical bidimer, a loner, a vee, and another loner. Jagged boundaries are indicated by brackets.

in place, so that the monomer is *mapped* to $(n-1, n-x_0-1)$, and the dimers change orientation. These equivalent operations preserve the tatami condition. Diagonals are used and illustrated frequently in the next section.

2 Square Grids

Let $T(n, m)$ be the number of $n \times n$ tatami tilings with m monomers.

Theorem 1. *If n and m have the same parity, and $m < n$, then $T(n, m) = m2^m + (m+1)2^{m+1}$.*

Proof. We show that any $n \times n$ tiling with m monomers and $m < n$ has exactly one bidimer or vortex and we show that m is a function of the shortest distance from this source to the boundary. Such a feature determines all tiles in the tiling except a number of diagonals that can be flipped independently. Proving the result becomes a matter of counting the number of allowable positions for the bidimer or vortex. For example, the 20×20 tiling in Fig. 8 has a vertical bidimer which forces the placement of the green and blue tiles, while the remaining

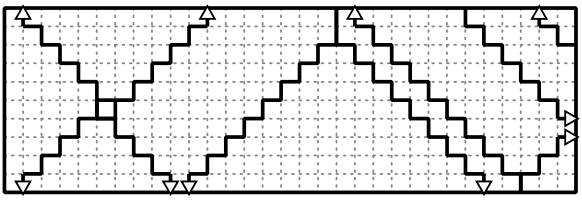


Fig. 7. A T-diagram

diagonals are coloured in alternating grey and magenta. There are eight such diagonals, so there are 2^8 tilings of the 20×20 grid, with a vertical bidimer in the position shown. Each of these 2^8 tilings has exactly 10 monomers.

Let the centers of bidimers and vortices be the crossing point of an X-drawn through them symmetrically, where the origin is the grid point at the bottom left of the grid as in Figures 8-10. Suppose the center of a feature is at the point (x_f, y_f), then without loss of generality, we may re-orient the tiling, by rotating and reflecting, so that $y_f \leq x_f \leq n/2$, to count the monomers and diagonals as follows.

The number of diagonals for each type of bidimer or vortex centered at (x_f, y_f), can be abstracted from the figures, by counting certain grid squares on boundaries. Namely, the grid squares indicated by the brackets filled with grey in Figures 8-10, which are not covered by green or blue dimers. Omitting some details which can be gleaned from Figures 8-10, we tabulate these numbers, abbreviating top, bottom, left, right, and corner with the letters t,b,l,r, and c, respectively.

feature center	positions	grid squares	diagonals	c-monomers
$y_f = x_f$	bl and tr	0	0	0
$y_f < x_f$	bl and tr	$2(x_f - y_f) - 1$	$x_f - y_f - 1$	1
$y_f = x_f = n/2$	tl and br	0	0	0
$y_f + x_f < n$	tl and br	$2(n - x_f - y_f) - 1$	$n - x_f - y_f - 1$	1

The number of monomers is equal to the number of diagonals plus corner-monomers, plus, possibly, a vortex. When $y_f = x_f = n/2$, this number is either 0 or 1, and if $y_f = x_f < n/2$, it is $n - x_f - y_f = n - 2y_f$, or $n - 2y_f + 1$. When $y_f < x_f$, the number of monomers is also $n - 2y_f$ or $n - 2y_f + 1$.

In both the cases, $y_f = x_f$ and $y_f < x_f$, the number of monomers is $n - 2y_f$ if the feature is a bidimer, and $n - 2y_f + 1$ if it is a vortex. Therefore, the number of monomers is decided by the distance from the center of this feature to the nearest boundary of the grid. The number of positions for features whose tilings have m monomers are tabulated below the following explanation.

We count the number of positions for bidimers for a fixed y_f, whose tilings have $n - 2y_f$ monomers as in Fig. 9 and Fig. 8. Abandoning the range restriction on x_f, there are four such positions when $x_f = y_f$, up to rotation, and otherwise

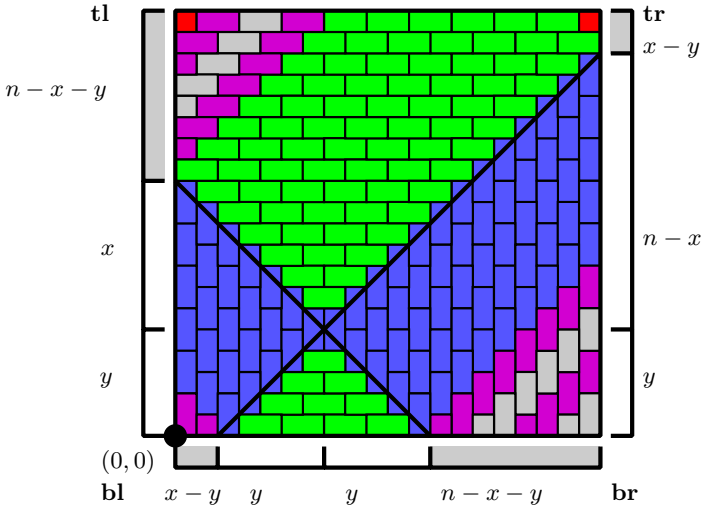


Fig. 8. Vertical bidimer

we count the four rotations of the case where $y_f < x_f < n - y_f$. Thus there are $4(n - y_f - 1 - y_f)$ positions for x_f, and $4(n - 2y_f - 1) = 4(m - 1)$, where m is the number of monomers.

The same argument works for the vortex in Fig. 10, at distance y'_f from the nearest boundary, in spite of the fact that x'_f and y'_f are not integers. Here the number of monomers is $n - 2y'_f + 1$. Once again, the number of positions for the feature is four when $x'_f = y'_f$, and otherwise it is $4(n - 2y'_f - 1)$, which is equal to $4(m - 2)$ when expressed in terms of m.

We tabulate what we have derived above. The words horizontal, vertical, counterclockwise and clockwise are abbreviated as h, v, cc and c, respectively.

Position	Feature	Positions	Diagonals
$x_f = y_f$	h and v bidimers	4	$m - 1$
$x_f = y_f$	cc and c vortices	4	$m - 2$
$x_f < y_f$	h and v bidimers	$4(m - 1)$	$m - 2$
$x_f < y_f$	cc and c vortices	$4(m - 2)$	$m - 3$

Each term in the following sum comes from a row of the table, in the same respective order, and similarly, the three factors in each sum term come from the last three columns of the table.

$$\begin{aligned} T(n,m) =& 2 \cdot 4 \cdot 2^{m-1} + 2 \cdot 4 \cdot 2^{m-2} + 2 \cdot 4(m-1) \cdot 2^{m-2} + 2 \cdot 4(m-2) \cdot 2^{m-3} \\ =& 2 \cdot 2^{m+1} + 2 \cdot 2^m + (m-1)2^{m+1} + (m-2)2^m \\ =& m2^m + (m+1)2^{m+1}. \end{aligned}$$

That is, there are $m2^m$ tilings with vortices and $(m + 1)2^{m+1}$ without. □

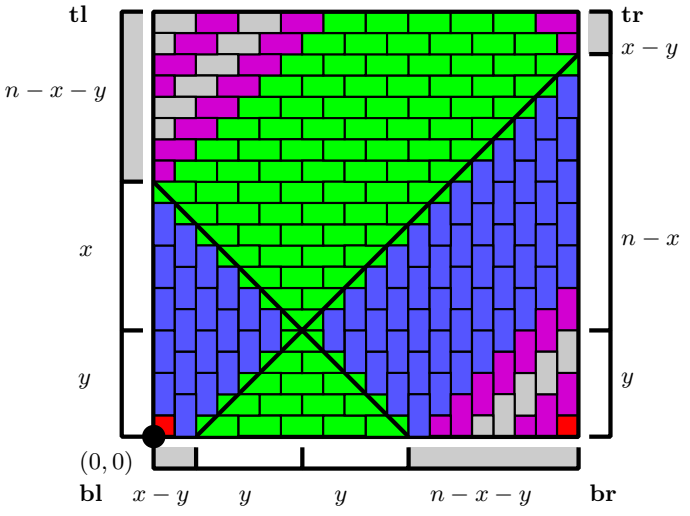


Fig. 9. Horizontal bidimer

This completes the enumeration of $n \times n$ tatami tilings with m monomers.

Let this not be the end of the square tatami tiling story. We are compelled by other mathematicians, including Knuth, to organize the $n \times n$ tatami tilings with n monomers into n classes of size 2^{n-1}. We present a proof which directly counts diagonal flips. The proof in [3] uses induction on the dimensions of the grid to show that $n2^{n-1} = 4S(n)$, where

$$S(n) = 2^{n-2} + 4S(n-2), \text{ where } S(1) = \frac{1}{4} \text{ and } S(2) = 1.$$

The recurrence arises from relating the collisions of diagonals in an $(n-2)\times(n-2)$ tiling with those of an $n \times n$ tiling.

Theorem 2 (Erickson, Ruskey, Schurch, Woodcock, [3]). $T(n, n) = n2^{n-1}$

Proof (n classes of size 2^{n-1}). We use the setup in [3]. Every tatami tiling of the $n \times n$ grid with n monomers can be obtained by performing a set of flips (of diagonals) on a running bond, in which no monomer is moved more than once, and the corner monomers remain fixed. As an immediate consequence, every tiling has exactly two corner monomers which are in adjacent corners, and as such, it has four distinct rotational symmetries.

Let $\mathcal{S}$ be the $n \times n$ tilings with n monomers whose upper corners have monomers. It is sufficient to divide the $n2^{n-3}$ tilings of $\mathcal{S}$, into n classes of size 2^{n-3}, and then use the rotational symmetries of these to obtain the n classes of size 2^{n-1} of all the tilings.

In this context, a flipped diagonal or monomer refers to one which was flipped from its initial position in the running bond, in the set of flips referred to above.

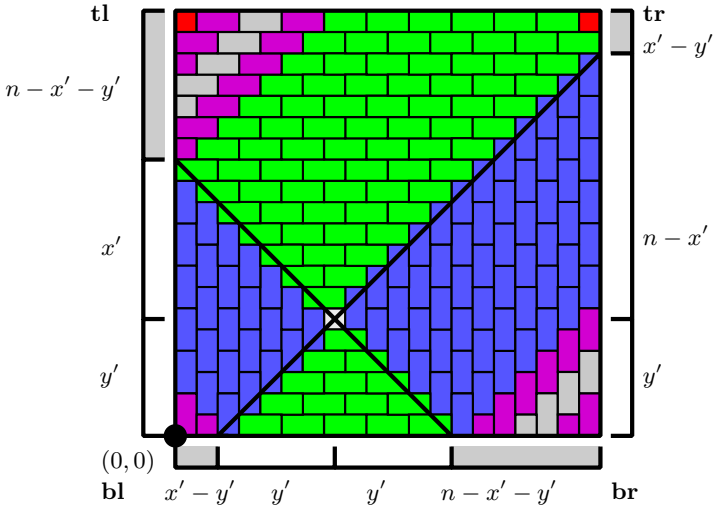


Fig. 10. Counter clockwise vortex. Note that x'_f and y'_f are not integers.

Each monomer is either flipped on its longest or shortest diagonal, or not flipped at all (see Fig. 11). Two monomers flipped on their longest diagonals cannot have been flipped on diagonals of the same length, because any two longest diagonals either interfere with each other, or they have different parities. A tiling in $\mathcal{S}$ has either,

(a) a monomer flipped on (one of) its longest diagonal which is the unique diagonal of maximum length, or
(b) no monomer that is flipped on its longest diagonal.

Suppose n is odd. The tilings in $\mathcal{S}$ are obtained from a vertical running bond with monomers in the top corners, as illustrated for $n = 17$ in Fig. 11. Here, the lengths of a monomer's diagonals, defined by the number of tiles they contain, differ except when the monomer is in the central column of the grid.

Flip a monomer μ on (one of) its longest diagonal, δ, as in Fig. 12(a). There are $n - 3$ monomers that may still be flipped. We claim that if these are flipped only along diagonals strictly shorter than δ, then each monomer has exactly one flip available to it, and the flips can be made independently, so that the number of combinations is 2^{n-3}. This is obviously the case in Fig. 12(a), where the available flips are shown by the diagonal lines.

We set up the general argument. Label each monomer with its x coordinate in the running bond, recalling that a grid square's coordinate is its bottom left corner. The diagonal to its left has size $x + 1$, and the one to the right has size $n - x$. Diagonals interfere with each other on the left boundary if and only if they map monomers from top and bottom boundaries, whose labels α and β sum to more than n. Diagonals interfere on the right if and only if they map monomers

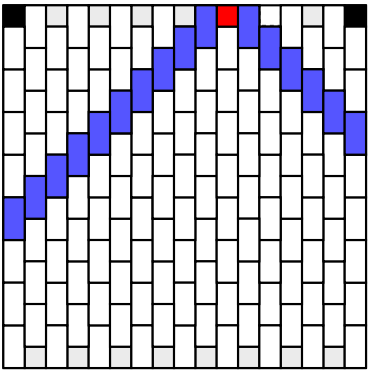
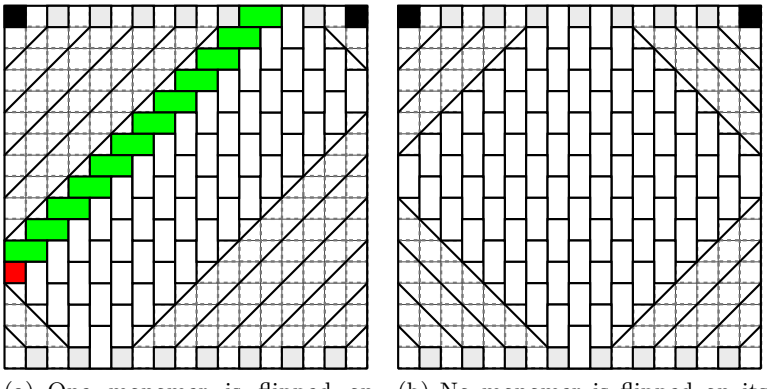


Fig. 11. Each monomer is initially in two diagonals

from top and bottom boundaries which do not interfere on the left. This labeling serves to prove that our statement about Fig. 12(a) holds in general.

There are $n-2$ choices for μ, and when μ is on the central column there are two choices of diagonals to flip it on. This gives $n-1$ classes of size 2^{n-3}.

The case where no monomer is flipped on its longest diagonal gives the nth class of size 2^{n-3}, because neither the middle column monomer nor the two corner monomers are flipped (see Fig. 12(b)). This concludes the odd case.



(a) One monomer is flipped on (one of) its longest diagonal.

(b) No monomer is flipped on its longest diagonal.

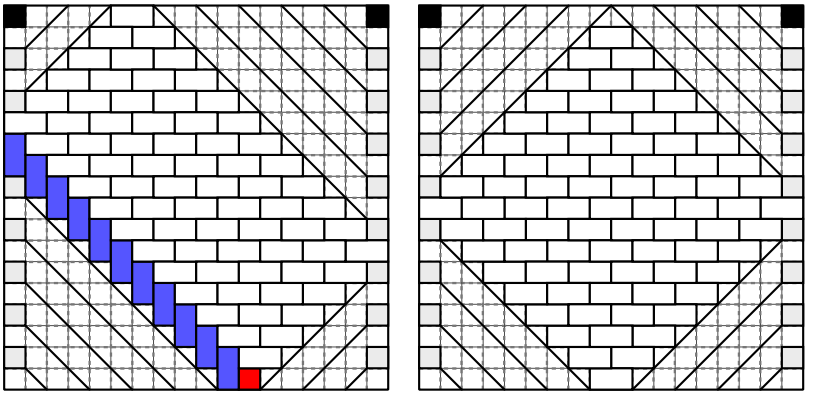
Fig. 12. Odd n

Let n be even. The tilings in $\mathcal{S}$ are obtained from a horizontal running bond, and, contrary to the odd case, all monomers initially have two diagonals whose lengths differ.

Flip a monomer μ on its longest diagonal. Again, there are $n-3$ monomers remaining, and each one has one available diagonal which is strictly shorter than

that of μ, and they can each be flipped on these independently. This can be seen to hold in Fig. 13(a), and a labeling similar to the one described for odd n shows that it holds in general.

The case where no monomer is flipped on its longest diagonal introduces an asymmetry for even n, resulting from the fact that every monomer has two diagonals that differ in length. The class of size 2^{n-2} is easily divided into two classes, however, since we are merely dealing with combinations of an $n-2$ set. □



(a) One monomer is flipped on its longest diagonal.

(b) Every monomer is flipped on its shortest diagonal.

Fig. 13. Even n

It comes as a surprise at first, but the total number of $n \times n$ tilings is equal to the sum of the squares of all parts in all compositions of n. We prove this by showing they satisfy the following expression.

Corollary 1. *The number of $n \times n$ tatami tilings is $2^{n-1}(3n-4)+2$.*

Proof. From [3], we have that $T(n,m) = 0$ when $m > n$. Let $T(n) = \sum_{m \geq 0} T(n,m)$, so that

$$T(n) = n2^{n-1} + \sum_{i=1}^{\lfloor n/2 \rfloor} \left((n-2i)2^{n-2i} + (n-2i+1)2^{n-2i+1} \right),$$

and notice that the sum simplifies to

$$T(n) = n2^{n-1} + \sum_{i=1}^{n-1} i2^i.$$

Now we use the fact that $2^k + 2^{k+1} + \cdots + 2^{n-1} = 2^n - 1 - 2^k + 1 = 2^n - 2^k$ to rearrange the sum.

$$\begin{aligned} T(n) =& n2^{n-1} + \sum_{i=1}^{n-1} i2^i \\ =& n2^{n-1} + (n-1)2^n - \sum_{i=1}^{n-1} 2^i \\ =& 2^{n-1}(3n-4) + 2 \end{aligned}$$

We omit the details of this next, and final curiosity. A reference to the sequence in question can be found in sequence A027992 of the The On-Line Encyclopedia of Integer Sequences ([6]).

Corollary 2. *The number of $n \times n$ tatami tilings is equal to the sum of the squares of all parts in all compositions of n.*

3 Conclusions and Further Research

The number of red squares in the Hasse diagram in [10] is the sum of the squares of all parts in all compositions of n, and it would be aesthetically pleasing to see a natural bijection from the tatami tilings of the $n \times n$ grid to these red squares (see Fig. 14).

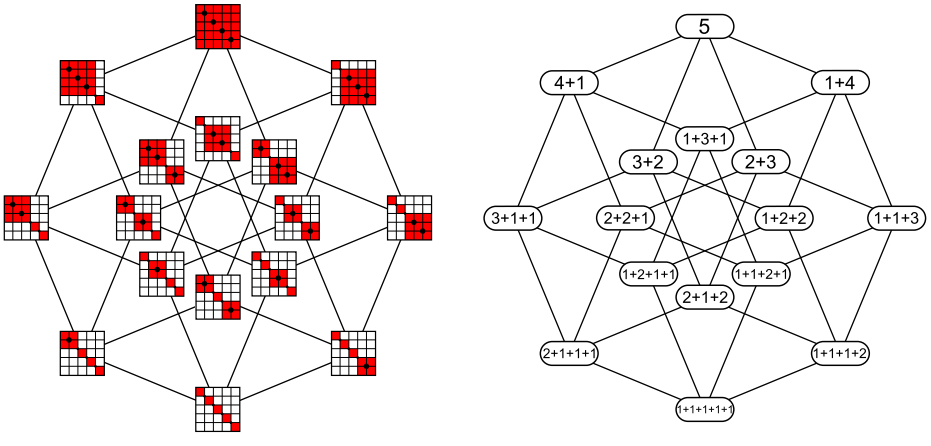


Fig. 14. Is there a mapping from $n \times n$ tatami tilings to red squares?

Perhaps of broader interest is the enumeration of $T(n, m, v, h)$, the number of $n \times n$ tilings with m monomers and v and h vertical and horizontal dimers, respectively. Some progress has been made on this by Knuth, as well as the authors of [3] (Section 4.1, Conjecture 4).

Acknowledgements. The authors are grateful to Prof. Frank Ruskey for his continued participation in tatami research and insightful comments on this paper, as well as Prof. Don Knuth for his valuable communications about tatami tilings. We also thank the anonymous referees for their feedback and pointing out important references on classical tilings.

References

[1] Alhazov, A., Morita, K., Iwamoto, C.: A note on tatami tilings. In: Proceedings of the 2009 LA Winter Symposium Mathematical Foundation of Algorithms and Computer Science, vol. 1691, pp. 1–7 (2010)
[2] Benedetto, K.P., Loehr, N.A.: Tiling problems, automata, and tiling graphs. Theoretical Computer Science 407(1-3), 400–411 (2008)
[3] Erickson, A., Ruskey, F., Schurch, M., Woodcock, J.: Monomer-dimer tatami tilings of rectangular regions. The Electronic Journal of Combinatorics 18(1), 24 (2011)
[4] Gale, D., Golomb, S.W., Haas, R.: Mathematical entertainments. The Mathematical Intelligencer 18(2), 38–47 (1996)
[5] Hock, J.L., McQuistan, R.B.: A note on the occupational degeneracy for dimers on a saturated two-dimensional lattice space. Discrete Applied Mathematics 8(1), 101–104 (1984)
[6] Jovovic, V.: Comment on a027992 (2005), `http://oeis.org/A027992`
[7] Knuth, D.E.: The Art of Computer Programming, vol 4A: Combinatorial Algorithms, Part 1, 1st edn. Addison-Wesley Professional (2011)
[8] Merlini, D., Sprugnoli, R., Cecilia Verri, M.: Strip tiling and regular grammars. Theoretical Computer Science 242(1-2), 109–124 (2000)
[9] Morrison, P., Morrison, P.: 100 or so books that shaped a century of science. American Scientist 87(6), 1 (1999)
[10] Piesk, T.: Binary and compositions 5 (2010), `http://commons.wikimedia.org/wiki/File:Binary_and_compositions_5.svg`
[11] Ruskey, F., Woodcock, J.: Counting fixed-height tatami tilings. The Electronic Journal of Combinatorics 16, 20 (2009)
[12] Stanley, R.P.: On dimer coverings of rectangles of fixed width. Discrete Applied Mathematics 12(1), 81–87 (1985)

Quasi-Cyclic Codes over $\mathbb{F}_{13}$

T. Aaron Gulliver

Department of Electrical and Computer Engineering, University of Victoria
Victoria, BC Canada, V8W 3P6
agullive@ece.uvic.ca

Abstract. Let $d_q(n,k)$ be the maximum possible minimum Hamming distance of a linear $[n,k]$ code over $\mathbb{F}_q$. Tables of best known linear codes exist for all fields up to $q = 9$. In this paper, linear codes over $\mathbb{F}_{13}$ are constructed for k up to 6. The codes constructed are from the class of quasi-cyclic codes. In addition, the minimum distance of the extended quadratic residue code of length 44 is determined.

1 Introduction

Let $\mathbb{F}_q$ denote the Galois field of q elements, and let $V(n,q)$ denote the vector space of all ordered n-tuples over $\mathbb{F}_q$. A linear $[n,k]$ code C of length n and dimension k over $\mathbb{F}_q$ is a k-dimensional subspace of $V(n,q)$. The elements of C are called codewords. The (Hamming) weight of a codeword is the number of non-zero coordinates. The minimum weight of C is the smallest weight among all non-zero codewords of C. The minimum weight of a linear code equals the minimum distance between codewords. An $[n,k,d]$ code is an $[n,k]$ code with minimum weight d. Let A_i be the number of codewords of weight i in C. Then the numbers $A_0, A_1, \ldots, A_n$ are called the weight distribution of C.

A central problem in coding theory is that of optimising one of the parameters n, k and d for given values of the other two. One version is to find $d_q(n,k)$, the largest value of d for which there exists an $[n,k,d]$ code over $\mathbb{F}_q$. Another is to find $n_q(k,d)$, the smallest value of n for which there exists an $[n,k,d]$ code over $\mathbb{F}_q$. A code which achieves either of these values is called *optimal*. Tables of best known linear codes exist for all fields up to $q = 9$ [6]. In this paper, linear codes over $\mathbb{F}_{13}$ are constructed for k up to 6.

The Griesmer bound is a well-known lower bound on $n_q(k,d)$

$$n_q(k,d) \geq g_q(k,d) = \sum_{j=0}^{k-1} \left\lceil \frac{d}{q^j} \right\rceil, \tag{1}$$

where $\lceil x \rceil$ denotes the smallest integer $\geq x$. For $k \leq 2$, the Griesmer bound is met for all q and d. The Singleton bound [13] is a lower bound on $n_q(k,d)$ and is given by

$$n_q(k,d) \geq d + k - 1 \tag{2}$$

C.S. Iliopoulos and W.F. Smyth (Eds.): IWOCA 2011, LNCS 7056, pp. 236–246, 2011.
© Springer-Verlag Berlin Heidelberg 2011

Codes that meet this bound are called maximum distance separable (MDS). MDS codes exist for all values of $n \leq q+1$. Thus for $q = 13$, MDS codes exist for all lengths 14 or less.

For larger lengths and dimensions, far less is known about codes over $\mathbb{F}_{13}$. MDS self-dual codes ($k = n/2$), of lengths 2, 4, 6, 8, 10 and 14 are given in [2], as well as self-dual $[12, 6, 6]$, $[16, 8, 8]$, $[20, 10, 10]$, $[22, 11, 10]$ and $[24, 12, 10]$ codes. de Boer earlier discovered a self-dual $[18, 9, 9]$ code, and $[23, 3, 20]$ and $[23, 17, 6]$ codes [7]. The $[18, 9, 9]$, $[24, 12, 10]$ and $[30, 15, 12]$ extended quadratic residue (QR) codes are given in [14]. Using Magma [3], it was determined that the next extended QR code over $\mathbb{F}_{13}$ has parameters $[44, 22, 16]$. In this paper, codes with dimensions $k = 3 - 6$ are constructed. These codes establish lower bounds on the minimum distance. Many of these meet the Singleton and/or Griesmer bounds, and so are optimal.

A *punctured code* of C is a code obtained by deleting a coordinate from every codeword of C. A *shortened code* of C is a code obtained by taking only those codewords of C having a zero in a given coordinate position and then deleting that coordinate. The following bounds can be established based on these constructions

1) $d_q(n+1, k) \leq d_q(n, k) + 1$,

and

2) $d_q(n+1, k+1) \leq d_q(n, k)$.

Using the codes given in this paper, they provide many additional lower bounds.

The next section presents the class of quasi-cyclic codes, and the construction results are given in Section 3.

2 Quasi-Cyclic Codes

A code C is said to be quasi-cyclic (QC) if a cyclic shift[1] of any codeword by p positions is also a codeword in C [8]. A cyclic code is a QC code with $p = 1$. The length of a QC code considered here is $n = mp$. With a suitable permutation of coordinates, many QC codes can be characterized in terms of $(m \times m)$ circulant matrices. In this case, a QC code can be transformed into an equivalent code with generator matrix

$$G = [R_0 \; R_1 \; R_2 \; ... \; R_{p-1}], \tag{3}$$

where $R_i, i = 0, 1, \ldots, p-1$, is a circulant matrix of the form

$$R_i = \begin{bmatrix} r_{0,i} & r_{1,i} & r_{2,i} & \cdots & r_{m-1,i} \\ r_{m-1,i} & r_{0,i} & r_{1,i} & \cdots & r_{m-2,i} \\ r_{m-2,i} & r_{m-1,i} & r_{0,i} & \cdots & r_{m-3,i} \\ \vdots & \vdots & \vdots & & \vdots \\ r_{1,i} & r_{2,i} & r_{3,i} & \cdots & r_{0,i} \end{bmatrix}. \tag{4}$$

[1] A cyclic shift of an m-tuple $(x_0, x_1, \ldots, x_{m-1})$ is the m-tuple $(x_{m-1}, x_0, \ldots, x_{m-2})$.

The algebra of $m \times m$ circulant matrices over $\mathbb{F}_q$ is isomorphic to the algebra of polynomials in the ring $\mathbb{F}_q[x]/(x^m - 1)$ if R_i is mapped onto the polynomial $r_i(x) = r_{0,i} + r_{1,i}x + r_{2,i}x^2 + \cdots + r_{m-1,i}x^{m-1}$, formed from the entries in the first row of R_i [13]. The $r_i(x)$ associated with a QC code are called the *defining polynomials* [8]. The set $\{r_0(x), r_1(x), \ldots, r_{p-1}(x)\}$ defines an $[mp, m]$ QC code with $k = m$.

The construction of QC codes requires a representative set of defining polynomials. These are the equivalence class representatives of a partition of the set of polynomials of degree less than m. Two polynomials, $r_j(x)$ and $r_i(x)$ are said to be *equivalent* if they belong to the same class, which here means

$$r_j(x) = \gamma x^l r_i(x) \bmod (x^m - 1),$$

for some integer $l > 0$ and scalar $\gamma \in \mathbb{F}_q \backslash \{0\}$. The number of representative defining polynomials, $N(m)$, for $\mathbb{F}_{13}$ is given below

m	$N(m)$
2	8
3	63
4	604
5	6189
6	67116

The QC codes presented here were constructed using a stochastic optimization algorithm, tabu search, similar to that in [4] and [11]. This algorithm will be described in the next section. By restricting the search to the class of QC codes, and using a stochastic heuristic, codes with high minimum distance can be found with a reasonable amount of computational effort.

3 The Construction Algorithm

Imposing a structure on the codes being considered results in a search space that is smaller than for the original problem. The more restrictions on the structure, the smaller the search problem. This results in a tradeoff, since good codes may be missed if too much structure is imposed on the code. However, it is often the case that good codes have significant structure, and this partially explains why the approach presented here works so well.

It is not necessary to check the weight of every codeword in a QC code in order to determine d. Only a subset, $N < M$, of the codewords need be considered since the Hamming weight of $i(x)b_s(x) \bmod (x^m - 1)$ is equal to the weight of $i(x)\gamma x^l b_s(x) \bmod (x^m - 1)$ for all $l \geq 0$ and $\gamma \in \mathrm{GF}(q) \setminus \{0\}$. Note that this argument also applies to the set of defining polynomials. Thus, for example, with $q = 13$ and $m = 3$, from the above table $N = 63$.

To simplify the process of searching for good codes, the weights of the subset of codewords can be stored in an array, and a matrix, D, can be formed from the arrays for the defining polynomials to be considered

$$D = \begin{array}{c|cccccc} & b_1(x) & b_2(x) & \cdots & b_s(x) & \cdots & b_y(x) \\ \hline i_1(x) & w_{11} & w_{12} & \cdots & w_{1s} & \cdots & w_{1y} \\ i_2(x) & w_{21} & w_{22} & \cdots & w_{2s} & \cdots & w_{2y} \\ \vdots & \vdots & \vdots & & \vdots & & \vdots \\ i_t(x) & w_{t1} & w_{t2} & \cdots & w_{ts} & \cdots & w_{ty} \\ \vdots & \vdots & \vdots & & \vdots & & \vdots \\ i_z(x) & w_{z1} & w_{z2} & \cdots & w_{zs} & \cdots & w_{zy}, \end{array}$$

where $i_t(x)$ is the tth information polynomial, $b_s(x)$ is the sth defining polynomial, and w_{ts} is the Hamming weight of $i_t(x)b_s(x) \bmod (x^m - 1)$. Since $i_t(x)$ and $b_s(x)$ correspond to the same polynomials, D is a square ($y = z = N$), symmetric (by letting $i_t(x) = b_t(x)$ for all $1 \leq t \leq N$) matrix. For example, if $q = 13$ and $m = 2$, the matrix is

$$D = \begin{array}{c|cccccccc} & 1 & x+1 & x+2 & x+3 & x+4 & x+5 & x+6 & x+12 \\ \hline 1 & 1 & 2 & 2 & 2 & 2 & 2 & 2 & 2 \\ x+1 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 0 \\ x+2 & 2 & 2 & 2 & 2 & 2 & 2 & 1 & 2 \\ x+3 & 2 & 2 & 2 & 2 & 1 & 2 & 2 & 2 \\ x+4 & 2 & 2 & 2 & 1 & 2 & 2 & 2 & 2 \\ x+5 & 2 & 2 & 2 & 2 & 2 & 1 & 2 & 2 \\ x+6 & 2 & 2 & 1 & 2 & 2 & 2 & 2 & 2 \\ x+12 & 2 & 0 & 2 & 2 & 2 & 2 & 2 & 2 \end{array}$$

The complete weight distribution for a QC code composed of any set of $b_s(x)$ can be constructed from D. The search for a good code consists of finding p columns of D with a large minimum row sum, since the weight of a minimum distance codeword must be contained in these sums.

Having decided on the values of m and p (and thus also $n = mp$), the entries of the integer matrix D can be calculated and the problem formulated as a combinatorial optimization problem. The goal is to find

$$\max_{S} \min_{1 \leq j \leq N} \sum_{s \in S} w_{j,s}, \tag{5}$$

where $S \subseteq \{1, 2, \ldots, N\}$ and $|S| = p$. In general, one can take a multiset S with p elements, but it was found in past studies that for the new codes obtained, no defining polynomial occurs more than once, so S is here required to be a set.

The optimization method used in this work is *tabu search* [5]. This method can produce good near-optimal (optimal in some cases) solutions to difficult optimization problems with a reasonable amount of computational effort [12]. Tabu search is a local search algorithm, which means that starting from an initial solution, a series of solutions is obtained so that every new solution only differs slightly from the previous one. A potential new solution is called a *neighbor* of the old solution, and all neighbors of a given solution constitute the *neighborhood* of that solution. To evaluate the quality of solutions, a *cost function* is needed. Tabu search always proceeds to a best possible solution in the neighborhood of the current solution. In a simple version, if there are several equally good neighbors with the best cost, a random choice is made (note that it is possible that the best neighbor has a worse cost than the current solution has). To ensure that the search does not loop on a subset of moves or solutions, attributes of recent solutions are stored in a so-called tabu list; new moves or solutions with these attributes are then not allowed for L moves.

Tabu search is applied here to the problem of finding QC codes, defined as a minimization problem, in the following way. First, the problem is not formulated as generally as in (5), as the desired minimum distance, d, of the code is fixed. A solution is any set $S \subseteq \{1, 2, \ldots, N\}$ of p columns of D, the neighborhood of a solution is the set of solutions obtained by replacing one column with a column that is not in the code, and the cost function is of the form

$$C = \sum_{j=1}^{N} \max\left(0, d - \sum_{s \in S} w_{j,s}\right)$$

A solution with cost 0 now corresponds to a code with minimum distance at least d. If such a solution is found, the search ends. Otherwise, the search is continued (and possibly restarted occasionally), until a given time or iteration limit is reached. The tabu list is simply the indexes of the new columns. Thus, if a column is replaced by another, the new column must not be replaced during the next L moves.

The values of L used were small, in the range $p/10 \leq L \leq p/5$. If a code was not found within 1000–2000 iterations, the search was restarted from a new random initial solution. As many as 1000 restarts were performed for given values of m and p. The total number of iterations to find a code varied between about one hundred and a few million.

The best QC codes found are given in Tables 1 to 4. The defining polynomials are listed with the lowest degree coefficient on the left, i.e., 7321 corresponds to the polynomial $x^3 + 2x^2 + 3x + 7$, with leading zeroes left out for brevity. The digits 10, 11 and 12 are denoted by (10), (11) and (12), respectively.

As an example, consider the [18,3] code in Table 1 with $m = 3$ and $p = 6$ defining polynomials. These polynomials give the following generator matrix

$$G = \left[\begin{array}{c|c|c|c|c|c} 016 & 175 & 125 & 117 & 135 & 143 \\ 601 & 517 & 512 & 711 & 513 & 314 \\ 160 & 751 & 251 & 171 & 351 & 431 \end{array}\right]$$

with weight distribution

i	A_i
0	1
15	456
16	468
17	720
18	552

This code is optimal since it meets the Griesmer bound (1), and so establishes that $d_{13}(18, 3) = 15$.

For $m = 5$ and $p = 4$, the best code found has generator matrix

$$G = \left[\begin{array}{c|c|c|c} 00018 & 14(10)(12)4 & 01(12)8(11) & 12(11)3(12) \\ 80001 & 414(10)(12) & (11)01(12)8 & (12)12(11)3 \\ 18000 & (12)414(10) & 8(11)01(12) & 3(12)12(11) \\ 01800 & (10)(12)414 & (12)8(11)01 & (11)3(12)12 \\ 00180 & 4(10)(12)41 & 1(12)8(11)0 & 2(11)3(12)1 \end{array}\right]$$

with weight distribution

i	A_i
0	1
15	9300
16	11640
17	51960
18	98520
19	125220
20	74652

This code is also optimal since it meets the Griesmer bound (1), and so establishes that $d_{13}(20, 5) = 15$. In addition, the codes for $m = 3$ and 4 with $p = 5$ meet the Griesmer bound. Note that all codes with $n \leq 14$ given in the tables are MDS.

Table 1. Best QC codes over $\mathbb{F}_{13}$ with $p = 3$

code	d	$r_i(x)$
[6,3]	4	1, 12(12)
[9,3]	7	1, 15(11), 117
[12,3]	10	1, 135, 112, 153
[15,3]	12	11, 153, 12, 143, 11(10)
[18,3]	15	16, 175, 125, 117, 135, 143
[21,3]	18	115, 11(10), 146, 132, 16, 1, 176
[24,3]	20	1, 174, 1(11), 112, 19(12), 138, 114, 12(12)
[27,3]	23	112, 117, 12, 162, 17, 12(12), 13, 114, 19
[30,3]	26	124, 176, 11(12), 156, 13, 129, 13(11), 14, 17, 132
[33,3]	29	12, 1(10), 13(11), 18, 15, 12(12), 15(11), 132, 156, 126, 116
[36,3]	32	129, 11(10), 11, 11(11), 19, 146, 12(12), 14(11), 11(12), 145, 13, 18
[39,3]	34	1(11), 138, 12, 11, 14(11), 11(11), 16, 125, 112, 114, 113, 156, 15
[42,3]	37	15, 12, 119, 158, 126, 156, 138, 11(12), 11, 176, 117, 1(11), 112, 13(10)
[45,3]	40	18, 1(11), 163, 11, 134, 135, 126, 156, 174, 19(12), 17, 132, 125, 13, 162
[48,3]	43	116, 1(10), 1(11), 114, 14(11), 142, 1(10)6, 12(12), 19(12), 163, 11(12), 12 174, 146, 11, 1(12)
[51,3]	45	113, 16, 11, 17, 142, 146, 18, 118, 132, 194, 1(12), 156, 15, 117, 176, 116, 115
[54,3]	48	17, 11, 13(12), 132, 118, 19, 153, 185, 13(10), 14(11), 146, 162, 142, 11(11), 135 175, 15, 11(10)
[57,3]	51	1(11), 11(10), 175, 123, 12(12), 115, 156, 1, 163, 124, 13, 18, 11(12), 11(11), 146 112, 19, 17(12), 132
[60,3]	54	162, 11(12), 13, 118, 113, 115, 13(12), 117, 19, 18, 112, 12, 185, 138, 153, 123, 17 156, 126, 132
[63,3]	56	116, 143, 119, 185, 146, 15(11), 11, 134, 19(12), 117, 156, 115, 19, 135, 12(11), 15 158, 125, 154, 113, 18
[66,3]	59	118, 1, 11(11), 13(12), 135, 154, 17, 11, 13(11), 163, 12(10), 129, 112, 115, 15 12(11), 19(12), 119, 132, 19, 18, 123
[69,3]	62	18, 142, 19, 11(11), 194, 17, 14(11), 13(11), 118, 132, 11, 123, 125, 112, 113, 12(10) 156, 17(12), 16, 117, 129, 116, 1(11)
[72,3]	65	1(12), 134, 1(11), 17(12), 119, 1, 11, 15(11), 163, 13(10), 124, 114, 142, 13, 128 14(11), 154, 126, 11(12), 11(11), 156, 19, 117, 176

Table 2. Best QC codes over $\mathbb{F}_{13}$ with $p = 4$

code	d	$r_i(x)$
[8,4]	5	1135, 1326
[12,4]	9	104, 1197, 135
[16,4]	12	17, 1(10)3, 14(11)8, 161(11)
[20,4]	16	116, 1(11), 1186, 142, 134(10)
[24,4]	19	1326, 11, 1745, 111(11), 186, 1165
[28,4]	23	14, 13, 1159, 163(11), 1252, 112(12), 1294
[32,4]	26	103, 1(10)3, 1182, 114, 1143, 1(10)(10), 1132, 1(11)(10)
[36,4]	30	1155, 113, 139, 1117, 13(10)(11), 153(11), 125, 136, 122
[40,4]	33	12, 1315, 1, 115(10), 141, 117(10), 1825, 112(12), 1(12)4, 1184
[44,4]	37	14, 1118, 112(10), 11(11)(12), 113(10), 102, 1214, 12(10)4, 1(11) 13(10), 13(12)2
[48,4]	40	102, 1115, 12, 1166, 12(10)4, 133, 18(11), 145(11), 12(10)5, 1197 1825, 112(12)
[52,4]	44	11, 1, 1125, 1546, 12(11)6, 1(12)5, 11(10)4, 1293, 1135, 1197, 17(10) 1458, 168
[56,4]	48	103, 19, 1(11)6, 1112, 145(11), 17(10), 141, 13(10)6, 1376, 1385 119(10), 11(10)9, 11(12)8, 1(10)7
[60,4]	51	11, 135, 153, 1598, 169, 13(11)(12), 123, 1112, 12(12), 1416, 128(12) 1346, 1219, 118(10), 181
[64,4]	55	135, 14(10)5, 151, 11(11)(10), 1418, 166, 13(10)5, 13(11)2, 1564, 16(10) 14(11)(12), 1193, 1153, 14, 1116, 111(10)
[68,4]	58	1, 112, 14(10)2, 1592, 1265, 17(10), 151(11), 167(12), 193, 145(12) 1416, 117, 1115, 189, 1464, 14(12), 135
[72,4]	62	13, 1623, 1243, 1462, 1475, 12, 172, 1128, 161(11), 164, 1535, 12(12) 1625, 1148, 1284, 134, 18(12), 1328
[76,4]	66	171, 1284, 1376, 12(11)3, 113(12), 12(10), 1, 11(10)3, 161(12), 1154, 143 1289, 12(10)(11), 119(11), 1(11)(12), 129(12), 187, 1278, 1318
[80,4]	69	105, 11(10)(11), 1825, 1314, 1(12)(12), 102, 14(10)(11), 113, 11(12), 1423 127(12), 124(10), 1148, 1243, 1215, 12(11)6, 1329, 127, 1(11), 1382
[84,4]	73	196, 1376, 158, 167, 18(10)3, 1343, 129, 14(12), 1, 1498, 1187, 133 143(10), 1237, 1(11)5, 1598, 1217, 1172, 125, 12(11)2, 1238
[88,4]	76	11, 1245, 1, 1585, 113(12), 112(12), 1329, 1(12)(10), 1876, 197, 199, 1138 121(11), 1(11)6, 17(11), 1354, 15(11)6, 1134, 17(10), 1684, 1319, 14(10)
[92,4]	80	102, 1169, 149, 16(12), 1, 146, 1(12)(12), 1384, 1172, 1425, 1114, 1(10)1 115, 1217, 192, 12(11)3, 147(12), 1193, 1456, 1454, 1756, 127(10), 11(12)6
[96,4]	84	113, 1415, 1, 1254, 1(12)(12), 137(11), 13(11)3, 119(11), 187, 13, 11(11)9 1194, 1623, 13(12)2, 1264, 11(10), 1516, 11(11)5, 145(11), 128(11), 1148 106, 121, 129(10)

Table 3. Best QC codes over $\mathbb{F}_{13}$ with $p = 5$

code	d	$r_i(x)$
[10,5]	6	13(10), 10(10)(10)
[15,5]	10	154, 14(10)56, 11(12)9(10)
[20,5]	15	18, 14(10)(12)4, 1(12)8(11), 12(11)3(12)
[25,5]	19	10(12), 1(12)(11)(12), 14168, 11893, 17(10)(12)3
[30,5]	23	1561(11), 11659, 1163(12), 1(10)6(11), 126(10)(12), 1842
[35,5]	27	10(10), 112(12)6, 16842, 11458, 11(10)(12), 1212(12), 11(10)92
[40,5]	31	1, 13(12)1, 12434, 1157, 158(12)3, 133(10), 11(12)3(11), 111
[45,5]	36	11(10), 19, 11453, 1156(12), 12(12)5, 11247, 16746, 1222, 1635(11)
[50,5]	40	121(10)2, 14, 12619, 1(12)47, 12(12)2(12), 1454, 11586, 12835, 13792 13(10)(12)
[55,5]	45	138(12)8, 11, 1(10)3(10)6, 159, 172(11), 12348, 1673(11), 10(12)(12), 15684 128(11)8, 1776
[60,5]	49	13, 1589, 11284, 1333, 11(11), 153, 1456, 12(12)52, 13(12)6(11), 12(11)89 1(11)1(12), 11724
[65,5]	54	16(12), 1(10)94, 128(10)(11), 10(11)9, 1(11)11, 13(11)6, 11(12)49, 13138 1152(11), 1351(10), 11139, 1124(10), 14(11)
[70,5]	58	103, 14, 17(12)34, 12(11)(10)(12), 124(10)(11), 112, 1713, 11(11)89, 14182 106(12), 11374, 11553, 1(10)17, 145(11)3
[75,5]	62	1137, 11, 103, 1161(12), 1(10)9, 1987, 1(12)32, 1783, 11766, 142(11)3, 11618 12(12)43, 141(12)2, 127(11)5, 1152(12)
[80,5]	67	152, 102, 14(11)3(11), 15(10), 11(12)7(10), 1385, 11(11)38, 11229, 11295 15385, 1334, 18(11)6, 1115(12), 12154, 1841, 11(11)8
[85,5]	71	1944, 13, 1268(11), 1(11)92, 12(12)72, 119, 1131(10), 1841, 15(11)5 129(10)6, 1497(12), 11(12)82, 15325, 11584, 14742, 14(10)(11)3, 1397(10)
[90,5]	76	122, 1675(11), 1262(12), 12825, 113(11)5, 15(12)(11), 1(11)8, 1(10)4 12(10)3(10), 1394(12), 119(12)9, 1172(12), 12723, 1552, 121, 12(12)8(11) 14(11)(10)(11), 19(12)
[95,5]	80	14(10), 12978, 14, 11474, 11(11)7(11), 14(11)2(11), 1285, 127, 12132, 14646 1955, 12(11)63, 134(10)2, 14934, 11(12)96, 1(12), 1627(12), 14163, 119(12)8
[100,5]	84	135, 11(11)96, 15(10), 13494, 1241(11), 127(10)(12), 13(10)23, 11(10)2(11) 137(11), 13156, 13(10)28, 13(10)6(10), 1(10)95, 128(12)3, 12496, 1134(11) 159(11)8, 11917, 19(10)(10), 13(11)2(11)
[105,5]	89	108, 11445, 14(11)2, 12328, 14(11)82, 11833, 135(11)4, 18(10)9(12), 11269 114(12)5, 13172, 173(10), 144(12), 11(10)3, 142(12)4, 16975, 117(12)7 113(11)8, 11(12)(11)3, 12(10)3(12), 11875
[110,5]	93	106, 1121(11), 14(10)(11), 111, 161(12)6, 12694, 193, 14172, 11564, 1117(12) 13(10)4, 1(11)3(10), 115(10)8, 12349, 1131(12), 14374, 161(11)2, 1721 11(12)7(11), 1437, 11935, 11719
[115,5]	98	116, 137(10)5, 113, 15(11)(10)6, 13(11)(12)(10), 11(11)95, 119(12)(10) 15694, 15156, 126(12)9, 1(10)98, 119(12)5, 1275(10), 1588, 11917, 1229 12(10)(12)4, 1118(10), 11(11)(12), 114(11)(10), 118(12)5, 1427(10), 1515(11)
[120,5]	102	118(11), 169, 1127, 1, 124(12)3, 10(11), 11, 1338, 19(12)1(12), 13258 1863, 128(10)4, 1274, 1161(10), 13438, 116(10)(12), 18(12)(11) 131(10)(11), 11814, 11363, 117(10)6, 12(10)28, 11(12)93, 13862

Table 4. Best QC codes over $\mathbb{F}_{13}$ with $p = 6$

code	d	$r_i(x)$
[12,6]	7	12212, 11(10)93
[18,6]	11	12, 10(12)5(11), 128(10)3
[24,6]	16	185, 1827(12)4, 12835, 114947
[30,6]	21	13, 11183(10), 1133(12)4, 14(10)2(11)(12), 13537
[36,6]	26	1002, 118217, 115184, 126596, 12179, 1112
[42,6]	32	11(10), 12(12)235, 111893, 1121(12)(12), 138745, 119(12)24 13(12)6(11)(10)
[48,6]	37	114, 107, 1(11)267, 1131(12), 128(11)(10)(11), 1346(11)5 11(11)6(11)4, 1(12)511
[54,6]	42	105(11)7, 118633, 19, 111299, 1698(10)3, 1269(11)8, 121976, 1451(11)4 11(12)9(10)(11)
[60,6]	47	1(11)(12), 104(10), 12(12)4(10)4, 1469(10), 119(10)6, 1(10)25(11) 11(11)38(11), 14(10)46, 15564, 12361
[66,6]	52	13, 11, 1214, 10633, 1124(10)7, 141(11)8(10), 14187(11), 1455(11) 11(12)7(11)(12), 135843, 15577
[72,6]	57	1, 11(11)5, 12(10)(10)(11), 11(11)543, 147(12)92, 1299, 128958 111(12)(11)2, 15768, 11(12)61(10), 19247, 12773
[78,6]	63	12(12)1, 19, 153(10)3(11), 13784(10), 19(10)3, 10326, 12(12)(10)85 125268, 11(11)65, 13564, 187(11)9, 1192(10)7, 1(10)269
[84,6]	68	119, 149, 18332, 13(10)17(12), 14623, 131768, 105(10)3, 11837(10) 11123(11), 121435, 1241(10)8, 151(10)7, 11385, 11(10)535
[90,6]	73	106, 11, 11(11)68(11), 11398(11), 14(11)4(10), 143(11)2, 119(11)29 125286, 10169, 17556, 13469(11), 129474, 1535, 108(10)(10), 13(11)2(10)5
[96,6]	78	132345, 14535(12), 14(10)(12)82, 11(10)395, 1029, 112453, 115(10)68 19452, 1231(12), 132(11)(12), 11286, 1(12)(12)(12)(12), 116684, 13(11)476 147(12)(12), 1591
[102,6]	83	102, 1432, 116735, 1263, 12(10)8, 1, 11123(12), 11585(12), 14689(11) 19432, 17(10)96, 124592, 117758, 1347(11), 1419(12)4, 12(11)7(12)(11) 14(12)(12)6
[108,6]	88	1, 107, 149(11)1(12), 14(10)8(10)2, 121796, 11(12)173, 10249, 14142(12) 13(12)8(10)(11), 116, 116895, 126714, 11323(11), 1114(10)6, 1(11)684, 1901 19(12)84, 13673
[114,6]	94	104(11)7, 118(11)(10)8, 11(10)418, 1688(12), 15, 1(10)8(10)8, 125894 112452, 1282(12)(10), 1(10)4(11)(10), 1178(10)2, 121752, 16(12)59, 14618 135(12)15, 19(11)(10)2, 117(10)15, 113564, 16235(11)
[120,6]	99	125, 119(11)56, 1432(12)6, 1, 16726, 101(12), 1382(11)(10), 19133, 107(11) 112(11)(11), 1427(10)6, 116459, 121(11)6, 1187(12)(10), 131645, 1(10)46 146(10)(12)5, 112(10)(12)2, 1421(11)(10), 14(10)(11)9
[126,6]	105	16, 11, 117(12)6, 125(12), 11668(12), 14(11)(10)92, 11(11)(12)18 13(11)8(11)5, 10759, 114578, 111934, 11493(12), 1(10)91, 123575, 132193 114188, 13(12)5(12)4, 119427, 1154(10)3, 14147(12), 12(11)3(12)4
[132,6]	110	13, 17(10)72, 11, 19(12), 14295, 121(12)63, 119615, 11572(10), 116784 14(11)1(10)2, 1(12)(10)(12)(11), 129234, 111114, 11598(11), 14935(11) 132398, 1(12)918, 1417(11)2, 141(11)53, 10927, 115(11)5(12), 1(10)(12)14
[138,6]	115	112146, 12, 1(10)83, 119125, 17, 116(10)35, 1397, 11(11)683, 11(11)843 145(11)68, 13026, 1(10)4(11)8, 10743, 1846(11), 1251(12)9, 11635 112(11)(12)5, 129373, 164(12)2, 15(12)56, 14(11)818, 1328(11)4, 11032
[144,6]	120	101, 1574(11), 121642, 19(11)28, 127(12)6(10), 139(10)62, 1(12)969 16(11)2(11), 14(10)5(12), 125438, 1419(10)6, 119324, 12(12)45(11), 16(12) 13(12)3(12)2, 142(10)84, 1326(11)4, 15(11)(12)38, 1419, 129585, 13(11)434 114897, 114(10)64, 12(11)6(10)

References

1. Berlekamp, E.R.: Algebraic Coding Theory. McGraw-Hill, New York (1968)
2. Betsumiya, K., Georgiou, S., Gulliver, T.A., Harada, M., Koukouvinos, C.: On self-dual codes over some prime fields. Disc. Math. 262, 37–58 (2003)
3. Bosma, W., Cannon, J.: Handbook of Magma Functions, Department of Mathematics, University of Sydney, `http://magma.maths.usyd.edu.au/magma/`
4. Daskalov, R.N., Gulliver, T.A.: New good quasi-cyclic ternary and quaternary linear codes. IEEE Trans. Inform. Theory 43, 1647–1650 (1997)
5. Glover, F.: Tabu search—Part I. ORSA J. Comput. 1, 190–206 (1989)
6. Grassl, M.: Bounds on the minimum distance of linear codes and quantum codes, `http://www.codetables.de`
7. de Boer, M.A.: Almost MDS codes. Design, Codes Crypt. 9, 143–155 (1996)
8. Greenough, P.P., Hill, R.: Optimal ternary quasi-cyclic codes. Designs, Codes and Crypt. 2, 81–91 (1992)
9. Gulliver, T.A., Bhargava, V.K.: Some best rate 1/p and rate (p-1)/p systematic quasi-cyclic codes over GF(3) and GF(4). IEEE Trans. Inform. Theory 38, 1369–1374 (1992)
10. Gulliver, T.A.: New optimal ternary linear codes. IEEE Trans. Inform. Theory 41, 1182–1185 (1995)
11. Gulliver, T.A., Bhargava, V.K.: New good rate (m-1)/pm ternary and quaternary quasi-cyclic codes. Designs, Codes and Crypt. 7, 223–233 (1996)
12. Honkala, I.S., Östergård, P.R.J.: Applications in code design. In: Aarts, E.H.L., Lenstra, J.K. (eds.) Local Search in Combinatorial Optimization. Wiley, New York (2003)
13. MacWilliams, F.J., Sloane, N.J.A.: The Theory of Error-Correcting Codes. North-Holland, New York (1977)
14. Newhart, D.W.: On minimum weight codewords in QR codes. J. Combin. Theory Ser. A 48, 104–119 (1988)

Acyclic Colorings of Graph Subdivisions

Debajyoti Mondal[1], Rahnuma Islam Nishat[2,⋆], Sue Whitesides[2,⋆], and Md. Saidur Rahman[3]

[1] Department of Computer Science, University of Manitoba
[2] Department of Computer Science, University of Victoria
[3] Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology (BUET)
debajyoti_mondal_cse@yahoo.com,
{rnishat,sue}@uvic.ca, saidurrahman@cse.buet.ac.bd

Abstract. An acyclic coloring of a graph G is a coloring of the vertices of G, where no two adjacent vertices of G receive the same color and no cycle of G is bichromatic. An acyclic k-coloring of G is an acyclic coloring of G using at most k colors. In this paper we prove that any triangulated plane graph G with n vertices has a subdivision that is acyclically 4-colorable, where the number of division vertices is at most $2n - 6$. We show that it is NP-complete to decide whether a graph with degree at most 7 is acyclically 4-colorable or not. Furthermore, we give some sufficient conditions on the number of division vertices for acyclic 3-coloring of subdivisions of partial k-trees and cubic graphs.

Keywords: Acyclic coloring, Subdivision, Triangulated plane graph.

1 Introduction

A *coloring* of a graph G is an assignment of colors to the vertices of G such that no two adjacent vertices receive the same color. A coloring of G is an *acyclic coloring* if G has no bichromatic cycle in that coloring. The *acyclic chromatic number* of G is the minimum number of colors required in any acyclic coloring of G. See Figure 1 for an example.

The large number of applications of acyclic coloring has motivated much research [4,7]. For example, acyclic coloring of planar graphs has been used to obtain upper bounds on the volume of 3-dimensional straight-line grid drawings of planar graphs [6]. Consequently, acyclic coloring of planar graph subdivisions can give upper bounds on the volume of 3-dimensional polyline grid drawings, where the number of division vertices gives an upper bound on the number of bends sufficient to achieve that volume. As another example, solving large scale optimization problems often makes use of sparse forms of Hessian matrices; acyclic coloring provides a technique to compute these sparse forms [7].

Acyclic coloring was first studied by Grünbaum in 1973 [8]. He proved an upper bound of nine for the acyclic chromatic number of any planar graph G,

⋆ Work is supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) and University of Victoria.

C.S. Iliopoulos and W.F. Smyth (Eds.): IWOCA 2011, LNCS 7056, pp. 247–260, 2011.
© Springer-Verlag Berlin Heidelberg 2011

with $n \geq 6$ vertices. He also conjectured that five colors are sufficient for acyclic coloring of any planar graph. His upper bound was improved many times [1,9,10] and at last Borodin [3] proved that five is both an upper bound and a lower bound. Testing acyclic 3-colorability is NP-complete for planar bipartite graphs with maximum degree 4, and testing acyclic 4-colorability is NP-complete for planar bipartite graphs with the maximum degree 8 [13].

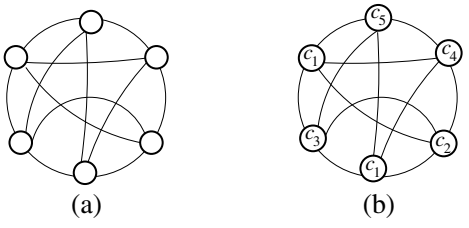


Fig. 1. (a) A graph G and (b) an acyclic coloring of G using five colors c_1–c_5

Subdividing an edge (u, v) of a graph G is the operation of deleting the edge (u, v) and adding a path $u(= w_0), w_1, w_2, \ldots, w_k, v(= w_{k+1})$ through new vertices $w_1, w_2, \ldots, w_k$, $k \geq 1$, of degree two. A graph G' is said to be a *subdivision* of a graph G if G' is obtained from G by subdividing some of the edges of G. A vertex v of G' is called an *original vertex* if v is a vertex of G; otherwise, v is called a *division* vertex. Wood [15] observed that every graph has a subdivision with two division vertices per edge that is acyclically 3-colorable. Recently Angelini and Frati [2] proved that every plane graph has a subdivision with one division vertex per edge that is acyclically 3-colorable.

Main Results : We study acyclic colorings of subdivisions of graphs and prove the following claims.

(1) Every cubic graph with n vertices has a subdivision that is acyclically 3-colorable, where the number of division vertices is $3n/4$. Every triconnected plane cubic graph has a subdivision that is acyclically 3-colorable, where the number of division vertices is at most $n/2$. Every Hamiltonian cubic graph has a subdivision that is acyclically 3-colorable, where the number of division vertices is at most $n/2 + 1$. See Section 2.
(2) Every partial k-tree, $k \leq 8$, has a subdivision with at most one division vertex per edge that is acyclically 3-colorable. See Section 2.
(3) Every triangulated plane graph G with n vertices has a subdivision with at most one division vertex per edge that is acyclically 4-colorable, where the total number of division vertices is at most $2n - 6$. See Section 3.
(4) It is NP-complete to decide whether a graph with degree at most 7 is acyclically 4-colorable or not. See Section 4.

2 Preliminaries

In this section we present some definitions and preliminary results that are used throughout the paper. See also [12] for graph theoretic terms.

Let $G = (V, E)$ be a connected graph with vertex set V and edge set E. The degree $d(v)$ of a vertex $v \in V$ is the number of neighbors of v in G. A *subgraph* of a graph $G = (V, E)$ is a graph $G' = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E$. If G' contains exactly the edges of G that join vertices in V', then G' is called the *subgraph induced by* V'. If $V' = V$ then G' is a *spanning subgraph* of G. A *spanning tree* is a spanning subgraph of G that is a tree. The *connectivity* $\kappa(G)$ of a graph G is the minimum number of vertices whose removal results in a disconnected graph or a single-vertex graph. G is said to be *k-connected* if $\kappa(G) \geq k$.

Let $P = u_0, u_1, u_2, \ldots, u_{l+1}$, $l \geq 1$, be a path of G such that $d(u_0) \geq 3$, $d(u_1) = d(u_2) = \ldots = d(u_l) = 2$, and $d(u_{l+1}) \geq 3$. Then we call the subpath $P' = u_1, u_2, \ldots, u_l$ of P a *chain* of G. A *subsequence* is a sequence that can be derived from another sequence by deleting some elements without changing the order of the remaining elements. An *ear* of a graph G is a maximal path whose internal vertices have degree two in G. An *ear decomposition* of G is a decomposition $P_1, \ldots, P_k$ such that P_1 is a cycle and P_i, $2 \leq i \leq k$, is an ear of $P_1 \cup \ldots \cup P_i$.

Throughout the paper, division vertices are colored gray in all the figures. We now have the following two facts.

Fact 1. *Let G be a graph with two distinct vertices u and v and let G' be a graph obtained by adding a chain $w_1, \ldots, w_k$ between the vertices u and v of G. Let G be acyclically 3-colorable such that the colors of u and v are different. Then G' is acyclically 3-colorable.*

Proof. In an acyclic coloring of G that colors vertices u and v differently, let the colors of vertices u and v be c_1 and c_2, respectively. For each w_i, $i = 1, 2, \ldots, k$, we assign color c_3 when i is odd and color c_1 when i is even as in Figure 2(a). Clearly, no two adjacent vertices of G' have the same color. Therefore, the coloring of G' is a valid 3-coloring. Suppose for a contradiction that the coloring of G' is not acyclic. Then G' must contain a bichromatic cycle C. The cycle C either contains the chain $u, w_1, w_2, \ldots, w_k, v$ or is a cycle in G. C cannot contain the chain since the three vertices u, v and w_1 are assigned three different colors c_1, c_2 and c_3, respectively. Thus we can assume that C is a cycle in G. Since G does not contain any bichromatic cycle, C cannot be a bichromatic cycle, a contradiction. □

Fact 2. *Let G be a biconnected graph with n vertices and let $P_1 \cup \ldots \cup P_k$ be an ear decomposition of G where each ear P_i, $2 \leq i \leq k$, contains at least one internal vertex. Then G has a subdivision G', with at most $k-1$ division vertices, that is acyclically 3-colorable.*

Proof. We prove the claim by induction on k. The case $k = 1$ is trivial since P_1 is a cycle, which is acyclically 3-colorable. Therefore we assume that $k > 1$ and that the claim is true for the graphs $P_1 \cup \ldots \cup P_i$, $1 \leq i \leq k-1$. By induction, $G - P_k$ has a subdivision G'' that is acyclically 3-colorable and that has at most $k-2$ division vertices. Let the end vertices of P_k in G be u and v. If u and v have different colors in G'' then we can prove in a similar way as in the proof of Fact 1 that G has a subdivision G' that is acyclically 3-colorable and that has the same number of division vertices as G'', which is at most $k-2$. Otherwise, u and v have the same color in G''. Let the color of u and v be c_1 and let the two other colors in G'' be c_2 and c_3. If P_k contains more than one internal vertex then we assign the colors c_2 and c_3 to the vertices alternately. If P_k contains only one internal vertex v then we subdivide an edge of P_k once. We color v with c_2 and the division vertex with c_3 as shown in Figure 2(b). In both cases we can prove in a similar way as in the proof of Fact 1 that G' has no bichromatic cycle. Moreover, the number of division vertices in G' is at most $(k-2)+1 = k-1$. □

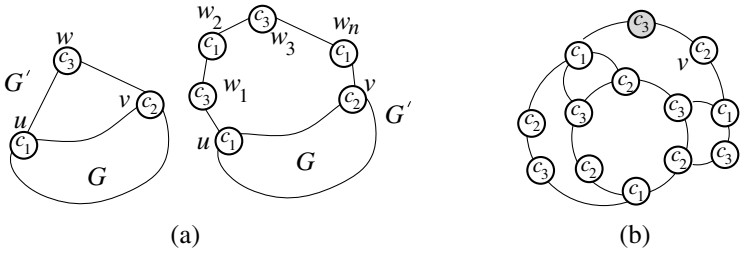


Fig. 2. Illustration for the proof of (a) Fact 1 and (b) Fact 2

Let $G = (V, E)$ be a 3-connected plane graph and let (v_1, v_2) be an edge on the outer face of G. Let $\pi = (V_1, V_2, ..., V_l)$ be an ordered partition of V. Then we denote by G_k, $1 \leq k \leq l$, the subgraph of G induced by $V_1 \cup V_2 \cup ... \cup V_k$ and by C_k the outer cycle of G_k. We call the vertices of the outer face the *outer vertices*. An *outer chain* of G_k is a chain on C_k. We call π a *canonical decomposition* of G with an edge (v_1, v_2) on the outer face if the following conditions are satisfied [12].

(a) V_1 is the set of all vertices on the inner face that contains the edge (v_1, v_2). V_l is a singleton set containing an outer vertex v, $v \notin \{v_1, v_2\}$.
(b) For each index k, $2 \leq k \leq l-1$, all vertices in V_k are outer vertices of G_k and the following conditions hold:
  (1) if $|V_k| = 1$, then the vertex in V_k has two or more neighbors in G_{k-1} and has at least one neighbor in $G - G_k$; and
  (2) If $|V_k| > 1$, then V_k is an outer chain of G_k.

Figure 3 illustrates a canonical decomposition of a 3-connected plane graph.

A *cubic graph* G is a graph such that every vertex of G has degree exactly three. Every cubic graph has an acyclic 4-coloring [14]. We can get an acyclic

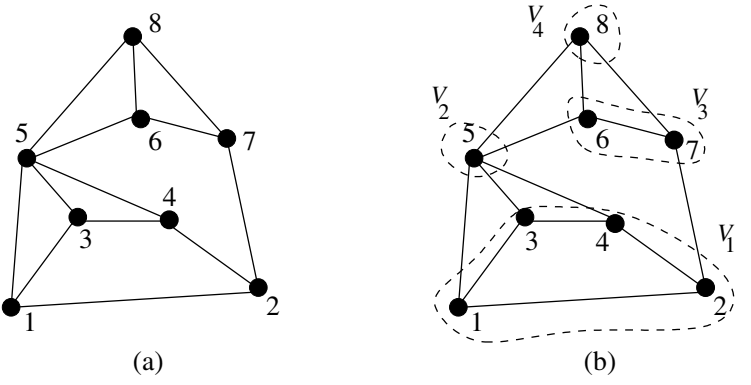


Fig. 3. (a) A 3-connected plane graph G and (b) a canonical decomposition of G

3-coloring of a subdivision G' of G with $3n/4$ division vertices from an acyclic 4-coloring of G as follows. Let c_4 be the color of the vertices that belong to the smallest color class and let the other colors be c_1, c_2 and c_3. We first assign to each vertex v with color c_4 a different color $c \in \{c_1, c_2, c_3\}$. If all three neighbors of v have different colors, we assign any one of the three colors c_1, c_2, c_3 to v. Otherwise, we assign v the color that is not assigned to any of its neighbors. We then subdivide each of the three edges incident to v with a vertex u such that u is assigned a color c_1, c_2 or c_3, which is not assigned to the end vertices of the edge. It is now straightforward to observe that the resulting subdivision G' of G is acyclically colored with 3 colors. Since the number of vertices with color c_4 is at most $n/4$, the number of division vertices in G' is at most $3n/4$.

In the following two lemmas we show two subclasses of cubic graphs for which we can obtain acyclic 3-colorings using smaller number of division vertices.

Lemma 1. *Let G be a triconnected plane cubic graph with n vertices. Then G has a subdivision G' with at most one division vertex per edge that is acyclically 3-colorable and has at most $n/2$ division vertices.*

Proof. Let $\pi = \{V_1, V_2, \ldots, V_k\}$ be a canonical decomposition of G. G_1 is a cycle, which can be colored acyclically with three colors c_1, c_2 and c_3. Since every vertex of G has degree three, each $V_i, 1 < i < k$, has exactly two neighbors in G_{i-1}. Therefore, V_i corresponds to an ear of G_i and $V_1 \cup \ldots \cup V_{k-1}$ corresponds to an ear decomposition of G_{k-1}. By Fact 2, G_{k-1} has a subdivision G'_{k-1} that is acyclically 3-colorable with at most $k-2$ division vertices. We now add the singleton set V_k to G'_{k-1}. First, suppose that all the three neighbors of V_k have the same color c_1. Then V_k is assigned color c_2 and any two edges incident to V_k are subdivided with division vertices of color c_3 as in Figure 4(a). In all other cases, at most one edge incident to V_k is subdivided. Thus any cycle that passes through the vertex V_k uses three different colors. Since G'_{k-1} has at most $k-2$ division vertices and the last partition needs at most two division vertices, the total number of division vertices in the subdivision G' of G is equal to the

number of partitions in π. Note that the addition of V_k creates two inner faces and the addition of each V_i, $1 < i < k$, creates one inner face. Let the number of inner faces of G be F. Then the number of partitions is $F - 1 = n/2$ by Euler's formula. Therefore, G' has at most $n/2$ division vertices. □

Lemma 2. *Any Hamiltonian cubic graph G, not necessarily planar, with n vertices has a subdivision G' that is acyclically 3-colorable and has $n/2+1$ division vertices.*

Proof. Let C be a Hamiltonian cycle in G. Since the number of vertices in G is even by the degree-sum formula, we can color the vertices on C with colors c_1 and c_2. We next subdivide an edge on C and each of the other edges in G that are not on C to get G'. We assign color c_3 to all the division vertices. See Figure 4(b).

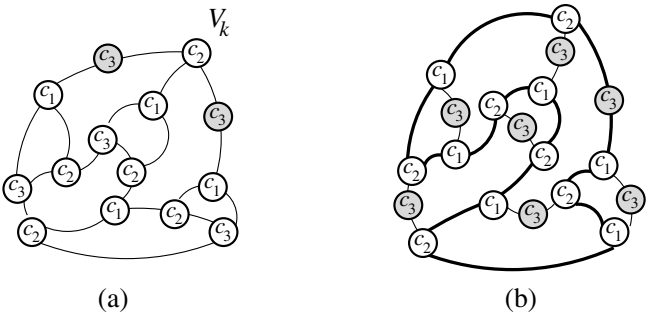


Fig. 4. Illustration for the proof of (a) Lemma 1 and (b) Lemma 2

Each cycle C' in G' corresponds to a unique cycle C'' in G that contains only the original vertices of C'. If C' in G' corresponds to the Hamiltonian cycle C in G, then C' is not bichromatic. Since every vertex in G' has degree at most 3, no cycle can be formed with only edges that are not on C in G. Let C' be a cycle in G' that corresponds to a cycle C'' in G where $C'' \neq C$. Then C'' must contain at least one edge e on C and one edge e' not on C. According to the coloring of G', the end vertices of e in G must have different colors c_1, c_2 and the division vertex on the edge e' has the remaining color c_3. Therefore G' does not contain any bichromatic cycle. The total number of edges in G is $3n/2$. We have subdivided all the edges of G other than $(n-1)$ edges on C. As a result, the total number of division vertices in G' is $3n/2 - (n-1) = n/2 + 1$. □

A graph G with n vertices is a *k-tree* if G satisfies the following (a)-(b).

(a) If $n = k$, then G is the complete graph with k vertices.
(b) If $n > k$, then G can be constructed from a k-tree G' with $n-1$ vertices by adding a new vertex to exactly k vertices of G', where the induced graph of these k-vertices is a complete graph.

Let G be a k-tree with vertex set V. Then by definition, there is an ordered partition $\pi = (V_1, V_2, ..., V_m)$ of V that satisfies the following:

(a) V_1 contains k vertices inducing a complete graph.
(b) Let G_k, $1 \leq k \leq m$, be the subgraph of G induced by $V_1 \cup V_2 \cup ... \cup V_k$. Then G_k, $k > 1$, is a k-tree obtained by adding V_k to G_{k-1}, where V_k is a singleton set and its neighbors in G_{k-1} form a k-clique.

A partial k-tree is a subgraph of a k-tree. It is straightforward to observe that k-trees are acyclically $(k+1)$-colorable.

Lemma 3. *For $k \leq 8$, every partial k-tree G with n vertices has a subdivision G' with at most one division vertex per edge that is acyclically 3-colorable.*

Proof. For $n \leq 3$, G is itself acyclically 3-colorable. We thus assume that $n > 3$ and that all partial k-trees with less than n vertices have a subdivision with at most one division vertex per edge that is acyclically 3-colorable. Let G be a partial k-tree obtained from a k-tree K. Let $\pi = (V_1, V_2, ..., V_m)$ be an ordered partition of the vertex set of K and let $\pi' = (V'_1, V'_2, ..., V'_{m'})$ be an ordered partition of the vertex set of G, where $V'_1 \subseteq V_1$ and $V'_2, ..., V'_{m'}$ is a subsequence of $V_2, ..., V_m$. Now we add $V'_{m'}$ to $G_{m'-1}$ to obtain G. By induction $G_{m'-1}$ has a subdivision $G'_{m'-1}$ that is acyclically 3-colorable, where the number of division vertices per edge of $G_{m'-1}$ is at most one. Let $V'_{m'} = v$. By definition of k-tree, v is connected to at most k original vertices of $G'_{m'-1}$. However, the neighbors of v may not induce a complete graph since G is a partial k-tree. Let G'' be the graph obtained by adding v to $G'_{m'-1}$. Then G'' is a subdivision of G. To get G' from G'', we consider the following three cases.

Case 1 : The neighbors of v in G'' have the same color c_1. Assign color c_2 to v and subdivide each edge (v, u), where u is a neighbor of v. Finally, assign color c_3 to all these new division vertices. See Figure 5(a). Thus any cycle that passes through v uses three different colors.

Case 2 : The neighbors of v in G'' have color c_1 and c_2. Then assign color c_3 to v. For each neighbor u of v, if u has color c_1, subdivide the edge (v, u) and assign color c_2 to the division vertex. Similarly, for each neighbor u of v, if u has color c_2, subdivide the edge (v, u) and assign color c_1 to the division vertex as in Figure 5(b). So any cycle that passes through v, uses three different colors.

Case 3 : The neighbors of v have all three colors c_1, c_2 and c_3. Since $k \leq 8$ there is at least one color assigned to less than or equal to two neighbors of v. Let the color be c_3. Assign color c_3 to v. If only one neighbor u_1 of v has color c_3, subdivide edge (v, u_1) and assign color c_1 to the division vertex. If two neighbors u_1, u_2 of v have color c_3, subdivide each of the edges (v, u_1) and (v, u_2) once. Then assign color c_1 to the division vertex of edge (u, v_1) and color c_2 to the division vertex of edge (u, v_2). For each neighbor $u \notin \{u_1, u_2\}$ of v, if u has color c_1, then subdivide the edge (v, u) and assign color c_2 to the division vertex. Similarly, for each neighbor u of v, if u has color c_2, then subdivide the edge (v, u) and assign color c_1 to the division vertex. See Figure 5(c). Note that any cycle that passes through v but does not contain both u_1 or u_2, must have three

different colors. Any cycle that passes through v, u_1 and u_2 has color c_3 on v and colors c_1, c_2 on the two division vertices on the edges $(v, u_1), (v, u_2)$.

In all the three cases above, any cycle that passes through vertex v is not a bichromatic cycle. All the other cycles are cycles of $G'_{m'-1}$, thus are not bichromatic. Thus the computed coloring in each of Cases 1–3 is an acyclic 3-coloring of a subdivision G' of G. By construction, the number of division vertices on each edge of G is at most one.

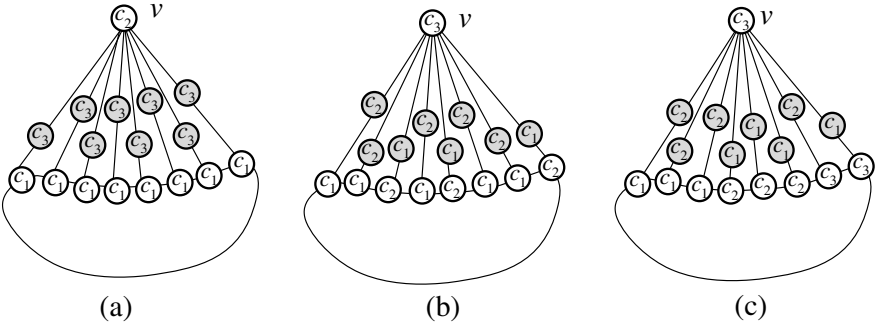


Fig. 5. Illustration for the proof of Lemma 3

□

An *independent set* S of G is a set of vertices in G, where no two vertices in S are adjacent in G. The following lemma is of independent interest.

Lemma 4. *Let S be an independent set of a graph G. If $G - S$ is acyclic then G is acyclically* 3*-colorable.*

Proof. If $G - S$ is acyclic then $G - S$ is a tree or a forest and hence, it is 2–colorable. Color the vertices of $G - S$ with colors c_1 and c_2. Add the vertices of S to $G - S$ and assign the vertices color c_3. Since S is an independent set, a cycle in G contains at least one edge (u_1, u_2) from $G - S$ and at least one vertex u_3 from S. Since, by the coloring method given above, u_1, u_2 and u_3 have different colors, there is no bichromatic cycle in G. □

3 Acyclic Coloring of Plane Graphs

In this section we prove our results for acyclic 3 and 4-colorability of plane graph subdivisions. We first introduce "canonical ordering" of triangulated plane graphs. Let G be a triangulated plane graph on $n \geq 3$ vertices. We denote by $C_0(G)$ the outer cycle of G. Let the three vertices on $C_0(G)$ be v_1, v_2 and v_n in counterclockwise order. Let $\pi = (v_1, v_2, \ldots, v_n)$ be an ordering of all vertices in G. For each integer k, $3 \leq k \leq n$, we denote by G_k the plane subgraph of G induced by the k vertices $v_1, v_2, \ldots, v_k$. We call π a *canonical ordering* of G with respect to the outer edge (v_1, v_2) if it satisfies the following conditions [12]:

(a) G_k is 2-connected and internally triangulated for each k, $3 \leq k \leq n$.
(b) If $k+1 \leq n$, then the vertex v_{k+1} is located on the outer face of G_k, and all neighbors of v_{k+1} in G_k appear on $C_0(G_k)$ consecutively.

Observe that the vertex partition obtained by a canonical decomposition of a triangulated plane graph G determines a vertex ordering, which corresponds to a canonical ordering of G.

Let E^* be the set of edges that do not belong to any $C_0(G_k)$, $3 \leq k \leq n$. We call these edges the *internal edges* of G because they never appear on the outer face of any G_k. We call all the other edges of G, the *external edges*. Let $V^* = V - \{v_1, v_2\}$ and let $G^* = (V^*, E^*)$. Now we prove that G^* is a tree.

Lemma 5. *For any triangulated plane graph G with a canonical ordering $\pi = (v_1, v_2, \ldots, v_n)$, the subgraph $G^* = (V^*, E^*)$ is a tree.*

Proof. We prove that G^* is a tree by first showing that G^* is connected and then showing that $|E^*| = |V^*| - 1$.

To show that G^* is connected, we show that each internal node v_k, $3 \leq k \leq n$, has a path to v_n in G^*. For a contradiction, let k be the maximum index such that v_k, $k < n$, does not have such a path to v_n. Since $v_k \in C_0(G_k)$ but $v_k \notin C_0(G)$, there exists an integer l, $k < l \leq n$, such that $v_k \in C_0(G_{l-1})$ but $v_k \notin C_0(G_l)$. Hence by property (b) of π, (v_k, v_l) must be an internal edge in G. Since $l > k$, by assumption there must be a path from v_l to v_n in G^*. Therefore v_k has a path to v_n in G^* which is a contradiction.

Each v_k, $3 \leq k \leq n$, is connected to G_{k-1} by exactly two external edges. Since (v_1, v_2) is also an external edge, the number of external edges in G is $2(n-2)+1 = 2n-3$. By Euler's formula, G has $3n-6$ edges in total. Therefore, $|E^*| = 3n-6-(2n-3) = n-3 = |V^*|-1$. Therefore, G^* is a tree. □

We use Lemma 5 to prove the following theorem on acyclic 3-colorability of subdivisions of triangulated plane graphs. This theorem is originally proved by Angelini and Frati [2]. However, our proof is simpler and relates acyclic coloring of graph subdivisions with canonical ordering, which is an important tool for developing graph algorithms.

Theorem 1. *Any triangulated plane graph G has a subdivision G' with one division vertex per edge that is acyclically 3-colorable.*

Proof. Let $G = (V, E)$ be a triangulated plane graph and let $\pi = (v_1, v_2, ..., v_n)$ be a canonical ordering of G. Let E' be the set of external edges and let $E^* = E - E'$ be the set of internal edges of G. Let $G_s = (V, E')$. We now compute a subdivision G'_s of G_s and color G'_s acyclically with three colors as follows.

We assign colors c_1, c_2 and c_3 to the vertices v_1, v_2 and v_3, respectively. For $3 \leq k \leq n$, as we traverse $C_0(G_k)$ in clockwise order starting at v_1 and ending at v_2, let l_{v_k} be the first neighbor of v_k encountered and let r_{v_k} be the other neighbor of v_k on $C_0(G_k)$. Then assign v_k a color other than the colors of l_{v_k} and r_{v_k}.

We now subdivide each edge in E' with one division vertex to get G'_s. Finally, we assign each division vertex a color other than the colors of its two neighbors. It is easy to see that every edge in G_s along with its division vertex uses three different colors. Therefore, the resulting coloring of G'_s is an acyclic 3-coloring.

We now add the edges of E^* to G'_s and subdivide each of these edges with one division vertex to obtain G'. We assign each new division vertex a color other than the colors of its two neighbors. By Lemma 5, E^* is the edge set of a tree. Therefore, any cycle in G' must contain an edge from E'. Consequently, the cycle must use three different colors. Figure 6(a) shows an example of G', where the edges of E' are shown by solid lines and the edges of E^* are shown by dashed lines. □

We now extend the technique used in the proof of Theorem 1 to obtain the following theorem on acyclic 4-colorability of triangulated plane graphs.

Theorem 2. *Any triangulated plane graph G has a subdivision G' with at most one division vertex per edge that is acyclically 4-colorable, where the number of division vertices in G' is at most $2n - 6$.*

Proof. We define π and G_s as in the proof of Theorem 1. We first compute a subdivision G'_s of G_s and color G'_s acyclically with three colors as follows. We assign colors c_1, c_2 and c_3 to the vertices v_1, v_2 and v_3, respectively. For $3 \leq k \leq n$, as we traverse $C_0(G_k)$ in clockwise order starting at v_1, let l_{v_k} be the first neighbor of v_k encountered and let r_{v_k} be the other neighbor of v_k on $C_0(G_k)$. Then for each vertex v_k we consider the following two cases.

Case 1: The colors of l_{v_k} and r_{v_k} are the same. In this case we assign v_k a color other than the color of l_{v_k} and r_{v_k}. Then we subdivide edge (v_k, r_{v_k}) with one division vertex and assign the division vertex a color other than the colors of its two neighbors.

Case 2: The colors of l_{v_k} and r_{v_k} are different. In this case we assign v_k a color other than the color of l_{v_k} and r_{v_k} and do not subdivide any edge.

At each addition of v_k, Cases 1 and 2 ensure that any cycle passing through v_k has three different colors. Hence, the resulting subdivision is the required G'_s and the computed coloring of G'_s is an acyclic 3-coloring.

We now add the edges of E^* to G'_s and subdivide each of these edges with one division vertex to obtain G'. We assign each new division vertex the fourth color. Any cycle that does not contain any internal edge is contained in G'_s and hence, uses three different colors. On the other hand, any cycle that contains an internal edge must use the fourth color and two other colors from the original vertices on the cycle. Therefore, the computed coloring of G' is an acyclic 4-coloring. Figure 6(b) shows an example of G'. We have not subdivided any edges between the vertices v_1, v_2 and v_3. Moreover, for each v_k, we subdivided exactly one external edge. Therefore the number of division vertices is at most $(3n - 6) - (n - 3) - 3 = 2n - 6$. □

Observe that canonical ordering and Schnyder's realizer of a triangulated plane graph are equivalent notions [11]. Using the fact that $G^* = (V^*, E^*)$ is a tree of Schnyder's realizer [5], one can obtain alternate proofs for Theorems 1 and 2.

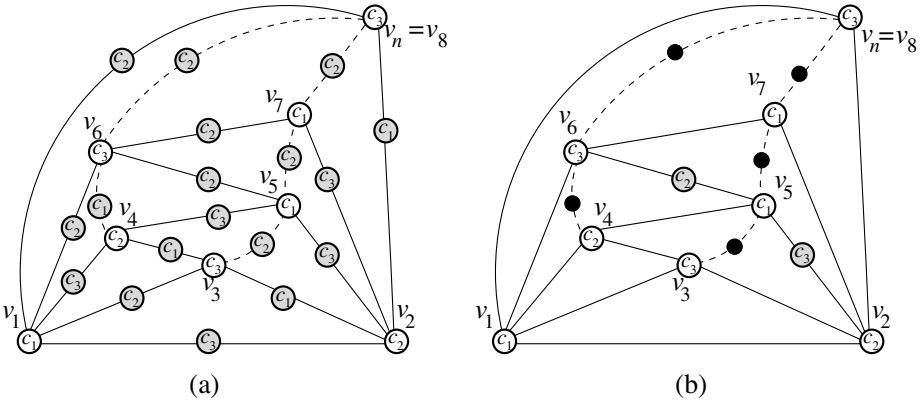


Fig. 6. Illustration for the proof of (a) Theorem 1 and (b) Theorem 2

4 NP-Completeness

In this section we prove that it is NP-complete to decide whether a graph with maximum degree 7 has an acyclic 4-coloring or not. We denote the problem by Acyclic 4-Color Max-deg 7. The equivalent decision problem is given below.

Instance: A graph G with maximum degree 7.
Question: Can the vertices of G be acyclically colored with 4 colors?

Theorem 3. Acyclic 4-Color Max-deg 7 *is NP-complete.*

Proof. The problem is in NP. If a valid 4-coloring of the vertices of G is given, we can check in polynomial time whether that is an acyclic coloring or not. We consider each pair of colors and the subgraph induced by the vertices of those two colors. We check whether that subgraph contains a cycle. If none of the $\binom{4}{2}$ subgraphs contains any cycles, the 4-coloring is an acyclic coloring.

We will prove the NP-hardness by reducing the problem of deciding acyclic 3-colorability of plane graphs with maximum degree 4 to our problem. The problem of acyclic 3-colorability, which was proved to be NP-complete by Angelini and Frati [2], is given below.

Instance: A plane graph H with maximum degree 4.
Question: Can the vertices of H be acyclically colored with 3 colors?

Let H be an instance of the problem of deciding acyclic 3-colorability of plane graphs with maximum degree 4, as in Figure 7. Let p be the number of vertices in H. Then we construct a plane 3-tree G_{4p} of $4p$ vertices as in Figure 7 as follows. We first take a triangle with vertices v_1, v_2 and v_3. Next we take a vertex v_4 in the inner face of the triangle and connect v_4 to v_1, v_2 and v_3 to get G_4. In any valid coloring of G_4, v_1, v_2, v_3 and v_4 must be assigned four different colors and hence the coloring is acyclic. Let the colors assigned to v_1, v_2, v_3 and v_4 be c_1, c_2, c_3 and c_4, respectively. Now we place a new vertex v_5 inside the face

bounded by the triangle v_2, v_3, v_4 and connect v_5 with the three vertices on the face to get G_5. It is obvious that G_5 is 4–colorable and v_5 must be assigned the same color as v_1 in a 4-coloring of G_5. In this recursive way, we construct the graph G_{4p} with $4p$ vertices, where each inner vertex of G_{4p} has degree exactly six. In any valid 4-coloring of G_{4p}, each of the four colors is assigned to exactly p vertices.

We now prove that any valid 4-coloring of a plane 3-tree G_n with n vertices is an acyclic coloring. The proof is by induction on n. When $n \leq 4$, any 4-coloring of G_3 is an acyclic coloring. We thus assume that $n > 4$ and that any valid 4-coloring of a plane 3-tree with less than n vertices is an acyclic coloring. By definition of plane 3-tree, G_n has a vertex v of degree three. We remove v from G_n to get another plane 3-tree G_{n-1} with $n-1$ vertices. By the induction hypothesis, any 4-coloring of G_{n-1} is an acyclic coloring. We now add v to G_{n-1} to get G_n. By construction of plane 3-trees, v must be placed in a face of G_{n-1} and must be connected to the three vertices on the face. Let the colors assigned to three neighbors of v be c_1, c_2 and c_3. Then v is assigned color c_4. Now, any cycle that goes through v must also go through at least two of the neighbors of v. Hence any cycle containing v contains vertices of at least three colors. Therefore, G_n has no bichromatic cycle.

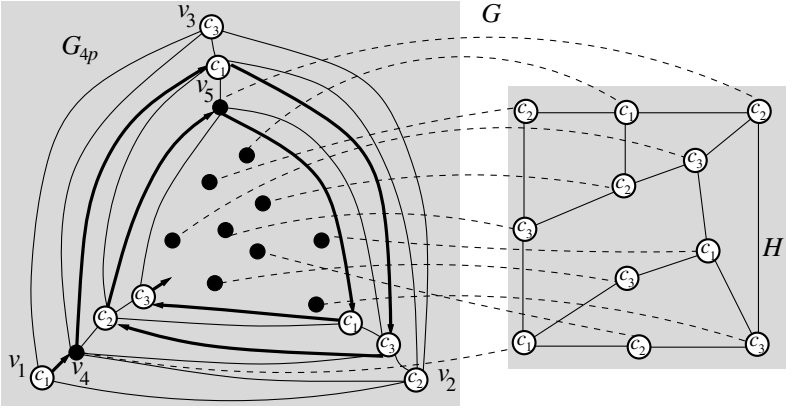


Fig. 7. Illustration for the proof of Theorem 3

Let S be the set of the vertices of G_{4p} that are assigned color c_4 in a 4-coloring when the outer vertices use the colors c_1, c_2, c_3. We connect each vertex of H to exactly one vertex of S as illustrated in Figure 7, so that the edges connecting vertices of H and G_{4p} form a matching. Let the resulting graph be G. It is easy to see that the degree of each vertex of G is at most seven. We argue that G has an acyclic 4-coloring if and only if H has an acyclic 3-coloring.

First we assume that G admits an acyclic 4-coloring. Let the colors assigned to the vertices of G be c_1, c_2, c_3 and c_4. Let the colors assigned to the outer vertices of G_{4p} be c_1, c_2 and c_3. Then each vertex in S has color c_4 and hence no

vertex in H receives color c_4. Therefore, the vertices of H are acyclically colored with three colors c_1, c_2 and c_3.

We now assume that H has an acyclic 3-coloring where the colors assigned to the vertices of H are c_1, c_2 and c_3. We assign the three colors to the three outer vertices v_1, v_2 and v_3 of G_{4p}. Clearly the common neighbor v_4 of the three outer vertices must be assigned a fourth color c_4. In the same way, all the vertices of S get the color c_4. Suppose for a contradiction that G contains a bichromatic cycle C. C cannot be a cycle of H. Since any 4-coloring of G_{4p} is acyclic, C cannot be a cycle of G_{4p}. Therefore, C must contain vertices from both G_{4p} and H. Since the edges connecting G_{4p} and H form a matching, no two vertices of G_{4p} have the same neighbor in H. Therefore, C must contain at least one edge e of H. The end vertices of e have two of the three colors c_1, c_2, c_3. Since C must contain a vertex in G_{4p} with color c_4, C contains vertices of at least three colors and hence cannot be a bichromatic cycle. Therefore, the 4-coloring of G described above is acyclic. □

5 Open Problems

Acyclic colorings of plane graph subdivisions with fewer division vertices will be an interesting direction to explore. We ask the following question:

What the minimum positive constant c such that every triangulated planar graph with n vertices has an acyclic k-coloring, $k \in \{3, 4\}$, with at most cn division vertices?

Every cubic graph is acyclically 4-colorable [14]. On the other hand, we have proved that testing acyclic 4-colorability is NP-complete for graphs with the maximum degree 7. The problem of obtaining acyclic 4-colorings for graphs with maximum degree greater than three and less than seven remains open, as does using our results to improve volume bounds on 3-dimensional polyline grid drawings.

References

1. Albertson, M.O., Berman, D.M.: Every planar graph has an acyclic 7-coloring. Israel Journal of Mathematics 28(1-2), 169–174 (1977)
2. Angelini, P., Frati, F.: Acyclically 3-colorable planar graphs. In: Rahman, M. S., Fujita, S. (eds.) WALCOM 2010. LNCS, vol. 5942, pp. 113–124. Springer, Heidelberg (2010)
3. Borodin, O.V.: On acyclic colorings of planar graphs. Discrete Mathematics 306(10-11), 953–972 (2006)
4. Coleman, T.F., Cai, J.: The cyclic coloring problem and estimation of sparse hessian matrices. SIAM Journal on Algebraic and Discrete Methods 7(2), 221–235 (1986)
5. Dhandapani, R.: Greedy drawings of triangulations. Discrete & Computational Geometry 43(2), 375–392 (2010)
6. Dujmović, V., Morin, P., Wood, D.R.: Layout of graphs with bounded tree-width. SIAM Journal of Computing 34, 553–579 (2005)

7. Gebremedhin, A.H., Tarafdar, A., Pothen, A., Walther, A.: Efficient computation of sparse hessians using coloring and automatic differentiation. INFORMS Journal on Computing 21, 209–223 (2009)
8. Grünbaum, B.: Acyclic colorings of planar graphs. Israel Journal of Mathematics 14(4), 390–408 (1973)
9. Kostochka, A.V.: Acyclic 6-coloring of planar graphs. Diskretn. Anal. 28, 40–56 (1976)
10. Mitchem, J.: Every planar graph has an acyclic 8-coloring. Duke Mathematical Journal 41(1), 177–181 (1974)
11. Miura, K., Azuma, M., Nishizeki, T.: Canonical decomposition, realizer, Schnyder labeling and orderly spanning trees of plane graphs. International Journal of Foundations of Computer Science 16(1), 117–141 (2005)
12. Nishizeki, T., Rahman, M.S.: Planar Graph Drawing. World Scientific (2004)
13. Ochem, P.: Negative results on acyclic improper colorings. In: European Conference on Combinatorics (EuroComb 2005), pp. 357–362 (2005)
14. Skulrattanakulchai, S.: Acyclic colorings of subcubic graphs. Information Processing Letters 92(4), 161–167 (2004)
15. Wood, D.R.: Acyclic, star and oriented colourings of graph subdivisions. Discrete Mathematics & Theoretical Computer Science 7(1), 37–50 (2005)

Kinetic Euclidean Minimum Spanning Tree in the Plane

Zahed Rahmati and Alireza Zarei

Department of Mathematical Sciences,
Sharif University of Technology, Tehran, Iran
{rahmati@alum.,zarei@}sharif.edu

Abstract. This paper presents the first kinetic data structure (*KDS*) for maintenance of the Euclidean minimum spanning tree (*EMST*) on a set of n moving points in 2-dimensional space. We build a *KDS* of size $O(n)$ in $O(n \log n)$ preprocessing time by which their *EMST* is maintained efficiently during the motion. In terms of the *KDS* performance parameters, our *KDS* is *responsive*, *local*, and *compact*.

Keywords: computational geometry; Euclidean minimum spanning tree; kinetic data structures.

1 Introduction

Problem statement. For a weighted graph $G(V, E)$, a minimum spanning tree of G is a connected sub-graph $G'(V, E')$ of G where the sum of the weights of its edges is the minimum possible. For a set $P = \{p_1, p_2, \ldots, p_n\}$ of n points, there is a complete weighted graph with P as its nodes. The weight of each edge of this graph is the Euclidean distance between its two endpoints. A minimum spanning tree of this graph is known as the Euclidean minimum spanning tree (*EMST*) of the underlying points.

The *EMST* has many applications in solving geometric and graph problems and has been studied extensively. Some of which are described below.

In this paper, we consider the kinetic version of the *EMST* problem in the plane. This problem was first posed by Basch *et al.* [4] and has been open since 1997. In this setting, the points are moving independently in the plane and the goal is to maintain the combinatorial structure of their *EMST* during the motion. We assume that the position of a point p_i at time t, denoted by $p_i(t) = (x_i(t), y_i(t))$, is defined by two algebraic functions (for x and y coordinates) of constant maximum degree (in terms of time). Moreover, we assume that the points move without collision, *i.e.* $\forall_{i \neq j} \forall_{t \in \mathcal{R}} p_i(t) \neq p_j(t)$. This assumption is required by the kinetic Delaunay triangulation algorithm on which our solution is built.

Related work. Whereas *EMST* is a special version of the minimum spanning tree problem, we can use any of the classic minimum spanning tree algorithms to solve the *EMST* problem. Therefore, the *EMST* can be obtained in $O(E+n \log n)$

C.S. Iliopoulos and W.F. Smyth (Eds.): IWOCA 2011, LNCS 7056, pp. 261–274, 2011.
© Springer-Verlag Berlin Heidelberg 2011

time using the Prim's algorithm [14] or in time $O(E \log E)$ using the Kruskal's algorithm [12] where $E = O(n^2)$ is the number of the edges in the complete graph over the n points. Based on the geometric properties of the *EMST*, this problem can be solved in optimal $O(n \log n)$ time in the plane [7].

A good method for computing the *EMST* in the plane obtained just after the Delaunay triangulation (*DT*) problem was solved in $O(n \log n)$ time [6,11]. The edges of the *EMST* of points P are a subset of the edges of their *DT* and the number of the edges of the *DT* is $O(n)$ which result a $O(n \log n)$ time algorithm for the *EMST* problem using the Prim or Kruskal algorithms.

Afterward, the *dynamic* and *kinetic* versions of this problem were investigated where respectively, point insertion/deletion and point movement are allowed. Eppstein [8] proposed an algorithm that maintains the *EMST* of a set of points in the plane allowing point insertion and deletion (the dynamic version). This algorithm requires $O(n^{\frac{1}{2}} \log^2 n)$ amortized time per update (point insertion/deletion). The first algorithm for the kinetic situation was proposed by Fu and Lee [9]. Their algorithm requires $O(kn^4 \log n)$ preprocessing time and $O(m)$ space where k is the maximum degree of the algebraic functions defining the points motion and m is the maximum number of the changes of the *EMST* from time $t = 0$ to $t = \infty$. Then, the *EMST* of the points at any given time can be constructed in query time $O(n)$. For the restricted version of the kinetic *EMST* problem, in which the distances between each pair of points are defined by linear functions of time, Agarwal *et al.* [1] proposed an algorithm that runs in time $O(n^{\frac{1}{2}} \log^{\frac{3}{2}} n)$ per combinatorial change of the *EMST*. They propose this algorithm for maintaining the minimum spanning tree of a general graph where the edge weights are linear functions of time and supports edge insertion and deletion as well.

The kinetic data structure framework. The kinetic data structure (*KDS*) framework was initially introduced by Basch *et al.* [3]. In this framework, a set of *certificates* is defined to maintain a special attribute of a set of moving objects. Validity of these certificates implies the correctness of the goal attribute. Whenever a certificate fails, it means that the computed value of the attribute must be updated and the new set of certificates must be built. Therefore, it is enough to compute the failure time of these certificates, called *events*, and put them in an event queue.

The most important part of this framework is a set of criteria that determines the performance of a *KDS*. The performance of a *KDS* is measured according to the following four criteria [2,3]:

- **Responsiveness:** The response time of a *KDS* is the processing time of an event spent by the repair mechanism. If the response time is a polylogarithmic function of the number of the moving objects, the *KDS* is *responsive*.
- **Compactness:** The size of a *KDS* is defined by the space used by its data structures and certificates. A *KDS* is called *compact* if its size is within a polylogarithmic factor of linear in the total number of the moving objects.
- **Locality:** The locality of a *KDS* is defined by the maximum number of events associated with one particular object, at any fixed time. A *KDS* is

called *local* if this number is always a polylogarithmic function of the total number of the moving objects.

- **Efficiency:** The efficiency of a *KDS* deals with the number of events processed during the motion. Not every event (certificate failure) of a *KDS* necessarily implies a change in the attribute being maintained. Processing an event may produce only internal changes to the data structures while the desired attribute is still valid. These events are called *internal events*. An event that produces a change in the target attribute is called an *external event*. The efficiency of a *KDS* is defined as the ratio between the total number of the internal events and the total number of the external events (enumerated from time $t = 0$ to $t = \infty$). A *KDS* is called *efficient* if this ratio is polylogarithmic in the number of the moving objects.

Our Results. To the best of our knowledge there is no algorithm for solving the *exact EMST* problem in general kinetic settings satisfying the *KDS* performance metrics. This was our motivation in considering this problem.

It is known that the edges of the *EMST* of a set of points are a subset of the edges of their Delaunay triangulation (*DT*). Using this fact, we use the *KDS* proposed by Guibas *et al.* [10] for tracking changes of the *DT*. As soon as a *DT* change happens the necessary updates are applied on the *EMST*. Moreover, if the ordering of the edge-lengths of two edges of the *DT* is changed, it may produce a change in the *EMST*. One can maintain the edges of the *DT* in a sorted list and whenever the ordering of two edges in this list is changed, apply the required changes to the *EMST*. Our contribution in this paper is to do this while satisfying some of the performance criteria of the *KDS* framework.

Besides one instance of the *KDS* proposed by Guibas *et al.* [10] for maintaining the *DT*, we build a set of data structures of total size $O(n)$ in $O(n \log n)$ preprocessing time by which a certificate failure (event) is handled in $O(\log^2 n)$ time. According to the *KDS* performance metrics, our *KDS* is responsive, local in expectation and compact.

2 Certificates and Events

As mentioned in the introduction, our approach is based on the fact that the edges of the *EMST* of a set of points is a subset of the edges of their Delaunay triangulation. On the other hand, the minimum spanning tree of a graph depends on the orderings of its edge weights. Therefore, we need to track two types of changes to correctly update and maintain the *EMST* of a set of moving points:

1. Changes in the order of each pair of consecutive edges in the sorted list (according to the edge's length) of the Delaunay triangulation edges of the points.
2. Changes in the Delaunay triangulation of the points which causes an edge removal from or an edge insertion into the potential edges of the *EMST*.

Let $\mathcal{E}(DT)$ and $\mathcal{E}(EMST)$ be respectively the set of edges of the Delaunay triangulation and the edges of the $EMST$ of a set of moving points P in the plane and let $path(p_i, p_j)$ be the simple path between p_i and p_j in the $EMST$.

The first change type corresponds to a pair of edges e and e' in $\mathcal{E}(DT)$ such that (for small enough value of ϵ)

- at time $t - \epsilon$ the Euclidean length of e is smaller than that of e', and
- at time $t + \epsilon$ the Euclidean length of e is greater than that of e'.

Then, e may be replaced by e' in $\mathcal{E}(EMST)$ at time t. Such a change is called an *order event* with parameters e, e' and t. It is simple to prove the following lemma about the order-events:

Lemma 1. *An order-event of parameters e, e' and t changes the $EMST$ if and only if at time $t - \epsilon$ we have $e \in \mathcal{E}(EMST)$, $e' \notin \mathcal{E}(EMST)$ and $e \in path(p_i, p_j)$ where p_i and p_j are the end points of e'.*

We call such order-events *effective order-events*.

The second change type means that there is a pair of edges e and e' such that (for small enough value of ϵ)

- at time $t - \epsilon$ we have $e \in \mathcal{E}(DT)$ and $e' \notin \mathcal{E}(DT)$,
- at time t the four endpoints of e and e' lie on a circle, and
- at time $t + \epsilon$ we have $e \notin \mathcal{E}(DT)$ and $e' \in \mathcal{E}(DT)$.

Then, e' may appear in $\mathcal{E}(EMST)$ at some time $t' > t$. Such a change is called a *DT-event* with parameters e, e' and t. If we add the points of a sufficiently large bounding box of the points to the set of input points, the convex hull of the points is always this box and DT-events do not affect it. Having this bounding box, we prove that a DT-event does not directly affect the $EMST$.

Lemma 2. *For any DT-event of parameters e, e' and t there is an arbitrarily small value of ϵ such that neither e nor e' exist in $\mathcal{E}(EMST)$ at times $t - \epsilon$ and $t + \epsilon$.*

Proof. We prove this lemma by showing that exactly at time t and before removing e from the Delaunay triangulation $e \notin \mathcal{E}(EMST)$ and exactly at time t and after adding e' to the Delaunay triangulation $e' \notin \mathcal{E}(EMST)$. Any DT-event has a configuration shown in Figure 1 where four points lie on a circle. In this figure $e = p_2p_4$ and $e' = p_1p_3$. Assume that p_1 is the point on the opposite side of e relative to the center of this circle (if e is a diameter of this circle then there is no distinction between the points p_1 and p_3 in this proof). Then, $|p_1p_2| < |e|$ and $|p_1p_4| < |e|$. We prove by contradiction that $e \notin \mathcal{E}(EMST)$ at time $t - \epsilon$. If $e \in \mathcal{E}(EMST)$, at most one of the edges p_1p_2 and p_1p_4 can be a member of $\mathcal{E}(EMST)$. Without loss of generality, assume that $p_1p_2 \notin \mathcal{E}(EMST)$. Then, either $e \in path(p_1, p_2)$ which we can obtain a smaller $EMST$ by using p_1p_2 instead of e, or $e \notin path(p_1, p_2)$ which means that $p_1p_4 \notin \mathcal{E}(EMST)$ and $e \in path(p_1, p_4)$ and therefore, we can obtain a smaller $EMST$ by using p_1p_4 instead of e. In both cases we have the contradiction that $e \notin \mathcal{E}(EMST)$ at time $t - \epsilon$. By the same argument we can prove that e' does not exist in the $EMST$ at time $t + \epsilon$. □

According to Lemmas 1 and 2, the actual set of certificates that may affect our goal attribute (the *EMST*) is the set of $O(n)$ certificates that defines the order of the weights of $\mathcal{E}(DT)$; see Theorem 1

Theorem 1. *The EMST of a set of moving points is changed if and only if an effective order-event happens.*

Using this theorem, we can maintain the sorted list of the edges of the Delaunay triangulation and whenever the ordering of a pair of consecutive edges is changed we must check to see whether it defines an effective order-event. If so, the required changes are applied to the *EMST*. Doing this naively, each event can be processed in $O(n)$ time which is not responsive in terms of the *KDS* metrics.

In the next sections, we describe how these certificates are handled to maintain a responsive, local, and compact *EMST* during the motion.

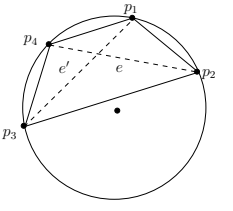


Fig. 1. The *EMST* of a set of points does not contain the degenerate edges of the Delaunay triangulation

However, we need to track and process DT-events to have the correct value of $\mathcal{E}(DT)$ that is necessary to have the correct set of the order-event certificates. We employ the method proposed by Guibas *et al.* [10] to detect the DT-events. In this method, after preprocessing requiring $O(n \log n)$ time and $O(n)$ space, the Voronoi diagram of a set of moving points can be maintained by processing the required events. Any event is processed in $O(\log n)$ time and the total number of processed events from $t = 0$ to $t = \infty$ is $O(n^2 \lambda_s(n))$ where $\lambda_s(n)$ is the maximum length of a Davenport-Schinzel sequence of length n and order s. Thereby, $s = 4q$ where q is the maximum degree of the polynomial curves defining the points motion.

3 Building the Kinetic Data Structure

Besides the data structures and algorithms proposed by Guibas *et al.* [10] which triggering the DT-events, our KDS contains three parts:

- **The DT Edges and Certificates:** We store the Euclidean lengths of $\mathcal{E}(DT)$ in a balanced binary search tree, $T(DT)$, and for each pair of consecutive nodes of this tree we compute the closest time at which the order of these nodes is changed. These times are put in a priority queue $Q(DT)$. The root of $Q(DT)$ contains the closest time at which the order of the lengths of two edges of $\mathcal{E}(DT)$ is changed. Moreover, we make links between the Delaunay triangulation edges and their length entries in $T(DT)$ and $Q(DT)$ for removal purposes.
- **The *EMST* Planar Structure:** Assume that *SD* is the subdivision produced from the overlay of the convex hull of the points (which is the added bounding box) and their *EMST*. This part of our *KDS* is a variation of the

DCEL data structure [5] that maintains the status of *SD* and has three structures for vertices, edges and faces which are respectively denoted by $\mathcal{V}$, $\mathcal{E}$ and $\mathcal{F}$. For each point $p_i \in P$, there is an entry $\mathcal{V}(p_i)$ that points to the root of a search tree in which the edges of *SD* that are adjacent to p_i are sorted according to their radial order around p_i. For each edge e_i in *SD* there is an entry in $\mathcal{E}$ that points to the two directed half-edges of e_i in its both sides. The direction of these half-edges are such that their adjacent faces lie to their left. Moreover, each half-edge have a pointer to its occurrence in $\mathcal{F}$ to be defined as follows. For each face f_i of *SD* there is an entry $\mathcal{F}(f_i)$ that points to the root of a search tree in which the half-edges of the boundary of f_i are sorted according to their order on this boundary. To be precise, each $\mathcal{F}(f_i)$ is a balanced binary search tree that supports merge and split operations efficiently as well as search, insert and delete in $O(\log n)$ time [15]. we denote these trees by MS-BBST. Moreover, we assign the maximum ordering values to the bounding half-edges of a face f_i that are not member of the *EMST* (these half-edges belongs to the convex hull). This convention makes our next discussions easier. Figure 2 sketches a typical configuration of these structures.

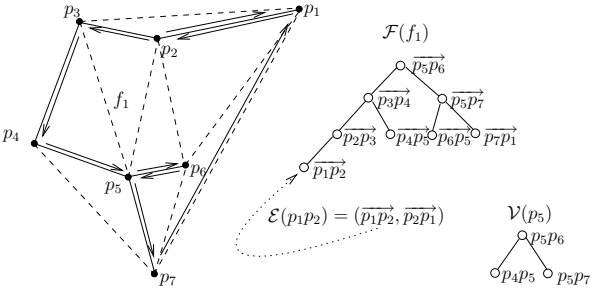


Fig. 2. A typical configuration of the $\mathcal{V}$, $\mathcal{E}$ and $\mathcal{F}$ structures

- **The Relation between $\mathcal{E}(DT)$ and $\mathcal{E}(EMST)$:** The last part of our *KDS* exhibits relations between edges of the current *EMST* and its future potential edges which are those edges of *DT* that are not members of the current *EMST*. In other words, during the motion some edges of $\mathcal{E}(DT) - \mathcal{E}(EMST)$ may be inserted into the *EMST*. We describe next, how these updates are handled by our *KDS*. For each edge $p_ip_j \in \mathcal{E}(DT) - \mathcal{E}(EMST)$ there is a simple path $path(p_i, p_j)$ in the *EMST* that connects p_i and p_j. Assume that p_sp_t has the maximum Euclidean length among edges of $path(p_i, p_j)$. Then, $|p_ip_j| > |p_sp_t|$ and if $|p_ip_j|$ gets to decrease while the points are moving it will be added to the *EMST* just after the moment that its length reaches $|p_sp_t|$ (we assume that p_sp_t has still the maximum Euclidean length among edges of $path(p_i, p_j)$). For such situations we say that p_ip_j is a *potential candidate* for p_sp_t. In other words, it is possible to have an effective order-event of parameters p_sp_t, p_ip_j and t for any potential candidate edge p_ip_j of p_sp_t.

In this part of our *KDS*, we store the set of all potential candidates of each edge $p_sp_t \in \mathcal{E}(EMST)$ and p_sp_t itself in a MS-BBST. In the MS-BBST of an edge p_sp_t denoted by $PK(p_sp_t)$, the ordering of the nodes are according to their *distance* from the edge p_sp_t to be defined later.

Now, we describe how these data structures are initially constructed and analyze their complexities.

Lemma 3. *For a set of n points, their $T(DT)$ and $Q(DT)$ can be constructed in $O(n \log n)$ time and the size of these structures is $O(n)$.*

Proof. We can compute the Delaunay triangulation in $O(n \log n)$ time. We know that $|\mathcal{E}(DT)| = O(n)$. Then, the lengths of these edges are computed in $O(n)$ time and they are inserted in a balanced binary search tree $T(DT)$ in $O(n \log n)$ time. Finally, for each consecutive pair of nodes in $T(DT)$ we compute the time at which this ordering changes and put this event time into a priority queue $Q(DT)$. The size of this queue is also $O(n)$ and can be constructed in $O(n \log n)$ time. □

Lemma 4. *For a set of n points, the $\mathcal{V}$, $\mathcal{E}$ and $\mathcal{F}$ data structures can be constructed in $O(n \log n)$ time and their total size is $O(n)$.*

Proof. After computing the Delaunay triangulation, the *EMST* and the convex hull of the points, the overlay *SD* can be constructed in $O(n)$ time and the subdivision can be obtained as a standard DCEL data structure [5]. Having this DCEL, the $\mathcal{V}$, $\mathcal{E}$ and $\mathcal{F}$ data structures is constructed in a linear trace on this DCEL in $O(n \log n)$ time. The size of the subdivision *SD* is $O(n)$ which implies that the total size of these three data structures is also $O(n)$. □

Before analyzing the *PK* structures we need to know more about them. Assume that $p_sp_t \in \mathcal{E}(EMST)$. Removing this edge from the *EMST* will break this graph into two connected components $C_1(P_1, E_1)$ and $C_2(P_2, E_2)$. Only those edges p_ip_j of $\mathcal{E}(DT)$ that $p_i \in P_1$ and $p_j \in P_2$ can be used to reconnect C_1 and C_2 and obtain a spanning tree (not necessarily minimum). Let denote this set of edges by $cut(p_sp_t)$. Figure 3a shows such cuts for edges p_4p_5 and p_5p_6. It is simple to argue that each edge of $PK(p_sp_t)$ exists in $cut(p_sp_t)$, but the reverse is not necessarily true. For all edges $p_sp_t \in path(p_i, p_j)$, p_ip_j exists in $cut(p_sp_t)$ but p_ip_j belongs only to $PK(p_sp_t)$ where p_sp_t has the maximum length among the edges of the path $path(p_i, p_j)$.

Using the following steps we can find the edges of $PK(p_sp_t)$ for all edges $p_sp_t \in \mathcal{E}(EMST)$:

1. For each face f_k of the *SD* subdivision we build a dual directed tree $D(f_k)$ as follows. Each triangle of DT that is inside the face f_k corresponds to a node in $D(f_k)$ and two nodes of $D(f_k)$ are connected by an edge if their corresponding triangles have an edge in common. The face f_k has exactly one boundary edge which is not in $\mathcal{E}(EMST)$ and is an edge of the convex hull

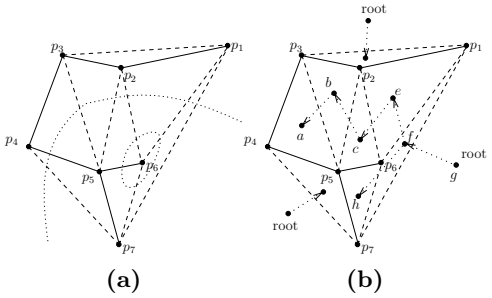


Fig. 3. (a) $cut(p_4p_5) = \{p_4p_7, p_4p_5, p_3p_5, p_2p_5, p_2p_6, p_1p_6, p_1p_7\}$ and $cut(p_5p_6) = \{p_5p_6, p_2p_6, p_1p_6, p_7p_6\}$. (b) Dual trees of the faces of a Delaunay triangulation.

of the points. We add one extra node to $D(f_k)$ as its root that is connected to the node corresponding to the triangle that is adjacent to this convex hull edge (see Figure 3b). The direction of the edges of $D(f_k)$ is set to be from parent to child.

2. We assign a left label $l(\overrightarrow{p_ip_j})$ and a right label $r(\overrightarrow{p_ip_j})$ to each directed edge $\overrightarrow{p_ip_j} \in D(f_k)$ as follows (In the following items $\max(a, b)$ denotes the edge of maximum length among edges a and b):

   2.1. If p_j is a leaf node, both edges of its dual triangle lie on the *EMST*. Let p_mp_l and p_mp_r be these edges which p_l lies on the left of the directed chain $\overrightarrow{p_ip_jp_m}$ and p_r lies on the right of this chain. We set $l(\overrightarrow{p_ip_j})$ and $r(\overrightarrow{p_ip_j})$ to be p_mp_l and p_mp_r, respectively.

   2.2. If p_j has only one child p_k, one of the edges of its dual triangle lies on the *EMST*. Let e be this edge. If e lies on the left (right) of the chain $\overrightarrow{p_ip_jp_k}$, we set $l(\overrightarrow{p_ip_j})$ and $r(\overrightarrow{p_ip_j})$ to be $\max(l(\overrightarrow{p_jp_k}), e)$ and $r(\overrightarrow{p_jp_k})$ ($l(\overrightarrow{p_jp_k})$ and $\max(r(\overrightarrow{p_jp_k}), e)$), respectively.

   2.3. Otherwise, p_j has two children. Assume that p_l is the left child (according to the direction of $\overrightarrow{p_ip_j}$) and p_r is the right one. If $r(\overrightarrow{p_jp_l}) \neq l(\overrightarrow{p_jp_r})$, we set $l(\overrightarrow{p_ip_j})$ and $r(\overrightarrow{p_ip_j})$ to be $\max(l(\overrightarrow{p_jp_l}), l(\overrightarrow{p_jp_r}))$ and $\max(r(\overrightarrow{p_jp_l}), r(\overrightarrow{p_jp_r}))$, respectively. Otherwise, assume that $e = r(\overrightarrow{p_jp_l}) = l(\overrightarrow{p_jp_r})$ and p_m and p_s are the dual nodes of the triangles adjacent to e and e lies to the left of a chain $\overrightarrow{p_{m'}p_mp_{m''}}$ and lies to the right of a chain $\overrightarrow{p_{s'}p_sp_{s''}}$ of $D(f_k)$ (If $l(\overrightarrow{p_mp_{m''}}) = r(\overrightarrow{p_sp_{s''}})$, we use this value ($l(\overrightarrow{p_mp_{m''}})$) as the new value of e and find the nodes p_m and p_s corresponding to this value of e as defined before. This is done until we obtain different values for $l(\overrightarrow{p_mp_{m''}})$ and $r(\overrightarrow{p_sp_{s''}})$.). Then, we set $l(\overrightarrow{p_ip_j})$ and $r(\overrightarrow{p_ip_j})$ to be $\max(l(\overrightarrow{p_jp_l}), l(\overrightarrow{p_mp_{m''}}))$ and $\max(r(\overrightarrow{p_jp_r}), r(\overrightarrow{p_sp_{s''}}))$, respectively. If p_m (p_s) is a leaf node in $D(f_k)$, its dual triangle must have another edge in the *EMST* which is used instead of $l(\overrightarrow{p_mp_{m''}})$ ($r(\overrightarrow{p_sp_{s''}})$) to obtain $l(\overrightarrow{p_ip_j})$ ($r(\overrightarrow{p_ip_j})$).

   For example, assuming that in Figure 3b we have $|p_5p_7| < |p_4p_5| < |p_5p_6|$ and $|p_2p_3| < |p_3p_4| < |p_1p_2|$. Then,

$$l(\overrightarrow{ba}) = p_4p_5,\ r(\overrightarrow{ba}) = p_3p_4;\ l(\overrightarrow{cb}) = p_4p_5,\ r(\overrightarrow{cb}) = p_3p_4;$$
$$l(\overrightarrow{ec}) = p_5p_6,\ r(\overrightarrow{ec}) = p_3p_4;\ l(\overrightarrow{fe}) = p_5p_6,\ r(\overrightarrow{fe}) = p_1p_2;$$
$$l(\overrightarrow{fh}) = p_5p_7,\ r(\overrightarrow{fh}) = p_5p_6;\ l(\overrightarrow{gf}) = p_4p_5,\ r(\overrightarrow{gf}) = p_1p_2;$$

3. An edge $e \in \mathcal{E}(DT) - \mathcal{E}(EMST)$ is added to $PK(e')$ where $e' = \max(l(\overrightarrow{p_ip_j}), r(\overrightarrow{p_ip_j}))$ and $\overrightarrow{p_ip_j}$ is the dual edge of e in a $D(f_k)$ tree. The *distance* of this entry in $PK(e')$ is set to be the height of the subtree of $D(f_k)$ with root p_j. There are two faces adjacent to e' in its left and right sides. Therefore, we may have two edges in $PK(e')$ with the same distance one for each side of e'. In order to obtain a well-defined ordering and distinct values of distances, we set the sign of distances of all edges of $PK(e')$ that come from one side of e' to be positive and the other ones to be negative.
4. For each edge $e' \in \mathcal{E}(EMST)$, it will be added to $PK(e')$ with zero as its distance value.

We used three simple observations inside the above steps which we ignore their proves.

1. $D(f_k)$ is a tree.
2. Each internal node of $D(f_k)$ has at most two children.
3. Each face f_k has exactly one boundary edge that is an edge of the convex hull of the points.

Lemma 5. *The above procedure correctly computes the edges of $PK(e)$ for all edges $e \in \mathcal{E}(EMST)$.*

Proof. According to our definition, an edge $p_ip_j \in \mathcal{E}(DT) - \mathcal{E}(EMST)$ is added to $PK(p_sp_t)$ if and only if p_sp_t exists in $path(p_i, p_j)$ and has the maximum length among edges of $path(p_i, p_j)$.

Assume that after running the above procedure p_ip_j has been added to $PK(p_sp_t)$. This implies that $p_ip_j \in cut(p_sp_t)$ and therefore, p_sp_t is a member of $path(p_i, p_j)$. On the other hand, for each edge $p_{s'}p_{t'} \in path(p_i, p_j)$ there is a path in $D(f_k)$ along which the $p_{s'}p_{t'}$ label can reach to either the left or the right label of the edge a where f_k is the face containing edge p_ip_j and a is the dual edge of p_ip_j in $D(f_k)$. The only exception to this claim is due to the 2.3. step of the building procedure in witch the labels $l(\overrightarrow{p_jp_r})$ and $r(\overrightarrow{p_jp_l})$ are omitted if they are equal. It is simple to prove that such labels do not belong to $path(p_i, p_j)$. For example, the label p_5p_6 can not reach to the edge $\overrightarrow{gf}$ in Figure 3b even if it has the maximum length among all edges of the $EMST$ and it is apparent that $p_5p_6 \notin path(p_7, p_1)$. Therefore, if p_sp_t is the left or right label of the edge a with the maximum length, $|p_sp_t|$ must have the maximum among all edges of $path(p_i, p_j)$ which proves the *only if* part.

To prove the *if* part, assume that p_sp_t exists in $path(p_i, p_j)$ and has the maximum length among edges of $path(p_i, p_j)$ where $p_ip_j \in \mathcal{E}(DT) - \mathcal{E}(EMST)$. Trivially, $p_ip_j \in cut(p_s, p_t)$ and lies inside the face f_k where p_sp_t lies on its boundary. On the other hand, any edge $p_{s'}p_{t'} \in \mathcal{E}(EMST)$ that can be the label of the dual edge a of p_ip_j in $D(f_k)$ lies on the path $path(p_i, p_j)$. These implies

that the label of the edge a must be equal to $p_s p_t$ which means that $p_i p_j$ must be added to $PK(p_s p_t)$. □

Lemma 6. *For a set of n points, the PK structures of all edges of the EMST can be constructed in $O(n \log n)$ time and their total size is $O(n)$.*

Proof. By the same argument the DCEL structure of the *DT*, *EMST* and *SD* of the points can be constructed in $O(n \log n)$ time. Having this DCEL, the first step can be done in $O(n)$ time. The second step can be done in $O(n)$ time as well. Finally, the last steps insert $O(n)$ items into their corresponding MS-BBST in total $O(n \log n)$ time. The number of *PK* structures is $O(n)$ but their total size is still $O(n)$. □

Summarizing the above discussions, we have the following theorem about the complexity of our *KDS*.

Theorem 2. *The proposed KDS can be constructed in $O(n \log n)$ time and requires $O(n)$ space.*

4 Event Handling

In this section we describe how the events are processed to correctly maintain the *EMST* as well as updating our *KDS* during the motion.

4.1 Processing DT-Events

According to Lemma 2, when a DT-event with parameters $p_i p_j$, $p_k p_l$ and t happens it does not have a direct effect on the *EMST* and, therefore, it is enough to update our *KDS* with respect to this event.

First, we update the *PK* data structures. These data structures are affected because the edge $p_i p_j$ must be removed from the *DT* and $p_k p_l$ must be inserted instead. $p_i p_j$ belongs to $PK(e)$ for some $e \in \mathcal{E}(EMST)$ and because we have a pointer from $p_i p_j$ to its position in $PK(e)$ it can be removed in $O(\log n)$ time. Now, we should find the edge e' such that the new edge $p_k p_l$ must be inserted in $PK(e')$. We determine the edge e' by checking the status of the four edges $p_i p_k$, $p_i p_l$, $p_j p_k$ and $p_j p_l$.

Assume that $p_i p_k \in PK(e_1), p_i p_l \in PK(e_2), p_j p_k \in PK(e_3)$ and $p_j p_l \in PK(e_4)$. It is simple to prove that it is impossible to have four distinct values for e_1, e_2, e_3 and e_4. So assume that two of these four edges are the same. Without loss of generality, assume that e_1 is equal to one of the other edges. We continue with two cases where either $e_1 = e_2$ or $e_1 = e_4$. The other case ($e_1 = e_3$) is the same as the case of $e_1 = e_2$.

For the first case where $e_1 = e_2$, the edge $p_k p_l$ must be added to the *PK* data structure of one the edges e_3 or e_4 that has greater Euclidean length. Assuming that $|e_3| > |e_4|$, $p_k p_l$ is added to $PK(e_3)$ and its position (ordering) is just before (resp. after) the position of the edge $p_j p_k$ if e_3 lies before (resp. after) $p_j p_k$ in the ordering of $PK(e_3)$.

In the other case where $e_1 = e_4$ the edge p_kp_l is added to $PK(e_1)$ between positions of the edges p_ip_k and p_jp_l in $PK(e_1)$.

Doing the above updates, we obtain the correct values of PK for the time $t+\epsilon$ just after the event.

The second part of our *KDS*, the $\mathcal{V}$, $\mathcal{E}$ and $\mathcal{F}$ data structures only depend on the subdivision *SD*. This subdivision is the overlay of the *EMST* and the convex hull of the points. DT-events does not change the *EMST*. Hence, a DT-event changes these data structures if it affects the convex hull of the points. According to our bounding box assumption which defines the convex hull, the DT-events do not affect the $\mathcal{V}$, $\mathcal{E}$ and $\mathcal{F}$ data structures.

Finally, on processing a DT-event of parameters p_ip_j, p_kp_l and t, the $T(DT)$ and $Q(DT)$ data structures are updated as follows. The edge p_ip_j is removed from $T(DT)$ and its associated certificates (at most two certificates for its previous and next edges in $T(DT)$) are removed from $Q(DT)$. Then, the new DT edge, p_kp_l is inserted into $T(DT)$ and the new certificates (at most two) associated to this edge are determined and inserted into $Q(DT)$.

4.2 Processing Order Events

For an *order event* of parameters p_ip_j, p_kp_l and t we should apply the necessary changes to the *EMST* as well as to our *KDS*. These changes are applied as follows.

We first update the order of the edges p_ip_j and p_kp_l in $T(DT)$ by swapping their positions in $T(DT)$. Then, the previous certificates of these edges that are no longer valid are removed from $Q(DT)$ and the new certificates, according to the new adjacent edges of p_ip_j and p_kp_l in $T(DT)$ are computed and inserted into $Q(DT)$.

An order event changes the $\mathcal{V}$, $\mathcal{E}$ and $\mathcal{F}$ data structures if it results a change in the *EMST*. Otherwise, these data structures are not affected due to an order event.

In order to identify the effect of the assumed order event we distinguish between four different cases:

a) $p_ip_j \notin \mathcal{E}(EMST)$ and $p_kp_l \notin \mathcal{E}(EMST)$.
b) $p_ip_j \in \mathcal{E}(EMST)$ and $p_kp_l \in \mathcal{E}(EMST)$.
c) $p_ip_j \notin \mathcal{E}(EMST)$ and $p_kp_l \in \mathcal{E}(EMST)$.
d) $p_ip_j \in \mathcal{E}(EMST)$ and $p_kp_l \notin \mathcal{E}(EMST)$.

In cases (a) and (c) none of the PK, $\mathcal{V}$, $\mathcal{E}$ and $\mathcal{F}$ data structure is changed. Therefore, the *EMST* is not changed in these cases.

In case (b) the order event does not change the *EMST* of the points and therefore, it does not change the $\mathcal{V}$, $\mathcal{E}$ and $\mathcal{F}$ data structures. However, it should be checked to see whether there is an edge e in both $PK(p_ip_j)$ and $cut(p_kp_l)$. If there are such edges, they must be removed from $PK(p_ip_j)$ and added to $PK(p_kp_l)$. Trivially, if p_ip_j and p_kp_l do not lie on the boundary of a single face of *SD*, there is no edge e in both $PK(p_ip_j)$ and $cut(p_kp_l)$.

Therefore, we first check this condition using the $\mathcal{E}$ and $\mathcal{F}$ data structures: Each edge has two half-edges in $\mathcal{E}$ and each half-edge points to its corresponding half-edge in $\mathcal{F}(f_s)$ for some face f_s of SD. If we obtain the same face for a half-edge of p_ip_j and a half-edge of p_kp_l, a subset Δ of edges of $PK(p_ip_j)$ must be removed and these edges must be inserted into $PK(p_kp_l)$. The subset Δ has the following properties:

- The subset Δ is a connected part of the sorted list of edges in $PK(p_ip_j)$ *i.e.* if $e_1 \in \Delta$ has the minimum distance and $e_2 \in \Delta$ has the maximum distance among edges of Δ, there is no edge $e' \in PK(p_ip_j) - \Delta$ that its distance is between the distances of e_1 and e_2.
- For each $p_sp_t \in \Delta$, $p_lp_k \in f$ where f is the face of SD that contains the edge p_sp_t. Moreover, f is adjacent to only one of the half-edges of p_kp_l.

Using the above properties, we can find the two extreme edges of Δ by navigating along two paths from the root of $PK(p_ip_j)$ to two leaves. At any node of these paths we check the second property using the $\mathcal{F}(f)$ data structure to determine the direction of the next step downward.

After finding Δ, it will be removed from $PK(p_ip_j)$ and inserted into its appropriate position in $PK(p_kp_l)$. Assume that this appropriate position is between the edges p_sp_t and $p_{s'}p_{t'}$ of $PK(p_kp_l)$. We have the following property about these edges:

- For each edge $p_mp_n \in \Delta$, $path_f(p_s, p_t) \subset path(p_m, p_n)$ and $path_f(p_m, p_n) \subset path(p_{s'}, p_{t'})$ where $path_f(p_i, p_j)$ is the path between p_i and p_j on the boundary of f that uses only the half-edges of the edges of the $EMST$.

Using this property, we can find the position of Δ in $PK(p_kp_l)$ by navigating a path from the root of $PK(p_kp_l)$ to a leaf. It is notable that in some cases there is no edge p_sp_t or $p_{s'}p_{t'}$ in $PK(p_kp_l)$ and Δ must be added to the start or the end of $PK(p_kp_l)$.

In case (d) if $p_kp_l \notin PK(p_ip_j)$, none of the PK, $\mathcal{V}$, $\mathcal{E}$ and $\mathcal{F}$ data structures and the $EMST$ is changed. Otherwise, by adding p_kp_l to the $EMST$ and removing p_ip_j we obtain an $EMST$ of smaller weight. Therefore, in such conditions the $EMST$ and some parts of our KDS must be updated. To do this p_kp_l is added to the $EMST$ and p_ip_j is removed from it. Whereas p_ip_j no longer exists in the $EMST$ and p_kp_l is a new edge of the $EMST$, $PK(p_ip_j)$ is no longer required and we must build the $PK(p_kp_l)$ data structure. It is interesting to note that just after the event $PK(p_kp_l)$ is equal to $PK(p_ip_j)$ just before the event and we can use the existing $PK(p_ip_j)$ as the value of the required $PK(p_kp_l)$. Moreover, p_ip_j is removed from $\mathcal{E}$, $\mathcal{V}(p_i)$ and $\mathcal{V}(p_j)$. Finally, $\mathcal{F}(f_s)$ and $\mathcal{F}(f_t)$ are updated accordingly where f_s and f_t are the two faces adjacent to p_ip_j. As shown in Figure 4, a set of half-edges of the boundary of f_s is removed from $\mathcal{F}(f_s)$ and this set

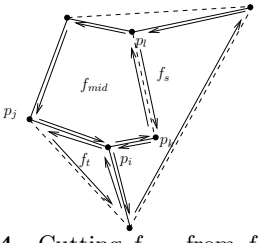


Fig. 4. Cutting f_{mid} from f_s and adding it to f_t

is inserted into $\mathcal{F}(f_t)$. Precisely, the half-edge $\overrightarrow{p_ip_j}$ is removed from $\mathcal{F}(f_t)$, the sequence of half-edges from p_i to p_k are removed from $\mathcal{F}(f_s)$ and are inserted into $\mathcal{F}(f_t)$ just after the point p_i, the half-edge $\overrightarrow{p_kp_l}$ is inserted into $\mathcal{F}(f_t)$ just after the point p_k, the sequence of half-edges from p_l to p_j are removed from $\mathcal{F}(f_s)$ and are inserted into $\mathcal{F}(f_t)$ just after the point p_l, the half-edge $\overrightarrow{p_jp_i}$ is removed from $\mathcal{F}(f_s)$, and the half-edge $\overrightarrow{p_lp_k}$ is inserted into $\mathcal{F}(f_s)$ just after the point p_l.

Except for the case (b) of processing an order-event, other processes required for a DT-event and order-event includes a constant number of logarithmic operations (search, insert, delete, merge, split) on data structures of linear size. The excepted part of the processing of an order event can be done in $O(\log^2 n)$ time using the PK and $\mathcal{F}$ MS-BBST data structures. Therefore,

Theorem 3. *Each DT-event and order event can be handled in $O(\log n)$ and $O(\log^2 n)$ time.*

5 Performance Analysis

In this section, we analyze the performance of the proposed *KDS* according to the *KDS* performance criteria.

Theorem 4. *The proposed KDS for a set of n moving points has the following properties:*

- *It processes $O(n^4)$ events.*
- *Each event can be handled in $O(\log^2 n)$ time.*
- *Each point participates in $O(1)$ (in average) number of certificates.*
- *It uses $O(n)$ space and requires $O(n \log n)$ preprocessing time.*

Therefore, the proposed KDS is responsive, compact and local (in average).

Proof. Assume that x and y coordinates of moving points are defined by algebraic functions of maximum degree s. There are $O(n^2)$ items (lengths of edges) that can appear in $T(DT)$. Although at any fixed time only $O(n)$ items exist in $T(DT)$, to obtain an upper bound we assume that all these items exist in $T(DT)$. The total number of swaps in this sorted list is $O(n^4)$ which dominates the number of the DT-events (the number of the Delaunay triangulation events is $O(n^2\lambda_s(n))$). This $O(n^4)$ bound is a consequence of our assumption about the motion of the points: x and y coordinates of the points change according to some algebraic functions of constant maximum degree which means that s is constant. However, we do not know how much this upper bound is tight.

The second property is concluded from Theorem 3. the Delaunay triangulation is a planar graph which means that the average number of edges of its points is constant. This concludes the third property. The last property is concluded from Theorem 2. □

Efficiency is somehow the main *KDS* performance evaluation metric. As discussed in Section 1, this metric depends on the upper bound of the number of

the internal events and the lower bound of the number of the external events. We proved an upper bound of $O(n^4)$ for the number of internal events in Theorem 4. But, the number of external events is smaller [13] which means that this *KDS* is not efficient.

6 Conclusion

In this paper, we considered the kinetic version of the planar Euclidean minimum spanning tree. We proposed a *KDS* that can be used to track the combinatorial changes of the *EMST* of a set of moving points. Our *KDS* is the first responsive, local in average and compact solution for this problem.

Proving the tight bounds of the number of the changes of the *EMST* is the immediate open direction in continuing this research. Extending to higher dimensions as well as to dynamic environments in which the points are added or removed are the other future directions.

References

1. Agarwal, P.K., Eppstein, D., Guibas, L.J., Henzinger, M.R.: Parametric and kinetic minimum spanning. In: 39th IEEE Sympos Found Comput Sci, pp. 596–605 (1998)
2. Basch, J.: Kinetic data structures. PhD Thesis, Stanford University (1999)
3. Basch, J., Guibas, L.J., Hershberger, J.: Data structures for mobile data. Journal of Algorithms 31, 1–28 (1999)
4. Basch, J., Guibas, L.J., Zhang, L.: Proximity problems on moving points. In: 13th Annual Symposium on Computational Geometry, pp. 344–351 (1997)
5. de Berg, M., Cheong, O., van Kreveld, M., Overmars, M.: Computational Geometry: Algorithms and Applications, 3rd edn. Springer, Santa Clara (2008)
6. Boissonnat, J.-D., Teillaud, M.: On the randomized construction of the Delaunay. Theoretical Computer Science 112, 339–354 (1993)
7. Chang, R.C., Lee, R.C.T.: An $O(nlogn)$ minimal spanning tree algorithm for n points in the plane. BIT 26, 7–16 (1986)
8. Eppstein, D.: Dynamic Euclidean minimum spanning trees and extrema of binary functions. Discrete and Computational Geometry 13, 111–122 (1995)
9. Fu, J.-J., Lee, R.C.T.: Minimum Spanning Trees of Moving Points in the Plane. IEEE Transactions on Computers 40, 113–118 (1991)
10. Guibas, L.J., Mitchell, J.S.B., Roos, T.: Voronoi Diagrams of Moving Points in the Plane. In: Proceedings of the 17th International Workshop, pp. 113–125 (1991)
11. Guibas, L.J., Knuth, D.E., Sharir, M.: Randomized incremental construction of Delaunay and Voronoi diagrams. Algorithmica 7, 381–413 (1992)
12. Kruskal, J.B.: On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. Proceedings of the American Mathematical Society 7, 48–50 (1956)
13. Katoh, N., Tokuyama, T., Iwano, K.: On minimum and maximum spanning trees of linearly moving points. Discrete and Computational Geometry 13, 161–176 (1995)
14. Prim, R.C.: Shortest connection networks and some generalizations. Bell System Technical Journal 36, 1389–1401 (1957)
15. Tarjan, R.E.: Data structures and network algorithms. Society for Industrial and Applied Mathematics (1983)

Generating All Simple Convexly-Drawable Polar Symmetric 6-Venn Diagrams

Khalegh Mamakani, Wendy Myrvold, and Frank Ruskey

Dept. of Computer Science, University of Victoria, Canada

Abstract. An n-Venn diagram consists of n curves drawn in the plane in such a way that each of the 2^n possible intersections of the interiors and exteriors of the curves forms a connected non-empty region. A Venn diagram is convexly-drawable if it can be drawn with all curves convex and it is *simple* if at most two curves intersect at any point. A Venn diagram is called *polar symmetric* if its stereographic projection about the infinite outer face is isomorphic to the projection about the innermost face. We outline an algorithm that shows there are exactly 375 simple convexly drawable polar-symmetric 6-Venn diagrams.

Keywords: Venn diagram, polar-symmetry.

1 Introduction

Named after John Venn($1834-1923$), who used diagrams of overlapping circles to represent propositions [10], Venn diagrams are commonly used in set theory to visualize the relationship between different sets. When talking about Venn diagrams, the traditional three circles diagram with 3-fold symmetry often comes to mind (Figure 1(a)). Although it is not possible to use circles to draw Venn diagrams of more than 3 sets, more than 3 sets can be represented if the curves of the Venn diagram are other simple closed curves. Figure 1(b) shows a 5-set Venn diagram composed of 5 congruent ellipses which was discovered by Grünbaum[5], and Figure 1(c) shows a 7-set Venn diagram with 7-fold rotational symmetry called Adelaide which was discovered independently by Grünbaum [7] and Edwards [3].

Intensive research has been done recently on generating and drawing Venn diagrams of more than three sets, particularly in regard to symmetric Venn diagrams, which are those where rotating the diagram by $360/n$ degrees results in the same diagram. Henderson considered rotationally symmetric Venn diagrams and he showed that they could exist only if the number of curves is prime [9]. Griggs, Killian, and Savage published a constructive method for producing symmetric Venn diagrams with a prime number of curves [8]. Venn diagrams exist for any number of curves and several constructions of them are known [12].

Another type of symmetry, introduced by Grünbaum [6], is called *polar-symmetry* which can be imagined by first projecting the diagram onto the surface of a sphere with the regions corresponding to the full and empty sets at the north

C.S. Iliopoulos and W.F. Smyth (Eds.): IWOCA 2011, LNCS 7056, pp. 275–286, 2011.
© Springer-Verlag Berlin Heidelberg 2011

and south poles, respectively. The Venn diagram is *polar-symmetric* if it is invariant under polar flips, meaning that the north and south hemispheres are congruent. In other words, for a polar-symmetric Venn diagram on the plane turning the diagram inside-out (the innermost face becomes the outermost face) gives the same Venn diagram. Note that all three Venn diagrams of Figure 1 are polar-symmetric, as well as being rotationally symmetric.

The only other attempt that we know of to exhaustively list some interesting class of 6-Venn diagrams is the work of Jeremy Carroll, who discovered that there are precisely 126 such Venn diagrams where all curves can be drawn as triangles [2]. He used a brute force search algorithm for all possible face sizes of a Venn diagram. However, the problem of generating polar-symmetric six-set Venn diagrams has not been studied before. In this paper we are restricting our attention to the special (and most studied) class of Venn diagrams that are both simple and drawable with convex curves. We introduce two representations of these diagrams. Inspired by Carroll's work, an algorithm for generating all possible simple convexly-drawable polar-symmetric six-set Venn diagrams is developed — an algorithm which determines that there are exactly 375 simple convexly-drawable polar-symmetric 6-Venn diagrams.

Although our results are oriented towards 6-Venn diagrams, they could in principle be applied to general n-Venn diagrams, but the computations will quickly become prohibitive. Nevertheless, we believe that the data structures and ideas introduced here (i.e., the representations) will be useful in further investigations, and in particular to resolving one of the main outstanding open problems in the area of Venn diagrams: is there a simple 11-Venn diagram? We intend to use the data structures proven useful here to attack a restricted, but natural, version of that problem: is there a simple convexly-drawable polar-symmetric 11-Venn diagram?

The study of symmetric Venn diagrams in interesting not only because symmetry is core aspect of mathematical enquiry, but also because we are often led to diagrams of great inherent beauty. Furthermore, the geometric dual of a simple Venn diagram is a maximal planar spanning subgraph of the the hypercube, and so results about Venn diagrams often have equivalent statements as results about the hypercube.

The remainder of paper is organized as follows. In Section 2 we introduce basic definitions. Representations of simple convexly-drawable Venn diagrams are explained in Section 3. The generating algorithm and results are explained in the last two sections.

2 Basic Definitions

A closed curve in the plane is *simple* if it doesn't intersect itself. Each simple closed curve decomposes the plane into two connected regions, the interior and the exterior. An *n-Venn diagram* is a collection of n finitely intersecting simple closed curves $\mathcal{C} = \{C_0, C_1, \ldots, C_{n-1}\}$ in the plane, such that there are exactly 2^n nonempty and connected regions of the form $X_0 \cap X_1 \cap \cdots \cap X_{n-1}$, where

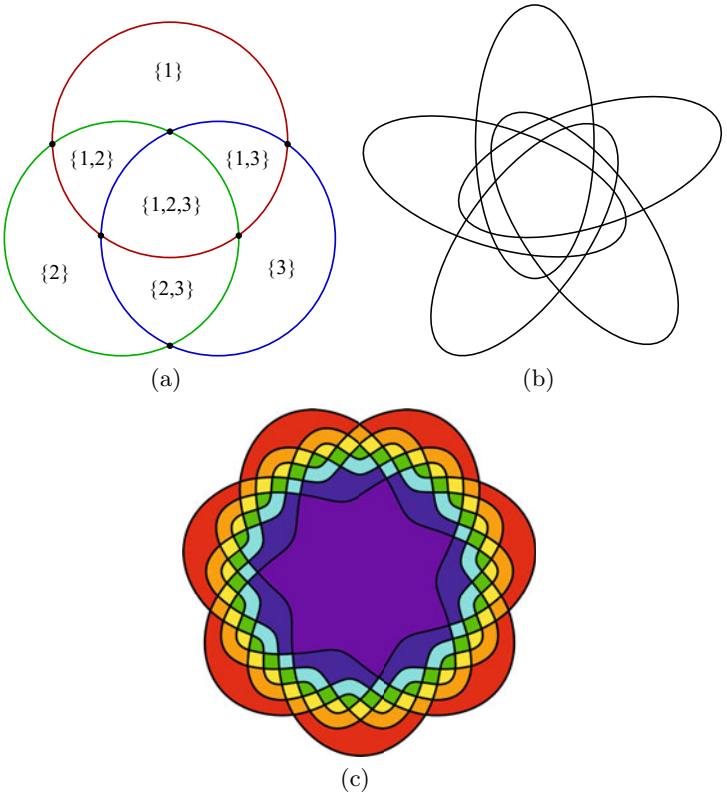


Fig. 1. (a) A 3-Venn diagram whose curves are circles. (b) A 5-Venn diagram whose curves are ellipses. (c) A symmetric 7-Venn known as "Adelaide."

X_i is either the unbounded open exterior or open bounded interior of curve C_i. Each connected region corresponds to a subset of the set $\{0, 1, \ldots, n-1\}$. Two Venn diagrams are *isomorphic* if one of them can be changed into the other or its mirror image by a one-to-one continuous transformation of the plane onto itself.

A *k-region* in a diagram is a region that is in the interior of precisely k curves. In an n-Venn diagram, each k-region corresponds to a k-element subset of a set with n elements. So, there are $\binom{n}{k}$ k-regions. A Venn diagram is *monotone* if every k-region is adjacent to both some $(k-1)$-region (if $k > 0$) and also to some $(k+1)$-region (if $k < n$). A diagram is monotone if and only if it is drawable with each curve convex [1]. A *simple Venn diagram* is one in which exactly two curves cross each other at each intersection point. Figure 2 shows a simple monotone 6-Venn diagram.

Consider a Venn diagram as being projected onto the surface of a unit sphere where the empty region of the diagram encloses the north pole of the sphere and the innermost region contains the south pole. A *cylindrical projection* of the

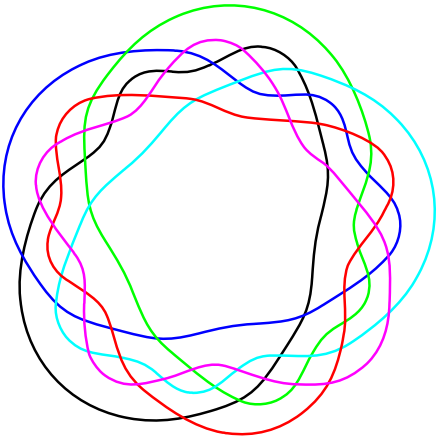


Fig. 2. A simple monotone 6-Venn diagram

Venn diagram can be obtained by mapping the surface of sphere to a 2π by 2 rectangle on the plane, where the equator of the sphere maps to a horizontal line of length 2π and the north and south pole of the sphere are mapped to the top and bottom sides of the rectangle respectively. In this representation, the top of cylinder represents the empty region and bottom of cylinder represents the innermost region. A Venn diagram is said to be *polar symmetric* if it is invariant under polar flips. In the cylindrical representation the polar flip is equivalent to turning the cylinder upside-down. For a Venn diagram on the plane, a polar flip is equivalent to turning the diagram inside-out, with the innermost face becoming the outermost. It is known that there are exactly 6 simple monotone 7-Venn diagrams with rotational and polar symmetry [4],[3]. Figure 3 shows the cylindrical representation of the polar symmetric 6-Venn diagram shown in Figure 2. Note that in the cylindrical representation of a monotone Venn diagram every curve is x-monotone because of the monotonicity of the diagram; *i.e.*, every vertical line intersects each curve at a single point at most.

A simple Venn diagram can be viewed as a planar graph where the intersection points of the Venn diagram are the vertices of the graph and the sections of the curves that connect the intersection points are the edges of the graph. For a

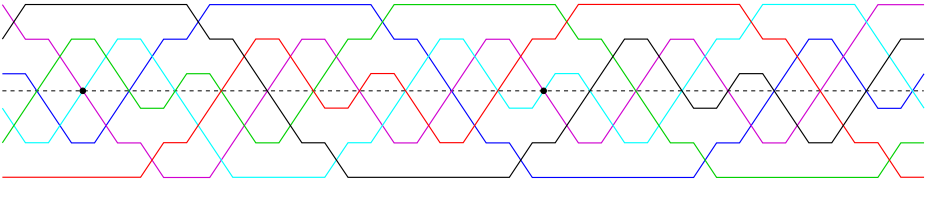


Fig. 3. Cylindrical representation of the 6-Venn diagram of Figure 2

planar graph with f faces, v vertices and e edges, Euler's formula states that $f + v = e + 2$. A graph of an n-Venn diagram has 2^n faces. In a simple Venn diagram each vertex of this graph has degree 4; *i.e.* $e = 2v$, so a simple n-Venn diagram has $2^n - 2$ vertices.

3 Representing Venn Diagrams

In this section we introduce two representations for simple monotone Venn diagrams. First we discuss the binary matrix representation where each 1 in the matrix represents an intersection point of the corresponding Venn diagram. Having the matrix representation of a diagram, it is easy to check if it is a Venn diagram or not. In the second part we show how to represent simple monotone Venn diagrams using compositions. We use this representation to find all candidate diagrams. Then we filter non-Venn diagrams using the matrix representation.

3.1 Matrix Representation

For a simple monotone n-Venn diagram, every 1-region is adjacent to the empty region. So the empty region surrounds a "ring" of $\binom{n}{1}$ 1-regions. An intersection point is said to be part of ring i if of the four incident regions, two are in ring i and the other two are in ring $i-1$ and $i+1$. Since each region is started by one intersection point and ended by another one, there are $\binom{n}{1}$ intersection points in the first ring. Similarly, every 2-region is adjacent to at least one 1-region. So, there are $\binom{n}{2}$ 2-regions that form a second ring surrounded by the first ring and which contains $\binom{n}{2}$ intersection points. In general, in a simple monotone $n-$Venn diagram, there are $n-1$ rings of regions, where all regions in a ring are enclosed by the same number of curves and every region in ring i, $1 \leq i \leq n-1$, is adjacent to at least one region in ring $i-1$ and also to at least one region in ring $i+1$. The number of intersection points in the i^{th} ring is the same as the number of regions in the i^{th} ring, which is $\binom{n}{i}$. The rings have different colors in Figure 1(c).

Thus a simple monotone $n-$Venn diagram can be represented by a $n-1$ by m binary matrix, $m \leq 2^n - 2$, where each 1 in the matrix represents an intersection point in the Venn diagram. Row i of the matrix corresponds to ring i of the Venn diagram. A Venn matrix has the following properties :

- There are $\binom{n}{i}$ 1's in the i^{th} row of the matrix, $1 \leq i \leq n-1$.
- There are no two adjacent 1's in any row or column of the matrix.
- A valid matrix must represent exactly $2^n - 2$ distinct regions of the corresponding Venn diagram. The two other regions are the outermost and the innermost regions. Figure 4 shows the matrix representation of the Venn diagram of Figures 2 and 3.

The *rank* of a region of a Venn diagram is defined by $\sum_{i=0}^{n-1} 2^i x_i$ where $x_i = 1$ if curve i encloses the region and $x_i = 0$ otherwise. Given a matrix representation P, we need to check that no two ranks are the same to check if it represents a valid Venn diagram.

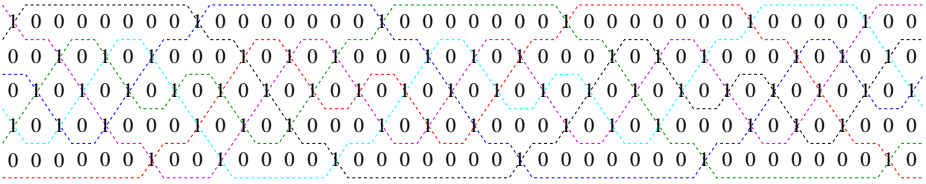


Fig. 4. Matrix representation of the 6-Venn diagram of Figures 2 and 3

For a given matrix P, suppose vector $C = [c_0, c_1, \cdots, c_{n-1}]$ represents the curve labels of the corresponding diagram in cylindrical representation along some column where c_0 is the label of the outermost(top) curve and c_{n-1} is the label of the innermost(bottom) curve. Then a region at ring i, $1 \leq i \leq n-1$, is enclosed by curves $c_0, \cdots, c_{i-1}$ and the rank of the region is $\sum_{k=0}^{i-1} 2^{c_k}$. To get the curve labels for each region we need to update C based on the entries of matrix P at each column. For a given column j of the matrix, each entry of 1 represents an intersection point. So for each row i, if $p_{ij} = 1$ then we need to exchange c_i and c_{i+1} to get the next curve labels. Starting from left to right with $C = [0, 1, \cdots, n-1]$ as the initial curve labels, then we can compute the rank of each region. Matrix P represents a valid simple monotone Venn diagram if we get exactly $2^n - 2$ regions with distinct ranks and $C = [0, 1, \cdots, n-1]$ at the end given that we start with $C = [0, 1, \cdots, n-1]$. We used the matrix representation in [4] to generate all monotone simple symmetric 7-Venn diagrams.

3.2 Compositions

In this part we introduce the other representation we use to generate Venn diagrams. In this representation, we use a sequence of non-negative integers to show the size of faces in each ring and and also to specify the position of intersection points of the next ring.

Definition 1. *Let $a_1, a_2, \cdots, a_k$ be non-negative integers such that :*

$$\sum_{i=1}^{k} a_i = n$$

Then $(a_1, a_2, \cdots, a_k)$ is called a composition of n into k parts or a $k-$composition of n.

In a simple monotone n-Venn diagram there are $\binom{n}{i+1}$ intersection points at ring $i+1$ that are distributed among $\binom{n}{i}$ intersection points at ring i. So if we pick a particular point at ring i as the reference point, then we can specify the exact location of points at ring $i+1$ using a composition of $\binom{n}{i+1}$ into $\binom{n}{i}$ parts.

Definition 2. *Let $\mathcal{C}(n,k)$ be the set of all compositions of n into k parts. For a simple monotone n-Venn diagram V, starting from an arbitrary point of the first ring, we label the intersection points of V from 1 to 2^n-2 in clockwise direction. Let ℓ_i denote the label of the starting point at ring i. The composition representation P of V is a set of $n-2$ pairs of form $\langle \ell_i, c_i \rangle$, where*

$$\ell_1 = 1,\ \ell_i = \ell_{i-1} + \binom{n}{i-1},\ c_i \in \mathcal{C}\left(\binom{n}{i+1}, \binom{n}{i}\right)$$

Figure 5 shows the composition representation of the 6-Venn diagram of Figure 2 and 3.

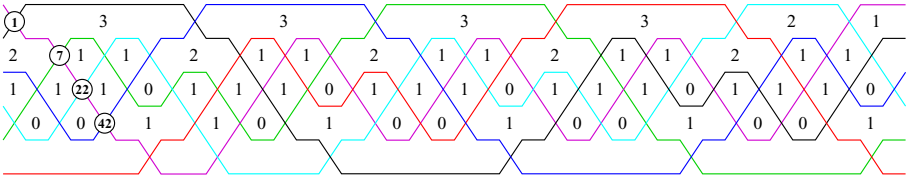


Fig. 5. Composition representation of the 6-Venn diagram of Figures 2 and 3

We now list several observations that will help us cut down on the size of the search space.

Observation 1. *For any simple monotone n-Venn diagram V, the largest part of c_i in the composition representation is at most $n-i-1$.*

Proof. A region at ring i is enclosed by i curves above its starting and ending points. Since the size of a region is at most n and no two edges belong to the same curve [11], at most $n-i$ remaining curves can be used to shape the region. As shown in Figure 6, to put p intersection points between the two end points of the region on the next ring, we need $p+1$ curves, $p-1$ curves for the bottom side and two curves for the left and right sides. So, $p \leq n-i-1$.

Observation 2. *In the composition representation of any simple monotone Venn diagram with more than 3 curves, there are no two non-adjacent 1's in c_1.*

Proof. Suppose, there is such a Venn diagram $\mathcal{V}$, then the first ring of the Venn diagram will be like Figure 7 , where regions A and D correspond to non-adjacent 1's in the composition and $A \neq D$. Then $A \cap D = \emptyset$ which contradicts the assumption that $\mathcal{V}$ is a Venn diagram. So in the first ring composition there are at most two 1's which must be adjacent.

Observation 3. *There are no two faces of size 3 adjacent to another face of size 3 in a simple monotone $n-$Venn diagram $\mathcal{V}$.*

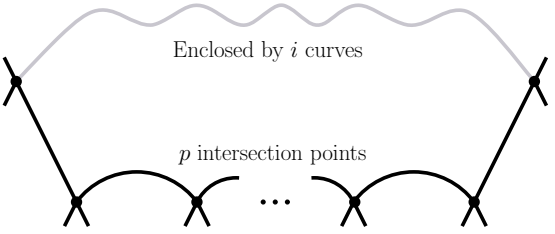


Fig. 6. Largest part of the composition at level i

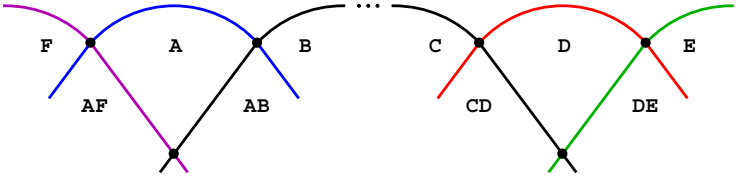


Fig. 7. Non-Adjacent 1's in the first ring composition

Proof. There are only two cases, shown in Figure 8, that two faces of size 3 could be adjacent to a single face of size 3. However, both cases result in a two part disconnected region(the shaded region) which contradicts the fact that $\mathcal{V}$ is a Venn diagram.

Observation 4. *There are no two consecutive 0's in c_2 for the composition representation of any simple monotone n-Venn diagram.*

Proof. By observation 3

Definition 3. *Let $r_1, r_2 \in \mathcal{C}(n,k)$ be two compositions of n into k parts. r_1 and r_2 are rotationally distinct if it is not possible to get r_2 from any rotation of r_1 or its reversal.*

Let $\mathcal{F}_n$ denote the set of all rotationally distinct compositions of $\binom{n}{2}$ into n parts such that for any $r \in \mathcal{F}_n$ there are no two non-adjacent parts of 1 and all parts are less than or equal to $n-2$.

Theorem 5. *If c_1 is the composition corresponding to the first ring of a simple monotone n-Venn diagram, then $c_1 \in \mathcal{F}_n$.*

Proof. Given a simple monotone n-Venn diagram, suppose we get the composition representation P of $\mathcal{V}$ by picking a particular intersection point x in the first ring as the reference point. Now let P' be another representation of $\mathcal{V}$ using any other intersection point different than x as the reference point. It is clear that c_1' in P' is a rotation of c_1 in P. Also for any composition representation P'' of the mirror of $\mathcal{V}$ the first composition c_1'' in P'' is a rotation of the reversal

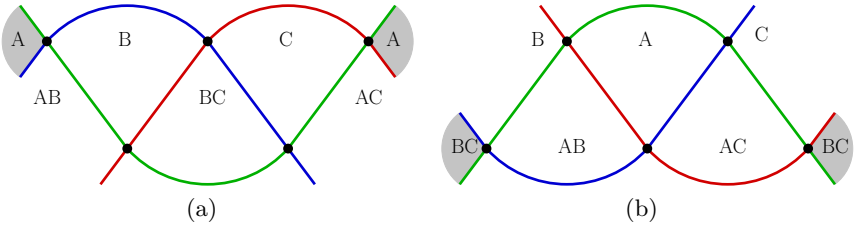


Fig. 8. Possible cases for two faces of size 3 being adjacent to a single face of size 3

of c_1. By the observations 1 and 2 the largest part of c_1 is $n-2$ and there are no two non-adjacent 1's in c_1. Therefore, there is a composition $c \in \mathcal{F}_n$ which is rotationally identical to c_1.

4 Generating Algorithm

Given the upper/lower half of the cylindrical representation of a polar symmetric Venn diagram, one can generate the whole diagram by creating a copy of the given half, turning it upside down and rotating it until the two parts match together. So, to generate a polar symmetric monotone Venn diagram, we need only to generate the first $\lceil \frac{n-2}{2} \rceil$ compositions.

Two halves of the diagram can match only if gluing them using the intersection points doesn't create any faces of size 2. Given the last composition of the upper half, for each positive part a_j there are $a_j - 1$ edges that bound the corresponding face from the bottom and there is a gap between two faces corresponding to two consecutive parts of the composition. So, we can map the composition to a bit-string where each 1 represents a bounding edge of a face and each 0 represents the gap between two faces. The length of bit-string is the same as the sum of all parts of the composition. In other words the composition $(a_1, a_2, \cdots, a_k)$ is mapped to the following bit-string.

$$\overbrace{11\cdots1}^{a_1-1\ bits}\ 0\ \overbrace{11\cdots1}^{a_2-1\ bits}\ 0\cdots0\ \overbrace{11\cdots1}^{a_k-1\ bits}\ 0$$

We can find all matchings of the two halves by computing the bitwise "and" of the bit-string and its reverse for all left rotations of the reverse bit-string. Any result other than 0 means that there is at least one face of size 2 in the middle. Then for each matching we compute the matrix representation of the resulting diagram. The matrix can be obtained by sweeping the compositions from left to right and computing the position of each intersection point. Checking each resulting matrix for all compositions gives us all possible polar symmetric 6-Venn diagrams.

Algorithm 1. GenPolarSymSixVenn

```
begin
    foreach composition (a_1, a_2, ··· , a_6) ∈ F_6 do
        foreach composition (b_1, b_2, ··· , b_15) ∈ C(20, 15) do
            create the corresponding upper and lower halves
            for i ← 1 to 20 do
                glue the upper and lower halves
                if there are no parallel edges in the diagram then
                    compute matrix X representing the diagram
                    if isVenn(X) then
                        Print(X)
                rotate lower half one point to the left
end
```

5 Results

Using the exhaustive search we found 375 simple monotone polar symmetric 6-Venn diagrams. This result was independently checked by using a separate program that is based on a different search method. That search method will be explained in the eventual expanded version of this paper.

Table 1. Number of polar symmetric 6-Venn diagrams for $\mathcal{F}_6$

Composition	Venn Diagrams	Composition	Venn Diagrams
4 4 3 2 1 1	25	2 3 4 3 2 1	5
4 3 4 2 1 1	0	4 2 2 4 2 1	0
3 4 4 2 1 1	38	3 3 2 4 2 1	9
4 4 2 3 1 1	0	2 4 2 4 2 1	0
4 3 3 3 1 1	12	3 2 3 4 2 1	3
3 4 3 3 1 1	9	4 3 2 2 3 1	9
4 2 4 3 1 1	0	3 4 2 2 3 1	15
4 3 2 4 1 1	0	4 2 3 2 3 1	0
4 4 2 2 2 1	15	3 3 3 2 3 1	9
4 3 3 2 2 1	2	3 2 4 2 3 1	0
3 4 3 2 2 1	30	4 2 2 3 3 1	12
4 2 4 2 2 1	0	3 3 2 3 3 1	4
3 3 4 2 2 1	6	4 2 2 2 4 1	0
2 4 4 2 2 1	13	4 3 2 2 2 2	15
4 3 2 3 2 1	7	4 2 3 2 2 2	22
3 4 2 3 2 1	8	3 3 3 2 2 2	6
4 2 3 3 2 1	6	4 2 2 3 2 2	1
3 3 3 3 2 1	41	3 3 2 3 2 2	21
2 4 3 3 2 1	22	3 2 3 2 3 2	3
3 2 4 3 2 1	7		

Table 1 shows the number of Venn diagrams for each particular composition of the first level. Figure 9 shows one Venn diagram for each of those 29 compositions that have at least one Venn diagram. Renderings of each of the 375 diagrams may be found at the website:
`http://webhome.cs.uvic.ca/~ruskey/Publications/SixVenn/SixVenn.html`.

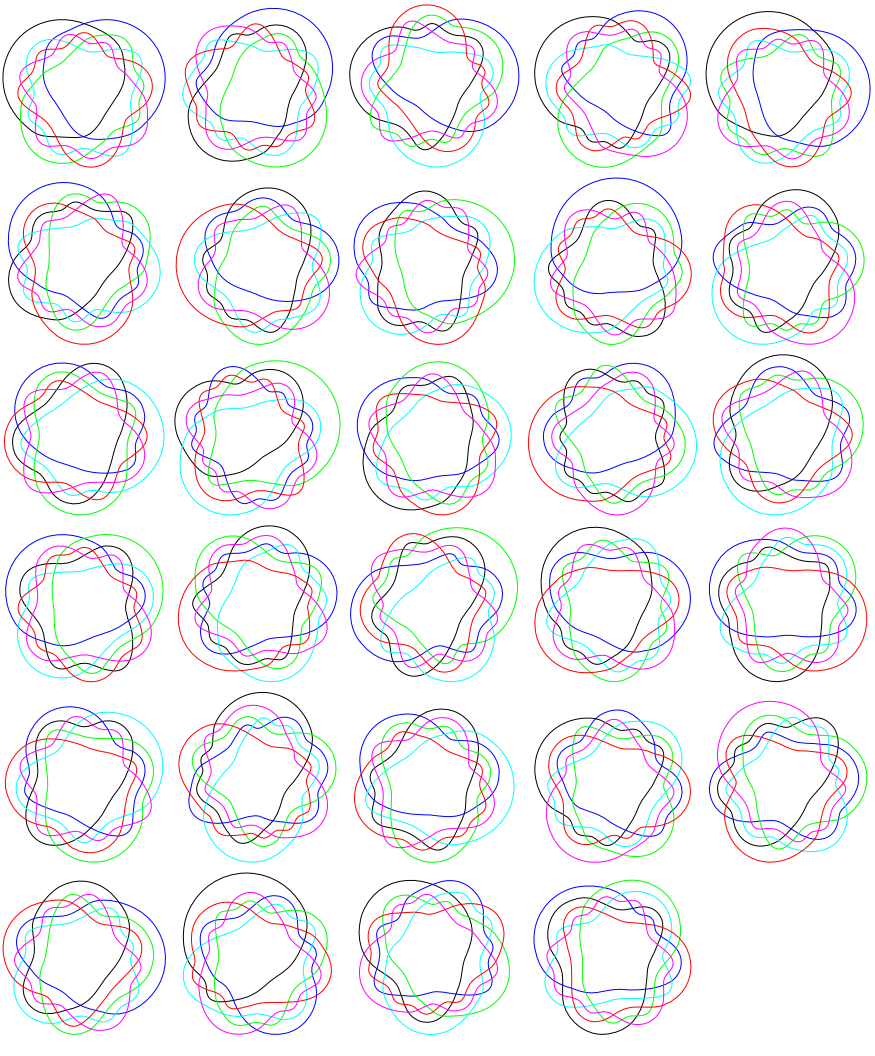


Fig. 9. 29 simple monotone polar symmetric 6-Venn diagrams, representing all possible compositions for the outermost ring

References

1. Bultena, B., Grünbaum, B., Ruskey, F.: Convex Drawings of Intersecting Families of Simple Closed Curves. In: 11th Canadian Conference on Computational Geometry, pp. 18–21 (1999)
2. Carroll, J.: Drawing Venn triangles. Technical Report HPL-2000-73, HP Labs (2000)
3. Edwards, A.W.F.: Seven-set Venn Diagrams with Rotational and Polar Symmetry. Combinatorics, Probability, and Computing 7, 149–152 (1998)
4. Cao, T., Mamakani, K., Ruskey, F.: Symmetric Monotone Venn Diagrams with Seven Curves. In: Boldi, P. (ed.) FUN 2010. LNCS, vol. 6099, pp. 331–342. Springer, Heidelberg (2010)
5. Grünbaum, B.: Venn Diagrams and Independent Families of Sets. Mathematics Magazine, 13–23 (January-February 1975)
6. Grünbaum, B.: Venn Diagrams I. Geombinatorics I(4), 5–12 (1992)
7. Grünbaum, B.: Venn Diagrams II. Geombinatorics II(2), 25–32 (1992)
8. Griggs, J., Killian, C.E., Savage, C.D.: Venn Diagrams and Symmetric Chain Decompositions in the Boolean Lattice. Electronic Journal of Combinatorics 11(1), #R2 (2004)
9. Henderson, D.W.: Venn diagrams for more than four classes. American Mathematical Monthly 70, 424–426 (1963)
10. Venn, J.: On the diagrammatic and mechanical representation of propositions and reasonsings. Philosophical Magazine and Journal of Science, Series 5 10(59) (1880)
11. Chilakamarri, K.B., Hamburger, P., Pippert, R.E.: Venn diagrams and planar graphs. Geometriae Dedicata 62, 73–91 (1996)
12. Ruskey, F., Weston, M.: A survey of Venn diagrams. The Electronic Journal of Combinatorics (1997); Dynamic survey, Article DS5 (online) (revised 2001, 2005)

The Rand and Block Distances of Pairs of Set Partitions

Frank Ruskey[1,⋆] and Jennifer Woodcock[1]

Dept. of Computer Science, University of Victoria, Canada

Abstract. The *Rand distance* of two set partitions is the number of pairs $\{x, y\}$ such that there is a block in one partition containing both x and y, but x and y are in different blocks in the other partition. Let $R(n, k)$ denote the number of distinct (unordered) pairs of partitions of n that have Rand distance k. For fixed k we prove that $R(n, k)$ can be expressed as $\sum_j c_{k,j} \binom{n}{j} B_{n-j}$ where $c_{k,j}$ is a non-negative integer and B_n is a Bell number. For fixed k we prove that there is a constant K_n such that $R(n, \binom{n}{2} - k)$ can be expressed as a polynomial of degree $2k$ in n for all $n \geq K_n$. This polynomial is explicitly determined for $0 \leq k \leq 3$.

The *block distance* of two set partitions is the number of elements that are not in common blocks. We give formulae and asymptotics based on $N(n)$, the number of pairs of partitions with no blocks in common. We develop an $O(n)$ algorithm for computing the block distance.

1 Introduction and Motivation

In statistics, particularly as it is applied to cluster analysis, it is sometimes useful to have a measure of the difference between two set partitions [4]. The Rand distance is one such measure, and was introduced in Rand [8]. In this paper we will initiate a combinatorial study of the properties of the Rand distance, taken over all unordered pairs of partitions of an n-set. We will also introduce another measure, which we call the block distance, and determine some of its properties. For example, we will determine an exact expression for the number of pairs of partitions that have no blocks in common. Furthermore, we will show how to compute the block distance efficiently.

The *Rand distance* of two set partitions is the number of unordered pairs $\{x, y\}$ such that there is a block in one partition containing both x and y, but x and y are in different blocks in the other partition. We use $\mathcal{R}(P, Q)$ to denote the Rand distance between two set partitions P and Q. For example, $\mathcal{R}(\{\{1,2\},\{3\}\},\{\{1\},\{2,3\}\}) = 2$ (the pairs are $\{1,2\}$ and $\{2,3\}$) and $\mathcal{R}(\{\{1,2,3\}\},\{\{1\},\{2\},\{3\}\}) = 3$ (the pairs are $\{1,2\}$, $\{1,3\}$, and $\{2,3\}$). In general, if P and Q are partitions of an n-set, then $0 \leq \mathcal{R}(P, Q) \leq \binom{n}{2}$.

Let $R(n, k)$ be the number of distinct (unordered) pairs of partitions of an n-set that have Rand distance k. See Table 2 in Section 3.1. This table was computed from exhaustive computer listings of all partitions of $\{1, 2, \ldots, n\}$ up

⋆ Research supported in part by NSERC.

C.S. Iliopoulos and W.F. Smyth (Eds.): IWOCA 2011, LNCS 7056, pp. 287–299, 2011.
© Springer-Verlag Berlin Heidelberg 2011

to $n = 11$. The column sums are $\binom{B_n}{2}$. Note that the numbers for fixed n are not unimodal in general.

We define the *block distance* $\mathcal{B}(P, Q)$ between two partitions of n as the number of elements in the blocks that are not common to both P and Q. For example,

$$\mathcal{B}(\{\{1,2\},\{3\}\},\{\{1\},\{2\},\{3\}\}) = 2$$

since the only block that is common to both partitions is $\{3\}$ and there are 2 elements in the remaining blocks. By $B(n, k)$ we denote the number of pairs of partitions of n that have block distance k. See Table 1 in Section 2. The Rand distance can be cleverly computed using a linear number of arithmetic operations; see Filkov and Skiena [2] and we will show that the block distance is also efficiently computable.

Organizationally, we will finish this section by giving some background on set partitions. In the succeeding two sections, we discuss first the block distance and then the Rand distance. The focus is mainly on the elucidation of some enumerative results along with a clever $O(n)$ algorithm for computing the block distance.

1.1 Background on Set Partitions

A *partition* of a set S is collection of disjoint subsets of S, say $\{S_1, S_2, \cdots, S_k\}$ whose union is S. Each S_i is referred to as a *block*. The number of partitions of an n-set into k blocks is the Stirling number (of the second kind), which is denoted as $\left\{{n \atop k}\right\}$. We use $[n]$ to denote $\{1, 2, \ldots, n\}$.

In the computer, partitions are usually represented by *restricted growth strings*. We assume that the blocks of a partition X are numbered $S_1, S_2, \ldots, S_k$ according to the size of the smallest element in each block. That is, S_1 contains 1, S_2 contains the smallest element not in S_1, and so on. Then the restricted growth string $r[1..n]$ of X is defined by taking $r[i]$ to be the distance of the block containing i. The Gray code algorithms for generating restricted growth strings developed in [9] and discussed in [5] were used to generate the numbers in Tables 1 and 2; as each string was generated the $O(n)$ algorithms for computing the Rand distance and the block distance were applied.

The n-th Bell number, B_n, is the total number of partitions of an n-set, irrespective of block size. Thus $B_n = \sum_k \left\{{n \atop k}\right\}$. The exponential generating function (egf) of the Bell numbers is well-known (e.g., Stanley [10], pg. 34)) to be

$$B(z) = \sum_{n \geq 1} B_n \frac{z^n}{n!} = e^{e^z - 1}. \tag{1}$$

The number of pairs of partitions is $\binom{B_n}{2}$. For $n = 1, 2, 3, \ldots, 10$ these numbers are

$$0, 1, 10, 105, 1326, 20503, 384126, 8567730, 223587231, 6725042325.$$

They give the row sums in Tables 1 and 2.

We use several times a generalization of the fact that if $f(z) = \sum_{n\geq 0} f_n z^n/n!$ is the egf of a sequence f_n, then $zf(z)$ is the egf of the sequence nf_{n-1}. See Knuth, Graham, Patashnik [3], page 350. Furthermore, for $k \geq 0$,

$$\begin{aligned} z^k f(z) &= \sum_{n\geq k} n(n-1)\cdots(n-k+1) f_{n-k} \frac{z^n}{n!} \\ &= k! \sum_{n\geq 0} \binom{n}{k} f_{n-k} \frac{z^n}{n!}. \end{aligned} \tag{2}$$

Thus $k!\binom{n}{k} f_{n-k}$ is the n-th coefficient of $z^k f(z)$.

2 The Block Distance

Recall that the *block distance* $\mathcal{B}(P, Q)$ of two partitions of n is the number of elements in the blocks that are *not* common to both P and Q, and that $B(n, k)$ is the number of pairs of partitions of n that have block distance k. See Table 1.

Table 1. The values of $B(n, k)$ for $1 \leq k \leq n \leq 9$

$n\backslash k$	2	3	4	5	6	7	8	9
2	1							
3	3	7						
4	12	28	65					
5	50	140	325	811				
6	225	700	1950	4866	12762			
7	1092	3675	11375	34062	89334	244588		
8	5684	20384	68250	227080	714672	1956704	5574956	
9	31572	119364	425880	1532790	5360040	17610336	50174604	148332645

Let $N(n) = B(n, n)$; this is the number of unordered pairs of partitions that have no blocks in common. The numerical values of $N(n)$, for $0 \leq n \leq 10$, are

$$0, 0, 1, 7, 65, 811, 12762, 244588, 5574956, 148332645, 4538695461.$$

Determining $N(n)$ for $i = 1, \ldots, n$ is sufficient to determine $B(n, k)$ since, by direct combinatorial considerations,

$$B(n, k) = N(k)\binom{n}{k} B_{n-k}. \tag{3}$$

We also note that

$$\binom{B_n}{2} = \sum_{k=0}^{n} B(n, k) = \sum_{k=0}^{n} N(k)\binom{n}{k} B_{n-k}. \tag{4}$$

Letting $N(z)$ be the egf of the $N(n)$ numbers, from (4) we obtain the equation

$$P(z) := \sum_{n\geq 0} \binom{B_n}{2} \frac{z^n}{n!} = N(z)e^{e^z-1}.$$

And thus

$$N(z) = P(z)e^{1-e^z}. \tag{5}$$

The egf e^{1-e^z} is known; it is the egf of the "complementary Bell numbers" (OEIS A000587). The complementary Bell numbers, C_n, for $n = 0, 1, 2, \ldots, 14$ are

$$1, -1, 0, 1, 1, -2, -9, -9, 50, 267, 413, -2180, -17731, -50533, 110176.$$

It is known that

$$C_n = \sum_{k=0}^{n} (-1)^k \left\{ {n \atop k} \right\}.$$

Thus, from (5) we get a "closed-form" formula for $N(n)$, namely

$$N(n) = \sum_{j=0}^{n} \binom{n}{j} C_j \binom{B_{n-j}}{2} = \sum_{j=0}^{n} \binom{n}{j} \binom{B_{n-j}}{2} \sum_{k=0}^{j} (-1)^k \left\{ {j \atop k} \right\}.$$

2.1 Linear Time Algorithm to Compute the Block Distance

In this subsection we present a linear time algorithm to compute the block distance of two partitions.

Closely related to the restricted growth string, we define the *block string*, $b[1..n]$, of P as follows: $b[i]$ is the smallest element in the block containing i. Every block string has the characterizing property that $b[1] = 1$, and for $i > 1$,

$$b[i] \in \{i,\ b[1], b[2], \ldots, b[i-1]\}.$$

It is relatively simple to convert a restricted growth string into the corresponding block string in $O(n)$ time.

The following code takes as input a restricted growth function $r[1..n]$ and returns the corresponding block string $b[1..n]$. It uses a temporary array $m[1..n]$ that maintains the invariant $b[i] = m[r[i]]$.

for $i \in \{1, 2, \ldots, n\}$ **do** $m[i] := 0$;
for $i := 1, 2, \ldots, n$ **do**
  if $m[r[i]] = 0$ **then** $m[r[i]] := i$;
  $b[i] := m[r[i]]$;

Before describing the algorithm for computing the block distance, we encourage the reader to consider the following small example. Suppose

$$P = \{1\}\{2\}\{3,4\}\{5,7\}\{6\}, \qquad Q = \{1,2\}\{3,4,6\}\{5,7\}.$$

Then the restricted growth strings for P and Q are

$$rP = 1, 2, 3, 3, 4, 5, 4, \qquad rQ = 1, 1, 2, 2, 3, 2, 3$$

and the block strings are

$$p = 1, 2, 3, 3, 5, 6, 5, \qquad q = 1, 1, 3, 3, 5, 3, 5.$$

Comparing the elements, we find that the blocks labelled 1, 2, 3, and 6 are not common to P and Q and that the block labelled 5 is common to P and Q. Since there are 5 elements in blocks 1, 2, 3, and 6, the block distance of P and Q is 5.

The algorithm maintains a boolean array $C[1..n]$ with the property that, upon termination, $C[i]$ is true if i is in a block common to P and Q, and is false otherwise. The block distance is thus equal to the number of entries in this array that are false.

The algorithm makes two passes over p, one pass over q, and one pass over C. Consider $p[i]$ and $q[i]$; there are three mutually exclusive cases: (a) $p[i] \neq q[i]$ and i is not in a common block, (b) $p[i] = q[i]$ and i is in a common block, and (c) $p[i] = q[i]$ and i is not in a common block. (Because we are using the block string and not the restricted growth string, it is not possible that $p[i] \neq q[i]$ and i is in a common block.) In the first pass we test only for case (a). In the second pass we (indirectly) distinguish cases (b) and (c).

The key observation is this: If i is *not* in a common block and $p[i] = q[i]$, then there is some value $j \neq i$ such that j is in the same block as i in P but is in a different block than i in Q, or vice-versa. In other words, $p[i] = p[j] \neq q[j]$ or $p[j] \neq q[j] = q[i]$. Thus, in the first pass $C[p[j]]$ and $C[q[j]]$ were set to false.

So on the second pass, we test whether $C[p[i]]$ is false to determine whether i is in a common block or not. On the final pass, we find the block distance by counting the number of false values in C. Below is the code in detail.

```
for i ∈ {1, 2, ..., n} do C[i] := true
for i := 1, 2, ..., n do
    if p[i] ≠ q[i] then C[p[i]] := C[q[i]] := false;
for i := 1, 2, ..., n do
    if ¬C[p[i]] then C[i] := false;
c := 0;
for i ∈ {1, 2, ..., n} do
    if ¬C[i] then c := c + 1;
return(c);
```

3 Results on the Rand Distance

Now recall that $\mathcal{R}(P, Q)$ is the number of unordered pairs $\{x, y\}$ such that there is a block in one partition containing both x and y, but x and y are in different blocks in the other partition, and that $R(n, k)$ is the number of distinct (unordered) pairs of partitions of an n-set that have Rand distance k. See Table 2. Let $R(n)$ be the sum of the Rand distance over all unordered pairs of partitions.

Table 2. The values of $R(n,k)$ for $2 \le n \le 11$ and $1 \le k \le 55$. This table is inverted in the sense that k increases down columns and n varies along the columns.

$k\backslash n$	2	3	4	5	6	7	8	9	10	11
1	1	3	12	50	225	1092	5684	31572	186300	1163085
2		6	30	150	780	4200	23772	141624	887220	5835060
3		1	32	280	1720	10885	69272	452508	3060360	21482340
4			24	300	3360	25200	183960	1341648	9883440	74471760
5			6	240	3426	42672	391356	3266172	26969040	222185304
6			1	220	4100	56889	696178	7234374	67288830	612903720
7				60	2400	60165	941088	12259368	141778440	1469224350
8				15	2700	57750	1182888	18992502	256463820	3164268690
9				10	1075	46585	1150520	23324140	399874640	5762811670
10				1	471	31374	1165416	28129626	547907454	9538994388
11					150	24528	815640	26605908	670419540	13513772745
12					35	14140	780570	26190612	742419510	18112131840
13					45	4725	413840	21568932	744780330	20675910420
14					15	1890	369180	17119818	701747010	23653643310
15					1	1302	178080	13040280	607809750	22677991578
16						252	115780	8948079	520591950	22923998460
17						210	43512	6244308	377521875	19287053775
18						140	20734	3679032	312082260	17554312490
19						105	6860	2431044	198307620	13495597225
20						21	7098	1250109	158606532	11143736604
21						1	3508	640908	87210930	8029798920
22							574	315828	63688410	6035010960
23							840	197568	33243120	4254456690
24							665	57288	25703205	2872892550
25							476	46116	11343906	1924619235
26							210	30366	6764940	1215058680
27							28	25732	3272500	789847190
28							1	7695	2003805	453548480
29								4104	1532340	306871290
30								2226	757080	177358500
31								3780	211410	112440900
32								2205	212625	53211510
33								1344	198345	35497935
34								378	138600	16793040
35								36	82512	13781493
36								1	21080	10664335
37									16200	6744100
38									15750	2483415
39									14910	1445565
40									13545	802164
41									7245	1320165
42									3270	860640
43									630	580965
44									45	215325
45									1	104313
46										62205
47										103950
48										70455
49										74250
50										45045
51										21945
52										7095
53										990
54										55
55										1

Theorem 1

$$R(n) = \sum_{k=0}^{\binom{n}{2}} k\ R(n,k) = \binom{n}{2} B_{n-1}(B_n - B_{n-1}).$$

Proof Choose a pair $\{x, y\}$. The number of partitions in which this pair appears in the same block is B_{n-1}. The number of partitions in which this pair appears in different blocks is the difference $B_n - B_{n-1}$. Thus in total, each pair contributes $B_{n-1}(B_n - B_{n-1})$ to the sum. Since there are $\binom{n}{2}$ ways to choose a pair, the proof is finished. □

The average value of the Rand distance is thus

$$\frac{R(n)}{\binom{B_n}{2}} = \frac{n(n-1)B_{n-1}(B_n - B_{n-1})}{B_n(B_n - 1)}.$$

Since the Bell numbers grow exponentially,

$$\frac{R(n)}{\binom{B_n}{2}} \sim n^2 \frac{B_{n-1}}{B_n},$$

which experimentally appears to be $\Theta(n \log n)$.

3.1 Determining $R(n, k)$ for Small Values of k

We now consider $R(n, k)$ for small values of k.

Clearly $R(n, 0) = 0$.

Theorem 2. *For all $n \geq 1$,*

$$R(n,1) = \binom{n}{2} B_{n-2}.$$

Proof. The only way that the Rand distance can be 1 is if there is a block $\{x, y\}$ in one partition and two blocks $\{x\}, \{y\}$ in the other, and all other blocks in one partition are present in the other. There are $\binom{n}{2}$ ways to choose the pair and B_{n-2} ways to determine the other blocks. □

Corollary 1. *The egf of the $R(n, 1)$ numbers is*

$$\sum_{n\geq 1} R(n,1)\frac{z^n}{n!} \ = \ \frac{z^2}{2}B(z) \ = \ \frac{z^2}{2}e^{e^z-1}.$$

Proof. Apply (2) with $k = 2$. □

The previous two results were warm-ups for the more technical results that follow.

Theorem 3. *For fixed k, there are non-negative integer constants $c_{k,j}$ such that, for all $n \geq 1$,*

$$R(n,k) = \sum_{j=\lceil (1+\sqrt{1+8k})/2 \rceil}^{2k} c_{k,j} \binom{n}{j} B_{n-j}.$$

Proof. Any two partitions P and Q will have a largest subpartition X that is common to both P and Q. As a consequence, $\mathcal{R}(P,Q) = \mathcal{R}(P \setminus X, Q \setminus X)$. In the sum above j represents $n - |X|$, given that $\mathcal{R}(P,Q) = k$. Thus, $c_{k,j}$ is the number of pairs of j-element set partitions with no common blocks and that have Rand distance k. It remains to prove that the restrictions on the index of summation are correct.

The lower bound in the summation follows from the fact that the maximum Rand distance between two partitions of n is $\binom{n}{2}$ and thus $k \leq \binom{j}{2}$. Solving the implied quadratic yields $j \geq (1+\sqrt{1+8k})/2$, which gives us the lower bound. We hereafter use $\alpha = \lceil (1+\sqrt{1+8k})/2 \rceil$ for ease of reading.

For the upper bound, consider two partitions P and Q of an j-set that have no block in common, and have Rand distance k. We claim that $k \geq \lceil j/2 \rceil$. Consider some arbitrary integer $x \in \{1, 2, \ldots, j\}$. Since P and Q have no common blocks, there is some integer y that is in the same block as x in one partition, and in another block in the other partition. Thus we have j distinct ordered pairs (x, y), one for each different value of x. At least $\lceil j/2 \rceil$ of them have to be distinct as unordered pairs, and each such unordered pair contributes 1 to the Rand distance. Thus $k \geq \lceil j/2 \rceil$ as claimed. From this it follows that $j \leq 2k$, which is the upper bound in the sum above. □

Theorem 4. *For all $n \geq 1$,*

$$R(n,2) = 6\binom{n}{3} B_{n-3} + 6\binom{n}{4} B_{n-4}.$$

Proof. Theorem 3 tells us that

$$R(n,2) = c_{2,3}\binom{n}{3} B_{n-3} + c_{2,4}\binom{n}{4} B_{n-4}.$$

From the $k = 2$ row of Table 2 we then have the following two equations.

$$R(3,2) = 6 = c_{2,3}\binom{3}{3} B_0 + c_{2,4}\binom{3}{4} B_{-1} = c_{2,3} \quad \text{and}$$

$$R(4,2) = 30 = c_{2,3}\binom{4}{3} B_1 + c_{2,4}\binom{4}{4} B_0 = c_{2,3}4 + c_{2,4}.$$

This system of equations can be solved to obtain $c_{2,3} = c_{2,4} = 6$. □

Corollary 2. *The egf of the $R(n,2)$ numbers is*

$$\sum_{n \geq 1} R(n,2) \frac{z^n}{n!} \;=\; \left(z^3 + \frac{z^4}{4}\right) B(z) \;=\; \left(z^3 + \frac{z^4}{4}\right) e^{e^z - 1}.$$

In a similar fashion we can solve systems of linear equations to obtain the following theorems and corollaries.

Theorem 5. *For all $n \geq 1$,*

$$R(n,3) = \binom{n}{3}B_{n-3} + 28\binom{n}{4}B_{n-4} + 120\binom{n}{5}B_{n-5} + 60\binom{n}{6}B_{n-6}.$$

Corollary 3. *The egf of the $R(n,3)$ numbers is*

$$\sum_{n\geq 1} R(n,3)\frac{z^n}{n!} \;=\; \left(\frac{z^3}{6} + \frac{7z^4}{6} + z^5 + \frac{z^6}{12}\right)B(z) \;=\; \left(\frac{z^3}{6} + \frac{7z^4}{6} + z^5 + \frac{z^6}{12}\right)e^{e^z-1}.$$

Theorem 6. *For all $n \geq 1$, the value of $R(n,4)$ is*

$$24\binom{n}{4}B_{n-4} + 180\binom{n}{5}B_{n-5} + 1560\binom{n}{6}B_{n-6} + 2520\binom{n}{7}B_{n-7} + 840\binom{n}{8}B_{n-8}.$$

Corollary 4. *The egf of the $R(n,4)$ numbers is*

$$\sum_{n\geq 1} R(n,4)\frac{z^n}{n!} \;=\; \left(z^4 + \frac{3z^5}{2} + \frac{13z^6}{6} + \frac{z^7}{2} + \frac{z^8}{48}\right)B(z).$$

We summarize the known values of $c_{k,j}$ in Table 3. Although we don't know the value of $c_{k,j}$ in general, we can determine a few specific infinite sequences, which are given in the next lemma.

Lemma 1. *For all $k \geq 1$,*

$$c_{k,2k} = \frac{(2k-1)!}{(k-1)!} \quad \textit{and} \quad c_{k,\alpha} = R(\alpha,k).$$

Proof. For a pair of $2k$ element set partitions P and Q to have Rand distance k with no common blocks, the $2k$ elements must be paired, and each pair of elements is a block in either P or Q. Further, if $\{a,b\}$ is a block in set P then set B contains the singleton blocks $\{a\}$ and $\{b\}$ and vice versa. Since the order of the blocks doesn't matter, we can assume the blocks (pairs) are sorted by their smallest elements. So, for $i = 1,2,\ldots,k$, once we have chosen the elements for blocks $1,2,\ldots,i-1$, the first element in block i must be the smallest remaining element and there are $2k-(2(i-1)+1) = 2k-2i+1$ choices for the second element in block i. Thus the number of ways to pair the elements is

$$\prod_{i=1}^{k}(2k-2i+1) = \frac{(2k-1)!}{2^{k-1}(k-1)!}$$

If we assume, without loss of generality, that a pair, say $\{a,b\}$, is in partition P, then there are 2^{k-1} unique ways to distribute the remaining pairs between P and Q. So we have

$$c_{j,2k} = \frac{(2k-1)!}{2^{k-1}(k-1)!}2^{k-1} = \frac{(2k-1)!}{(k-1)!}.$$

Table 3. Known values of $c_{k,j}$ for $2 \leq j \leq 11$. The bold value at the beginning of each row is $c_{k,\alpha} = R(\alpha, k)$.

$k\backslash j$	2	3	4	5	6	7	8	9	10	11
1	**1**									
2		**6**	6							
3		**1**	28	120	60					
4			**24**	180	1560	2520	840			
5			**6**	210	1986	18900	63840	60480	15120	
6			**1**	215	2780	28224	253246	1340640	2520000	1663200
7				**60**	2040	43365	463128	3998736	26878320	82328400
8				**15**	2610	38850	721728	8575200	74028240	554843520
9				**10**	1015	39060	778400	13061020	172444150	1568364600
10				**1**	465	28077	914480	17680572	270474480	3714220092
11					**150**	23478	619416	19277748	407335320	6281694045
12					**35**	13895	667450	19168422	482217540	10078945140
13					**45**	4410	376040	17848152	529667460	12553128060
14					**15**	1785	354060	13798458	530778780	15995950740
15					**1**	1295	167664	11437644	477563400	16021896264
16						**252**	113764	7906059	431141400	17216673870
17						**210**	41832	5852700	315103995	15141561930
18						**140**	19614	3492426	275308740	14124874940
19						**105**	6020	2369304	174009780	11315379955
20						**21**	6930	1186227	146107962	9400242852
21						**1**	3500	609336	80801970	7071057840
22							**574**	310662	60530130	5334533160
23							**840**	190008	31267440	3888920970
24							**665**	51303	25130325	2590267020
25							**476**	41832	10882746	1799914809
26							**210**	28476	6461280	1140678990
27							**28**	25480	3015180	753854310
28							**1**	7686	1926855	431506790
29								**4104**	1491300	290015550
30								**2226**	734820	169030620
31								**3780**	173610	110115390
32								**2205**	190575	50872635
33								**1344**	184905	33316140
34								**378**	134820	15268440
35								**36**	82152	12873861
36								**1**	21070	10432455
37									**16200**	6565900
38									**15750**	2310165
39									**14910**	1281555
40									**13545**	653169
41									**7245**	1240470
42									**3270**	824670
43									**630**	574035
44									**45**	214830
45									**1**	104302
46										**62205**
47										**103950**
48										**70455**
49										**74250**
50										**45045**
51										**21945**
52										**7095**
53										**990**
54										**55**
55										**1**

Since $B_i = 0$ when $i < 0$, $B_0 = 1$, and $\binom{i}{i} = 1$,

$$R(\alpha, k) = \sum_{j=\alpha}^{2k} c_{k,j} \binom{\alpha}{j} B_{\alpha - j} = c_{k,\alpha}.$$

□

Lemma 2. *For all $j \geq 1$,*

$$c_{\binom{j}{2},j} = R\left(j, \binom{j}{2}\right) = 1.$$

For all $j \geq 4$:

$$c_{\binom{j}{2}-1,j} = R\left(j, \binom{j}{2} - 1\right) = \binom{j}{2}.$$

For all $j \geq 5$:

$$c_{\binom{j}{2}-2,j} = R\left(j, \binom{j}{2} - 2\right) = \binom{\binom{j-1}{2}}{2}.$$

For all $j \geq 2 + x$:

$$c_{\binom{j}{2}-x,j} = R\left(j, \binom{j}{2} - x\right).$$

Proof. Omitted in this extended abstract. □

3.2 The Numbers $R(n, \binom{n}{2} - k)$ for Small k

We now consider the numbers at the bottom of the columns in Table 2. Clearly $R(n, \binom{n}{2}) = 1$ (the pair is $\{1, 2 \ldots n\}$ and $\{1\}\{2\} \ldots \{n\}$).

Theorem 7. *For all $n \geq 4$,*

$$R(n, \binom{n}{2} - 1) = \binom{n}{2}, \text{ and } R(3, 2) = 6.$$

Proof. For $n \geq 4$, the two partitions are the full set $\{1, 2, \ldots, n\}$ and the partition consisting of one pair and $n - 2$ singleton sets. □

Theorem 8. *For all $n \geq 5$,*

$$R(n, \binom{n}{2} - 2) = \binom{\binom{n-1}{2}}{2} = \frac{1}{8} n(n-1)(n-2)(n-3),$$

and $R(3, 1) = 3$, $R(4, 4) = 24$.

Proof. For $n \geq 5$ the two partitions are the full set $\{1, 2, \ldots, n\}$ and the partition consisting of two pairs and $n-4$ singleton sets. The order of the two pairs does not matter so we have

$$R(n, \binom{n}{2} - 2) = \frac{1}{2}\binom{n}{2}\binom{n-2}{2},$$

which can be shown to be equal to the two values given in the statement of the theorem. □

The numbers in Theorems 7 and 8 are a shifted versions of OEIS A000217 and OEIS A050534, respectively.

Theorem 9. *For all $n \geq 6$,*

$$R(n, \binom{n}{2} - 3) = \frac{1}{6}\binom{n}{2}\binom{n-2}{2}\binom{n-4}{2} + \binom{n}{3},$$

and $R(4, 3) = 32$, $R(5, 7) = 60$.

Proof. For $n \geq 5$ the two partitions are either the full set $\{1, 2, \ldots, n\}$ and the partition consisting of three pairs and $n-6$ singleton sets, or the full set $\{1, 2, \ldots, n\}$ and the partition consisting of one triple and $n-3$ singleton sets. □

Theorem 10. *For fixed k there is a constant K_k such that $R(n, \binom{n}{2} - k)$ is a polynomial of degree $2k$ in n for all $n \geq K_k$.*

Proof. Omitted in this extended abstract. □

Acknowledgements. We wish to thank Rod Canfield for helpful discussions.

References

1. Canfield, E.R.: Engel's inequality for the Bell numbers. Journal of Combinatorial Theory, Series A 72, 184–187 (1995)
2. Filkov, V., Skiena, S.: Integrating Microarray Data by Consensus Clustering. International Journal on Artificial Intelligence Tools 13(4), 863–880 (2004)
3. Graham, R.L., Knuth, D.E., Patashnik, O.: Concrete Mathematics. Addison-Wesley (1989)
4. Hubert, L.: Comparing Partitions. Journal of Classification 2, 193–218 (1985)
5. Knuth, D.E.: The Art of Computer Programming, vol 4: Combinatorial Algorithms, Part 1. Addison-Wesley, Reading (2011)
6. Moser, L., Wyman, A.: An asymptotic formula for the Bell numbers. Transactions of the Royal Society of Canada III 49, 49–54 (1955)
7. Munagi, A.O.: Set Partitions with Successions and Separations. Int. J. Math and Math. Sc. 2005(3), 451–463 (2005)

8. Rand, W.: Objective criteria for the evaluation of clustering methods. J. American Statistical Assoc. 66(336), 846–850 (1971)
9. Ruskey, F.: Simple Combinatorial Gray Codes Constructed by Reversing Sublists. In: Ng, K.W., Balasubramanian, N.V., Raghavan, P., Chin, F.Y.L. (eds.) ISAAC 1993. LNCS, vol. 762, pp. 201–208. Springer, Heidelberg (1993)
10. Stanley, R.R.: Enumerative Combinatorics, vol. 1. Wadsworth (1986)

On Minimizing the Number of Label Transitions around a Vertex of a Planar Graph

Bojan Mohar and Petr Škoda

Department of Mathematics, Simon Fraser University,
8888 University Drive, Burnaby, BC, V5A 1S6, Canada
{mohar,pskoda}@sfu.ca

Abstract. We study the minimum number of label transitions around a given vertex v_0 in a planar multigraph G in which the edges incident with v_0 are labelled with integers $1, \ldots, l$, where the minimum is taken over all embeddings of G in the plane. For a fixed number of labels, a linear-time FPT algorithm that (given the labels around v_0) computes the minimum number of label transitions around v_0 is presented. If the number of labels is unconstrained, then the problem of deciding whether the minimum number of label transitions is at most k is NP-complete.

Keywords: label transitions, planar graph, fixed-parameter tractable.

1 Introduction

Let G be a planar 2-connected multigraph. Suppose that the edges incident with a vertex $v_0 \in V(G)$ are labelled by integers $1, 2, \ldots, l$. We are interested in finding an embedding of G in the plane such that the number of label transitions around v_0 is minimized. By a *label transition* we mean two edges that are consecutive in the local rotation around v_0 and whose labels are different. The motivation for this problem comes from investigations of minimum genus embeddings of graphs with small separations. In particular, to compute genus of a 2-sum of two graphs [11], see also [4], [5] and [10], it is necessary to know if a graph admits a planar embedding with only four label transitions (where $l = 2$).

By deleting the vertex v_0 from G and putting all edge labels onto vertices incident with the deleted edges, we obtain an equivalent formulation of the same problem. Both representations are useful and will be treated in this paper. Let H be the graph obtained from G by deleting v_0. Note that H is connected. For each $v \in V(H)$ let $\lambda(v)$ be the set of all labels of edges joining v and v_0 in G. If v is not a neighbor of v_0, then $\lambda(v) = \emptyset$. The pair (H, λ) carries the whole information about G and the labels of edges around v_0 (except for multiplicity of the edges with the same label).

Let $\mathcal{L}$ be a set of labels. The graph H together with the labelling $\lambda : V(H) \to 2^{\mathcal{L}}$ is a *labelled graph*. Let $\widehat{H}$ be the graph obtained from a labelled graph H by adding a vertex v_0 to H and joining it to each vertex v by $|\lambda(v)|$ edges and labelling these edges by elements of $\lambda(v)$. The vertex v_0 is called the *center* of

C.S. Iliopoulos and W.F. Smyth (Eds.): IWOCA 2011, LNCS 7056, pp. 300–309, 2011.
© Springer-Verlag Berlin Heidelberg 2011

$\widehat{H}$. If the graph $\widehat{H}$ is planar (which can be checked in linear time, see [7]), we are back to an instance of the original problem.

Given (H, λ) or $\widehat{H}$, v_0, and the labelling of edges incident with v_0, consider an embedding Π of $\widehat{H}$ in the plane (all embeddings in this paper are into the plane). Define the *label sequence* $Q = Q(\Pi)$ of Π to be the cyclic sequence of labels of edges coming to v_0 in the clockwise order of the local rotation around v_0 in Π. The *origin* of a label $L \in Q$ that came from an edge vv_0 is the vertex v. A *label transition* in Q is a pair of (cyclically) consecutive labels A, B in Q such that $A \neq B$. The *number of transitions* $\tau(Q)$ of Q is the number of label transitions in Q. The *number of transitions* $\tau(\widehat{H})$ of $\widehat{H}$ is the minimum $\tau(Q(\Pi))$ taken over all planar embeddings Π of $\widehat{H}$. When considering label transitions, the graphs H and $\widehat{H}$ are used interchangeably, i.e., $\tau(H) \equiv \tau(\widehat{H})$.

The following problem will be of our main interest:

Min-Trans. Given a planar 2-connected multigraph G with edges incident to a fixed vertex v_0 labelled by $1, \ldots, l$ and an integer k, determine if $\tau(G) \leq k$.

In the following, we show that Min-Trans can be solved in linear time when the number of labels l is fixed.

Theorem 1. *For every fixed integer l, there is a linear-time algorithm that determines the minimum number of transitions $\tau(G)$ of a given planar 2-connected multigraph G with edges incident to a fixed vertex v_0 labelled by $1, \ldots, l$. This algorithm is fixed-parameter tractable.*

We also show that this is best possible in the sense that Min-Trans becomes NP-complete when l is part of the input.

Theorem 2. Min-Trans *is NP-complete if the number of labels l is unconstrained. The problem remains NP-complete even when each label occurs precisely twice.*

2 Bounded Number of Labels

In this section we develop most of the formalism needed to prove Theorem 1. In particular, it is observed that we can restrict our attention to a special class of cactus graphs; also, the basic structure of the algorithm is presented.

Let H and G be labelled graphs. If every label sequence of H is also a label sequence of G, and vice versa, then H and G are said to be *equivalent*.

A connected graph G is called a *cactus* if every block of G is either an edge or a cycle. A labelled cactus G is *leaf-labelled* if every endblock of G is an edge, every vertex of G has at most one label, and a vertex of G is labelled if and only if it is a leaf.

The following lemma shows that it is enough to prove Theorem 1 for the case when H is a leaf-labelled cactus. The main idea is that the "interiors" of 2-connected components are not significant for our problem. The proof is given in the full paper.

Lemma 1. *Let H be a connected labelled graph. If $\widehat{H}$ is planar, then there exists a leaf-labelled cactus G which is equivalent to H. Furthermore, G can be constructed in linear time.*

In our algorithm, we use a rooted version of graphs. A *root* r in a leaf-labelled cactus H can be any vertex of H. The root is marked by a special label $L_r \notin \mathcal{L}$. We then speak of a *rooted leaf-labelled cactus*, or simply a cactus (H, r). The restriction on labels in a rooted leaf-labelled cactus is slightly relaxed, every leaf still has a unique label (possibly L_r) but a non-leaf vertex can also be labelled if it is the root. When a label sequence Q of H is cut at the label L_r (and L_r is deleted), we obtain a linear sequence called a *rooted label sequence* of H. Let $\mathcal{Q}(H)$ denote the set of all rooted label sequences of H. Similarly to the unrooted graphs, two rooted graphs are *equivalent* if they admit the same rooted label sequences.

There exists a tree-like structure, called a PC-tree (see [12]), that captures all embeddings of a cactus in the plane. PC-trees and their rooted version, PQ-trees, are used in testing planarity [2]. We note that Min-Trans reduces to the problem of minimizing the number of label transitions over all cyclic permutations of the leaves of a PC-tree that are compatible with the PC-tree.

For every embedding Π of $\widehat{H}$ there is the flipped embedding Π' of $\widehat{H}$ where each clockwise rotation in Π is a counter-clockwise rotation in Π'. The following lemma formulates this for a rooted label sequence of H. For a linear sequence Q, let Q^R denote the sequence obtained by reversing Q.

Lemma 2. *Let (H, r) be a rooted leaf-labelled cactus. If Q is a rooted label sequence of H, then the reversed sequence Q^R is also a rooted label sequence of H.*

The following lemmas establish a recursive construction of rooted label sequences. Let us recall that for a cut vertex v of H, *v-bridge* in H is a subgraph of H consisting of a connected component of $H - v$ together with all edges joining this component and v.

Lemma 3. *Let (H, r) be a rooted leaf-labelled cactus where r is a leaf. Let u be the neighbor of r. If u is labelled, then H has a unique rooted label sequence $Q = \lambda(u)$. Otherwise, (H, r) is equivalent to $(H - r, u)$.*

Proof. If u is labelled, then u is a leaf and H contains precisely one label $\lambda(u)$ and therefore $\lambda(u)$ is the unique rooted label sequence of H. Otherwise, take an embedding of $\widehat{H - r}$ in the plane. Recall that u as the root of $H - r$ is given a special label L_u and thus there is an edge connecting u and the center of $\widehat{H - r}$. Subdividing this edge gives a planar embedding of $\widehat{H}$ with the same rooted label sequence. Similarly, one can obtain an embedding of $\widehat{H - r}$ from an embedding of $\widehat{H}$ with the same rooted label sequence. □

Lemma 4. *Let (H, r) be a rooted leaf-labelled cactus with r in a cycle C of length k. For $v \in V(H)$, let D_v be the union of v-bridges in H that do not contain C.*

If D_r is empty, then every rooted label sequence Q of H can be partitioned into $k-1$ (possibly empty) consecutive parts P_v, $v \in V(C) \setminus \{r\}$, where P_v is a rooted label sequence of (D_v, v) and P_v appear in Q in one of the two cyclic orders corresponding to the two orientations of C.

Proof. Let Q be a rooted label sequence of H such that the conclusion of the lemma is not true. If labels contained in one of the subgraphs D_v do not form a consecutive subsequence of Q, we obtain a contradiction as in the proof of Lemma 5. Suppose now that Q contains a cyclic subsequence $L_1L_3L_2L_4$ (in this order) such that the origins $u_1, \ldots, u_4$ of $L_1, \ldots, L_4$ are in $D_{v_1}, \ldots, D_{v_4}$ and $v_1, \ldots, v_4$ appear on C in this order. Let Π be an embedding of $\widehat{H}$ that corresponds to Q and let G be the graph obtained from $\widehat{H}$ by deleting the center v_0 and adding an edge uv for every two consecutive edges uv_0, vv_0 in the local rotation around v_0. Π can be easily modified to a planar embedding Π' of G. It is easy to check that $u_1, \ldots, u_4, v_1, v_3$ are the branch-vertices of a subdivision of $K_{3,3}$ in G, a contradiction with G being planar (see [9]). □

Lemma 5. *Let (H, r) be a rooted leaf-labelled cactus with r a cut vertex and let $B_1, \ldots, B_k$ be the r-bridges in H. Every rooted label sequence of H can be partitioned into k consecutive parts where each of the k parts is a rooted label sequence of one of (B_i, r). Furthermore, if Q_i is a rooted label sequence of (B_i, r) $(1 \leq i \leq k)$ and $(i_1, \ldots, i_k)$ is a permutation of $(1, \ldots, k)$, then the concatenation $Q_{i_1}Q_{i_2}\cdots Q_{i_k}$ is a rooted label sequence of H.*

Proof. Suppose for a contradiction that there is a rooted label sequence Q of H with a cyclic subsequence $L_1L_2L_3L_4$ (in this order) such that L_1 and L_3 have origins in B_1 and L_2, L_4 have origins outside B_1. Let Π be an embedding of $\widehat{H}$ that corresponds to Q and $v_1, \ldots, v_4$ the origins of $L_1, \ldots, L_4$. Let G be the graph obtained from $\widehat{H}$ by deleting the center v_0 and adding an edge uv for every two consecutive edges uv_0, vv_0 in the local rotation around v_0. The embedding Π can be extended to a planar embedding Π' of G such that the added edges form a facial cycle. Since v_1 and v_3 are in B_1, there is a path P in $B_1 - r$ joining v_1 and v_3. Similarly, there is a path Q in $H - (B_1 - r)$ joining v_2 and v_4. Since P and Q are disjoint and both embedded inside C, their endvertices cannot interlace on C. This contradiction proves the claim and implies that all labels in each B_i appear consecutively in every rooted label sequence of H. This proves the first part of the lemma.

The second part is an easy consequence of the fact that arbitrary embeddings of $\widehat{B_i}$ $(1 \leq i \leq k)$ can be combined into an embedding of $\widehat{H}$ so that the cyclic order of r-bridges around r is $B_{i_1}, B_{i_2}, \ldots, B_{i_k}$. □

We are interested in rooted label sequences that have minimum number of transitions. But to combine them later on, it is important to know what is the first and the last label in the rooted label sequence. This motivates the following definition. Let $\mathcal{Q}$ be a set of (linear) label sequences. We say that a sequence $Q \in \mathcal{Q}$ is *AB-minimal* for labels $A, B \in \mathcal{L}$, if

$$\tau(AQB) = \min\{\tau(ASB) \mid S \in \mathcal{Q}\}$$

where AQB is the sequence obtained from Q by adding labels A and B at the beginning and at the end of Q, respectively. A rooted label sequence Q of (H, r) is *AB-minimal* if Q is AB-minimal in $\mathcal{Q}(H)$. Minimal sequences are composed of minimal sequences as shows the following lemma. This allows us to restrict our attention to minimal sequences. The proof is not difficult and is included in the full paper.

Lemma 6. *Let $\mathcal{Q}$ be the set of all sequences that are concatenations of a sequence in $\mathcal{Q}_1$ and a sequence in $\mathcal{Q}_2$ (in this order). Then for $A, B \in \mathcal{L}$, every AB-minimal sequence Q in $\mathcal{Q}$ is a concatenation of an AC-minimal sequence in $\mathcal{Q}_1$ and a CB-minimal sequence in $\mathcal{Q}_2$ for some label $C \in \mathcal{L}$.*

Let (H, r) be a rooted leaf-labelled cactus. We can describe "optimal" embeddings of $\widehat{H}$ in the plane by a set of AB-minimal rooted label sequences of H, one for each pair of labels $A, B \in \mathcal{L}$. Let $\rho_H[A, B]$ be the minimum number of label transitions in an AB-minimal rooted label sequence of H. Note that the values of ρ_H differ by at most 2 since adding labels A and B to a sequence increases the number of label transitions by at most 2. Hence we can represent ρ_H by the minimum $\rho_H[A, B]$ over all labels A, B and by the individual differences from this minimum. Let n_H be the *minimum number of label transitions* in a rooted label sequence of H and let $p_H[A, B] = \rho_H[A, B] - n_H$ be the *type* of H. Note that $p_H[A, B] \in \{0, 1, 2\}$. The type p_H of H is viewed as a function $p_H : \mathcal{L} \times \mathcal{L} \to \{0, 1, 2\}$ and also as a number between 1 and $t \equiv 3^{l^2}$. Note that the number t of different types is a constant. We will show that cacti of the same type "behave" the same. We call the pair (p_H, n_H) the *descriptor* of H. For simplicity, we also call ρ_H the descriptor of H since it is easy to compute ρ_H from (p_H, n_H) and vice versa.

Note that the "unrooted" number of transitions $\tau(H)$ can be obtained from the descriptor of H as

$$\tau(H) = \min_{A \in \mathcal{L}} \rho_H[A, A].$$

It is time to consider how a descriptor of a rooted leaf-labelled cactus can be computed from descriptors of its subtrees. The first non-trivial case is when the root lies on a cycle. This case is easily dealt with using Lemmas 4 and 6 since minimal sequences of the subtrees attached to the cycle have fixed cyclic order. The following lemma is proven in the full paper.

Lemma 7. *Let (H, r) be a rooted leaf-labelled cactus such that r is a vertex of degree 2 in a cycle C of length k in H, and let B_v, $v \in V(C)$, be the union of v-bridges in H that do not contain C. Then the descriptor of H can be computed from descriptors of (B_v, v), $v \in V(C)$, in time $\mathcal{O}(l^3 k)$.*

Computing the descriptor of a leaf-labelled cactus rooted at a cut vertex turns out to be main issue. Let (H, r) be a rooted leaf-labelled cactus where r is a cut vertex of H and let $B_1, \ldots, B_k$ be the r-bridges in H. Let $b_H(i)$ be the number of r-bridges in H of type i, $i = 1, \ldots, t$. We view b_H as an integer vector in $\mathbb{Z}^t$ with $\sum_{i=1}^{t} b_H(i) = k$. A non-negative integer vector $b \in \mathbb{Z}^t$ is called a

bridge vector and $\mathrm{sum}(b) = \sum_{i=1}^{t} b(i)$ the *sum* of b. Note that there are at most $\mathcal{O}(k^{t+1})$ different non-negative integer vectors b in $\mathbb{Z}^t$ with the sum at most k.

Each bridge vector b describes a problem to be solved: How to order $k = \mathrm{sum}(b)$ bridges of types given by b around a vertex so that the number of label transitions on the boundaries between bridges is minimized. For fixed labels A, B, let $\mathcal{R}_{AB}(b)$ be the set sequences of $k+1$ labels $A_0, A_1, \ldots, A_k$ such that $A_0 = A$ and $A_k = B$. For an ordering of types in b, $P = p_1, p_2, \ldots, p_k$, and a sequence $R \in \mathcal{R}_{AB}(b)$, $R = L_0, \ldots, L_k$, let $m(P,R) = \sum_{i=1}^{k} p_i[L_{i-1}, L_i]$. Let $m_b[A,B]$ be the the minimum $m(P,R)$ taken over all orderings P of b and all sequences $R \in R_{AB}(b)$. This minimum depends only on the types of the bridges, not on the minimum number of label transitions of the bridges.

The fact that computation of m_b is a solution to the posed problem and gives a way how to compute the descriptor of a leaf-labelled cactus rooted at a cutvertex is not difficult to see. The precise relation of m_b and the descriptor is stated in the following lemma and the details of the proof are provided in the full paper.

Lemma 8. *Let (H,r) be a rooted leaf-labelled cactus and let $B_1, \ldots, B_k$ be the r-bridges in H. Then*

$$\rho_H[A,B] = m_{b_H}[A,B] + \sum_{i=1}^{k} n_{B_i}.$$

This gives rise to the following dynamic program. Given a non-zero bridge b vector, there are only t possibilities for the type p of the first bridge whose label sequence starts a minimal label sequence of H (the existence of such a bridge follows from Lemma 5). By deleting the type p from b, we obtain a smaller bridge vector b_p. The array m_{b_p} is computed recursively and then combined with p to obtain m_b. However, using this approach would yield a polynomial-time algorithm that is not fixed parameter tractable (since there are $\Theta(n^t)$ bridge vectors of sum at most n). In the next section, we sidestep this problem and present a linear-time algorithm for computing m_b.

Let us describe an algorithm for Min-Trans that, as we show in the next section, yields Theorem 1. We assume that the input graph has at least 3 labels to avoid trivialities.

Algorithm 1

Input: a labelled graph G
Output: minimal number of transitions $\tau(G)$

1 Construct the leaf-labelled cactus H that is equivalent to G (Lemma 1).
2 Root H at an arbitrary unlabelled vertex r.
3 $\rho_H \longleftarrow$ `Descriptor`(H,r).
4 Compute $\tau(G)$ from ρ_H.
5 **return** $\tau(G)$.

Function. Descriptor(H, r)

Input: a rooted leaf-labelled cactus (H, r)
Output: the descriptor of H, ρ_H

```
1  switch according to the role of r do
2      case r is a leaf and its neighbor u is labelled
3          Note that F has just two vertices r and u.
4          The descriptor ρ_H corresponds to the single-label sequence λ(u).
5      case r is a leaf and its neighbor u is not labelled
6          ρ_H ⟵ Descriptor(H − r, u).
7      case r is in a cycle C and is of degree two
8          foreach v ∈ V(C) \ {r} do
9              Let B_v be the union of all v-bridges that do not contain C.
10             ρ_{B_v} ⟵ Descriptor(B_v, v).
11         Use Lemma 7 to compute ρ_H from ρ_{B_v}.
12     case r is a cut vertex
13         foreach r-bridge B_i do
14             ρ_{B_i} ⟵ Descriptor(B_i, r).
15         Construct the bridge vector b from ρ_{B_i}.
16         Compute m_b.
17         Use Lemma 8 to compute ρ_H from m_b and ρ_{B_i}.
18 return ρ_H.
```

Note that throughout Algorithm 1, each vertex of H appears as a root in `Descriptor` at most twice; once as a cut vertex and once either as a leaf or on a cycle. Therefore, each of the cases can happen at most n times, $n = |V(H)|$, and the basic recursion runs in linear time. By Lemma 7, the case when the root is in a cycle takes time $\mathcal{O}(l^3 n)$ since the sum of lengths of all cycles is bounded by n. If we can compute m_b for a bridge vector b in constant time, then Algorithm 1 runs in linear time. This is the goal of the next section.

3 Dealing with Bridge Vectors

In the previous section we have sketched an algorithm for computing the minimum number of label transitions in a planar multigraph. In this section we outline an algorithm for computing m_b of a bridge vector b in constant time (Lemma 11), the last ingredient for the proof of Theorem 1. We start by observing that m_b is bounded independently of the bridge vector b. Let us recall that $t = 3^{l^2}$.

Lemma 9. *Let b be a bridge vector. Then for $A, B \in \mathcal{L}$,*

$$m_b[A, B] \leq 2t + 2.$$

Proof. For every type $p = 1, \ldots, t$, there are labels A_p, B_p such that $p[A_p, B_p] = 0$. By Lemma 2, $p[B_p, A_p] = 0$ as well. Let $k = \mathrm{sum}(b)$ and let $P = p_1, p_2, \ldots, p_k$ be the sequence of types in b in the increasing order. Let R be the sequence of labels $L_0, L_1, \ldots, L_k$ be the sequence of labels such that $L_0 = A$, $L_k = B$, and for $i = 1, \ldots, k-1$, $L_i = A_{p_i}$ if i is odd and $L_i = B_{p_i}$ if i is even.

Note that for $i = 2, \ldots, k-1$, if $p_{i-1} = p_i$, then either $p_i[L_{i-1}, L_i] = p_i[A_{p_i}, B_{p_i}]$ or $p_i[L_{i-1}, L_i] = p_i[B_{p_i}, A_{p_i}]$ and so $p_i[L_{i-1}, L_i] = 0$. Since $p_i[A', B'] \leq 2$ for all labels A', B' and there are at most $t-1$ transitions between different types,

$$m(P, R) = \sum_{i=1}^{k} p_i[L_{i-1}, L_i] \leq 2(t-1) + 4.$$

Thus, $m_b[A, B] \leq m(P, R) \leq 2t + 2$. □

Let K be the complete edge-colored and edge-weighted multigraph on vertex set $\mathcal{L}$ where two vertices $A, B \in \mathcal{L}$ are joined by t edges such that pth edge is colored by p and given weight $p[A, B]$. Note that there are t loops at every vertex of K. For a walk W in K, the *weight* $w(W)$ of W is the sum of weights of edges in W. Let $P = p_1, \ldots, p_k$ be an ordering of types in a bridge vector b and $R = L_0, \ldots, L_k$ be a sequence of labels. The sequences P, R generate a walk W in K of length k where in the ith step the edge $L_{i-1}L_i$ with color p_i is used. The weight of W is $m(P, R)$. The walk W uses $b(p)$ edges of color p. The converse statement also holds: A walk W that uses $b(p)$ edges of color p gives an ordering P of types in b and a label sequence R such that $m(P, R) = w(W)$. This gives the following lemma.

Lemma 10. *Let b be a bridge vector, A, B labels, and w an integer. There is an AB-walk W of weight w in K such that W uses $b(p)$ edges of color p if and only if there is an ordering P of types in b and a label sequence $R = A, \ldots, B$ such that $m(P, R) = w$.*

In order to show that m_b can be computed in constant time, a bridge vector b' equivalent to b such that the sum of b' is bounded by a constant is found. The array $m_{b'}$ (and thus also m_b) is then computed in constant time by brute force. The main idea arise from the correspondence between orderings of bridge vectors and walks in the constant-sized graph K. Given a walk W in K with enough edges, it is shown that one can find a set of disjoint cycles in W that contain even number of edges of each color. The removal of the cycles then yields a smaller version of the problem that can be easily tested for extension to the original problem. This will be elaborated in detail in the full paper. We obtain the following lemma showing that, for a bridge vector b, the array m_b can be computed in constant time.

Lemma 11. *Let b be a bridge vector. Then for labels A, B, $m_b[A, B]$ can be computed in time $\mathcal{O}(l^2 t (l^2 t)^{4l^2 t})$.*

It is likely that a fixed-parameter tractable solution can also be described by the use of min-max algebra for shortest paths, see [3] and [1], [6].

Finally, let us conclude the section by sketching the proof of Theorem 1.

Proof (of Theorem 1). The proof of the correctness of Algorithm 1 consists of several lemmas. Lemma 1 shows that any input graph can be transformed to an equivalent leaf-labelled cactus. Lemmas 3, 4, and 5 justify our recursive approach for computing the descriptors of rooted cacti.

The linearity of Algorithm 1 was established at the end of Sec. 2 provided that we can compute m_b in constant time. The cornerstone of the argument was that Lemma 8 allows us to deal with bridge vectors instead to collection of bridges. By Lemma 11, we can compute m_b for a bridge vector in time $\mathcal{O}(l^4t(l^2t)^{4l^t})$ (applying the lemma for each pair of labels). This is the slowest part of the algorithm and, since there are at most n cut-vertices in the graph, the algorithm runs in time $\mathcal{O}(l^4t(l^2t)^{4l^t}n)$.

4 NP-Completeness

When the number of labels is not bounded, Min-Trans becomes harder. In this section we give a proof of Theorem 2 by providing a polynomial-time reduction from the Hamiltonian Cycle Problem (see [8]).

Proof (of Theorem 2). An embedding of G with small number of transitions is a certificate for Min-Trans which asserts that Min-Trans is in NP. To show that Min-Trans is NP-complete, we give a polynomial-time reduction from the Hamiltonian Cycle Problem.

Let G be a graph of order n and let H be the graph whose vertex set is $V(H) = \{v_1\} \cup V(G) \cup (E(G) \times \{0,1\})$. We connect v_1 to each vertex in $V(G)$ and for each edge $uv \in E(G)$ we join $(uv, 0)$ with u and $(uv, 1)$ with v. Only the leaves of H are labelled. Vertex (e, i) is labelled e. Thus, the number of labels is $|E(G)|$ and each label occurs precisely twice. It is immediate that H can be constructed in polynomial time in $|V(G)|$.

We ask if the number of transitions $\tau(H)$ is smaller or equal to k for

$$k = \sum_{v \in V(G)} (\deg(v) - 1) = 2|E(G)| - |V(G)|.$$

In the affirmative, there is a planar embedding Π of $\widehat{H}$ with $\tau(\Pi) \leq k$. The local rotation around v_1 gives a cyclic order π of vertices of G. Root H at v_1. By Lemma 5, every label sequence of H is a concatenations of seqeuences Q_v, $v \in V(G)$, such that Q_v consists of labels on leaves of H attached to v. Since labels in Q_v are the edges adjacent to v, they are different and thus $\tau(Q_v) = \deg(v) - 1$. Hence,

$$\tau(H) \geq \sum_{v \in V(G)} (\deg(v) - 1) = k. \tag{1}$$

To get an equality here, we need that there are no more label transitions between neighboring sequences Q_v.

Let $e_1(v)$ and $e_2(v)$ be the first and the last label in Q_v. We have an equality in (1) if and only if for every two consecutive vertices u, v in π, $e_1(u) = e_2(v)$. This gives a cyclic sequence C of n edges that visits every vertex precisely once. Hence C is a hamiltonian cycle in G.

On the other hand, a hamiltonian cycle C in G gives a cyclic order on vertices of G. This and the cyclic order of the edges of C give a construction of an embedding of $\widehat{H}$ with $\tau(H) = k$. □

References

1. Butkovič, P., Cuninghame-Green, R.A.: On matrix powers in max-algebra. Linear Algebra Appl. 421, 370–381 (2007)
2. Chiba, N., Nishizeki, T., Abe, S., Ozawa, T.: A linear algorithm for embedding planar graphs using PQ-trees. J. Comput. Syst. Sci. 30, 54–76 (1985)
3. Cuninghame-Green, R.: Minimax Algebra, vol. 166. Springer, Heidelberg (1979)
4. Decker, R.W., Glover, H.H., Huneke, J.P.: The genus of the 2-amalgamations of graphs. Journal of Graph Theory 5(1), 95–102 (1981)
5. Decker, R.W., Glover, H.H., Huneke, J.P.: Computing the genus of the 2-amalgamations of graphs. Combinatorica 5, 271–282 (1985)
6. Gavalec, M.: Linear matrix period in max-plus algebra. Linear Algebra and its Applications 307(1-3), 167–182 (2000)
7. Hopcroft, J., Tarjan, R.: Efficient planarity testing. J. ACM 21, 549–568 (1974)
8. Karp, R.: Reducibility among combinatorial problems. In: Miller, R., Thatcher, J. (eds.) Complexity of Computer Computations, pp. 85–103. Plenum Press (1972)
9. Kuratowski, K.: Sur le problème des courbes gauches en topologie. Fund. Math. 15, 271–283 (1930)
10. Mohar, B., Thomassen, C.: Graphs on Surfaces. Johns Hopkins Univ. Press, Baltimore (2001)
11. Mohar, B., Škoda, P.: Obstructions for xy-alternating embeddings of graphs in surfaces (in preparation)
12. Shih, W.K., Hsu, W.L.: A new planarity test. Theor. Comput. Sci. 223, 179–191 (1999)

A New View on Rural Postman Based on Eulerian Extension and Matching

Manuel Sorge*, René van Bevern**, Rolf Niedermeier, and Mathias Weller***

Institut für Softwaretechnik und Theoretische Informatik, TU Berlin
{manuel.sorge,rene.vanbevern,rolf.niedermeier,mathias.weller}@tu-berlin.de

Abstract. We provide a new characterization of the NP-hard arc routing problem Rural Postman in terms of a constrained variant of minimum-weight perfect matching on bipartite graphs. To this end, we employ a parameterized equivalence between Rural Postman and Eulerian Extension, a natural arc addition problem in directed multigraphs. We indicate the NP-hardness of the introduced matching problem. In particular, we use it to make some partial progress towards answering the open question about the parameterized complexity of Rural Postman with respect to the number of weakly connected components in the graph induced by the required arcs. This is a more than thirty years open and long-time neglected question with significant practical relevance.

1 Introduction

The Rural Postman (RP) problem with its special case, the Chinese Postman problem, is a famous arc routing problem in combinatorial optimization. Given a directed, arc-weighted graph G and a subset R of its arcs (called "required arcs"), the task is to find a minimum-cost closed walk in G that visits all arcs of R. The manifold practical applications of RP include snow plowing, garbage collection, and mail delivery [1, 2, 3, 6, 7, 15]. Recently, it has been observed that RP is closely related (more precisely, "parameterized equivalent") to the arc addition problem Eulerian Extension (EE). Here, given a directed and arc-weighted multigraph G, the task is to find a minimum-weight set of arcs to add to G such that the resulting multigraph is Eulerian [10, 4]. RP and EE are NP-hard. In fact, their mentioned parameterized equivalence means that many algorithmic and complexity-theoretic results for one of them transfer to the other. In particular, this gives a new view on RP, perhaps leading to novel approaches to attack its computational hardness. A key issue in both problems is to determine the influence of the number c of connected components[1] on each problem's computational complexity [4, 9, 11, 13, 14]. Indeed, Frederickson [9] observed that RP (and, thus, EE) is polynomial-time solvable when c is constant. However, this left

* Partially supported by the DFG, project AREG, NI 369/9 and project PABI, NI 369/7.
** Supported by the DFG, project AREG, NI 369/9.
*** Supported by the DFG, project DARE, NI 369/11.

[1] More precisely, c refers to the number of weakly connected components in the input for EE and the number of weakly connected components in the graph induced by the required arcs for RP.

C.S. Iliopoulos and W.F. Smyth (Eds.): IWOCA 2011, LNCS 7056, pp. 310–323, 2011.
© Springer-Verlag Berlin Heidelberg 2011

open whether c influences the degree of the polynomial or whether RP can be solved in $f(c) \cdot n^{O(1)}$ time for some exponential function f. In other words, it remained open whether RP and EE are fixed-parameter tractable[2] with respect to the parameter c [4]. We remark that this parameter is presumably small in a number of applications [4, 7, 9], strongly motivating to attack this seemingly hard open question.

Our Results. In this work, we contribute new insights concerning the seemingly hard open question whether RP (and EE) is fixed-parameter tractable with respect to the parameter "number of components". To this end, our main contribution is a new characterization of RP in terms of a constrained variant of minimum-weight perfect matching on bipartite graphs. Referring to this problem as Conjoining Bipartite Matching (CBM), we show its NP-hardness and a parameterized equivalence to RP and EE. Moreover, we show that CBM is fixed-parameter tractable[3] when restricted to bipartite graphs where one partition set has maximum vertex degree two. This implies corresponding fixed-parameter tractability results for relevant special cases of RP and EE which would probably have been harder to formulate and to detect using the definitions of these problems. Indeed, we hope that CBM might help to finally answer the puzzling open question concerning the parameterized complexity of RP with respect to the number of components. In this paper we consider decision problems. However, our results easily transfer to the corresponding optimization problems.

For the sake of notational convenience and justified by the known parameterized equivalence [4], most of our results and proofs refer to EE instead of RP. Due to space constraints, most proofs are deferred to a full version of the paper.

2 Preliminaries and Preparations

Consider a directed multigraph $G = (V, A)$, comprising the vertex set V and the arc multiset A. For notational convenience, we define a *component graph* $\mathbb{C}_G$ as a clique whose vertices one-to-one correspond to the weakly connected components of G. Since we never consider strongly connected components, we omit the adverb "weakly". A *walk* W in G is a sequence of arcs in G such that each arc ends in the same vertex as the next arc starts in. We use $V(W)$ and $A(W)$ to refer to the set of vertices in which arcs of W start or end, and the multiset of arcs of W, respectively. The first vertex in the sequence is called the *initial* vertex of the walk and the last vertex in the sequence is called the *terminal* vertex of the walk. A walk W in G such that $A(W)$ is a submultiset of the multiset $A(G)$ is called a *trail* of G. A trail T in G such that every vertex in G has at most two incident arcs in $A(T)$ is called a *cycle* if the initial and terminal vertices of T are equal, and *path* otherwise. If G is clear from the context, we omit it. We use $\text{balance}(v) := \text{indeg}(v) - \text{outdeg}(v)$ to denote the *balance* of a vertex v in G and I_G^+ and I_G^- to denote the set of all vertices v in G with $\text{balance}(v) > 0$ and $\text{balance}(v) < 0$, respectively. A vertex v is *balanced* if $\text{balance}(v) = 0$.

[2] See Section 2 and the literature [5, 8, 12] for more on parameterized complexity analysis.

[3] The corresponding parameter "join set size" measures the instance's distance from triviality and translates to the parameter "number of components" in equivalent instances of EE and RP.

Our results are in the context of parameterized complexity [5, 8, 12]. A *parameterized problem* $L \subseteq \Sigma^* \times \mathbb{N}$ is called *fixed-parameter tractable* (*FPT*) with respect to a parameter k if $(x,k) \in L$ is decidable in $f(k) \cdot |x|^{O(1)}$ time, where f is a computable function only depending on k.

We consider two types of parameterized reductions between problems: A *polynomial-parameter polynomial-time many-one reduction* ($\leq_{\mathrm{m}}^{\mathrm{PPP}}$-reduction) from a parameterized problem L to a parameterized problem L' is a polynomial-time computable function g such that $(x,k) \in L \Leftrightarrow (x',k') \in L'$, with $(x',k') := g(x,k)$, and $k' \leq p(k)$, where p is a polynomial only depending on k. If such a reduction exists, we write $L \leq_{\mathrm{m}}^{\mathrm{PPP}} L'$. A *parameterized Turing reduction* ($\leq_{\mathrm{T}}^{\mathrm{FPT}}$-reduction) from a parameterized problem L to a parameterized problem L' is an algorithm that decides $(x,k) \in L$ in $f(k) \cdot |x|^{O(1)}$ time, where queries of the form $(x', g(k)) \in L'$ are assumed to be decidable in O(1) time and f, g are functions solely depending on k. If such a reduction exists, we write $L \leq_{\mathrm{T}}^{\mathrm{FPT}} L'$. If $L \leq_{\mathrm{T}}^{\mathrm{FPT}} L'$ and $L' \leq_{\mathrm{T}}^{\mathrm{FPT}} L$, then we say that L and L' are $\leq_{\mathrm{T}}^{\mathrm{FPT}}$-equivalent. Note that every $\leq_{\mathrm{m}}^{\mathrm{PPP}}$-reduction is a $\leq_{\mathrm{T}}^{\mathrm{FPT}}$-reduction. Also, if $L' \in \mathrm{FPT}$ and $L \leq_{\mathrm{T}}^{\mathrm{FPT}} L'$, then $L \in \mathrm{FPT}$.

In this work, we consider the problem of making a given directed multigraph Eulerian by adding arcs. A directed multigraph G is *Eulerian* if it is connected and each vertex is balanced. An *Eulerian extension* E for $G = (V, A)$ is a multiset over $V \times V$ such that $G' = (V, A \cup E)$ is Eulerian.

Eulerian Extension (EE)
Input: A directed multigraph $G = (V, A)$, an integer $\omega_{\max}$, and a weight function $\omega \colon V \times V \to [0, \omega_{\max}] \cup \{\infty\}$.
Question: Is there an Eulerian extension E of G whose weight is at most $\omega_{\max}$?

In the context of EE we speak of *allowed* arcs $a \in V \times V$, if $\omega(a) < \infty$.

2.1 Preprocessing Routines

A polynomial-time preprocessing routine for EE introduced by Dorn et al. [4] ensures that the balance of every vertex is in $\{-1, 0, 1\}$. This simplifies the problem and helps in constructions later on. Dorn et al. [4] showed that the corresponding transformation can be computed in $O(n(n+m))$ time. In the following, we assume that all input instances of EE have been transformed thusly, and hence, we assume that the following observation holds.

Observation 1. *Let v be a vertex in a pre-processed instance of* EE. *Then,* $\mathrm{balance}(v) \in \{-1, 0, 1\}$.

We use a second preprocessing routine to make further observations about trails in Eulerian extensions. This preprocessing is a variant of the algorithm used by Dorn et al. [4] to remove isolated vertices from the input graph. Basically, it replaces the weight of a vertex pair by the weight of the "lightest" path in the graph $(V, V \times V)$ with respect to ω. Note that the resulting weight function respects the triangle inequality. Dorn et al. [4] showed that this transformation can be computed in $O(n^3)$ time. In the following, we assume all input instances of EE to have gone through this transformation, and hence, we assume that the following holds.

Observation 2. *Let ω be a weight-function of a pre-processed instance of* EE. *Then, ω respects the triangle inequality, that is, for each x, y, z, it holds that $\omega(x, z) \leq \omega(x, y) + \omega(y, z)$.*

In the subsequent sections, we use this preprocessing in parameterized algorithms and reductions. To this end, note that both transformations are parameter-preserving, that is, they do not change the number of connected components.

The presented transformations lead to useful observations regarding trails in Eulerian extensions. For instance, we often need the following fact.

Observation 3. *For any Eulerian extension E of G, there is an Eulerian extension E' of at most the same weight such that any path p and any cycle c in E' does not visit a connected component of G twice, except for the initial and terminal vertex of p and c.*

2.2 Advice

Since Eulerian extensions have to balance every vertex, they contain paths starting in vertices with positive balance and ending in vertices with negative balance. These paths together with cycles have to connect all connected components of the input graph. In order to reduce the complexity of the problem, we use advice as additional information on the structure of optimal Eulerian extensions. Advice consists of hints which specify that there must be a path or cycle in an Eulerian extension that visits connected components in a distinct order. Hints however do not specify exactly which vertices these paths or cycles visit. For an example of advice, see Figure 1a.

Formally, a *hint* for a directed multigraph $G = (V, A)$ is an undirected path or cycle t of length at least one in the component graph $\mathbb{C}_G$ together with a flag determining whether t is a cycle or a path.[4] Depending on this flag, we call the hints *cycle hints* and *path hints*, respectively. We say that a set of hints H is an *advice* for the graph G if the hints are edge-disjoint.[5] For a trail t in G, $\mathbb{C}_G(t)$ is the trail in $\mathbb{C}_G$ that is obtained by making t undirected and, for every connected component C of G, substituting every maximum length subtrail t' of t with $V(t') \subseteq C$ by the vertex in $\mathbb{C}_G$ corresponding to C. We say that a path p in the graph $(V, V \times V)$ *realizes* a path hint h if $\mathbb{C}_G(p) = h$ and the initial vertex of p has positive balance and the terminal vertex has negative balance in G. We say that a cycle c in the graph $(V, V \times V)$ realizes a cycle hint h if $\mathbb{C}_G(c) = h$. We say that an Eulerian extension E *heeds the advice* H if it can be decomposed into a set of paths and cycles that realize all hints in H.

A topic in this work is how having an advice helps in solving an instance of Eulerian extension. In order to discuss this, we introduce the following version of EE.

Eulerian Extension with Advice (EEA)

Input: A directed multigraph $G = (V, A)$, an integer $\omega_{\max}$, a weight function $\omega\colon V \times V \to [0, \omega_{\max}] \cup \{\infty\}$, and advice H.

Question: Is there an Eulerian extension E of G that is of weight at most $\omega_{\max}$ and heeds the advice H?

[4] The flag is necessary because a hint to a path in $\mathbb{C}_G$ may correspond to a cycle in G.

[5] Note that there is a difference between advice in our sense and the notion of advice in computational complexity theory. There, an advice applies to every instance of a specific length.

We will see that the hard part of computing an Eulerian extension that heeds a given advice H is to choose initial and terminal vertices for path hints in H. In fact, it is possible to compute optimal realizations for all cycle hints in any given advice in $O(n^3)$ time.

Observation 4 ([16]). *Let $(G, \omega_{max}, \omega, H)$ be an instance of* EEA. *In $O(n^3)$ time we can compute an equivalent instance $(G', \omega_{max}, \omega, H')$ such that H' does not contain a cycle hint. Furthermore, the number of components at most decreases.*

In this regard we note that we can compute an optimal realization of a path hint for given endpoints in the corresponding directed multigraph. This is possible in quadratic time, mainly using Observation 2, forbidding arcs contained in one connected component, and Dijkstra's algorithm.

Observation 5. *Let $(G, \omega_{max}, \omega, H)$ be an instance of* EE*, let $h \in H$ be a path hint and let C_i, C_t be the connected components of G that correspond to the endpoints of h. Furthermore, let $(u, v) \in (C_i \times C_t) \cap (I_G^+ \times I_G^-)$. Then, we can compute a minimum-weight realization of h with initial vertex u and terminal vertex v in $O(n^2)$ time.*

Since we want to derive Eulerian extensions from an advice and every Eulerian extension for a graph connects all of the graphs connected components, we are mainly interested in "connecting" advice. We say that an advice for a directed multigraph G is *connecting*, if its hints connect all vertices in $\mathbb{C}_G$. Furthermore, if there is no connecting advice H' with $H' \subset H$, then H is called *minimal connecting* advice. We consider the following restricted version of EEA that allows only minimal connecting advice (note that, by Observation 4, we can assume the given advice to be cycle-free).

Eulerian Extension with Cycle-free Minimal Connecting Advice (EE∅CA)
Input: A directed multigraph $G = (V, A)$, an integer $\omega_{\max}$, a weight function $\omega\colon V \times V \to [0, \omega_{\max}] \cup \{\infty\}$, and minimal connecting cycle-free advice H.
Question: Is there an Eulerian extension E of G with weight at most $\omega_{\max}$ and heeding the advice H?

We can show that each minimal connecting cycle-free advice can be obtained from a forest in $\mathbb{C}_G$. Enumerating these forests allows us to generate all such advices for a given graph G with c connected components in $f(c) \cdot |G|^{O(1)}$ time, where f is some function only depending on c. Deferring the presentation of details to a long version of this work, we state that EE is parameterized Turing reducible to EE∅CA [16].

Lemma 1. EE *is $\leq_{\mathrm{T}}^{\mathrm{FPT}}$-reducible to* EE∅CA *in $16^{c \log(c)}|G|^{O(1)}$ time.*

3 Eulerian Extension and Conjoining Bipartite Matching

This section shows that Rural Postman (RP) is parameterized equivalent to a matching problem. By the parameterized equivalence of RP and Eulerian Extension (EE) given by Dorn et al. [4], we may concentrate on the equivalence of EE and matching instead.

First we introduce a variant of perfect bipartite matching. Let G be a bipartite graph, M be a matching of the vertices in G, and let P be a vertex partition with the cells $C_1, \ldots, C_k$. We call an unordered pair $\{i, j\}$ of integers $1 \leq i < j \leq k$ a *join*

and a set J of such pairs a *join set* with respect to G and P. We say that a join $\{i, j\} \in J$ is *satisfied* by the matching M of G if there is at least one edge $e \in M$ with $e \cap C_i \neq \emptyset$ and $e \cap C_j \neq \emptyset$. We say that a matching M of G is *J-conjoining* with respect to a join set J if all joins in J are satisfied by M. If the join set is clear from the context, we simply say that M is conjoining.

Conjoining Bipartite Matching (CBM)
Input: A bipartite graph $G = (V_1 \uplus V_2, E)$, an integer $\omega_{\max}$, a weight function $\omega\colon E \to [0, \omega_{\max}]$, a partition $P = \{C_1, \ldots, C_k\}$ of the vertices in G, and a join set J.
Question: Is there a matching M of the vertices of G such that M is perfect, M is conjoining and M has weight at most $\omega_{\max}$?

CBM can be interpreted as a job assignment problem with additional constraints: an assignment of workers to tasks is sought such that each worker is busy and each task is being processed. Furthermore, every worker must be qualified for his or her assigned task. Both the workers and the tasks are grouped and the additional constraints are of the form "At least one worker from group A must be assigned a task in group B". An assignment that satisfies such additional constraints may be favorable in settings where the workers are assigned to projects and the projects demand at least one worker with additional qualifications.

Over the course of the following subsections, we prove the following theorem.

Theorem 1. Conjoining Bipartite Matching *and* Eulerian Extension *are* $\leq_{\mathrm{T}}^{\mathrm{FPT}}$*-equivalent with respect to the parameters "join set size" and "number of connected components in the input graph."*

The proof of Theorem 1 consists of four reductions, one of which is a parameterized Turing reduction. The other three reductions are polynomial-time polynomial-parameter many-one reductions.

It is easy to see that the equivalence of EE and RP given by Dorn et al. [4] also holds for the parameters "number of components" and "number of components in the graph induced by the required arcs." Thus, we obtain the following from Theorem 1.

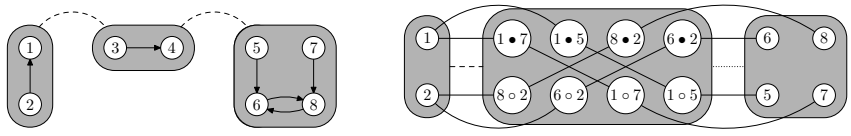
Theorem 2. Conjoining Bipartite Matching *and* Rural Postman *are* $\leq_{\mathrm{T}}^{\mathrm{FPT}}$*-equivalent with respect to the parameters "number of components in the graph induced by the required arcs" and "join set size."*

3.1 From Eulerian Extension to Matching

In this section we sketch a reduction from EE∅CA to CBM. By Lemma 1 this reduction leads to the following theorem.

Theorem 3. Eulerian Extension *is* $\leq_{\mathrm{m}}^{\mathrm{PPP}}$*-reducible to* Conjoining Bipartite Matching *with respect to the parameters "number of components" and "join set size."*

Outline of the Reduction. The basic idea of our reduction is to use vertices of positive balance and negative balance in an instance of EE∅CA as the two cells of the graph bipartition in a designated instance of CBM. Edges between vertices in the new instances



(a) EE∅CA instance (b) Long-hint gadget in CBM instance

Fig. 1. Example of the long-hint gadget. In (a) an EE∅A-instance is shown, consisting of a graph with three connected components and an advice that contains a single path hint h (dashed lines). In (b) a part of an instance of CBM is shown, comprising the cells that correspond to the initial and terminal vertices of h and a gadget to model h. The gadget consists of new vertices put into a new cell which is connected by two joins (dashed and dotted lines) to the cells corresponding to the initial and terminal vertices of h.

represent shortest paths between these vertices that consist of allowed extension arcs in the original instance. Every connected component in the original instance is represented by a cell in the partition in the matching instance and hints are basically modeled by joins.

Description of the Reduction. For the description of the reduction, we need the following definition.

Definition 1. *Let $C_1, \ldots, C_c$ be the connected components of a directed multigraph G, and let H be a cycle-free advice for G. For every $h \in H$ define* connect$(h) := \{i, j\}$, *where C_i, C_j are the components corresponding to the initial and terminal vertices of h.*

First, consider an EE∅CA-instance $(G, \omega_{\max}, \omega, H)$ such that H is a cycle-free minimal connecting advice that contains only hints of length one. We will deal with longer hints later. We create an instance I_{CBM} of CBM by first defining $B_0 = (I_G^+ \uplus I_G^-, E_0)$ as a bipartite graph. Here, the set E_0 consists of all edges $\{u, v\}$ such that $u \in I_G^+$, $v \in I_G^-$, and $\omega(u, v) < \infty$.[6] Second, we derive a vertex partition $\{V_1', \ldots, V_c'\}$ of B_0 by intersecting the connected components of G with $(I_G^+ \uplus I_G^-)$. The vertex partition obviously models the connected components in the input graph, and the need for connecting them according to the advice H is modeled by an appropriate join-set J_0, defined as $\{\text{connect}(h) : h \in H\}$. Finally, we make sure that matchings also correspond to Eulerian extensions weightwise, by defining the weight function $\omega'(\{u, v\})$ for every $u \in I_G^+, v \in I_G^-$ as $\omega(u, v)$ with $\omega'_{\max} = \omega_{\max}$.

By Observation 3 we may assume that every hint in H of length one is realized by a single arc. Since the advice connects all connected components, by the same observation, we may assume that all other trails in a valid Eulerian extension have length one. Finally, by Observation 1, we may assume that every vertex has at most one incident incoming or outgoing arc in the extension and, hence, we get an intuitive correspondence between conjoining matchings and Eulerian extensions.

To model hints of length at least two, we utilize gadgets similar to the one shown in Figure 1. The gadget comprises two vertices ($u \circ v$ and $u \bullet v$) for every pair (u, v) of vertices with one vertex in the component the hint starts and one in the component

[6] This serves the purpose of modeling the structure of "allowed" arcs in the matching instance.

the hint ends. The vertices $u \circ v$ and $u \bullet v$ are adjacent and each of these two vertices is connected with one vertex of the pair it represents. The edge $\{u \bullet v, u\}$ is weighted with the cost it takes to connect u, v with a path that realizes h. This cost is computed using Observation 5. The edges $\{u \bullet v, u \circ v\}$ and $\{u \circ v, v\}$ have weight 0. Intuitively these three edges in the gadget represent one concrete realization of h. If $u \circ v$ and $u \bullet v$ are matched, this means that this specific path does not occur in a designated Eulerian extension. However, by adding the vertices of the gadget as cell to the vertex partition and by extending the join set to the gadget, we enforce that there is at least one outgoing edge that is matched. If a perfect matching matches $u \circ v$ with u, then it also matches $u \bullet v$ with v and vice versa. This introduces an edge to the matching that has weight corresponding to a path that realizes h.

3.2 From Matching to Eulerian Extension with Advice

We reduce Conjoining Bipartite Matching to Eulerian Extension with Advice:

Theorem 4. Conjoining Bipartite Matching *is* $\leq_{\mathrm{m}}^{\mathrm{PPP}}$*-reducible to* Eulerian Extension with Advice *with respect to the parameters "join set size" and "connected components in the input graph".*

To reduce CBM to EEA we first observe that for every instance of CBM there is an equivalent instance such that every cell in the input vertex partition contains equal numbers of vertices from both cells of the graph bipartition. This observation enables us to model cells as connected components and vertices in the bipartite graph as unbalanced vertices in the designated instance of EEA.

Lemma 2. *For every instance of* CBM *there is an equivalent instance comprising the bipartite graph* $G = (V_1 \uplus V_2, E)$*, the vertex partition* $P = \{C_1, \ldots, C_k\}$ *and the join set* J*, such that*

(i) for every $1 \leq i \leq k$ *it holds that* $|V_1 \cap C_i| = |V_2 \cap C_i|$*, and*
(ii) the graph $(P, \{\{C_i, C_j\} : \{i, j\} \in J\})$ *is connected.*

This equivalent instance contains at most one cell more than the original instance.

Description of the Reduction. To reduce instances of CBM that conform to Lemma 2 to instances of EEA we use the simple idea of modeling every cell as connected component, vertices in V_1 as vertices with balance -1, vertices in V_2 as vertices with balance 1, and joins as hints.

Construction 1. Let the bipartite graph $B = (V_1 \uplus V_2, E)$, the weight function $\omega\colon E \to [0, \omega_{\max}]$, the vertex partition $P = \{C_1, \ldots, C_k\}$ and the join set J constitute an instance I_{CBM} of CBM that corresponds to Lemma 2. Let $v_1^1, v_1^2, \ldots, v_{n/2}^1, v_{n/2}^2$ be a sequence of all vertices chosen alternatingly from V_1 and V_2. Let the graph $G = (V, A) \coloneqq (V_1 \cup V_2, A_1 \cup A_2)$ where the arc sets A_1 and A_2 are as follows: $A_1 \coloneqq \{(v_i^1, v_i^2) : 1 \leq i \leq n/2\}$. For every $1 \leq j \leq k$ let $C_j = \{v_1, \ldots, v_{j_k}\}$, and let

$$A_2^j \coloneqq \{(v_i, v_{i+1}) : 1 \leq i \leq j_k - 1\} \cup \{(v_{j_k}, v_1\}$$

and define $A_2 := \bigcup_{j=1}^{k} A_2^j$. Define a new weight function ω' for every pair of vertices $(u, v) \in V \times V$ by

$$\omega'(u,v) := \begin{cases} \omega(\{u,v\}), & u \in V_2, v \in V_1, \{u,v\} \in E \\ \infty, & \text{otherwise.} \end{cases}$$

Finally, derive an advice H for G by adding a length-one hint h to H for every join $\{o, p\} \in J$ such that h consists of the edge that connects vertices in $\mathbb{C}_G$ that correspond to the connected components C_o, and C_p. The graph G, the weight function ω', the maximum weight $\omega_{\max}$ and the advice H constitute an instance of EEA.

Theorem 4 follows, since Construction 1 is a $\leq_{\mathrm{m}}^{\mathrm{PPP}}$-reduction.

3.3 Removing Advice

For Theorem 1, it remains to show, that the advice in an instance of EEA created by Construction 1 can be removed. That is, it remains to show the following theorem.

Theorem 5. Eulerian Extension with Advice *is* $\leq_{\mathrm{m}}^{\mathrm{PPP}}$*-reducible to* Eulerian Extension *with respect to the parameter "number of components in the input graph."*

The basic ideas for proving Theorem 5 are as follows. First, we remove every cycle-hint using Observation 4. We use the fact that every Eulerian extension has to connect all connected components of the input graph. Thus, for each hint h, we introduce a new connected component C_h. Let the components C_s, C_t correspond to the endpoints of hint h. To enforce that hint h is realized, we use the weight function to allow an arc from every vertex with balance 1 in C_s to a number of distinct vertices in C_h. This number is the number of vertices with balance -1 in C_t. That is, for every pair of unbalanced vertices in $C_s \times C_t$, we have an associated vertex in C_h. Then, for every inner vertex v on h, we copy C_h and connect it to the component corresponding to v. From one copy to another, using the weight function, we allow only arcs that start and end in vertices corresponding to the same pair of unbalanced vertices in $C_s \times C_t$. This enforces that every hint is realized and connects every component it visits. Using the weight function and Observation 5 we can ensure that the arcs corresponding to a realization of a hint have the weight of an optimal realization with the same endpoints. Using this construction, Theorem 5 can be proven which concludes the proof of Theorem 1.

4 Conjoining Bipartite Matching: Properties and Special Cases

This section investigates the properties of CBM introduced in Section 3. As discussed before, CBM might eventually help us derive a fixed-parameter algorithm for EE with respect to the parameter number of connected components. Section 4.1 first shows that also CBM is NP-complete. Section 4.2 then establishes tractability of the problem on restricted graph classes and translates this tractability result into the world of EE and RP.

4.1 NP-Hardness

NP-Hardness for Conjoining Bipartite Matching (CBM) does not follow from the parameterized equivalence to Eulerian Extension (EE) we gave in Section 3, since the reduction from EE we gave is a *parameterized* Turing reduction. To show that CBM is NP-hard, we polynomial-time many-one reduce from the well-known 3SAT, where a Boolean formula ϕ in 3-conjunctive normal form (3-CNF) is given and it is asked whether there is an assignment to the variables of ϕ that satisfies ϕ. Herein, a formula ϕ in 3-CNF is a conjunction of disjunctions of three literals each, where each literal is either x or $\neg x$ and x is a variable of ϕ. In the following, we represent each clause as three-element-set $\gamma \subseteq X \times \{+, -\}$, where $(x, +) \in \gamma$ means that x is contained in the clause represented by γ and $(x, -) \in \gamma$ means that $\neg x$ is contained in the clause represented by γ.

Construction 2. Let ϕ be a Boolean formula in 3-CNF with the variables $X := \{x_1, \ldots, x_n\}$ and the clauses $\gamma_1, \ldots, \gamma_m \subseteq X \times \{+, -\}$. We translate ϕ into an instance of CBM that is a yes-instance if and only if ϕ is satisfiable. To this end, for every variable x_i, introduce a cycle with $4m$ edges consisting of the vertex set $V_i := \{v_i^j : 1 \leq j \leq 4m\}$ and the edge set $E_i := \{e_i^k := \{v_i^k, v_i^{k+1}\} \subseteq V_i\} \cup \{e_i^{4m} := \{v_i^1, v_i^{4m}\}\}$. Let $G := (\bigcup_{i=1}^n V_i, \bigcup_{i=1}^n E_i)$, and let $\omega(e) := 0, e \in E_i$ for any $1 \leq i \leq n$, and define $\omega_{\max} := 1$. To construct an instance of CBM, it remains to find a suitable partition of the vertices of G and a join set.

Inductively define the vertex partition P_m of $V(G)$ and the join set J_m as follows: Let $J_0 = \emptyset$, and let $P_0 := \emptyset$. For every clause γ_j introduce the cell

$$C_j := \{v_i^{4j-1} : (x_i, +) \in \gamma_j \vee (x_i, -) \in \gamma_j\} \cup \{v_i^{4j-2} : (x_i, +) \in \gamma_j\} \cup \{v_i^{4j} : (x_i, -) \in \gamma_j\}.$$

Define $P_j := P_{j-1} \cup \{C_j\}$ and $J_j := J_{j-1} \cup \{\{0, j\}\}$.

Finally, define $C_0 := V(G) \setminus (\bigcup_{j=1}^m C_j)$. The graph G, the weight function ω, the vertex partition $P_m \cup \{C_0\}$ and the join set J_m constitute an instance of CBM.

Using this construction, we can prove the following theorem.

Theorem 6. CBM *is NP-complete, even in the unweighted case and when the input graph $G = (V \uplus W, E)$ has maximum degree two, and for every cell C_i in the given vertex partition of G it holds that $|C_i \cap V| = |C_i \cap W|$.*

Proof. CBM is contained in NP, because a perfect conjoining matching of weight at most $\omega_{\max}$ is a certificate for a yes-instance.

We prove that Construction 2 is a polynomial-time many-one reduction from 3SAT to CBM. Notice that in instances created by Construction 2 any matching has weight lower than $\omega_{\max}$ and, thus, the soundness of the reduction implies that CBM is hard even without the additional weight constraint. Also, since the cells in the instances of CBM are disjoint unions of edges, every cell in the partition P_m contains the same number of vertices from each cell of the graph bipartition.

It is easy to check that Construction 2 is polynomial-time computable. For the correctness we first need the following definition: For every variable $x_i \in X$ let

$$M_i^{\text{true}} := \{e_i^k \in E_i : k \text{ odd}\} \text{ and}$$
$$M_i^{\text{false}} := E_i \setminus M_i^{\text{true}} = \{e_i^k \in E_i : k \text{ even}\}.$$

Observe that all perfect matchings in G are of the form $\bigcup_{i=1}^{n} M_i^{\nu(x_i)}$, where ν is an assignment of truth values to variables in X. We show that the matching $\bigcup_{i=1}^{n} M_i^{\nu(x_i)}$ is a conjoining matching for G with respect to the join set J_m if and only if ν is satisfies ϕ. For this, it suffices to show that for every variable $x_i \in X$ it holds that

$$\{j : (x_i, +) \in \gamma_j\} = \{j : M_i^{\text{true}} \text{ satisfies the join } \{0, j\}\}, \text{ and} \tag{1}$$

$$\{j : (x_i, -) \in \gamma_j\} = \{j : M_i^{\text{false}} \text{ satisfies the join } \{0, j\}\}. \tag{2}$$

We only show that (1) holds; (2) can be proven analogously. Assume that $(x_i, +) \in \gamma_j$. By Construction 2, $v_i^{4j-2} \in C_j$, $v_i^{4j-3} \in C_0$ and thus, since

$$\{v_i^{4j-2}, v_i^{4j-3}\} = e^{4j-3} \in M_i^{\text{true}},$$

the matching M_i^{true} satisfies the join $\{0, j\}$. Now assume that $(x_i, +) \notin \gamma_j$, that is, either (1) $(x_i, \pm) \notin \gamma_j$ or (2) $(x_i, -) \in \gamma_j$. If $(x_i, \pm) \notin \gamma_j$, then V_i and C_j are disjoint and, thus, no matching in $G[V_i]$ can satisfy the join $\{0, j\}$. If $(x_i, -) \in \gamma_j$, then the only edges in E_i that can satisfy the join $\{0, j\}$ are e_i^{4j-2} and e_i^{4j}. Both edges are not in M_i^{true} and, thus, this matching cannot satisfy the join $\{0, j\}$. □

4.2 Tractability on Restricted Graph Classes

This section presents data reduction rules and employs them to sketch an algorithm for CBM on a restricted graph class, leading to the following theorem:

Theorem 7. Conjoining Bipartite Matching *can be solved in* $O(2^{j(j+1)}n + n^3)$ *time, where j is the size of the join set, provided that in the bipartite input graph $G = (V_1 \uplus V_2, E)$ each vertex in V_1 has maximum degree two.*

In this section, let $(G, \omega_{\max}, \omega, P = \{C_1, \ldots, C_c\}, J)$ be an instance of CBM, where in G is as in Theorem 7. The following lemma plays a central role in the proof of Theorem 7. It implies that, in a yes-instance, every component of G consists of an even-length cycle with a collection of pairwise vertex-disjoint paths incident to it.

Lemma 3. *If G has a perfect matching, then every connected component of G contains at most one cycle as subgraph.*

Proof. We show that if G contains a connected component that contains two cycles c_1, c_2 as subgraphs, then G does not have a perfect matching. First assume that c_1, c_2 are vertex-disjoint. Then, there is a path p from a vertex $v \in V(c_1)$ to a vertex $w \in V(c_2)$ such that p is vertex-disjoint from c_1 and c_2 except for v, w. It is clear that both $v, w \in V_2$ because they have degree three. Consider the vertices $V_1^{\text{cp}} := (V(c_1) \cup V(p) \cup V(c_2)) \cap V_1$ and the set $V_2^{\text{cp}} := (V(c_1) \cup V(p) \cup V(c_2)) \cap V_2$. The set V_2^{cp} is the set of neighbors of vertices in V_1^{cp}, because they have degree two and thus have neighbors only within p, c_1, and c_2. It is $|V_1^{\text{cp}}| = (|E(c_1)| + |E(p)| + |E(c_2)|)/2$ since neither of these paths and cycles overlap in a vertex in V_1. However, it is $|V_2^{\text{cp}}| = |V_1^{\text{cp}}| - 1$ because c_1 and p overlap in v and c_2 and p overlap in w. This is a violation of Hall's condition and thus G does not have a perfect matching.

The case where c_1 and c_2 share vertices can be proven analogously. (Observe that then there is a subpath of c_2 that is vertex-disjoint from c_1 and contains an even number of edges.) □

We now present four polynomial-time executable data reduction rules for CBM. The correctness of the first three rules is easy to verify, while the correctness of the fourth one is more technical and omitted. We note that all rules can be applied exhaustively in $O(n^3)$ time.

Reduction Rule 1 removes paths incident to the cycles of a graph G in a yes-instance. As a side-result, Reduction Rule 1 solves CBM in linear time on forests.

Reduction Rule 1. *If there is an edge* $\{v, w\} \in E(G)$ *such that* $\deg(v) = 1$*, then remove both* v *and* w *from* G*, and remove all joins* $\{i, j\}$ *from* J*, with* $v \in C_i, w \in C_j$*. Decrease* ω_{max} *by* $\omega(\{v, w\})$*.*

If exhaustively applying Reduction Rule 1 to G does not transform G such that each connected component is a cycle, which is checkable in linear time, then, by Lemma 3, G does not have a perfect matching and we can return "NO". Hence, in the following, assume that each connected component of G is a cycle. Reduction Rule 2 now deletes connected components that cannot satisfy joins.

Reduction Rule 2. *If there is a connected component* D *of* G *such that it contains no edge that could satisfy any join in* J*, then compute a minimum-weight perfect matching* M *in* $G[D]$*, remove* D *from* G *and decrease* ω_{max} *by* $\omega(M)$*.*

After exhaustively applying Reduction Rule 2, we may assume that each connected component of G contains an edge that could satisfy a join. We next present a data reduction rule that removes joins that are always satisfied. To this end, we need the following definition.

Definition 2. *For each connected component* D *(that is, each cycle) in* G*, denote by* $M_1(D)$ *a minimum-weight perfect matching of* D *with respect to* ω *and denote by* $M_2(D) := E(D) \setminus M_1(D)$ *the other perfect matching of* D*.*[7] *Furthermore, denote*

$$\sigma_1(D) := \{j \in J : \exists e \in M_1(D) : e \text{ satisfies } j\},$$
$$\sigma_2(D) := \{j \in J : \exists e \in M_2(D) : e \text{ satisfies } j\},$$

and the signature $\sigma(D)$ *of* D *as* $\{\sigma_1(D), \sigma_2(D)\}$*.*

Reduction Rule 3. *Let* D *be a connected component of* G*. If there is a join* $j \in \sigma_1(D) \cap \sigma_2(D)$*, then remove* j *from* J*.*

A final data reduction rule removes connected components that satisfy the same joins.

Reduction Rule 4. *Let* $S = \{D_1, \ldots, D_j\}$ *be a maximal set of connected components of* G *such that* $\sigma(D_1) = \ldots = \sigma(D_j)$ *and* $j \geq 2$*. Let* $M_1^* = \bigcup_{k=1}^{j} M_1(D_k)$*, let* $D_l \in S$ *such that* $\omega(M_2(D_l)) - \omega(M_1(D_l))$ *is minimum, and let* $M_1^{\sim} = M_1^* \setminus M_1(D_l)$*.*

[7] Note that in bipartite graphs every cycle is of even length.

(i) If the matching M_1^* is conjoining for the join set $\sigma_1(D_1) \cup \sigma_2(D_1)$, then remove each component in S from G, remove each join in $\sigma_1(D_1) \cup \sigma_2(D_1)$ from the join set J, and reduce ω_{max} by $\omega(M_1^*)$.

(ii) If the matching M_1^* is not conjoining for the join set $\sigma_1(D_1) \cup \sigma_2(D_1)$, then remove each component in $S \setminus \{D_l\}$ from G, remove any join in $\sigma_1(D_1)$ from the join set J, and reduce ω_{max} by $\omega(M_1^{\sim})$.

In either case, update the partition P accordingly.

Observation 6. *If Reduction Rule 4 is not applicable to G, then G contains at most one connected component for each of the $2^{|J|+1}$ possible signatures.*

Now, Theorem 7 follows: Exploiting Observation 6, a search-tree algorithm solving CBM can in $O(n)$ time choose a join $j \in J$ and choose one of the at most $2^{|J|+1}$ connected components of the graph that can satisfy j and match the component accordingly. Then, the algorithm can recurse on how the remaining $|J| - 1$ joins are satisfied.

Analyzing the pre-images that lead to tractable instances of CBM under the reductions we gave in Section 3, Theorem 7 can be translated to a tractability result for EE. A similar tractability result can also be shown for Rural Postman. Due to its length, we only state it for EE here.

Corollary 1. *Let the graph G and the weight function ω constitute an instance I_{EE} of* EE*. Let c be the number of connected components in G.*

(i) If every path or cycle in the set of allowed arcs w.r.t. ω has length at most one,

(ii) if G contains only vertices with balance between -1 and 1,

(iii) if every vertex in I_G^+ (every vertex in I_G^-) has only outgoing allowed arcs (incoming allowed arcs), and

(iv) if in every connected component C of G, either all vertices in $I_G^+ \cap C$ or in $I_G^- \cap C$ have at most two incident allowed arcs,

then it is decidable in $O(2^{c(c+\log(2c^4))}(n^4 + m))$ time whether I_{EE} is a yes-instance.

5 Conclusion

Clearly, the most important remaining open question is to determine whether Rural Postman is fixed-parameter tractable with respect to the number of connected components of the graph induced by the required arcs. This question also extends to the presumably harder undirected case. The newly introduced Conjoining Bipartite Matching (CBM) problem might also be useful in spotting new, computationally feasible special cases of Rural Postman and Eulerian Extension. The development of polynomial-time approximation algorithms for CBM or the investigation of other (structural) parameterizations for CBM seem worthwhile challenges as well. Finally, we remark that previous work [10, 4] also left open a number of interesting open problems referring to variants of Eulerian Extension. Due to the practical relevance of the considered problems, our work is also meant to further stimulate more research on these challenging combinatorial problems.

References

[1] Assad, A.A., Golden, B.L.: Arc routing methods and applications. In: Network Routing. Handbooks in Operations Research and Management Science, vol. 8, pp. 375–483. Elsevier B.V. (1995)

[2] Benavent, E., Corberán, A., Piñana, E., Plana, I., Sanchis, J.M.: New heuristic algorithms for the windy rural postman problem. Comput. Oper. Res. 32(12), 3111–3128 (2005)

[3] Cabral, E.A., Gendreau, M., Ghiani, G., Laporte, G.: Solving the hierarchical chinese postman problem as a rural postman problem. European J. Oper. Res. 155(1), 44–50 (2004)

[4] Dorn, F., Moser, H., Niedermeier, R., Weller, M.: Efficient Algorithms for Eulerian Extension. In: Thilikos, D.M. (ed.) WG 2010. LNCS, vol. 6410, pp. 100–111. Springer, Heidelberg (2010)

[5] Downey, R.G., Fellows, M.R.: Parameterized Complexity. Springer, Heidelberg (1999)

[6] Dror, M.: Arc Routing: Theory, Solutions, and Applications. Kluwer Academic Publishers (2000)

[7] Eiselt, H.A., Gendreau, M., Laporte, G.: Arc routing problems, part II: The rural postman problem. Oper. Res. 43(3), 399–414 (1995)

[8] Flum, J., Grohe, M.: Parameterized Complexity Theory. Springer, Heidelberg (2006)

[9] Frederickson, G.N.: Approximation algorithms for some postman problems. J. ACM 26(3), 538–554 (1979)

[10] Höhn, W., Jacobs, T., Megow, N.: On Eulerian extensions and their application to no-wait flowshop scheduling. J. Sched (to appear, 2011)

[11] Lenstra, J.K., Kan, A.H.G.R.: On general routing problems. Networks 6(3), 273–280 (1976)

[12] Niedermeier, R.: Invitation to Fixed-Parameter Algorithms. Oxford University Press (2006)

[13] Orloff, C.S.: A fundamental problem in vehicle routing. Networks 4(1), 35–64 (1974)

[14] Orloff, C.S.: On general routing problems: Comments. Networks 6(3), 281–284 (1976)

[15] Perrier, N., Langevin, A., Campbell, J.F.: A survey of models and algorithms for winter road maintenance. Part IV: Vehicle routing and fleet sizing for plowing and snow disposal. Comput. Oper. Res. 34(1), 258–294 (2007)

[16] Sorge, M.: On making directed graphs Eulerian. Diplomarbeit, Institut für Informatik, Friedrich-Schiller-Universität Jena (2011), Available electronically. arXiv:1101.4283 [cs.DM]

Hamilton Cycles in Restricted Rotator Graphs

Brett Stevens* and Aaron Williams

Carleton University, Canada
brett@math.carleton.ca,
haron@uvic.ca

Abstract. The rotator graph has vertices labeled by the permutations of n in one line notation, and there is an arc from u to v if a prefix of u's label can be rotated to obtain v's label. In other words, it is the directed Cayley graph whose generators are $\sigma_k := (1\ 2\ \cdots\ k)$ for $2 \leq k \leq n$ and these rotations are applied to the indices of a permutation. In a restricted rotator graph the allowable rotations are restricted from $k \in \{2, 3, \ldots, n\}$ to $k \in G$ for some smaller (finite) set $G \subseteq \{2, 3, \ldots, n\}$. We construct Hamilton cycles for $G = \{n-1, n\}$ and $G = \{2, 3, n\}$, and provide efficient iterative algorithms for generating them. Our results start with a Hamilton cycle in the rotator graph due to Corbett (IEEE Transactions on Parallel and Distributed Systems 3 (1992) 622–626) and are constructed entirely from two sequence operations we name 'reusing' and 'recycling'.

1 Introduction

Let Π_n denote the set of permutations of $[n] := \{1, 2, \ldots, n\}$ written in one-line notation as strings. For example, $\Pi_3 = \{1\,2\,3,\ 1\,3\,2,\ 2\,1\,3,\ 2\,3\,1,\ 3\,1\,2,\ 3\,2\,1\}$ and we henceforth omit spaces between individual symbols when appropriate. The operation σ_k is a *prefix-rotation*, or simply *rotation*, and it cyclically moves the first k symbols one position to the left. In other words, σ_k applies the permutation $(1\ 2\ \cdots\ k)$ to the indices of a string. For example, $541362\ \sigma_4 = 413562$ since 413 moves one position to the left and 5 "wraps around" into the fourth position. The operation is also known as a *prefix-shift of length* k in the literature.

1.1 Rotator Graphs and Hamilton Cycles

The *rotator graph* $\mathcal{R}_n$ has nodes labeled with the strings in Π_n, and arcs labeled σ_k directed from $\alpha \in \Pi_n$ to $\beta \in \Pi_n$ when $\beta = \alpha\ \sigma_k$. In group-theoretic terms, $\mathcal{R}_n$ is the directed Cayley graph $\overrightarrow{\mathsf{Cay}}(\{\sigma_2, \sigma_3, \ldots, \sigma_n\}, \mathbb{S}_n)$ with generators σ_k for $2 \leq k \leq n$ and where $\mathbb{S}_n$ is the symmetric group corresponding to Π_n. A *restricted rotator graph* for $G \subseteq [n]$ is $\mathcal{R}_n(G) = \overrightarrow{\mathsf{Cay}}(G, \mathbb{S}_n)$ where the generators are restricted to σ_k for $k \in G$. Figure 1 (a) illustrates $\mathcal{R}_3$.

A Hamilton cycle of $\mathcal{R}_n(G)$ can be described by a *Hamilton sequence* of integers $S = s_0, s_1, \cdots, s_{n!-1}$ where σ_{s_i} is the label of the $(i+1)$st arc in the

* Research supported in part by NSERC.

C.S. Iliopoulos and W.F. Smyth (Eds.): IWOCA 2011, LNCS 7056, pp. 324–336, 2011.
© Springer-Verlag Berlin Heidelberg 2011

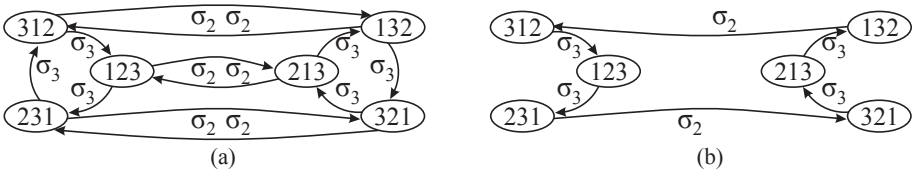


Fig. 1. (a) The rotator graph $\mathcal{R}_3$, and (b) a Hamilton cycle in $\mathcal{R}_3$

cycle and $s_i \in G$ for each $i \in \{0, 1, \ldots, n! - 1\}$. A Hamilton cycle of $\mathcal{R}_n(G)$ can also be described by the order of node labels along the cycle. In combinatorial generation, these orders are *cyclic Gray codes* since each string in Π_n appears exactly once, and successive strings differ by some σ_k for $k \in G$ where 'successive' includes last to first. For example, Figure 1 (b) contains a Hamilton cycle for $\mathcal{R}_3$ that can be described by

$$3, 3, 2, 3, 3, 2 \quad \text{or} \quad 321, 213, 132, 312, 123, 231. \tag{1}$$

Restricted rotator graphs are vertex-transitive; our Hamilton cycles and their associated Gray code orders for $\mathcal{R}_n(G)$ will all 'start' at $n\ n{-}1\ \cdots\ 1$. Orders of strings that do not necessarily have the Gray code properties are called *lists*.

An explicit Hamilton cycle in $\mathcal{R}_n$ was first constructed by Corbett [2]. Hamilton cycles were then constructed for different generalizations of $\mathcal{R}_n$ by Ponnuswamy and Chaudhary [11] and Williams [13]. Hamilton cycle constructions for $\mathcal{R}_n(\{n-1, n\})$ were proposed as an open problem by Knuth, and this was answered by Ruskey and Williams [12]. Observe that σ_n must be included in a restricted rotator graph in order to generate the entire symmetric group $\mathbb{S}_n$. Moreover, σ_n and σ_c are not sufficient for generating $\mathbb{S}_n$ if and only if c and n are both odd (in these cases the parity of a permutation cannot be changed). A well-known conjecture is that a Hamilton cycle exists in every connected undirected Cayley graph, where undirected Cayley graphs include the inverse of each generator. In particular, a Hamilton cycle was constructed for $\mathsf{Cay}(\{\sigma_2, \sigma_n\}, \mathbb{S}_n)$ by Compton and Williamson in a 50-page paper [1].

Corbett introduced the term "rotator graph" when considering point-to-point multiprocessor networks, where Hamilton cycles establish indexing schemes for sorting and for mapping rings and linear arrays [2] Applications of rotator graphs include fault-tolerant file transmission by Hamada et al [5] and parallel sorting by Corbett and Scherson [3]. Properties of rotator graphs have been examined including minimum feedback sets by Kuo et al [9] and node-disjoint paths by Yasuto, Ken'Ichi, and Mario [14]. Other variations of rotator graphs include incomplete rotator graphs [11], the bi-rotator graph (see Lin and Hsu [10]), and graphs where the labels can have repeated symbols [13]. The relationship between Hamilton cycles of $\mathcal{R}_n(n-1, n)$ and universal cycles of Π_n is discussed by Holroyd, Ruskey, and Williams along with applications [6,7].

1.2 New Results

We construct a new Hamilton cycle in $\mathcal{R}_n(\{n-1, n\})$ and the first Hamilton cycle in $\mathcal{R}_n(\{2, 3, n\})$. The chosen sets $\{n-1, n\}$ and $\{2, 3, n\}$ are natural since σ_{n-1} is the "largest" rotation other than σ_n, whereas σ_2 and σ_3 are the "smallest" pair of rotations given the previously mentioned difficulty of the Compton and Williamson result for $\mathsf{Cay}(\{\sigma_2, \sigma_n\}, \mathbb{S}_n)$ [1] and the trivial lack of connectivity in $\mathsf{Cay}(\{\sigma_3, \sigma_n\}, \mathbb{S}_n)$ when n is odd.

Our new constructions are intimately related to Corbett's original Hamilton cycle in $\mathcal{R}_n$. In fact, the beauty of our results is that all three Hamilton sequences can be described by two operations that we name 'reusing' and 'recycling'. We also provide an algorithm for constructing the Hamilton sequences. The algorithm is *loopless* since successive values in the sequence are obtained in worst-case O(1)-time (see Ehrlich [4] for the first use of this term).

Section 2 formally defines the 'reuse' and 'recycle' operations. Section 3 constructs the three Hamilton cycles and proves that two of the constructions are correct. Section 4 gives a loopless algorithm that generates the Hamilton sequences. Section 5 extends Corbett's recursive construction with an iterative description that is instrumental to the final proof of correctness. Section 6 completes the final proof of correctness by proving that Corbett's Hamilton sequence of $\mathcal{R}_n$ can be 'recycled' into a Hamilton cycle of $\mathcal{R}_{n+1}(\{n, n+1\})$. Section 7 concludes with open problems.

2 Sequence Building

This section defines two operations for building sequences of positive integers and examines the lists they create when they are treated as rotation indices.

2.1 Reusing and Recycling

In this subsection we define the reusing and recycling sequence operations, and describe how they are applied to create lists of strings. Given i and n satisfying $1 < i < n$, the result of *reusing* and *recycling* i with respect to n is

$$\mathsf{reuse}_n(i) = \overbrace{n, \ldots, n}^{n-1 \text{ copies}}, n-i+1 \text{ and } \mathsf{recycle}_n(i) = n, n, \overbrace{n-1, \ldots, n-1}^{i-1 \text{ copies}}, \overbrace{n, \ldots, n}^{n-i-1 \text{ copies}}$$

respectively. Notice that both operations create sequences of n symbols that are each at least 2 and at most n. For example,

$$\mathsf{reuse}_6(3) = 6, 6, 6, 6, 6, 4 \text{ and } \mathsf{recycle}_6(3) = 6, 6, 5, 5, 6, 6. \tag{2}$$

We build longer sequences by applying these operations to each symbol in a sequence. If $S = s_1, s_2, \ldots, s_t$ is a sequence with $1 < s_i < n$ for each i, then

$$\mathsf{reuse}_n(S) = \mathsf{reuse}_n(s_1), \mathsf{reuse}_n(s_2), \ldots, \mathsf{reuse}_n(s_t) \text{ and}$$
$$\mathsf{recycle}_n(S) = \mathsf{recycle}_n(s_1), \mathsf{recycle}_n(s_2), \ldots, \mathsf{recycle}_n(s_t).$$

We use sequences to create lists of strings by applying successive prefix-rotations. If $\alpha \in \Pi_n$ and $S = s_1, s_2, \ldots, s_t$ is a sequence with $1 < s_i \leq n$ for each i, then

$$\alpha \circ S = \beta_0, \beta_1, \ldots, \beta_t \text{ where } \beta_0 = \alpha \text{ and } \beta_i = \beta_{i-1}\ \sigma_{s_i} \text{ for } i = 1, 2, \ldots, t.$$

For example, if $\alpha = 612345$ then

$$\begin{aligned} \alpha \circ \mathsf{reuse}_6(3) &= 612345, 123456, 234561, 345612, 456123, 561234, 612534 \quad (3) \\ \alpha \circ \mathsf{recycle}_6(3) &= 612345, 123456, 234561, 345621, 456231, 562314, 623145 \end{aligned}$$

since $\mathsf{reuse}_6(3) = 6, 6, 6, 6, 6, 4$ and $\mathsf{recycle}_6(3) = 6, 6, 5, 5, 6, 6$ by (2). In some situations it is more convenient to leave off the last permutation in the list $\alpha \circ S$, and we use $\alpha \bullet S$ in these cases.

A symbol x is *periodic* in a list L of Π_n if the position of x moves once to the left (cyclically) between successive strings in L. For example, 6 is periodic in both lists from (3). More generally, the first symbol x of $\alpha \in \Pi_n$ is periodic in any list of the form $\alpha \circ \mathsf{reuse}(S)$ or $\alpha \circ \mathsf{recycle}(S)$. This is because the first rotation σ_n moves x from the first position to the last position, the next $n-1$ rotations move x one position to the left, and this pattern is repeated.

2.2 Rotation Identities

In this subsection we give two identities involving rotations. In addition to $\sigma_i = (1\ 2\ \cdots\ i)$ for prefix-rotations, let $\varsigma_i = (n\ n{-}1\ \cdots\ n{-}i{+}1)$ denote the *suffix-rotation* operation, and $\sigma'_i = (2\ 3\ \cdots\ i{+}1)$ denote a modified prefix-rotation that begins at the second symbol. We also let σ_i^j denote j successive copies of σ_i, and successive rotations are applied from left-to-right. Using these conventions we have the following simple identities

$$\underset{\text{“reuse equality”}}{\sigma_n^{n-1}\sigma_{n-i+1} = \varsigma_i} \quad \text{and} \quad \underset{\text{“recycle equality”.}}{\sigma_n^2\sigma_{n-1}^{i-1}\sigma_n^{n-i-1} = \sigma'_i} \qquad (4)$$

The “reusing equality” on the left follows from $(n\ n{-}1\ \cdots\ 1)(1\ 2\ \cdots\ n{-}i{+}1) = (n{-}i{+}1\ n{-}i{+}2\ \cdots\ n)$, while the “recycling equality” on the right is the second equality of Lemma 2 in [7]. The equalities allow the last string obtained by applying $\mathsf{reuse}_n(i)$ and $\mathsf{recycle}_n(i)$ to be computed directly. For example, when $i = 3$ and $n = 6$ we obtain the final strings in (3) as follows

$$\begin{aligned} 612345\sigma_6^5\sigma_4 &= 612345\varsigma_3 & 612345\sigma_6^2\sigma_5^2\sigma_6^2 &= 612345\sigma'_3 \quad (5) \\ &= 612534 & &= 623145. \end{aligned}$$

2.3 List Quotients

In Section 2.1 we saw that every nth string in $n\ n{-}1\ \cdots\ 1 \circ S$ begins with n, whenever S is obtained by reusing or recycling. Furthermore, Section 2.2 gave identities for these strings. This subsection examines these strings in more detail.

The *quotient* of a list L of Π_n with a symbol $x \in [n]$ is the list obtained from L by (1) removing the strings that do not begin with x, and (2) removing x from

the strings that begin with x. We denote this operation by x/L. Our first lemma uses recycling and is illustrated by the next example. If $S = 3, 3, 2, 3, 3, 2$ then

$$321 \bullet S = \underline{321}, 213, 132, 312, 123, 231 \text{ and} \tag{6}$$

$$4321 \bullet \mathsf{recycle}(S) = 4\underline{321}, 3214, 2143, 1423, 4\underline{213}, 2134, 1342, 3412, 4\underline{132}, 1324, 3241, 2431, 4\underline{312}, 3124, 1243, 2413, 4\underline{123}, 1234, 2341, 3421, 4\underline{231}, 2314, 3142, 1432.$$

Notice the quotient of the second list with 4 equals the first list (as underlined). That is, $4/(4321 \bullet \mathsf{recycle}(S)) = 321 \bullet S$. Lemma 1 proves this is true for any S.

Lemma 1. *If sequence S has values in $\{2, \ldots, n{-}1\}$ and $\alpha_i = i\ i{-}1\ \cdots\ 1$, then*

$$n/(\alpha_n \bullet \mathsf{recycle}_n(S)) = \alpha_{n-1} \bullet S.$$

Proof. The first string in both lists is α_{n-1} since $n/\alpha_n = \alpha_{n-1}$. Since n is periodic in $\alpha_n \circ \mathsf{recycle}_n(S)$, every nth string begins with n. Therefore, successive strings in $n/(\alpha_n \circ \mathsf{recycle}_n(S))$ are obtained by successive σ_{s_i} for $S = s_1, \ldots, s_t$ by the "recycling identity" in (4). Therefore, the two lists are equal. □

Our second lemma instead uses reusing and is illustrated by the next example

$$321 \bullet S = \underline{321}, \underline{213}, \underline{132}, \underline{312}, \underline{123}, \underline{231} \text{ and} \tag{7}$$

$$4321 \bullet \mathsf{reuse}(S) = 4\underline{321}, 3214, 2143, 1432, 4\underline{132}, 1324, 3241, 2413, 4\underline{213}, 2134, 1342, 3421, 4\underline{231}, 2314, 3142, 1423, 4\underline{123}, 1234, 2341, 3412, 4\underline{312}, 3124, 1243, 2431.$$

In this case the quotient of the second list with 4 equals the "double-reverse" of the first list. Given a string $a_1 a_2 \cdots a_n \in \Pi_n$ the *double-reverse* is

$$a_1 a_2 \cdots a_n^R = (n{-}a_n{+}1)\ \cdots\ (n{-}a_2{+}1)\ (n{-}a_1{+}1).$$

In a double-reverse the relative order of symbols is changed from $a_1 a_2 \cdots a_n$ to $a_n \cdots a_2 a_1$ and relative values are reversed from x to $n{-}x{+}1$. Given a list $L = \alpha_1, \ldots, \alpha_m$ the *double-reversal* of L is $L^R = \alpha_1^R,\ \ldots,\ \alpha_m^R$. For example,

$$\begin{aligned}(321, 132, 213, 231, 123, 312)^R &= 321^R, 132^R, 213^R, 231^R, 123^R, 321^R \\ &= 321, 213, 132, 312, 123, 231.\end{aligned}$$

This equation illustrates the relationship $4/(4321 \bullet \mathsf{reuse}(S)) = (321 \bullet S)^R$ in (7) (as underlined). Lemma 2 proves this is true for any S.

Lemma 2. *If sequence S has values in $\{2, \ldots, n{-}1\}$ and $\alpha_i = i\ i{-}1\ \cdots\ 1$, then*

$$n/(\alpha_n \circ \mathsf{reuse}_n(S)) = (\alpha_{n-1} \circ S)^R.$$

Proof. The first string in both lists is α_{n-1} since $n/\alpha_n = \alpha_{n-1}$ and $\alpha_{n-1}^R = \alpha_{n-1}$. Since n is periodic in $\alpha_n \circ \mathsf{reuse}_n(S)$, every nth string begins with n. Therefore, successive strings in $n/(\alpha_n \circ \mathsf{reuse}_n(S))$ are obtained by successive ς_{s_i} for $S = s_1, \ldots, s_t$ by the "reusing identity" in (4). Notice that suffix-rotations in a double-reversed string are 'equivalent' to prefix-rotations in the original string. That is, if $\alpha = \beta^R$, then $\alpha\ \sigma_i = \beta\ \varsigma_i^R$. Therefore, the two lists are equal. □

3 Three Hamilton Sequences

This section constructs Hamilton sequences for $\mathcal{R}_n$, $\mathcal{R}_n(\{n-1, n\})$, and $\mathcal{R}_n(\{2, 3, n\})$ through reusing and recycling. Two of the three main theorems are proven in this section, and the third is proven in Sections 5 and 6.

3.1 Hamilton Sequence for $\mathcal{R}_n$

This subsection proves that a Hamilton sequence for $\mathcal{R}_n$ can be obtained entirely with the reuse operation. The *Corbett sequence* is defined recursively as follows

$$C(n) = \begin{cases} 2,\ 2 & \text{if } n = 2 \\ \mathsf{reuse}_n(C(n-1)) & \text{if } n > 2. \end{cases} \tag{8}$$

Corbett proved that $C(n)$ is a Hamilton sequence for the rotator graph $\mathcal{R}_n$ [2]. Let $\Pi_C(n) = n\ n{-}1\ \cdots\ 1 \circ C(n)$ denote this *Corbett Gray code* of Π_n. Table 1 gives $C(n)$ and $\Pi_C(n)$ for $n = 3, 4$.

Table 1. (a)-(b) Corbett sequence for $n = 3, 4$, and (c)-(d) Corbett Gray code for $n = 3, 4$. Prefix-rotations in (c) and suffix-rotations of every fourth string in (d) are underlined according to (a) by the "reusing equality" in (4).

$C(3)$	$C(4) = \mathsf{reuse}_4(C(3))$	$\Pi_C(3)$	$\Pi_C(4) = 4321 \circ C(4)$
3,	4, 4, 4, 2,	$\underline{321}$,	4$\underline{321}$, 3214, 2143, 1432,
3,	4, 4, 4, 2,	$\underline{213}$,	4$\underline{132}$, 1324, 3241, 2413,
2,	4, 4, 4, 3,	$\underline{132}$,	42$\underline{13}$, 2134, 1342, 3421,
3,	4, 4, 4, 2,	$\underline{312}$,	4$\underline{231}$, 2314, 3142, 1423,
3,	4, 4, 4, 2,	$\underline{123}$,	4$\underline{123}$, 1234, 2341, 3412,
2	4, 4, 4, 3	$\underline{231}$	43$\underline{12}$ 3124, 1243, 2431
(a)	(b)	(c)	(d)

Theorem 1 extends Corbett's result by proving that any Hamilton sequence for $\mathcal{R}_{n-1}$ can be 'reused' into a Hamilton sequence for $\mathcal{R}_n$. Furthermore, we explicitly state the values used in the resulting sequence. (A simple induction proves that Corbett's 'canonical' sequence $C(n)$ uses each value in $\{2, 3, \ldots, n\}$.)

Theorem 1. *[2] If S is a Hamilton sequence in $\mathcal{R}_{n-1}(G)$, then $\mathsf{reuse}_n(S)$ is a Hamilton sequence in $\mathcal{R}_n(H)$, where $i \in H$ if and only if $i = n$ or $n - i + 1 \in G$.*

Proof. Let $\alpha = n\ n{-}1\ \cdots\ 1$. By Lemma 2 the nth strings in $\alpha \circ \mathsf{reuse}_n(S)$ form a Gray code for the strings in Π_n that begin with n. Each of these strings is followed by $n{-}1$ applications of σ_n by the definition of $\mathsf{reuse}_n(i)$. Therefore, $\alpha \circ \mathsf{reuse}_n(S)$ contains every string in Π_n and so $\mathsf{reuse}_n(S)$ is a Hamilton sequence. Finally, the values in H follow immediately from the definition of reusing. □

3.2 Hamilton Sequence for $\mathcal{R}_n(\{n-1, n\})$

This subsection states that a Hamilton sequence for $\mathcal{R}_n(n-1, n)$ can be obtained by recycling Corbett's Hamilton sequence. In other words, a Hamilton sequence for $\mathcal{R}_n(n-1, n)$ can be obtained by repeated reusing following by a single recycle. Let $D(n) = \mathsf{recycle}_n(C(n-1))$ denote this sequence and $\Pi_D(n) = n\ n{-}1\ \cdots\ 1 \circ D(n)$ denote its Gray code. Table 2 gives $D(n)$ and $\Pi_D(n)$ for $n = 4$.

Table 2. (a)-(b) Recycling the Corbett sequence and (c)-(d) the Corbett Gray code from $n = 3$ to $n = 4$. Prefix-rotations in (c) and modified prefix-rotations of every fourth string in (d) are underlined according to (a) by the "recycling equality" in (4).

$C(3)$	$D(4) = \mathsf{recycle}_4(C(3))$	$\Pi_C(3)$	$\Pi_D(4) = 4321 \circ D(4)$
3,	4, 4, 3, 3,	321,	4321, 3214, 2143, 1423,
3,	4, 4, 3, 3,	213,	4213, 2134, 1342, 3412,
2,	4, 4, 3, 4,	132,	4132, 1324, 3241, 2431,
3,	4, 4, 3, 3,	312,	4312, 3124, 1243, 2413,
3,	4, 4, 3, 3,	123,	4123, 1234, 2341, 3421,
2	4, 4, 3, 4	231	4231 2314, 3142, 1432
(a)	(b)	(c)	(d)

Theorem 2. *If $S = C(n-1)$ is the Corbett sequence for $\mathcal{R}_{n-1}$, then $\mathsf{recycle}_n(S)$ is a Hamilton sequence in $\mathcal{R}_n(\{n-1, n\})$.*

To illustrate the difficulty of Theorem 2, we point out that arbitrary Hamilton sequences for $\mathcal{R}_{n-1}$ cannot be recycled into Hamilton sequences for $\mathcal{R}_n$. For example, consider the following Hamilton sequence for $\mathcal{R}_4$ and its associated Gray code for Π_4

$$\begin{aligned} S &= 4,3,3,2,3,4,2,3,4,2,3,3,4,4,2,3,3,2,3,4,4,4,3,4 \qquad (9)\\ 4321 \circ S &= 4321, 3214, 2134, \underline{1324}, \underline{3124}, 1234, 2341, 3241, 2431, 4312, 3412, 4132,\\ &\quad 1342, 3421, 4213, 2413, 4123, 1243, 2143, 1423, 4231, \underline{2314}, \underline{3142}, 1432. \end{aligned}$$

Observe that 1324 is followed by 1324 $\sigma_2 = 3124$, and that 2314 is followed by 2314 $\sigma_4 = 3142$ in $4321 \circ S$. Therefore, Lemma 1 implies that 51324 is followed by $51324 \bullet \mathsf{recycle}_5(2)$, and 52314 followed by $52314 \bullet \mathsf{recycle}_5(4)$ in the recycled list $54321 \bullet \mathsf{recycle}_5(S)$. These two sublists appear below

$$\begin{aligned} &51324 \bullet \mathsf{recycle}_5(2) && 52314 \bullet \mathsf{recycle}_5(4) \qquad (10)\\ &= 51324 \bullet 5,5,4,5,5 && = 52314 \bullet 5,5,4,4,4\\ &= 51324, 13245, 32451, 24531, \underline{45312} && = 52314, 23145, 31452, 14532, \underline{45312}. \end{aligned}$$

Since both sublists contain 45312, the list $54321 \bullet \mathsf{recycle}_5(S)$ is not a Gray code. Furthermore, the reader can verify that $\mathsf{recycle}_6(\mathsf{reuse}_5(S))$ is also not a Hamilton sequence. In other words, an arbitrary Hamilton sequence S cannot be recycled into a Hamilton sequence, even when S is the result of reusing a previous Hamilton sequence. We prove Theorem 2 by developing results in Sections 5-6.

3.3 Hamilton Sequence for $\mathcal{R}_n(\{2,3,n\})$

This subsection proves that a Hamilton sequence for $\mathcal{R}_n(\{2,3,n\})$ can be obtained by recycling and then reusing Corbett's Hamilton sequence. In other words, a Hamilton sequence for $\mathcal{R}_n(2,3,n)$ can be obtained by repeated reusing followed by a single recycle and then a single reuse. Let $E(n) = \mathsf{reuse}_n(D(n{-}1))$ denote this sequence and $\Pi_E(n) = n\ n{-}1\ \cdots\ 1 \circ E(n)$ denote its Gray code. More generally, Theorem 3 proves that a Hamilton sequence for $\mathcal{R}_n(\{2,3,n\})$ can be obtained by reusing any Hamilton sequence for $\mathcal{R}_{n-1}(\{n{-}2,n{-}1\})$.

Theorem 3. *If S is a Hamilton sequence in $\mathcal{R}_{n-1}(\{n{-}2,n{-}1\})$, then $\mathsf{reuse}_n(S)$ is a Hamilton sequence in $\mathcal{R}_n(\{2,3,n\})$.*

Proof. By the statement of the theorem, S is a Hamilton sequence for $\mathcal{R}_{n-1}(G)$ for $G = \{n{-}2,n{-}1\}$. By theorem 1, $\mathsf{reuse}_n(S)$ is a Hamilton sequence in $\mathcal{R}_n(H)$ where $H = \{n{-}(n{-}2){+}1, n{-}(n{-}1){+}1, n\} = \{2,3,n\}$. □

4 Loopless Algorithm

In this section we show how to generate each symbol of Corbett's Hamilton sequence $C(n)$ for the rotator graph $\mathcal{R}_n$ in worst-case O(1)-time. Furthermore, our $\mathsf{CorbettLoopless}(n)$ algorithm is significant because

1. It adapts a well-known algorithm for generating multi-radix numbers, and
2. A modification generates Hamilton sequences in $\mathcal{R}_n(\{n{-}1,n\})$ or $\mathcal{R}_n(\{2,3,n\})$.

4.1 Staircase Sequence

The *staircase sequence* $S(n)$ is obtained from repeated applications of the *step* sequence operation as defined below

$$\mathsf{step}_n(i) = \overbrace{n,\ldots,n}^{n-1 \text{ copies}},i \quad \text{and} \quad S(n) = \begin{cases} 2 & \text{if } n = 1 \\ \mathsf{step}_n(S(n{-}1)) & \text{if } n > 1. \end{cases} \tag{11}$$

The step operation is identical to the reuse operation except the final symbol i has replaced $n{-}i{+}1$. Lemma 3 specifies each value of Corbett's sequence in terms of the staircase sequence and gives a simple condition for the occurrence of each value.

Lemma 3. *If the staircase sequence is $S(n) = s_1, s_2, \ldots, s_{n!}$ and the Corbett sequence is $C(n) = c_1, c_2, \ldots, c_{n!}$ and $n \geq 2$, then for each i satisfying $1 \leq i \leq n!$, we have $s_i = j$ if $n(n-1)\cdots(j+1)$ divides i but $n(n-1)\cdots(j+1)j$ does not divide i, and*

$$c_i = \begin{cases} n & \text{if } s_i = n \\ 2 & \text{if } s_i = n{-}1 \\ n{-}1 & \text{if } s_i = n{-}2 \\ 3 & \text{if } s_i = n{-}3 \\ n{-}2 & \text{if } s_i = n{-}4 \\ \cdots & \cdots \\ \lceil \frac{n+1}{2} \rceil & \text{if } s_i = 2. \end{cases}$$

Proof. The result is true for $n = 2$ since $S(2) = C(2) = 2, 2$. Assume the result is true for $S(k) = s'_1, \ldots, s'_{k!}$ and $C(k) = c'_0, \ldots, c'_{k!}$ for $k \geq 2$. When $n = k{+}1$,

$$S(n) = \mathsf{step}(S(n{-}1)) = \overbrace{n, \ldots, n}^{n-1 \text{ copies}}, s'_1, \ \ldots, \ \overbrace{n, \ldots, n}^{n-1 \text{ copies}}, s'_{n-1!}$$
$$C(n) = \mathsf{reuse}(C(n{-}1)) = \underbrace{n, \ldots, n}_{n-1 \text{ copies}}, n{-}c'_1{+}1, \ \ldots, \ \underbrace{n, \ldots, n}_{n-1 \text{ copies}}, n{-}c'_{n-1!}{+}1$$

so the result follows by induction by (11) and (8), respectively. □

Algorithm 1. Generate the staircase sequence $S(n)$ by $\mathsf{StaircaseLoopless}(n)$ and Corbett's Hamilton sequence $C(n)$ for rotator graph $\mathcal{R}_n$ by $\mathsf{CorbettLoopless}(n)$. Note: The final symbol output by $\mathsf{StaircaseLoopless}(n)$ is 1 instead of 2 by (11).

Require: $\mathsf{StaircaseLoopless}(n)$

```
 1:
 2: a_1 ··· a_n ← 0 ··· 0
 3: f_1 ··· f_n ← 1 ··· n
 4: loop
 5:    j ← f_1
 6:    output(n−j+1)
 7:    if j = n then
 8:       return
 9:    end if
10:    f_1 ← 1
11:    a_j ← a_j + 1
12:    if a_j = n−j then
13:       a_j ← 0
14:       f_j ← f_{j+1}
15:       f_{j+1} ← j + 1
16:    end if
17: end loop
```

Require: $\mathsf{CorbettLoopless}(n)$

```
 1: r_1 ··· r_n ← n 2 n−1 3 ··· ⌈(n+1)/2⌉ ⌈(n+1)/2⌉
 2: a_1 ··· a_n ← 0 ··· 0
 3: f_1 ··· f_n ← 1 ··· n
 4: loop
 5:    j ← f_1
 6:    output(r_j)
 7:    if j = n then
 8:       return
 9:    end if
10:    f_1 ← 1
11:    a_j ← a_j + 1
12:    if a_j = n−j then
13:       a_j ← 0
14:       f_j ← f_{j+1}
15:       f_{j+1} ← j + 1
16:    end if
17: end loop
```

4.2 Staircase Strings

Staircase sequences arise naturally in combinatorial generation. A string $\alpha = a_1 a_2 \cdots a_n$ is a *staircase string* if its symbols satisfy $1 \leq a_i \leq i$ for all $1 \leq i \leq n$. In other words, staircase strings are multi-radix numbers with radices $m_i = i$ for $1 \leq i \leq n$. Loopless Algorithm H in *The Art of Computer Programming* generates multi-radix numbers in reflected Gray code order, meaning that successive strings differ by ± 1 in a single symbol (see Knuth [8] pg. 20). In the special case of staircase strings, Algorithm H generates the $\pm$ indices according to the staircase sequence. For example, the Gray code appears below for $n = 3$

$$11\underline{1}, 11\underline{2}, 1\underline{1}3, 12\underline{3}, 12\underline{2}, 1\underline{2}1,$$

where the $\pm$ indices follow $S(3) = 3, 3, 2, 3, 3, 2$ (cyclically). StaircaseLoopless(n) in Algorithm 1 gives our presentation of Algorithm H, which is simplified by removing references to the multi-radix number, the $\pm$ direction array d, and by "hard-coding" the radices $m_i = i$ for $1 \le i \le n$. As in Knuth's presentation, array f stores focus pointers. To generate $C(n)$, we introduce an auxiliary array of constants

$$r_1, r_2, r_3, r_4, \cdots, r_{n-1}, r_n = n, 2, n{-}1, 3, \cdots, \left\lceil \frac{n}{2} \right\rceil, \left\lceil \frac{n}{2} \right\rceil$$

whose values are explained by Lemma 3. Finally, CorbettLoopless(n) in Algorithm 1 is obtained by replacing output($n{-}j{+}1$) on line 6 by output(r_j).

Theorem 4. CorbettLoopless(n) *is a loopless algorithm that generates Corbett's sequence* $C(n)$.

By Theorem 2 algorithm CorbettLoopless(n) can instead generate the Hamilton sequence $D(n)$ for $\mathcal{R}_{n+1}(\{n, n{+}1\})$ via recycling by replacing line 6 with

$$\mathsf{output}(n{+}1, n{+}1, \underbrace{n, \ldots, n}_{r_j - 1 \text{ copies}}, \underbrace{n{+}1, \ldots, n{+}1}_{n - r_j \text{ copies}}).$$

Similarly, by Theorem 1 the algorithm can generate the Hamilton sequence $E(n)$ for $\mathcal{R}_{n+2}(\{2, 3, n{+}2\})$ via recycling and reusing by replacing line 6 with

$$\mathsf{output}(\overbrace{n{+}2, \ldots, n{+}2}^{n+1 \text{ copies}}, 2, \overbrace{n{+}2, \ldots, n{+}2}^{n+1 \text{ copies}}, 2, \underbrace{\overbrace{n{+}2, \ldots, n{+}2}^{n+1 \text{ copies}}, 3, \ldots, \overbrace{n{+}2, \ldots, n{+}2}^{n+1 \text{ copies}}, 3}_{r_j - 1 \text{ copies}}, \underbrace{\overbrace{n{+}2, \ldots, n{+}2}^{n+1 \text{ copies}}, 2, \ldots, \overbrace{n{+}2, \ldots, n{+}2}^{n+1 \text{ copies}}, 2}_{n - r_j \text{ copies}}).$$

5 Corbett's Successor Rule

In Section 4 we showed how to generate Corbett's sequence $C(n)$ one symbol at a time, with Algorithm CorbettLoopless(n) creating the entire sequence and requiring two auxiliary arrays. Theorem 5 gives a *successor rule* that describes how each string in Corbett's Gray code $\Pi_C(n)$ can be computed from the previous string without additional state. The theorem is illustrated after its proof.

Theorem 5. *Suppose* $\alpha = a_1 a_2 \cdots a_n \in \Pi_n$. *Let* x *and* y *be the lengths of the longest prefix of the form* $n\ n{-}1\ n{-}2\ \cdots$ *and the longest suffix of the form* $\cdots\ 3\ 2\ 1$ *in* $a_2 a_3 \cdots a_n$, *respectively. The string that follows* α *in* $\Pi_C(n)$ *is*

$$\beta = \begin{cases} \sigma_{y+2}(\alpha) & \text{if } x > y \quad (12a) \\ \sigma_{n-x}(\alpha) & \text{otherwise } (x \le y). \quad (12b) \end{cases}$$

Proof. Suppose Corbett's sequence is $C(n) = c_1, c_2, \ldots, c_{n!}$, Corbett's Gray code is $\Pi_C(n) = \alpha_1, \alpha_2, \ldots, \alpha_{n!}$, and $n \ge 2$. Consider an arbitrary $\alpha_i = a_1 a_2 \cdots a_n$ in the Gray code. By using Lemma 2 and 3 the following conditions can be proven by induction on n

$$
\begin{aligned}
a_2 = n &\iff c_i \neq n, \text{and} \\
a_n = 1 \text{ and } a_2 = n &\iff c_i \notin \{n, 2\}, \text{and} \\
a_3 = n{-}1 \text{ and } a_n = 1 \text{ and } a_2 = n &\iff c_i \notin \{n, 2, n{-}1\}, \text{and} \\
a_{n-1} = 2 \text{ and } a_3 = n{-}1 \text{ and } a_n = 1 \text{ and } a_2 = n &\iff c_i \notin \{n, 2, n{-}1, 3\}, \text{and} \\
\cdots &\iff \cdots \\
\alpha = a_1\ n\ n{-}1\ \cdots\ p{+}2\ p{+}1\ a_{q+1}\ p{-}2\ p{-}3\ \cdots\ 2\ 1 &\iff c_i = p.
\end{aligned}
$$

where $p = \lceil \frac{n+1}{2} \rceil$ and $q = \lfloor \frac{n+1}{2} \rfloor$. The rule follows from these conditions. □

For example, if $\alpha = 4\underline{87}5623\overline{1}$ then $x = 2$ and $y = 1$ due to the underlined prefix and overlined suffix of 8756231, respectively. Therefore, the string after α in $\Pi_C(8)$ is $\alpha\ \sigma_3 = 48756231\ \sigma_3 = 87456231$ by (12a) since $x > y$ and $y = 2$.

Theorem 5 also allows the lookup table of size $n!$ to be avoided in Corbett's original application involving point-to-point multiprocessor networks [2].

6 Recycling Corbett's Sequence

In this section we prove a restatement of Theorem 2: *If $S = C(n)$ is the Corbett sequence for $\mathcal{R}_n$, then $\mathsf{recycle}_{n+1}(S)$ is a Hamilton sequence in $\mathcal{R}_{n+1}(\{n, n{+}1\})$.*

Proof. We prove an arbitrary string in Π_{n+1} appears in $n{+}1\ n\ \cdots\ 1 \circ \mathsf{recycle}(S)$ where $S = C(n)$. Let this arbitrary string equal $a_i\ a_{i+1}\ \cdots\ a_n\ n{+}1\ b_1\ b_2\ \cdots b_{i-1}$ for some i satisfying $1 \leq i \leq n{+}1$. We choose this expression for our arbitrary string since we will find α and β such that the following criteria hold

1. α has suffix $a_i\ a_{i+1}\ \cdots\ a_n$, and
2. β has prefix $b_1\ b_2\ \cdots\ b_{i-2}$, and
3. α is followed by β in $\Pi_C(n)$ by applying σ_r, and
4. $\alpha \circ \mathsf{recycle}(r)$ contains the arbitrary string $a_i\ a_{i+1}\ \cdots\ a_n\ n{+}1\ b_1\ b_2\ \cdots b_{i-1}$.

The result is trivial when $n{+}1$ is in the first, last, or second-last position of the arbitrary string. In the remaining cases we define the following

- $\gamma := g_1\ g_2\ ...g_n := b_1\ b_2\ ...\ b_{i-1}\ a_i\ a_{i+1}\ ...\ a_n$ and $p = \lfloor \frac{n}{2} \rfloor - 1$ and $q = \lceil \frac{n}{2} \rceil - 1$,
- x_b is the length of the longest $n\ n{-}1\ n{-}2\ \cdots$ prefix in $g_1\ g_2\ \cdots\ g_{i-2}$,
- y_a is the length of the longest $\cdots\ 3\ 2\ 1$ suffix in $g_i\ g_{i+1}\ \cdots\ g_n$,
- x' is the length of the longest $n\ n{-}1\ n{-}2\ \cdots$ prefix in $g_1\ g_2\ \cdots\ g_p$, and
- y' is the length of the longest $\cdots\ 3\ 2\ 1$ suffix in $g_{n-q+1}\ g_{n-q+2}\ \cdots\ g_n$.

One difference between (x_b, y_a) and (x', y') is that the former considers $b_1\ b_2\ \cdots\ b_{i-2}$ and $a_i\ a_{i+1}\ \cdots\ a_n$ separately, whereas the latter considers γ as a whole. Choose

$$
\alpha := \begin{cases}
b_{i-1}\ b_1\ b_2\ ...\ b_{i-2}\ a_i\ a_{i+1}\ ...\ a_n & \text{if } x_b \leq y_a \quad (13a) \\
g_{y'+2}\ g_1\ g_2\ ...\ g_{y'+1}\ g_{y'+3}\ g_{y'+4}\ ...g_n & \text{if } x_b > y_a \text{ and } x' > y' \quad (13b) \\
g_{n-x'}\ g_1\ g_2\ ...\ g_{n-x'-1}\ g_{n-x'+1}\ g_{n-x'+2}\ ...g_n & \text{if } x_b > y_a \text{ and } x' \leq y' \quad (13c)
\end{cases}
$$

In each case, we prove the first criterion holds for the choice of α. For (13a) this result is obvious. For (13b) there are two cases two consider. If $x_b \geq x'$, then

$$y' + 3 \leq x' + 2 \leq x_b + 2 \leq i$$

where the inequalities follow from $x' > y'$, $x_b \geq x'$, and $x_b \leq i - 2$, respectively. On the other hand, if $x_b < x'$ then it must be that $y_a = y'$ and so

$$y' + 3 = y_a + 3 \leq x_b + 2 \leq i$$

where the equalities and inequalities follow from $y' = y_a$, $x_b > y_a$, and $x_b \leq i-2$, respectively. In both cases, α has the suffix stated in the first criterion. For (13c) it must be that $i = n - y_a + 1$ and $x_b = x'$. Therefore,

$$n - x' + 1 = n - x_b + 1 \leq n - y_a = i - 1$$

where the equalities and inequalities follow from $x_b = x'$, $x_b < y_a$, and $i = n - y_a + 1$, respectively. Therefore, α has the suffix stated in the first criterion. To complete the proof, use the successor rule from Theorem 5 to verify the remaining criteria. □

Theorem 2 also affirms Conjecture 1 in [7]. That paper uses an equivalent notion of 'recycling' that acts on rotation Gray codes of Π_n instead of Hamilton sequences of $\mathcal{R}_n$. The conjecture is that Corbett's Gray code is 'recyclable' and Theorem 2 equivalently proves that Corbett's Hamilton sequence is 'recyclable'.

7 Open Problems

The following open problems are related to this research:

1. Efficiently generate an explicit Hamilton cycle in $\mathcal{R}_n(\{2, n\})$.
2. Necessary and sufficient conditions for recyclable Hamilton sequences of $\mathcal{R}_n$.
3. A loopless algorithm for generating a recyclable order of Π_n in an array.
4. The diameter of $\mathcal{R}_n(G)$ for $G = \{n-1, n\}$ and $G = \{2, 3, n\}$ and others.

For the fourth problem, we mention that Corbett showed the diameter of $\mathcal{R}_n$ is small [2] and discussed applications of this fact. For the third problem, we mention that there are many loopless algorithms that generate successive permutations in an array, but none are known to be 'recyclable' using the terminology from [7]. In fact, the known recyclable orders using rotations by Corbett [2] and Williams [13] cannot be generated by a loopless array-based algorithm since σ_n cannot be implemented in constant time.

Acknowledgement. The authors wish to thank all three referees for helpful comments. In particular, we wish to thank one referee who made several corrections and the observation that the direction array d could be removed from our initial presentation of Algorithm 1.

References

1. Compton, R.C., Williamson, S.G.: Doubly Adjacent Gray Codes for the Symmetric Group. Linear and Multilinear Algebra 35(3), 237–293 (1993)
2. Corbett, P.F.: Rotator Graphs: An Efficient Topology for Point-to-Point Multiprocessor Networks. IEEE Transactions on Parallel and Distributed Systems 3, 622–626 (1992)
3. Corbett, P.F., Scherson, I.D.: Sorting in Mesh Connected Multiprocessors. IEEE Transactions on Parallel and Distributed Systems 3, 626–632 (1992)
4. Ehrlich, G.: Loopless Algorithms for Generating Permutations, Combinations and Other Combinatorial Configurations. IEEE Transactions on Parallel and Distributed Systems 3, 626–632 (1992); Journal of the ACM 20(3), 500–513 (1973)
5. Hamada, Y., Bao, F., Mei, A., Igarashi, Y.: Nonadaptive Fault-Tolerant File Transmission in Rotator Graphs. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences E79-A 4, 477–482 (1996)
6. Holroyd, A., Ruskey, F., Williams, A.: Faster Generation of Shorthand Universal Cycles for Permutations. In: Thai, M.T., Sahni, S. (eds.) COCOON 2010. LNCS, vol. 6196, pp. 298–307. Springer, Heidelberg (2010)
7. Holroyd, A., Ruskey, F., Williams, A.: Shorthand Universal Cycles for Permutations. Algorithmica (to appear)
8. Knuth, D.E.: The Art of Computer Programming. Generating All Tuples and Permutations, Fascicle 2, vol. 4. Addison-Wesley (2005)
9. Kuo, C.-J., Hsu, C.-C., Lin, H.-R., Lin, K.-K.: An Efficient Algorithm for Minimum Feedback Vertex Sets in Rotator Graphs. Information Processing Letters 109(9), 450–453 (2009)
10. Lin, H.-R., Hsu, C.-C.: Topological Properties of Bi-Rotator Graphs. IEICE Transactions on Information and Systems E86-D(10), 2172–2178 (2003)
11. Ponnuswamy, S., Chaudhary, V.: Embedding of Cycles in Rotator and Incomplete Rotator Graphs. In: Proceedings of the Sixth IEEE Symposium on Parallel and Distributed Processing, October 26-29, pp. 603–610 (1994)
12. Ruskey, F., Williams, A.: An Explicit Universal Cycle for the $(n-1)$-Permutations of an n-set. ACM Transactions on Algorithms 6(3), article 45 (2010)
13. Williams, A.: Loopless Generation of Multiset Permutations Using a Constant Number of Variables by Prefix Shifts. In: Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2009, New York, NY, USA, January 4-6, pp. 987–996 (2009)
14. Yasuto, S., Ken'Ichi, K., Mario, N.: Node-Disjoint Paths Problem in a Rotator Graph. Joho Shori Gakkai Shinpojiumu Ronbunshu 14, 93–100 (2003)

Efficient Codon Optimization with Motif Engineering

Anne Condon and Chris Thachuk

Department of Computer Science, University of British Columbia,
Vancouver, BC, Canada V6T 1Z4
{condon,cthachuk}@cs.ubc.ca

Abstract. It is now common to add protein coding genes into cloning vectors for expression within non-native host organisms. Codon optimization supports translational efficiency of the desired protein product, by exchanging codons which are rarely found in the host organism with more frequently observed codons. Motif engineering, such as removal of restriction enzyme recognition sites or addition of immuno-stimulatory elements, is also often necessary. We present an algorithm for optimizing codon bias of a gene with respect to a well motivated measure of bias, while simultaneously performing motif engineering. The measure is the previously studied codon adaptation index, which favors the use, in the gene to be optimized, of the most abundant codons found in the host genome. We demonstrate the efficiency and effectiveness of our algorithm on the GENCODE dataset and provide a guarantee that the solution found is always optimal.

1 Introduction

Gene synthesis is now an economical and technically viable option for the construction of non-natural genes. Synthetic genes can be novel or derivatives of those found in nature. In either case, the expression levels of these genes, when inserted into the genome of a host organism, depend on many factors. One important factor is the bias of codon usage, relative to the host organism [16,7,10]. Note that for each amino acid in a protein, there may be many (up to six) valid codons, as given by the genetic code. Loosely speaking, the codon bias of a gene for the protein measures how well – or poorly – codons used in the gene match codon usage in the genome of a host organism (we describe specific measures later in this paper). Several studies have indicated that [10,3,11] codon optimization is necessary to ensure designed genes are maximally expressed within the host.

In addition to optimizing the codon bias of a gene relative to the genome of a host, it is often desirable to add or remove certain motifs via silent mutation, whereby DNA sequence is altered without changing the expressed amino acid sequence. Removal or addition of motifs can be treated as optimization criteria to be minimized or maximized. For example, with immuno-regulatory CpG

C.S. Iliopoulos and W.F. Smyth (Eds.): IWOCA 2011, LNCS 7056, pp. 337–348, 2011.
© Springer-Verlag Berlin Heidelberg 2011

motifs in mammalian expression vectors [13] it is desirable to minimize immuno-inhibitory elements and maximize immuno-stimulatory motifs. In the remainder of this work, we will refer to inclusion or exclusion of motifs, via silent mutation, as motif engineering.

A number of published software tools are capable of codon optimization, including DNA Works [8], Codon Optimizer [2], GeMS [9], Gene Designer [17], JCat [4], OPTIMIZER [12], the Synthetic Gene Designer [18], UpGene [3] and a method by Satya *et. al* [13]. Some of these methods also consider the other problem considered here, motif engineering. Of these, only the method of Satya *et. al* provides a mathematical guarantee of finding an optimal solution when one exists. However, their method – based on the graph theoretic approach of finding a critical path – runs in $O(n^2)$ time and space, where n is the length of the DNA sequence being optimized. In this work, we propose the first linear time and space codon optimization algorithm, which is guaranteed to find an optimal solution that also satisfies motif engineering constraints. We have focused our attention on optimizing codon usage according to the Codon Adaptation Index (CAI). The index, originally proposed by Sharp and Li [14], is based on the premise of each amino acid having a 'best' codon for a particular organism. This perspective evolved from the observation that protein expression is higher in genes using codons of high fitness and lower in genes using rare codons [6]. It is believed that this is due to the relative availability of tRNAs within a cell.

We also provide an experimental study of the performance of our algorithm on a biological data set comprising 3,157 coding sequence regions of the GENECODE subset of the ENCODE dataset [15].

The remainder of this paper is structured as follows. In the Preliminaries section, we formally define the problem of codon optimization. We detail the general objectives of the problem, and formalize the goals of motif engineering. We then present our algorithm, providing a proof of correctness and time and space analysis. In the Empirical Results section, we describe the performance of our algorithm, both in terms of run-time efficiency and also in terms of the quality of optimization achieved. Finally, we conclude with a summary of our major findings and directions for future work.

2 Preliminaries

A DNA strand is a string over the alphabet of DNA. A *codon* is a triple over the DNA alphabet and therefore there are at most $4^3 = 64$ distinct codons. An *amino acid sequence* is a string over the alphabet of amino acids, $\Sigma_{AA} = \{Ala, Arg, Asn, \ldots, Tyr, Val, stop\}$, with each symbol representing an amino acid and the special symbol '*stop*' denoting a string terminal. We assume there is a predetermined ordering of amino acids, for example, lexicographic.

Therefore, we can represent an amino acid sequence A as a sequence of integers, with $A = \alpha_1, \alpha_2, \ldots, \alpha_{|A|}$, where $1 \leq \alpha_k \leq 21$, for $1 \leq k \leq |A|$. We denote the i$^{\text{th}}$ amino acid by $\lambda(i)$. The *genetic code* is a mapping between amino acids and codons. However, as there are 64 possible codons and only 20 amino

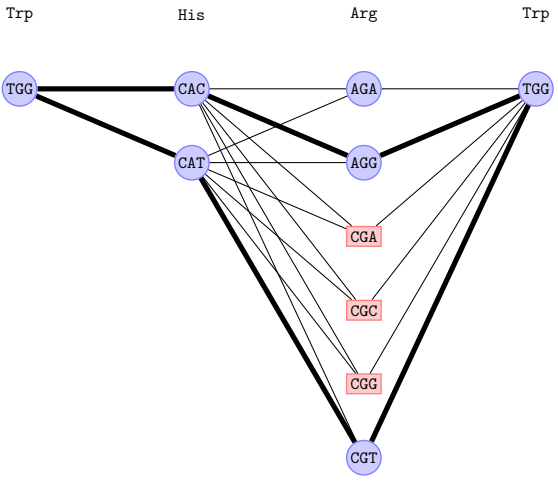


Fig. 1. Shown above is an instance consisting of four amino acids, a forbidden set $\mathcal{F} = \{$ `CGC`, `CGA`, `CGG`, `ACC`, `TAG` $\}$, a desired set $\mathcal{D} = \{$ `TTG`, `GGT`, `CCG` $\}$, and therefore $k = 1$. Arginine has six corresponding codons, however, three of them appear in $\mathcal{F}$ and are shown with red boxes. A valid codon assignment for this instance must contain no occurrence of a forbidden motif and one occurrence of a desired motif. There are two valid codon assignments for this problem instance, shown as paths with bold edges. The top (bottom) path denotes an assignment containing the desired motif `GGT` (`TTG`).

acids (plus one stop symbol), the code is degenerate, resulting in a one-to-many mapping from each amino acid to a set of corresponding codons.

We define $|\lambda(i)|$ to be the number of codons corresponding to the i$^{\text{th}}$ amino acid and $\lambda_j(i)$ to be the j^{th} such codon, $1 \leq j \leq |\lambda(i)|$, where again we use lexicographic ordering. Therefore, we can define a *codon design*, with respect to an amino acid sequence, as a sequence of codon indices. Again consider the problem instance in Figure 1. For the Arginine amino acid (`Arg`) which is the second amino acid in lexicographic order, $|\lambda(2)| = 6$ and $\lambda_3(2)$ is the codon `CGA`. The DNA sequence `TGA CAC CGA TGG` can be represented by the codon index sequence $S = 1, 1, 3, 1$.

A *codon's frequency* is the number of times that it appears in nature, divided by the total number of times that all codons corresponding to the same amino acid appear in nature. By "in nature", we mean codon frequencies present in some reference sequence or set of sequences such as a genome or set of genomes. As an example, if for some amino acid index i, $|\lambda(i)| = 2$, and the codon $\lambda_1(i)$ is observed 37 times in nature, while $\lambda_2(i)$ is observed 63 times, we can define the frequency of $\lambda_1(i)$ to be $\frac{37}{37+63} = 0.37$. Let $\rho_j(i)$ denote the frequency of the j^{th} codon of the i$^{\text{th}}$ amino acid, $1 \leq j \leq |\lambda(i)|$. Note that $\sum_{j=1}^{|\lambda(i)|} \rho_j(i) = 1.0$, for any i, assuming $\lambda(i)$ is in the reference set. In the example above, we say that $\lambda_2(i)$ is the *most frequent codon*. Note that it is possible for more than one codon to have this property.

A *codon's fitness* is the number of times that it appears in nature, divided by the number of occurrences of the corresponding most frequent codon (originally referred to as the relative adaptiveness of a codon [14]). Let $\tau_j(i)$ denote the fitness value of the j^{th} codon of the i^{th} amino acid. Returning to our previous example, if the i$^{\text{th}}$ amino acid has two codons with frequencies $\rho_1(i) = 0.37$ and $\rho_2(i) = 0.63$, then their fitness values, denoted by $\tau_1(i)$ and $\tau_2(i)$ respectively, are $0.37/0.63 \approx 0.59$ and $0.63/0.63 = 1.0$. Note that a most frequent codon will always have a fitness value of 1.0.

Motif Engineering. We focus our attention on designing codon sequences which minimize occurrences of forbidden motifs from a predetermined set, $\mathcal{F}$, while maximizing occurrences of desired motifs from a predetermined set, $\mathcal{D}$. A codon design – a sequence of codon assignments – is said to be *valid* with respect to an amino acid sequence it codes for if it satisfies the following constraints, in order: the DNA sequence corresponding to the codon design (1) contains the minimum possible number of forbidden motifs, and (2) contains the maximum possible number of desired motifs, given that (1) is satisfied. It is important to recognize that a valid design does not necessarily guarantee that the number of occurrences of desired motifs is the maximum number possible, of all possible codon designs. Again, consider the problem instance of Figure 1. Two codon designs result in a minimum number of forbidden motif occurrences (none), shown with paths having bold edges. Both of these paths also contain one occurrence of a desired motif. The top (bottom) path denotes an assignment containing the desired motif `GGT` (`TTG`). Therefore, a valid codon design for this instance, by our previous definition, contains no forbidden motifs and one desired motif. Notice that the DNA sequence `TGG CAC CGG TGG`, corresponding to a codon design $S = 1, 1, 5, 1$, actually contains more desired motifs (two) than a valid codon design; however, it does contain one forbidden motif and therefore cannot be valid.

We now develop some notation for motif engineering. For a sequence of amino acid indices $A = \alpha_1, \alpha_2, \ldots, \alpha_{|A|}$, a corresponding codon design $S = c_1, c_2, \ldots, c_{|A|}$, a set of forbidden motifs $\mathcal{F}$ and a set of desired motifs $\mathcal{D}$, let $M_{\mathcal{F}}(\lambda_{c_i}(\alpha_i) \ldots \lambda_{c_j}(\alpha_j))$ and $M_{\mathcal{D}}(\lambda_{c_i}(\alpha_i) \ldots \lambda_{c_j}(\alpha_j))$ be the number of occurrences of forbidden motifs and desired motifs, respectively, in the DNA sequence $\lambda_{c_i}(\alpha_i) \ldots \lambda_{c_j}(\alpha_j)$, where $j \geq i$. For convenience in our algorithms, we also introduce $M'_{\mathcal{F}}(\lambda_{c_i}(\alpha_i) \ldots \lambda_{c_j}(\alpha_j))$ and $M'_{\mathcal{D}}(\lambda_{c_i}(\alpha_i) \ldots \lambda_{c_j}(\alpha_j))$ which respectively determine the number of forbidden and desired motifs in $\lambda_{c_i}(\alpha_i) \ldots \lambda_{c_j}(\alpha_j)$ that end within the last codon position (the last 3 bases), here indexed by j. For instance consider the codon design $S = 1, 1, 5, 1$ of the problem instance in Figure 1. $M_{\mathcal{D}}(\texttt{TGGCACCGGTGG}) = 2$ as it contains the motifs `CCG` and `GGT`, however, $M'_{\mathcal{D}}(\texttt{TGGCACCGGTGG}) = 1$ as only the motif `GGT` ends within the last codon.

In practice, forbidden and desired motifs are short and we assume their length is bounded by a constant, g [13].

Observation 1. *If the largest forbidden or desired motif is of length g, then any forbidden or desired motif can span at most $k+1$ consecutive codons, where $k = \lceil g/3 \rceil$.*

Codon Optimization. The codon adaption index (CAI) is a metric defined in terms of the relative fitness of codons constituting a codon design. For some codon design $S = c_1, c_2, \ldots, c_{|A|}$, which correctly codes for a desired amino acid sequence $A = \alpha_1, \alpha_2, \ldots, \alpha_{|A|}$, the CAI value for S with respect to A, CAI(S, A), can be calculated as in Eqn. (1). Based on this definition, if S consists only of most frequent codons, it would have a CAI value of 1.0. Intuitively, the higher the CAI value, the better.

$$\text{CAI}(S, A) = \left(\prod_{i=1}^{|A|} \tau_{c_i}(\alpha_i) \right)^{\frac{1}{|A|}} \tag{1}$$

With the previously defined definitions, notation, and optimization criteria, we now formally define the problem of codon optimization with motif engineering.

The CAI Codon Optimization Problem with Motif Engineering
Instance: Amino acid sequence represented by the sequence of indices $A = \alpha_1, \alpha_2, \ldots, \alpha_{|A|}$, a set of forbidden motifs $\mathcal{F}$, and a set of desired motifs $\mathcal{D}$.

Problem: Find a codon design S^*, with $|S^*| = |A|$, corresponding to A such that S^* is *valid*, with respect to $\mathcal{F}$ and $\mathcal{D}$, and $CAI(S^*, A) = \max\{CAI(S, A) | S \in \mathbf{S}(A)\}$, where $\mathbf{S}(A)$ is the set of all valid codon designs corresponding to A. S^* is an *optimal codon design with respect to the CAI measure*.

3 A DP Algorithm for CAI Optimization

We now propose a linear time and space dynamic programming algorithm guaranteed to maximize the CAI measure, such that the codon design is *valid*. In terms of efficiency, this is a direct improvement in both run-time and space over the current state-of-the-art, previously proposed by Satya *et al.* [13]. Although we have chosen to first ensure forbidden motifs are minimized, then desired motifs maximized and finally the CAI value maximized, it should be clear that the algorithm we present can be adapted to optimize these criteria in any order.

One necessary feature of a codon optimization algorithm is an efficient means to detect if a forbidden motif from $\mathcal{F}$, or a desired motif from $\mathcal{D}$, is present in a potential design. For both algorithms proposed in this work, we utilize an Aho-Corasick search for this purpose. Briefly, the Aho-Corasick algorithm builds a keyword tree (trie) for $\mathcal{F}$ and transforms the structure into an automaton with the addition of failure links. Space and time complexity for building the initial structure is $O(h)$, where h is the sum of the lengths of the motifs in $\mathcal{F}$. Queries to determine if a sequence b contains any forbidden motif take $O(|b|)$ time [1]. Likewise, a second tree is constructed for the desired motifs in $\mathcal{D}$. For a detailed

description of the algorithm and existing applications of its use in computational biology, see Gusfield [5]. We note that Satya *et. al* [13] use the same approach for motif detection in their $\theta(n^2)$ algorithm. We note that any dictionary matching algorithm can be employed for the same task; however, Aho-Corasick automata were chosen due to their simpler implementation.

We first define three quantities that will be important in describing our algorithm. The first quantity, $F^i_{c_{i-k+1},\ldots,c_{i-1},c_i}$, denotes the minimum possible number of forbidden motifs in a DNA sequence which codes for an amino acid sequence $A = \alpha_1, \alpha_2, \ldots, \alpha_i$, given that the last k codons (of i total codons) have indices denoted as $c_{i-k+1}, \ldots, c_{i-1}, c_i$. Similarly, the second quantity, $D^i_{c_{i-k+1},\ldots,c_{i-1},c_i}$, denotes the maximum possible number of desired motifs, among those sequences which contain a minimum number of forbidden motifs. $P^i_{c_{i-k+1},\ldots,c_{i-1},c_i}$ denotes the maximum possible CAI score among all valid sequences.

Our algorithm stores a k-dimensional entry for each position i, $k \leq i \leq |A|$, of the input amino acid sequence, where $k = \lceil g/3 \rceil$ and g is the constant bounding the length of any forbidden or desired motif. The base case occurs when $i = k$ and is computed as follows. Every combination of codons for the first k amino acids is evaluated to determine, independently, the number of forbidden and desired motifs fully contained within the k consecutive codons (Eqn. (2) and Eqn. (3), respectively) and the CAI value (Eqn. (4)).

$$F^k_{c_1,c_2,\ldots,c_{k-1},c_k} = M_{\mathcal{F}}\left(\lambda_{c_1}(\alpha_1)\lambda_{c_2}(\alpha_2)\ldots\lambda_{c_{k-1}}(\alpha_{k-1})\lambda_{c_k}(\alpha_k)\right) \tag{2}$$

$$D^k_{c_1,c_2,\ldots,c_{k-1},c_k} = M_{\mathcal{D}}\left(\lambda_{c_1}(\alpha_1)\lambda_{c_2}(\alpha_2)\ldots\lambda_{c_{k-1}}(\alpha_{k-1})\lambda_{c_k}(\alpha_k)\right) \tag{3}$$

$$P^k_{c_1,c_2,\ldots,c_{k-1},c_k} = \prod_{i=1}^{k}\left(\tau_{c_i}(\alpha_i)\right) \tag{4}$$

The recursive case occurs for $i > k$. By Observation 1, a forbidden motif could span $k+1$ codons. Therefore, it is necessary to evaluate the last $k+1$ codons of a potential design to ensure codons are selected which 1) minimize forbidden motifs, then 2) maximize desired motifs, then 3) maximize the CAI score.

For any arbitrary assignment of the last k codons, we select the codon preceding them, denoted by the index c_{i-k}, such that the sum of forbidden motifs ending at position $i-1$, $F^{i-1}_{c_{i-k},\ldots,c_{i-2},c_{i-1}}$, and the count of new forbidden motifs which end in the new codon c_i, determined by the function $M'_{\mathcal{F}}$, is minimized. The number of forbidden motifs is recorded.

$F^i_{c_{i-k+1},\ldots,c_{i-1},c_i} = \min_{1 \leq c_{i-k} \leq |\lambda(\alpha_{i-k})|}$

$$\left\{F^{i-1}_{c_{i-k},\ldots,c_{i-2},c_{i-1}} + M'_{\mathcal{F}}\left(\lambda_{c_{i-k}}(\alpha_{i-k})\ldots\lambda_{c_{i-1}}(\alpha_{i-1})\lambda_{c_i}(\alpha_i)\right)\right\} \tag{5}$$

Similarly, D is calculated in the same manner, after ensuring that the minimal number of forbidden motifs criteria is first satisfied.

$D^i_{c_{i-k+1},\ldots,c_{i-1},c_i} = \max_{1 \le c_{i-k} \le |\lambda(\alpha_{i-k})|}$

$$\begin{cases} -\infty & \text{, if } F^{i-1}_{c_{i-k},\ldots,c_{i-2},c_{i-1}} + \\ & M'_{\mathcal{F}}\left(\lambda_{c_{i-k}}(\alpha_{i-k})\ldots\lambda_{c_{i-1}}(\alpha_{i-1})\lambda_{c_i}(\alpha_i)\right) \\ & \neq F^i_{c_{i-k+1},\ldots,c_{i-1},c_i} \\ D^{i-1}_{c_{i-k},\ldots,c_{i-2},c_{i-1}} & \\ +M'_{\mathcal{D}}\left(\lambda_{c_{i-k}}(\alpha_{i-k})\ldots\lambda_{c_i}(\alpha_i)\right) & \text{, otherwise} \end{cases} \tag{6}$$

Likewise, P is calculated to first ensure forbidden motifs are minimized, followed by desired motifs being maximized. Of these possible codon assignments, the one with the highest CAI value is selected and the score recorded.

$P^i_{c_{i-k+1},\ldots,c_{i-1},c_i} = \max_{1 \le c_{i-k} \le |\lambda(\alpha_{i-k})|}$

$$\begin{cases} -\infty & \text{, if } F^{i-1}_{c_{i-k},\ldots,c_{i-2},c_{i-1}} + \\ & M'_{\mathcal{F}}\left(\lambda_{c_{i-k}}(\alpha_{i-k})\ldots\lambda_{c_i}(\alpha_i)\right) \neq F^i_{c_{i-k+1},\ldots,c_{i-1},c_i} \\ & \vee D^{i-1}_{c_{i-k},\ldots,c_{i-2},c_{i-1}} \\ & +M'_{\mathcal{D}}\left(\lambda_{c_{i-k}}(\alpha_{i-k})\ldots\lambda_{c_i}(\alpha_i)\right) \neq D^i_{c_{i-k+1},\ldots,c_{i-1},c_i} \\ \tau_{c_i}(\alpha_i) \times P^{i-1}_{c_{i-k},\ldots,c_{i-2},c_{i-1}} & \text{, otherwise} \end{cases} \tag{7}$$

Eqn. (10) determines the optimal CAI score up to position i of the input amino acid sequence. Therefore, the optimal CAI value of some input sequence A of length $|A|$ is given by $\widetilde{P^{|A|}_k}$, where

$$\widetilde{F^i_k} = \min_{\substack{1 \le c_i \le |\lambda(\alpha_i)| \\ 1 \le c_{i-1} \le |\lambda(\alpha_{i-1})| \\ \vdots \\ 1 \le c_{i-k+1} \le |\lambda(\alpha_{i-k+1})|}} \left\{ F^i_{c_{i-k+1},\ldots,c_{i-1},c_i} \right\} \tag{8}$$

$$\widetilde{D^i_k} = \max_{\substack{1 \le c_i \le |\lambda(\alpha_i)| \\ 1 \le c_{i-1} \le |\lambda(\alpha_{i-1})| \\ \vdots \\ 1 \le c_{i-k+1} \le |\lambda(\alpha_{i-k+1})|}} \left\{ \begin{cases} D^i_{c_{i-k+1},\ldots,c_{i-1},c_i} & \text{, if } F^i_{c_{i-k+1},\ldots,c_{i-1},c_i} = \widetilde{F^i_k} \\ -\infty & \text{, otherwise} \end{cases} \right\} \tag{9}$$

$$\widetilde{P^i_k} = \max_{\substack{1 \le c_i \le |\lambda(\alpha_i)| \\ 1 \le c_{i-1} \le |\lambda(\alpha_{i-1})| \\ \vdots \\ 1 \le c_{i-k+1} \le |\lambda(\alpha_{i-k+1})|}} \left\{ \begin{cases} P^i_{c_{i-k+1},\ldots,c_{i-1},c_i} & \text{, if } F^i_{c_{i-k+1},\ldots,c_{i-1},c_i} = \widetilde{F^i_k} \\ & \wedge D^i_{c_{i-k+1},\ldots,c_{i-1},c_i} = \widetilde{D^i_k} \\ -\infty & \text{, otherwise} \end{cases} \right\} \tag{10}$$

The correctness of the algorithm can be shown by induction on the position in the amino acid sequence. Lemma 1 shows that Eqn. (7) gives an optimal score

under the assumption that the previous k codons are fixed. Since Eqn. (10) evaluates all combinations of the previous k codons, Theorem 1 states that an optimal design must be found, if one exists.

Lemma 1. *$P^i_{c_{i-k+1},\ldots,c_{i-1},c_i}$ of Eqn. (7) correctly determines the score of the optimal valid codon design up to the i^{th} codon position, having the codon assignment $c_{i-k+1},\ldots,c_{i-1},c_i$ for the last k codons, given that the maximum length of any motif is $3k$.*

Proof. We will argue by induction. The base case ($i = k$) is trivially *valid* as Eqn. (4) correctly determines the CAI score of the first k codons, by definition.

Assume $P^{i-1}_{c'_{i-k},\ldots,c'_{i-2},c'_{i-1}}$ correctly determines the score of an optimal *valid* codon assignment, up to position $i-1$, having the codon assignment $c'_{i-k},\ldots,c'_{i-2}$, c'_{i-1} for the last k codons. Similarly, assume F^{i-1} and D^{i-1} are also correct for the corresponding codon assignment. When moving one position ahead, from $i-1$ to i, we must consider the case of any new motifs we may introduce. By Observation 1, any new motif which ends within codon c_i could not extend past codon c_{i-k}. There are at most 6 possible codon assignments to position c_{i-k} that can directly precede a specific codon assignment $c_{i-k+1},\ldots,c_{i-1},c_i$ ending at position i as there are at most 6 codons for any amino acid. Therefore, the optimal assignment(s) to c_{i-k} must be a subset of these possibilities. $M'_{\mathcal{F}}\left(\lambda_{c_{i-k}}(\alpha_{i-k})\ldots\lambda_{c_{i-1}}(\alpha_{i-1})\lambda_{c_i}(\alpha_i)\right)$ calculates the number of new forbidden motifs introduced in the codon assignment $c_{i-k},\ldots,c_{i-1},c_i$ which end in codon c_i. By our assumption, $F^{i-1}_{c_{i-k},\ldots,c_{i-2},c_{i-1}}$ correctly determines the minimum number of forbidden motifs having codon assignment $c_{i-k},\ldots,c_{i-2},c_{i-1}$, ending at position $i-1$. Therefore, the sum of these two quantities correctly determines the minimum number of forbidden motifs. As codons $c_{i-k+1},\ldots,c_{i-1},c_i$ are fixed, and Eqn. (5) evaluates every possible assignment to c_{i-k} to determine a minimum, then it must be the case that $F^i_{c_{i-k+1},\ldots,c_{i-1},c_i}$ is the minimum number of forbidden motifs up to position i, having the codon assignment $c_{i-k+1},\ldots,c_{i-1},c_i$ for the last k codons. We argue similarly for $D^i_{c_{i-k+1},\ldots,c_{i-1},c_i}$ in Eqn. (6) with the addition that any assignment of c_{i-k} also be forbidden motif minimal ensured by line 1 of the equation.

Finally, consider $P^i_{c_{i-k+1},\ldots,c_{i-1},c_i}$. Line 1 of Eqn. (10) assigns the value $-\infty$ if the codon assignment $c_{i-k},\ldots,c_{i-1},c_i$ is not *valid*. For all assignments which are *valid*, the equation (line 2) determines the CAI score by multiplying the optimal score up to position $i-1$ (guaranteed optimal by our assumption) with the fitness of the codon represented by c_i for amino acid α_i. Since every assignment to codon c_{i-k} is evaluated and the maximum is determined, then it must be the case that $P^i_{c_{i-k+1},\ldots,c_{i-1},c_i}$ correctly determines the score of the optimal *valid* codon design up to the i$^{\text{th}}$ codon position, having the codon assignment $c_{i-k+1},\ldots,c_{i-1},c_i$ for the last k codons, given that the maximum length of any motif is $3k$. □

Theorem 1. *$\widetilde{P^i_k}$ of Eqn. (10) correctly determines the score of an optimal* valid *codon design, with respect to CAI value, up to the i^{th} codon, given that the maximum length of any motif is $3k$.*

Proof. Lemma 1 guarantees that $P^i_{c_{i-k+1},\ldots,c_{i-1},c_i}$ correctly determines the score of the optimal *valid* codon design up to the i^{th} codon, having the codon assignment $c_{i-k+1},\ldots,c_{i-1},c_i$ for the last k codons, given that the maximum length of any motif is $3k$. Therefore, if every possible assignment of the last k codons is evaluated, a maximum of 6^k possibilities, the score of an optimal *valid* codon design ending at position i can easily be determined.

First, consider that $\widetilde{F^i_k}$ correctly determines the minimum number of forbidden motifs possible, up to position i, by evaluating all possible assignments of that last k codons. Similarly, $\widetilde{D^i_k}$ evaluates the maximum number of desired motifs possible, by first ensuring that the minimum number of forbidden motifs criteria is satisfied. Finally, by evaluating all possible codon assignments of the last k codons, and determining the maximum score of all those which are *valid*, $\widetilde{P^i_k}$ must determine the optimal *valid* CAI score, up to position i. □

Under the assumption that the maximum length of any motif is constant, Theorem 2 proves that the overall time and space complexity is linear.

Theorem 2. *The dynamic programming algorithm for CAI optimization finds a valid nucleic acid sequence design for an amino acid sequence A in $O(|A|+h)$ time and $O(|A|+h)$ space, where h is the total length of forbidden and desired motifs and all motifs are of constant length.*

4 Empirical Results

Data Set. We use a filtered set of the 3,891 CDS (coding DNA sequence) regions of the GENECODE subset of the ENCODE dataset [15] (version hg17 NCBI build 35). This curated dataset comprises approximately 1% of the human genome and is representative of several its characteristics such as distribution of gene lengths and GC composition (54.31%). After filtering any sequences less than 75 bases in length, the remaining 3,157 CDS regions range in length from 75 to 8186 bases, averaging 173 bases with 267 bases standard deviation.

Codon Frequencies. In all cases, we use the codon frequencies of *Escherichia coli* as reported by the Codon Usage Database [http://www.kazusa.or.jp/codon].

Implementation and Hardware. All algorithms were implemented in C++ and compiled with g++ (GCC 4.1.0). Experiments were run on our reference Pentium IV 2.4 GHz processor machines, with 1GB main memory and 256 Kb of CPU cache, running SUSE Linux version 10.1.

4.1 Results

To evaluate the effectiveness and efficiency of our algorithm, a forbidden and desired motif set were constructed which could be considered typical in practice.

It is common for a gene synthesis experiment to use a single restriction enzyme. Furthermore, for reasons affecting gene expression, a common task is the removal of polyhomomeric regions (consecutive repeat region of identical nucleotides). Therefore, we have created a forbidden motif set containing ten elements including GAGTC, GACTC, AAAA, TTTT, GGGG, and CCCC where GAGTC is the motif for the *MlyI* restriction enzyme, GACTC is its reverse complement and the other motifs ensure no polyhomomeric regions greater than length three are permitted. The other four elements of the forbidden motif set (not shown) are immuno-inhibitory motifs originally used in the work of Satya *et. al* [13]. That work also used a desired motif set consisting entirely of thirty-three immuno-stimulatory motifs. We use this same desired motif set in our study.

Performance of the CAI Optimization Algorithm

Results are shown for all 3,157 sequences in Figure 2. On the left side of the figure, the difference in optimal CAI value and the original CAI value of each sequence, when forbidden motifs are minimized, is plotted against sequence length. Desired motifs were not considered. For all sequences, the CAI value is improved compared with the original, with an average improvement of approximately 0.27. Shown on the right is the difference in CAI value for each sequence when the forbidden motifs are minimized and then the desired motifs are maximized. For this case, the average improvement of CAI value drops to 0.18, with only 12 sequences (0.4%) reporting a worse CAI value than the original. A summary of CAI statistics is presented in Table 1. In virtually all cases, forbidden motifs were eliminated entirely. Less than 2% of all sequences contained more than one forbidden motif after optimization, with only 0.6% containing more than two. On average 10 motifs were added to optimized sequences, when desired motifs were considered. These results demonstrate that it is possible to engineer motifs while still optimizing codon usage considerably. The runtime of the algorithm scales linearly with sequence length as would be expected. Considering desired motifs, in addition to forbidden motifs, increases run-time by a small constant

Table 1. The mean values and standard deviations (averaged over 3,157 sequences) of CAI score, number of forbidden motifs, and number of desired motifs are shown for the original sequences (wild-types), the optimized sequences with forbidden motifs minimized and the optimized sequences with forbidden motifs minimized and then desired motifs maximized.

motif sets	CAI value (std. dev.)	# forbidden (std. dev.)	# desired (std. dev.)
none (wild-types)	0.6477 (0.06)	9.2372 (16.24)	0.4869 (1.06)
forbidden	0.9161 (0.04)	0.1384 (0.45)	0 (0.00)
forbidden and desired	0.8280 (0.05)	0.1384 (0.45)	10.1324 (14.84)

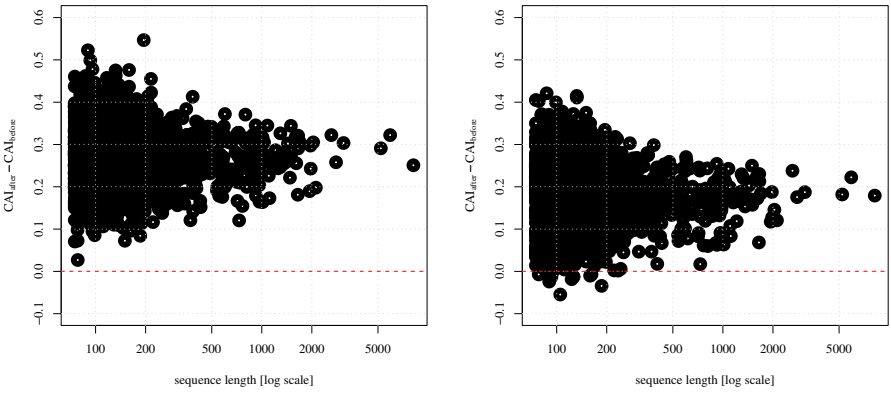


Fig. 2. Results are shown for the difference between the optimal CAI value and the original CAI value, plotted against sequence length, for each of the 3,157 sequences. On the left, results are shown when only the forbidden motif set is considered. The right side shows the results when both the forbidden and desired motif sets are considered.

factor, on average. In the worst case, the algorithm terminates in 0.43 CPU seconds for the longest sequence (8,141 bases).

5 Conclusions

In this work we have presented the first linear time and space algorithm for the problem of optimizing the codon adaptation index (CAI) value of a gene. The algorithm provides a guarantee that codon designs will be found which have a minimum number of forbidden motifs from some user defined set. The algorithm is also capable of adding desirable motifs, when applicable. A formal proof of correctness and time and space analysis was given. An extensive empirical analysis of the algorithm has shown it to be highly effective and efficient in practice. An efficient algorithm is a crucial first step towards designing genes while considering other important sequence features. For instance, designing genes with a guarantee that the resulting nucleic acid sequence does not form stable nucleic acid secondary structure is an interesting future direction, and one that may greatly effect translational efficiency.

Acknowledgments. The authors would like to thank the anonymous reviewers for their constructive suggestions to improve the presentation of this manuscript.

References

1. Aho, A.V.: Algorithms for finding patterns in strings, pp. 255–300. MIT Press, Cambridge (1990)
2. Fuglsang, A.: Codon optimizer: a freeware tool for codon optimization. Protein Expression and Purification 31(2), 247–249 (2003)

3. Gao, W., Rzewski, A., Sun, H., Robbins, P.D., Gambotto, A.: Upgene: Application of a web-based dna codon optimization algorithm. Biotechnology Progress 20(2), 443–448 (2004)
4. Grote, A., Hiller, K., Scheer, M., Münch, R., Nörtemann, B., Hempel, D.C., Jahn, D.: Jcat: a novel tool to adapt codon usage of a target gene to its potential expression host. Nucleic Acids Research 33(web Server issue), 526–531 (2005)
5. Gusfield, D.: Algorithms on strings, trees, and sequences. Cambridge Press, New York (1997)
6. Gustafsson, C., Govindarajan, S., Minshull, J.: Codon bias and heterologous protein expression. Trends in Biotechnology 22(7), 346–353 (2004)
7. Holm, L.: Codon usage and gene expression. Nucleic Acids Research 14(7), 3075–3087 (1986)
8. Hoover, D.M., Lubkowski, J.: Dnaworks: an automated method for designing oligonucleotides for pcr-based gene synthesis. Nucleic Acids Research 30(10), e43 (2002)
9. Jayaraj, S., Reid, R., Santi, D.V.: Gems: an advanced software package for designing synthetic genes. Nucleic Acids Research 33(9), 3011–3016 (2005)
10. Kane, J.F.: Effects of rare codon clusters on high-level expression of heterologous proteins in escherichia coli. Current Opinion in Biotechnology 6(5), 494–500 (1995)
11. Lithwick, G., Margalit, H.: Hierarchy of sequence-dependent features associated with prokaryotic translation. Genome Research 13(12), 2665–2673 (2003)
12. Puigbo, P., Guzman, E., Romeu, A., Garcia-Vallve, S.: OPTIMIZER: a web server for optimizing the codon usage of DNA sequences. Nucleic Acids Research 35(suppl.2), W126–W131 (2007)
13. Satya, R.V., Mukherjee, A., Ranga, U.: A pattern matching algorithm for codon optimization and cpg motif-engineering in dna expression vectors. In: CSB 2003: Proceedings of the IEEE Computer Society Conference on Bioinformatics, pp. 294–305. IEEE Computer Society, Washington, DC, USA (2003)
14. Sharp, P.M., Li, W.H.: The codon adaptation index–a measure of directional synonymous codon usage bias, and its potential applications. Nucleic Acids Research 15(3), 1281–1295 (1987)
15. The ENCODE Consortium: The ENCODE (ENCyclopedia of DNA elements) project. Science 306(5696), 636–640 (2004)
16. Varenne, S., Lazdunski, C.: Effect of distribution of unfavourable codons on the maximum rate of gene expression by an heterologous organism. Journal of Theoretical Biology 120(1), 99–110 (1986)
17. Villalobos, A., Ness, J.E., Gustafsson, C., Minshull, J., Govindarajan, S.: Gene Designer: a synthetic biology tool for constructing artificial DNA segments. BMC Bioinformatics 7, 285 (2006)
18. Wu, G., Bashir-Bello, N., Freeland, S.J.: The synthetic gene designer: a flexible web platform to explore sequence manipulation for heterologous expression. Protein Expression and Purification 47(2), 441–445 (2006)

An Algorithm for Road Coloring

A.N. Trahtman

Bar-Ilan University, Dep. of Math., 52900, Ramat Gan, Israel
trakht@macs.biu.ac.il

Abstract. A coloring of edges of a finite directed graph turns the graph into a finite-state automaton. The synchronizing word of a deterministic automaton is a word in the alphabet of colors (considered as letters) of its edges that maps the automaton to a single state. A coloring of edges of a directed graph of uniform outdegree (constant outdegree of any vertex) is synchronizing if the coloring turns the graph into a deterministic finite automaton possessing a synchronizing word.

The road coloring problem is the problem of synchronizing coloring of a directed finite strongly connected graph of uniform outdegree if the greatest common divisor of the lengths of all its cycles is one. The problem posed in 1970 has evoked noticeable interest among the specialists in the theory of graphs, automata, codes, symbolic dynamics as well as among the wide mathematical community.

A polynomial time algorithm of $O(n^3)$ complexity in the worst case and quadratic in the majority of studied cases for the road coloring of the considered graph is presented below. The work is based on the recent positive solution of the road coloring problem. The algorithm was implemented in the freeware package TESTAS.

Keywords: algorithm, road coloring, graph, deterministic finite automaton, synchronization.

Introduction

The road coloring problem was stated almost 40 years ago [2], [1] for a strongly connected directed finite deterministic graph of uniform outdegree where the greatest common divisor (gcd) of the lengths of all its cycles is one. The edges of the graph being unlabelled, the task is to find a labelling of the edges that turns the graph into a deterministic finite automaton possessing a synchronizing word. The outdegree of the vertex can be considered also as the size of an alphabet where the letters denote colors.

The condition on gcd is necessary [1], [9]. It can be replaced by the equivalent property that there does not exist a partition of the set of vertices on subsets V_1, V_2, ..., $V_k = V_1$ ($k > 2$) such that every edge which begins in V_i has its end in V_{i+1} [9], [16].

Together with the Černy conjecture [7], [8], [15], [20] the road coloring problem used to belong to the most fascinating problems in the theory of finite automata.

C.S. Iliopoulos and W.F. Smyth (Eds.): IWOCA 2011, LNCS 7056, pp. 349–360, 2011.
© Springer-Verlag Berlin Heidelberg 2011

The popular Internet Encyclopedia "Wikipedia" mentioned it many years on the list of the most interesting unsolved problems in mathematics.

For some results in this area, see [5], [6], [10], [11], [12], [13], [14], [16], [17], [19]. A detailed history of investigations can be found in [6]. The final positive solution of the problem is stated in [24].

An algorithm for road coloring oriented on DNA computing [13] is based on the massive parallel computing of sequences of length $O(n^3)$. The implementation of the algorithm as well as the implementation of effective DNA computing is still an open problem.

Another new algorithm for road coloring (ArXiv [4]) as well as our algorithm below is based on the proof of [24]. This proof is constructive and leads to an algorithm that finds a synchronized labelling with cubic worst-case time complexity. Both of the above mentioned algorithms use concepts and ideas of the considered proof together with the concepts from [9], [14], but use different methods to reduce the time complexity. A skillful study of the graph was added in [4].

The presented algorithm for the road coloring (see also ArXiv [22]) reduces the time complexity with the help of the study of two cycles with common vertex (Lemma 10). It gives us the possibility to reduce quite often the time complexity.

The theorems and lemmas from [24] and [23] are presented below without proof. The proofs are given only for new (or modified) results. The time complexity of the algorithm for a graph with n vertices and d outgoing edges of any vertex is $O(n^3d)$ in the worst case and quadratic in the majority of the studied cases. The space complexity is quadratic. This is the first embedded algorithm for road coloring.

The description of the algorithm is presented below together with some pseudo codes of the implemented subroutines. The algorithm is implemented in the freeware package TESTAS (http://www.cs.biu.ac.il/∼trakht/syn.html) [25]. The easy access to the package ensures the possibility to everybody to verify the considered algorithm.

The role of the road coloring is substantial also in education. "The Road Coloring Conjecture makes a nice supplement to any discrete mathematics course" [18]. The realization of the algorithm is demonstrated on the basis of a linear visualization program [25] and can analyze any kind of input graph.

Preliminaries

As usual, we regard a directed graph with letters assigned to its edges as a finite automaton, whose input alphabet Σ consists of these letters. The graph is called *a transition graph* of the automaton. The letters from Σ can be considered as colors and the assigning of colors to edges will be called *coloring*.

A finite directed strongly connected graph with constant outdegree of all its vertices where the gcd of lengths of all its cycles is one will be called an *AGW graph* (as introduced by Adler, Goodwyn and Weiss).

We denote by $|P|$ the size of the subset P of states of an automaton (of vertices of a graph).

If there exists a path in an automaton from the state $\mathbf{p}$ to the state $\mathbf{q}$ and the edges of the path are consecutively labelled by $\sigma_1, ..., \sigma_k$, then for $s = \sigma_1 ... \sigma_k \in \Sigma^+$ we shall write $\mathbf{q} = \mathbf{p}s$.

Let Ps be the set of states $\mathbf{p}s$ for $\mathbf{p} \in P$, $s \in \Sigma^+$. For the transition graph Γ of an automaton, let Γs denote the map of the set of states of the automaton.

A word $s \in \Sigma^+$ is called a *synchronizing* word of the automaton with transition graph Γ if $|\Gamma s| = 1$.

A coloring of a directed finite graph is *synchronizing* if the coloring turns the graph into a deterministic finite automaton possessing a synchronizing word.

Bold letters will denote the vertices of a graph and the states of an automaton.

A pair of distinct states $\mathbf{p}, \mathbf{q}$ of an automaton (of vertices of the transition graph) will be called *synchronizing* if $\mathbf{p}s = \mathbf{q}s$ for some $s \in \Sigma^+$. In the opposite case, if $\mathbf{p}s \neq \mathbf{q}s$ for any s, we call the pair a *deadlock*.

A synchronizing pair of states $\mathbf{p}$, $\mathbf{q}$ of an automaton is called *stable* if for any word u the pair $\mathbf{p}u, \mathbf{q}u$ is also synchronizing [9], [14].

We call the set of all outgoing edges of a vertex a *bunch* if all these edges are incoming edges of only one vertex.

The subset of states (of vertices of the transition graph Γ) of maximal size such that every pair of states from the set is a deadlock will be called an *F-clique*.

1 Some Properties of F-Cliques and Stable Pairs

The road coloring problem was formulated for *AGW* graphs [1] and only such graphs are considered in Sections 1 and 2.

Let us recall that a binary relation ρ on the set of the states of an automaton is called *congruence* if ρ is equivalence and for any word u from $\mathbf{p}\ \rho\ \mathbf{q}$ follows $\mathbf{p}u\ \rho\ \mathbf{q}u$. Let us formulate an important result from [9], [14] in the following form:

Theorem 1. *[14] Let us consider a coloring of an AGW graph Γ. Let ρ be the transitive and reflexive closure of the stability relation on the obtained automaton. Then ρ is a congruence relation, Γ/ρ is also an AGW graph and a synchronizing coloring of Γ/ρ implies a synchronizing recoloring of Γ.*

Lemma 1. [24], [9] *Let F be an F-clique of some coloring of an AGW graph Γ. For any word s the set Fs is also an F-clique and any state $\mathbf{p}$ belongs to some F-clique.*

Lemma 2. *Let A and B (with $|A| > 1$) be distinct F-cliques of some coloring of an AGW graph Γ such that $|A| - |A \cap B| = 1$. Then for all $\mathbf{p} \in A \setminus A \cap B$ and $\mathbf{q} \in B \setminus A \cap B$, the pair $(\mathbf{p}, \mathbf{q})$ is stable.*

Proof. By the definition of an F-clique, $|A| = |B|$ and $|B| - |A \cap B| = 1$, too. If the pair of states $\mathbf{p} \in A \setminus B$ and $\mathbf{q} \in B \setminus A$ is not stable, then for some word s the pair $(\mathbf{p}s, \mathbf{q}s)$ is a deadlock. Any pair of states from the F-clique A and from the F-clique B, as well as from the F-cliques As and Bs, is a deadlock. So any pair of states from the set $(A \cup B)s$ is a deadlock. One has $|(A \cup B)s| = |As| + 1 = |A| + 1 > |A|$. So the size of the set $(A \cup B)s$ of deadlocks is greater than the maximal size of F-clique. Contradiction.

Lemma 3. *If some vertex of an AGW graph Γ has two incoming bunches, then the origins of the bunches form a stable pair by any coloring.*

Proof. If a vertex $\mathbf{p}$ has two incoming bunches from $\mathbf{q}$ and $\mathbf{r}$, then the couple $\mathbf{q}$, $\mathbf{r}$ is stable for any coloring because $\mathbf{q}\sigma = \mathbf{r}\sigma = \mathbf{p}$ for any $\sigma \in \Sigma$.

2 The Spanning Subgraph of an *AGW* Graph

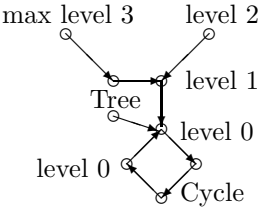
Definition 4. *Let us call a subgraph S of an AGW graph Γ, a spanning subgraph of Γ, if S contains all vertices of Γ and if each vertex has exactly one outgoing edge. (In usual graph-theoretic terms it is a 1-outregular spanning subgraph).*

A maximal subtree of a spanning subgraph S with its root on a cycle from S and having no common edges with the cycles of S is called a tree *of S.*

The length of a path from a vertex $\mathbf{p}$ through the edges of the tree of the spanning set S to the root of the tree is called a level *of $\mathbf{p}$ in S.*

A tree with a vertex of maximal level is called a maximal tree.

Remark 5. *Any spanning subgraph S consists of disjoint cycles and trees with roots on the cycles. Any tree and cycle of S is defined identically. The level of the vertices belonging to some cycle is zero. The vertices of the trees except the roots have positive level. The vertices of maximal positive level have no incoming edge in S. The edges labelled by a given color defined by any coloring form a spanning subgraph. Conversely, for each spanning subgraph, there exists a coloring and a color such that the set of edges labelled with this color corresponds to this spanning subgraph.*



Lemma 6. [24] *Let N be a set of vertices of maximal level in some tree of the spanning subgraph S of an AGW graph Γ. Then, via a coloring of Γ such that all edges of S have the same color α, for any F-clique F holds $|F \cap N| \leq 1$.*

Lemma 7. [24] *Let Γ be an AGW graph with a spanning subgraph R which is a union of cycles (without trees). Then the non-trivial graph Γ has another spanning subgraph with exactly one maximal tree.*

Lemma 8. *Let R be a spanning subgraph of an AGW graph Γ. Let T be a maximal tree of R with a vertex $\mathbf{p}$ of maximal positive level L and with a root $\mathbf{r}$ on a cycle H of R. Let us change the spanning subgraph by means of the following flips:*

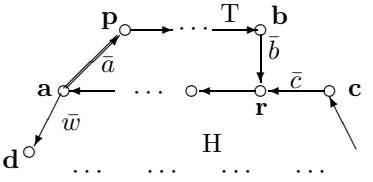
1)an edge $\bar{a} = \mathbf{a} \to \mathbf{p}$ *replaces the edge* $\bar{d} = \mathbf{a} \to \mathbf{d}$ *of* R *for appropriate vertices* $\mathbf{a}$ *and* $\mathbf{d} \neq \mathbf{p}$,

2) replacing edge $\bar{b} = \mathbf{b} \to \mathbf{r}$ *of* T *by an edge* $\mathbf{b} \to \mathbf{x}$ *for appropriate vertices* $\mathbf{b}$ *and* $\mathbf{x} \neq \mathbf{r}$,

3) replacing edge $\bar{c} = \mathbf{c} \to \mathbf{r}$ *of* H *by an edge* $\mathbf{c} \to \mathbf{x}$ *for appropriate vertices* $\mathbf{c}$ *and* $\mathbf{x} \neq \mathbf{r}$.

Suppose that one or two consecutive flips do not increase the number of edges in cycles (Condition) and no vertex of* Γ *has two incoming bunches (Condition**). Then there exists a spanning subgraph with a single maximal non-trivial tree.*

Proof. In view of Lemma 7, suppose that R has non-trivial trees. Further consideration is necessary only if the maximal tree T is not single.



Our aim is to increase the maximal level L using the three aforesaid flips. If one of the flips does not succeed, let us go to the next, assuming the situation in which the previous fails, and excluding the successfully studied cases. We check at most two flips together. Let us begin from

the edge $\bar{a}$) Suppose first $\mathbf{a} \notin H$. If $\mathbf{a}$ belongs to the path in T from $\mathbf{p}$ to $\mathbf{r}$ then a new cycle with part of the path and the edge $\mathbf{a} \to \mathbf{p}$ is added to R extending the number of vertices in its cycles in spite of Condition* of lemma. In the opposite case the level of $\mathbf{a}$ is $L + 1$ in a single maximal tree.

So let us assume $\mathbf{a} \in H$. In this case the vertices $\mathbf{p}$, $\mathbf{r}$ and $\mathbf{a}$ belong to a cycle H_1 of a new spanning subgraph R_1 obtained by removing $\bar{d}$ and adding $\bar{a}$. So we have the cycle $H_1 \in R_1$ instead of $H \in R$. If the length of the path from $\mathbf{r}$ to $\mathbf{a}$ in H is r_1, then H_1 has length $L + r_1 + 1$. A path from $\mathbf{r}$ to $\mathbf{d}$ of the cycle H remains in R_1. Suppose that its length is r_2. So the length of the cycle H is $r_1 + r_2 + 1$. The length of the cycle H_1 is not greater than the length of H in view of Condition*. So $r_1 + r_2 + 1 \geq L + r_1 + 1$, whence $r_2 \geq L$. If $r_2 > L$, then the length r_2 of the path from $\mathbf{d}$ to $\mathbf{r}$ in a tree of R_1 (as well as the level of $\mathbf{d}$) is greater than L. The tree containing $\mathbf{d}$ is the desired single maximal tree.

So we can assume for further consideration that $L = r_2$ and $\mathbf{a} \in H$. An analogous statement can be stated for any maximal tree.

The edge $\bar{b}$) Suppose that the set of outgoing edges of the vertex $\mathbf{b}$ is not a bunch. So one can replace in R the edge $\bar{b}$ by an edge $\bar{v} = \mathbf{b} \to \mathbf{v}$ ($\mathbf{v} \neq \mathbf{r}$).

The vertex $\mathbf{v}$ could not belong to T because in this case a new cycle is added to R in spite of Condition*.

If the vertex **v** belongs to another tree of R but not to the cycle H, then T is a part of a new tree T_1 with a new root of a new spanning subgraph R_1 and the path from **p** to the new root has a length greater than L. Therefore the tree T_1 is the unique maximal tree in R_1.

If **v** belongs to some cycle $H_2 \neq H$ in R, then together with replacing $\bar{b}$ by $\bar{v}$, we also replace the edge $\bar{d}$ by $\bar{a}$. So we extend the path from **p** to the new root **v** of H_2 at least by the edge $\bar{a} = \mathbf{a} \to \mathbf{p}$ and there is a unique maximal tree of level $L_1 > L$ which contains the vertex **d**.

Now it remains only the case when **v** belongs to the cycle H. The vertex **p** also has level L in a new tree T_1 with root **v**. The only difference between T and T_1 (just as between R and R_1) is the root and the incoming edge of this root. The new spanning subgraph R_1 has the same number of vertices in their cycles just as does R. Let $r_2^{'}$ be the length of the path from **d** to $\mathbf{v} \in H$.

For the spanning subgraph R_1, one can obtain $L = r_2^{'}$ just as it was done earlier in the case of the edge $\bar{a}$) for R. From $\mathbf{v} \neq \mathbf{r}$ follows $r_2^{'} \neq r_2$, though $L = r_2^{'}$ and $L = r_2$.

So for further consideration suppose that the set of outgoing edges of the vertex **b** is a bunch to **r**.

The edge $\bar{c}$) The set of outgoing edges of the vertex **c** is not a bunch in virtue of Condition** (**r** has another bunch from **b**.)

Let us replace in R the edge $\bar{c}$ by an edge $\bar{u} = \mathbf{c} \to \mathbf{u}$ such that $\mathbf{u} \neq \mathbf{r}$. The vertex **u** could not belong to the tree T because one has in this case a cycle with all vertices from H and some vertices of T whence its length is greater than $|H|$ and so the number of vertices in the cycles of a new spanning subgraph grows in spite of Condition*.

If the vertex **u** does not belong to T, then the tree T is a part of a new tree with a new root. The path from **p** to the new root is extended at least by a part of H starting at the former root **r**. The new level of **p** therefore is maximal and greater than the level of any vertex in another tree.

Thus in any case we obtain a spanning subgraph with a single non-trivial maximal tree.

Lemma 9. *For some coloring of any AGW graph Γ, there exists a stable pair of states.*

Proof. We exclude the case of two incoming bunches of a vertex in virtue of Lemma 3. There exists a coloring such that for some color α, the corresponding spanning subgraph R has maximum edges in cycles.

By Lemma 8, we must consider now a spanning subgraph R with a single maximal tree T. Let the root **r** of T belong to the cycle C.

By Lemma 1, in a strongly connected transition graph for every word s and F-clique F of size $|F| > 1$, the set Fs also is an F-clique of the same size and for any state **p** there exists an F-clique F such that $\mathbf{p} \in F$.

In particular, some F-clique F has a non-empty intersection with the set N of vertices of maximal level L. The set N belongs to one tree, whence by Lemma 6 $|N \cap F| = 1$. Let $\mathbf{p} \in N \cap F$.

The word α^{L-1} maps F on an F-clique F_1 of size $|F|$. One has $|F_1 \setminus C| = 1$ because any sequence of length $L-1$ of edges of color α in any tree of R leads to a cycle. For the set N of vertices of maximal level, $N\alpha^{L-1} \not\subseteq C$ holds. So $|N\alpha^{L-1} \cap F_1| = |F_1 \setminus C| = 1$, $\mathbf{p}\alpha^{L-1} \in F_1 \setminus C$ and $|C \cap F_1| = |F_1| - 1$.

Let the integer m be a common multiple of the lengths of all considered cycles colored by α. So for any $\mathbf{r}$ in C as well as in $F_1 \cap C$ holds $\mathbf{r}\alpha^m = \mathbf{r}$. Let F_2 be $F_1\alpha^m$. We have $F_2 \subseteq C$ and $C \cap F_1 = F_1 \cap F_2$.

Thus the two F-cliques F_1 and F_2 of size $|F_1| > 1$ have $|F_1| - 1$ common vertices. So $|F_1 \setminus (F_1 \cap F_2)| = 1$, whence by Lemma 2, the pair of states $\mathbf{p}\alpha^{\mathbf{L-1}}$ from $F_1 \setminus (F_1 \cap F_2)$ and $\mathbf{q}$ from $F_2 \setminus (F_1 \cap F_2)$ is stable. It is obvious that $\mathbf{q} = \mathbf{p}\alpha^{\mathbf{L+m-1}}$.

Theorem 2. [24] *Every AGW graph has a synchronizing coloring.*

Theorem 3. [23] *Let every vertex of a strongly connected directed graph Γ have the same number of outgoing edges. Then Γ has synchronizing coloring if and only if the greatest common divisor of lengths of all its cycles is one.*

The goal of the following lemma is to reduce the complexity of the algorithm.

Lemma 10. *Let Γ be an AGW graph having two cycles C_u and C_v. Suppose that either $C_u \cap C_v = \{\mathbf{p}_1\}$ or $C_u \cap C_v = \{\mathbf{p}_k, ..., \mathbf{p}_1\}$, where all incoming edges of $\mathbf{p}_i$ develop a bunch from $\mathbf{p}_{i+1}$ $(i < k)$.*

Let $u \in C_u$ and $v \in C_v$ be the distinct edges of the cycles C_u and C_v leaving $\mathbf{p}_1$. Let R_u be a spanning subgraph with all edges from C_u and C_v except u. The spanning subgraph R_v is obtained from R_u by removing v and adding u.

Then at least one of two spanning subgraphs R_u, R_v has a unique maximal tree whose root is $\mathbf{p}_1$.

Proof. Let us add to R_u the edge u and consider a set of trees with roots on the cycles C_u and C_v. The trees have no common vertices and have no vertices except a root on the cycles C_u and C_v. The same set of trees can be obtained by adding the edge v to R_v.

Let us define the levels of vertices of a tree as in the case of a spanning subgraph and consider the set of maximal trees (the trees with a maximal vertex level).

If all maximal trees have a common root, then R_u (and also R_v) is a spanning subgraph with a unique maximal tree.

If maximal trees have different roots, then let as take a maximal tree T with root $\mathbf{r}$ such that the length of the path P from $\mathbf{r}$ to $\mathbf{p}_1$ on the cycle C_u (or C_v) is maximal. If P belongs to C_u, then the tree T is extended by the path P, whence R_u has a unique maximal tree. In the opposite case, R_v has a unique maximal tree.

3 The Algorithm for Synchronizing Coloring

Let us start with transition graph of an arbitrary deterministic complete finite automaton.

3.1 Preliminary Steps

The study is based on Theorem 3. A synchronizing graph has a sink strongly connected component (SCC). Our aim is to reduce the study to sink SCC (if it exists) in order to remove non-synchronizing graphs without sink SCC and then check the condition on gcd.

The function CheckSinkSCC verifies the existence of sink SCC. We use the linear algorithm for finding strongly connected components SCC [3], [21].

Then we remove all SCC as having outgoing edges to other SCC. If only one SCC remains then let us continue. In the opposite case a synchronizing coloring does not exist.

We study a strongly connected graph (with one SCC). The function FindGCDofCycles finds the great common divisor (gcd) of lengths of cycles of the automaton and verifies the necessary conditions of synchronizability ($gcd = 1$).

Let $\mathbf{p}$ be an arbitrary fixed vertex. Suppose $d(\mathbf{p}) = 1$. Then we use a depth-first search from $\mathbf{p}$. For an edge $\mathbf{r} \to \mathbf{q}$ where $d(\mathbf{r})$ is already defined and $d(\mathbf{q})$ is not, suppose $d(\mathbf{q}) = d(\mathbf{r}) + 1$. If $d(\mathbf{q})$ is defined, let us add the non-zero difference $abs(d(\mathbf{q}) - 1 - d(\mathbf{r}))$ to the set D. The integer from D is a difference of lengths of two paths from $\mathbf{p}$ to $\mathbf{q}$. In a strongly connected graph, the gcd of all elements of D is also a gcd of lengths of all cycles [2], [23].

If $gcd = 1$ for all integers from D, then the graph has synchronizing coloring. In opposite case the answer is negative. So we reduce the investigation to an AGW graph.

Let us proceed with an arbitrary coloring of such a graph Γ with n vertices and constant outdegree d. The considered d colors define d spanning subgraphs of the graph.

We keep the preimages of vertices and colored edges by any transformation and homomorphism.

If there exists a loop in Γ around a state $\mathbf{r}$, then let us color the edges of a tree whose root is $\mathbf{r}$ with the same color as the color of the loop. The other edges may be colored arbitrarily. The coloring is synchronizing [1]. The function FindLoopColoring finds the coloring.

3.2 Help Subroutines

In the case of two incoming bunches of some vertex, the origins of these bunches develop a stable pair by any coloring (Lemma 3). We merge both vertices in the homomorphic image of the graph (Theorem 1) and obtain according to the theorem a new AGW graph of a smaller size. The pseudo code of corresponding procedure returns two such origins of bunches (a stable pair).

The linear search of two incoming bunches and of the loop can be made at any stage of the algorithm.

The function HomonorphicImage of linear complexity reduces the size of the considered automaton and its transition graph. The congruence classes of the homomorphism are defined by a stable pair (Theorem 1). A new AGW graph of a smaller size will be the output.

The main part of the algorithm needs the parameters of the spanning subgraph: levels of all vertices, the number of vertices (edges) in cycles, trees, next and former vertices. We keep the tree and the cycle of any vertex, the root of the tree. We form the set of vertices of *maximal* level and the set of *maximal* trees. The function FindParameters (spanning subgraph S, parameters) is linear and used by any recoloring step.

The subroutine MaximalTreeToStablePair of linear complexity finds a stable pair in a given spanning subgraph with unique maximal tree. The stable pair consists of two beginnings of incoming edges of the root of the unique maximal tree (Lemma 9).

3.3 A Possibility to Reduce the Complexity

Our algorithm as well as the algorithm of [4] is based on [24]. Only this section essentially differs in both these papers.

If there are two cycles with one common vertex (path) then we use Lemma 10 and find a spanning subgraph with single maximal tree. Then after coloring edges of spanning subgraph by a color α, we find a stable pair (beginnings of two incoming edges to the root of the tree).

The function TwoCyclesWithIntersection as a rule returns a pair of cycles with common vertex (path). The vast majority of digraphs contains such a pair of cycles. The goal of the subroutine is to omit the cubic complexity of the algorithm. The search of a stable pair is linear in this case and thus the whole algorithm is quadratic.

```
TwoCyclesWithIntersection (graph G)
levels of all vertices first are negative
level(r) = 1 and add r to stack
for every vertex q from stack
  do
    for every letter β
      do
        add qβ to stack
        if level(qβ) ≥ 0
          level(qβ) =level(q) + 1
            keep the cycle C of vertices qβ, q and break from both cycles
  remove q from stack
for every vertex r
  do
    if r ∉ C level((r) = −1 (for a search of second cycle)
for every vertex q from cycle C
  do
    r = qα
    for every letter β
      do
        if r ≠ qβ break
```

```
21      if r ≠ qβ break
22 add q to stack 1 (possible intersection of two cycles)
23 for every vertex r from stack 1
24   do
25      for every letter β
26        do
27           if level(rβ) < 0
28              level(rβ) =level(r) + 1
29              add rβ to stack 1
30              if rβ = q (found second cycle)
31                 develop trees with roots on both cycles, find maximal trees
32                 color the edge v from q on cycle of maximal tree by color 2
33                 color the edges of trees and both cycles except v by color 1
34                 FindParameters (spanning subgraph of color 1)
35                 MaximalTreeToStablePair (subgraph, p, s)
36                 return p, s (stable pair)
37      remove r from stack 1
38 return False
```

3.4 The Recoloring of the Edges

A repainting of the edges of the transition graph for to obtain a spanning subgraph with single maximal tree is a most complicated part of the algorithm. Let us fix the spanning subgraph R of edges of a given color α. We consider the flips from Lemmas 7 and 8. The flips change R. According to the Lemmas, after at most $3d$ steps either the number of edges in the cycles is growing or there exists a single maximal tree.

The subroutine of pseudo code Flips (spanning subgraph F) returns either a stable pair or enlarges the number of edges in cycles of the spanning subgraph. The subroutine uses linear subroutines FindParameters, MaximalTreeToStablePair and also has linear time complexity $O(nd)$.

We repeat the procedure with pseudo code Flips for a new graph if the number of edges in cycles after the flips grows. In the opposite case, we find a stable pair and then a homomorphic image of a smaller size. For a graph of given size, the complexity of this step is quadratic.

3.5 Main Procedure and Complexity

The Procedure Main uses all above-mentioned linear procedures and returns a synchronizing coloring (if exists) of the graph.

```
Main()
 1 arbitrary coloring of G
 2 if False(CheckSinkSCC(graph G))
 3   return False
 4 if FindLoopColoring(F=SCC of G)
```

```
5   return
6  if False(FindGCDofCycles(SCCF))
7   return False
8   while |G| > 1
9     ifFindLoopColoring(F)
10        change the coloring of generic graph G
11        return
12      for every letter β
13        do
14          if FindTwoIncomingBunches(spanning subgraph,stable pair)
15            HomonorphicImage(automaton A,stable pair,new A)
16            FindParameters (A = new A)
17            break
18          while Flips(spanning subgraph F of color β) = GROWS
19            F = new F
20            if FindTwoIncomingBunches( F,stable pair)
21              HomonorphicImage(automaton A,stable pair,new A)
22              FindParameters (A = new A)
23              break
24          MaximalTreeToStablePair (subgraph, stable pair)
25          HomonorphicImage(automaton A,stable pair,new A)
26          FindParameters (A = new A)
27 change the coloring of G on the base of the last homomorphic image
```

Some of above-mentioned linear subroutines are included in cycles on n and d, sometimes twice on n. So the upper bound of the time complexity is $O(n^3d)$.

Nevertheless, the overall complexity of the algorithm in a majority of cases is $O(n^2d)$. The upper bound $O(n^3d)$ of the time complexity is reached only if the number of edges in the cycles grows slowly, the size of the automaton decreases also slowly, loops do not appear and the case of two ingoing bunches emerges rarely (the worst case). The space complexity is quadratic.

References

1. Adler, R.L., Goodwyn, L.W., Weiss, B.: Equivalence of topological Markov shifts. Israel J. of Math. 27, 49–63 (1977)
2. Adler, R.L., Weiss, B.: Similarity of automorphisms of the torus. Memoirs of the Amer. Math. Soc. 98 (1970)
3. Aho, A., Hopcroft, J., Ulman, J.: The Design and Analisys of Computer Algorithms. Addison-Wesley (1974)
4. Béal, M.P., Perrin, D.: A quadratic algorithm for road coloring. arXiv:0803.0726v2 [cs.DM] (2008)
5. Budzban, G., Mukherjea, A.: A semigroup approach to the Road Coloring Problem. Probability on Alg. Structures. Contemporary Mathematics 261, 195–207 (2000)
6. Carbone, A.: Cycles of relatively prime length and the road coloring problem. Israel J. of Math. 123, 303–316 (2001)

7. Černy, J.: Poznamka k homogenym eksperimentom s konechnymi automatami. Math.-Fyz. Čas. 14, 208–215 (1964)
8. Černy, J., Piricka, A., Rosenauerova, B.: On directable automata. Kybernetika 7, 289–298 (1971)
9. Culik, K., Karhumaki, J., Kari, J.: A note on synchronized automata and Road Coloring Problem. In: 5th Int. Conf. on Developments in Language Theory, Vienna, pp. 459–471 (2001); J. of Found. Comput. Sci.13, 459–471 (2002)
10. Friedman, J.: On the road coloring problem. Proc. of the Amer. Math. Soc. 110, 1133–1135 (1990)
11. Gocka, E., Kirchherr, W., Schmeichel, E.: A note on the road-coloring conjecture. Ars Combin. 49, 265–270 (1998)
12. Hegde, R., Jain, K.: Min-Max theorem about the Road Coloring Conjecture. In: EuroComb 2005, DMTCS Proc., AE, pp. 279–284 (2005)
13. Jonoska, N., Karl, S.A.: A molecular computation of the road coloring problem. DNA Based Computers II, DIMACS Series in DMTCS 44, 87–96 (1998)
14. Kari, J.: Synchronizing Finite Automata on Eulerian Digraphs. In: Sgall, J., Pultr, A., Kolman, P. (eds.) MFCS 2001. LNCS, vol. 2136, p. 432. Springer, Heidelberg (2001)
15. Mateescu, A., Salomaa, A.: Many-Valued Truth Functions, Černy's Conjecture and Road Coloring. Bull. of European Ass. for TCS 68, 134–148 (1999)
16. O'Brien, G.L.: The road coloring problem. Israel J. of Math. 39, 145–154 (1981)
17. Perrin, D., Schŭtzenberger, M.P.: Synchronizing prefix codes and automata, and the road coloring problem. Symbolic Dynamics and Appl. Contemp. Math. 135, 295–318 (1992)
18. Rauff, J.V.: Way back from anywhere: exploring the road coloring conjecture. Math. and Comput. Education 01 (2009)
19. Roman, A.: Decision Version of the Road Coloring Problem Is NP-Complete. In: Kutyłowski, M., Charatonik, W., Gębala, M. (eds.) FCT 2009. LNCS, vol. 5699, pp. 287–297. Springer, Heidelberg (2009)
20. Rystsov, I.C.: Quasioptimal bound for the length of reset words for regular automata. Acta Cybernetica 12, 145–152 (1995)
21. Tarjan, R.E.: Depth first search and linear graph algorithms. SIAM J. Comput. 1, 146–160 (1972)
22. Trahtman, A.N.: A subquadratic algorithm for road coloring. arXiv:0801.2838 v1 [cs.DM] (2008)
23. Trahtman, A.N.: Synchronizing Road Coloring. In: 5-th IFIP WCC-TCS, vol. 273, pp. 43–53. Springer, Heidelberg (2008)
24. Trahtman, A.N.: The road coloring problem. Israel Journal of Math. 172(1), 51–60 (2009)
25. Trahtman, A.N., Bauer, T., Cohen, N.: Linear visualization of a Road Coloring. In: 9th Twente Workshop on Graphs and Comb. Optim. Cologne, pp. 13–16 (2010)

Complexity of the Cop and Robber Guarding Game

Robert Šámal*, Rudolf Stolař, and Tomas Valla**

Charles University, Faculty of Mathematics and Physics,
Institute for Theoretical Computer Science (ITI)
Malostranské nám, 2/25, 118 00, Prague, Czech Republic
{samal,ruda,valla}@kam.mff.cuni.cz

Abstract. The guarding game is a game in which several cops has to guard a region in a (directed or undirected) graph against a robber. The robber and the cops are placed on vertices of the graph; they take turns in moving to adjacent vertices (or staying), cops inside the guarded region, the robber on the remaining vertices (the robber-region). The goal of the robber is to enter the guarded region at a vertex with no cop on it. The problem is to determine whether for a given graph and given number of cops the cops are able to prevent the robber from entering the guarded region. The problem is highly nontrivial even for very simple graphs. It is known that when the robber-region is a tree, the problem is NP-complete, and if the robber-region is a directed acyclic graph, the problem becomes PSPACE-complete [Fomin, Golovach, Hall, Mihalák, Vicari, Widmayer: How to Guard a Graph? Algorithmica, DOI: 10.1007/s00453-009-9382-4]. We solve the question asked by Fomin et al. in the previously mentioned paper and we show that if the graph is arbitrary (directed or undirected), the problem becomes E-complete.

Keywords: pursuit game, cops and robber game, graph guarding game, computational complexity, E-completeness.

1 Introduction and Motivation

The *guarding game* (G, V_C, c), introduced by Fomin et al. [1], is played on a graph $G = (V, E)$ (or directed graph $\overrightarrow{G} = (V, E)$) by two players, the *cop-player* and the *robber-player*, each having his pawns (c cops and one robber, respectively) on V. There is a protected region (also called cop-region) $V_C \subset V$. The remaining region $V \setminus V_C$ is called robber-region and denoted V_R. The robber aims to enter V_C by a move to vertex of V_C with no cop on it. The cops try to prevent the robber from entering a vertex of V_C with no cop on it. The game is played in alternating turns. In the first turn the robber-player places the robber on some vertex of V_R. In the second turn the cop-player places his c cops on vertices of V_C

* Partially supported by grant GA ČR P201/10/P337.
** Supported by the GAUK Project 66010 of Charles University Prague. Supported by ITI, Charles University Prague, under grant 1M0021620808.

C.S. Iliopoulos and W.F. Smyth (Eds.): IWOCA 2011, LNCS 7056, pp. 361–373, 2011.
© Springer-Verlag Berlin Heidelberg 2011

(more cops can share one vertex). In each subsequent turn the respective player can move each of his pawns to a neighbouring vertex of the pawn's position (or leave it where it is). However, the cops can move only inside V_C and the robber can move only on vertices with no cops. At any time of the game both players know the positions of all pawns. The robber-player wins if he is able to move the robber to some vertex of V_C in a finite number of steps. The cop-player wins if the cop-player can prevent the robber-player from placing the robber on a vertex in V_C indefinitely. Note that after exponentially many (in the size of the graph G) turns the positions has to repeat and obviously if the robber can win, he can win in less than $2|V|^{(c+1)}$ turns, as $2|V|^{(c+1)}$ is the upper bound on the number of all possible positions of the robber and all cops, together with the information who is on move.

For a given graph G and guarded region V_C, the task is to find the minimum number c such that cop-player wins.

The guarding game is a member of a big class called the pursuit-evasion games, see, e.g., Alspach [4] for introduction and survey. The discrete version of pursuit-evasion games on graphs is called the Cops-and-Robber game. This game was first defined for one cop by Winkler and Nowakowski [5] and Quilliot [6]. Aigner and Fromme [7] initiated the study of the problem with several cops. The minimum number of cops required to capture the robber is called the cop number of the graph. In this setting, the Cops-and-Robber game can be viewed as a special case of search games played on graphs. Therefore, the guarding game is a natural variant of the original Cops-and-Robber game. The complexity of the decision problem related to the Cops-and-Robbers game was studied by Goldstein and Reingold [11]. They have shown that if the number of cops is not fixed and if either the graph is directed or initial positions are given, then the problem is E-complete. Another interesting variant is the "fast robber" game, which is studied in Fomin et al. [12]. See the annotated bibliography [10] for reference on further topics.

A different well-studied problem, the *Eternal Domination* problem (also known as *Eternal Security*) is strongly related to the guarding game. The objective in the Eternal Domination is to place the minimum number of guards on the vertices of a graph G such that the guards can protect the vertices of G from an infinite sequence of attacks. In response to an attack of an unguarded vertex v, at least one guard must move to v and the other guards can either stay put, or move to adjacent vertices. The Eternal Domination problem is a special case of the guarding game. This can be seen as follows. Let G be a graph on n vertices and we construct a graph H from G by adding a clique K_n on n vertices and connecting the clique and G by n edges which form a perfect matching. The cop-region is $V(G)$ and the robber-region is $V(K_n)$. Now G has an eternal dominating set of size k if and only if k cops can guard $V(G)$. Eternal Domination and its variant have been considered for example in [15,16,17,18,19,20,21,22].

In our paper we focus on the complexity issues of the decision problem related to the guarding game: Given the guarding game $\mathcal{G} = (G, V_C, c)$, who has the

winning strategy? Observe that the task of finding the minimum c such that $\mathcal{G}$ is cop-win is at least as hard as the decision version of the problem.

Let us define the computational problem precisely. The *directed guarding decision problem* is, given a guarding game $(\overrightarrow{G}, V_C, c)$ where $\overrightarrow{G}$ is a directed graph, to decide whether it is a cop-win game or a robber-win game. Analogously, we define the *undirected guarding decision problem* with the difference that the underlying graph G is undirected. The *guarding problem* is, given a a directed or undirected graph G and a cop-region $V_C \subseteq V(G)$, to compute the minimum number c such that the (G, V_C, c) is a cop-win.

The directed guarding decision problem was introduced and studied by Fomin et al. [1]. The computational complexity of the problem depends heavily on the chosen restrictions on the graph G. In particular, in [1] the authors show that if the robber's region is only a path, then the problem can be solved in polynomial time, and when the robber moves in a tree (or even in a star), then the problem is NP-complete. Furthermore, if the robber is moving in a directed acyclic graph, the problem becomes PSPACE-complete. Later Fomin, Golovach and Lokshtanov [13] studied the *reverse guarding game* which rules are the same as in the guarding game, except that the cop-player plays first. They proved in [13] that the related decision problem is PSPACE-hard on undirected graphs. Nagamochi [8] has also shown that that the problem is NP-complete even if V_R induces a 3-star and that the problem is polynomially solvable if V_R induces a cycle. Also, Thirumala Reddy, Sai Krishna and Pandu Rangan have proved [9] that if the robber-region is an arbitrary undirected graph, then the decision problem is PSPACE-hard.

Let us consider the class $\mathrm{E} = \mathrm{DTIME}(2^{O(n)})$ of languages recognisable by a deterministic Turing machine in time $2^{O(n)}$. We consider log-space reductions, this means that the reducing Turing machine is log-space bounded. Very little is known how the class E is related to PSPACE. However, it is known [3] that $\mathrm{E} \neq \mathrm{PSPACE}$. Fomin et al. [1] asked, whether the guarding decision problem for general graphs is PSPACE-complete too. We disprove this conjecture in the following theorem.

Theorem 1. *The directed guarding decision problem is E-complete under log-space reductions.*

Immediately, we get the following corollary.

Corollary 1. *The guarding problem is E-complete under log-space reductions.*

We would like to point out the fact that we can prove Theorem 1 without prescribing the starting positions of players.

We also state Theorem 2, a theorem similar to Theorem 1 for general undirected graphs. Unfortunately, we omit the proof of Theorem 2 due to the page limit imposed on the paper. We define the *guarding game with prescribed starting positions* $\mathcal{G} = (G, V_C, c, S, r)$, where $S : \{1, \ldots, c\} \to V_C$ is the initial placement of cops and $r \in V_R$ is the initial placement of robber. The *undirected guarding decision problem with prescribed starting positions* is, given a guarding game with

prescribed starting positions (G, V_C, c, S, r) where G is an undirected graph, to decide whether it is a cop-win game or a robber-win game. The *directed guarding decision problem with prescribed starting positions* is defined analogously.

Theorem 2. *The undirected guarding decision problem with prescribed starting positions is E-complete under log-space reductions.*

Here, we would like to point out the fact that with the exception of the result in [13], all known hardness results for cops and robbers, or pursuit evasion games are for the directed graph variants of the games [1,11]. For example, the classical Cop and Robbers game was shown to be PSPACE-hard on directed graphs by Goldstein and Reingold in 1995 [11] while for *undirected* graphs, even an NP-hardness result was not known until recently by Fomin, Golovach and Kratochvíl [14].

For the original Cops-and-Robber game, Goldstein and Reingold [11] have proved that if the number c of cops is not fixed and if either the graph is directed or initial positions are given, then the related decision problem is E-complete.

In a sense, we show analogous result for the guarding game as Goldstein and Reingold [11] have shown for the original Cops-and-Robber game. Similarly to Goldstein and Reingold, we can prove the complexity of the undirected guarding decision problem only when having prescribed the initial positions of players. Dealing with this issue now seems to be the main task in this family of games.

2 The Directed Case

In order to prove E-completeness of the directed guarding decision problem, we first note that the problem is in E.

Lemma 1. *The guarding decision problem (directed or undirected) is in E.*

The proof is standard and easy (we just have to realize that the cops are mutually indistinguishable so the number of all configurations is $2^{O(n)}$) and we omit it.

Let us first study the problem after the second move, where both players have already placed their pawns. We reduce the directed guarding decision problem with prescribed starting positions from the following formula-satisfying game $\mathcal{F}$.

A position in $\mathcal{F}$ is a 4-tuple $(\tau, F_R(C,R), F_C(C,R), \alpha)$ where $\tau \in \{1,2\}$, F_R and F_C are formulas in 12-DNF both defined on set of variables $C \cup R$, where C and R are disjoint and α is an initial $(C \cup R)$-assignment. The symbol τ serves only to differentiate the positions where the first or the second player is on move. Player I (II) moves by changing the values assigned to at most one variable in R (C); either player may pass since changing no variable amounts to a "pass". Player I (II) wins if the formula F_R (F_C) is true after some move of player I (II). More precisely, player I can move from $(1, F_R, F_C, \alpha)$ to $(2, F_R, F_C, \alpha')$ in one move if and only if α' differs from α in the assignment given to at most one variable in R and F_C is false under the assignment α; the moves of player II are defined symmetrically.

According to Stockmeyer and Chandra [2], the set of winning positions of player I in the game $\mathcal{F}$ is an E-complete language under log-space reduction.

Let us first informally sketch the reduction from $\mathcal{F}$ to $\mathcal{G}$, i.e., simulating $\mathcal{F}$ by an equivalent guarding game $\mathcal{G}$. The setting of variables is represented by positions of certain cops so that only one of these cops may move at a time (otherwise cop-player loses the game). The variables (or more precisely the corresponding cops) of C are under control of cop-player. However, in spite of being represented by cops, the variables of R are under control of the robber-player by a gadget in the graph $\overrightarrow{G}$, which allows him to force any setting of cops representing R.

When describing the features of various gadgets, we will often use the term *normal scenario*. By normal scenario S of certain gadget (or even the whole game) we mean a flow of the game that imitates the formula game $\mathcal{F}$. The graph G will be constructed in such a way that if the player (both cop-player and robber-player) does not exactly follow the normal scenario S, he loses the game in a few moves.

There are four cyclically repeating phases of the game, determined by the current position of the robber. The normal scenario is that robber cyclically goes through the following phases marked by four special vertices and in different phases he can enter certain gadgets.

1. "Robber Move" (RM): In this step the robber can enter the Manipulator gadget, allowing him to force setting of at most one variable in R.
2. "Robber Test" (RT): In this step the robber may pass through the *Robber Gate* into the protected region V_C, provided that the formula F_R is satisfied under the current setting of variables.
3. "Cop Move" (CM): In this step (and only in this step) one (and at most one) variable cell V_x for $x \in C$ is allowed to change its value. This is realized by a gadget called *Commander*.
4. "Cop Test" (CT): In this step, if the formula F_C is satisfied under the current setting of variables, the cops are able to block the entrance to the protected region forever (by temporarily leaving the *Cop Gate* gadget unguarded and sending a cop to block the entrance to V_C provided by the Robber Gate gadgets).

See Fig. 1 for the overview of the construction.

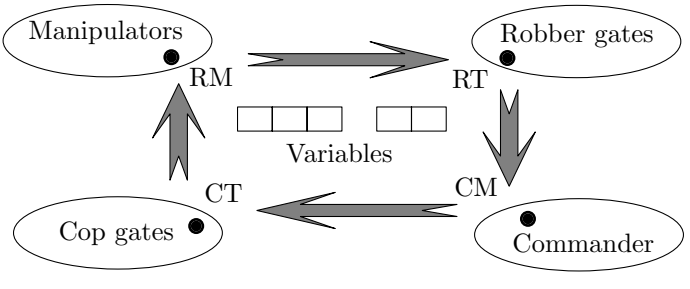


Fig. 1. The sketch of the construction

2.1 The Variable Cells

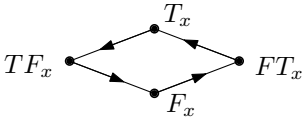


Fig. 2. Variable cell V_x

For every variable $x \in C \cup R$ we introduce a *variable cell* V_x, which is a directed cycle (T_x, TF_x, F_x, FT_x) (see Fig. 2). There is one cop (*variable cop*) located in every V_x and the position of the cop on vertices T_x, F_x represents the boolean values true and false, respectively. The prescribed starting position of the variable cop is T_x if $\alpha(x)$ is true, and F_x otherwise. All the vertices of V_x belong to V_C.

The cells are organised into blocks C and R. The block C is under control of cop-player via the Commander gadget, the block R is represented by cops as well, however, there are the *Manipulator* gadgets allowing the robber-player to force any setting of variables in R, by changing at most one variable in his turn.

Every variable cell V_y, $y \in R$ has assigned the Manipulator gadget M_y. Manipulator M_y consists of directed paths (RM, T'_y, T''_y, T_y) and (RM, F'_y, F''_y, F_y) and edges (T'_y, RT) and (F'_y, RT) (see Fig. 3).

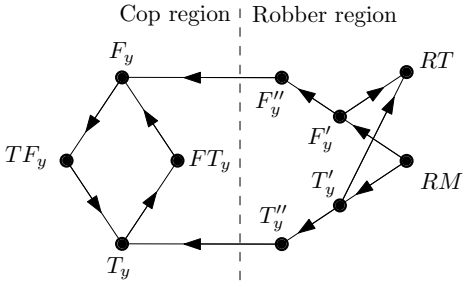


Fig. 3. The Manipulator gadget M_y

The vertices $\{T'_y, F'_y, T''_y, F''_y, RM, RT\} \subset V_R$, the rest belongs to V_C.

Lemma 2. *Let us consider variable cell V_y, $y \in R$, and the corresponding Manipulator M_y. Let the robber be at the vertex RM, let the cop be either on T_y or F_y and suppose no other cop can access any vertex of M_y in less than three moves. Then the normal scenario is following: By entering the vertex T'_y (F'_y), the robber forces the cop to move towards the vertex T_y (F_y). Robber then has to enter the vertex RT.*

Proof. If the cop refuses to move, the robber advances to T''_y or F''_y and easily reaches V_C before the cop can block him. On the other hand, if the robber moves to T''_y or F''_y even though the cop moved towards the opposite vertex, then cop finishes his movement to the opposite vertex and robber cannot move anymore. □

Note that this is not enough to ensure that the variable cop really reaches the opposite vertex and that only one variable cop from variable cells can move. We deal with this issue later.

When changing variables of C, we have to make sure that at most one variable is changed at a time. We ensure that by the gadget Commander (see Fig. 4), connected to every V_x, $x \in C$. It consists of vertices $\{f_x, g_x, h_x;\ x \in C\} \cup \{HQ\}$ and edges

$$\{(HQ, h_x), (h_x, HQ), (h_x, f_x), (T_x, f_x), (F_x, f_x), (g_x, f_x), (CM, g_x);\ x \in C\}.$$

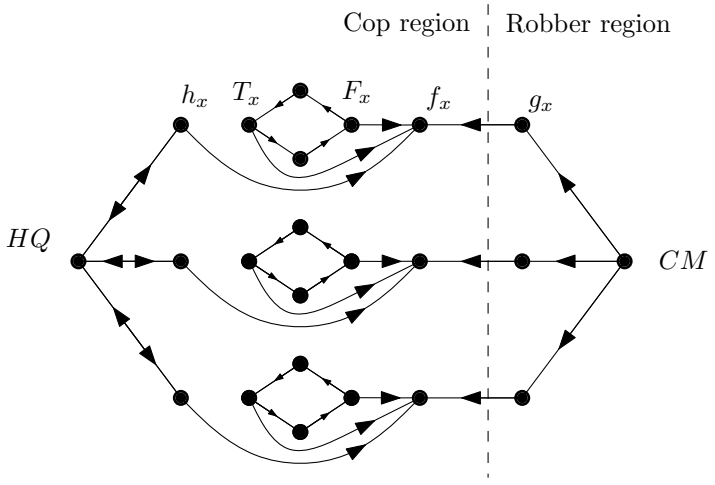


Fig. 4. The Commander gadget

The vertices $\{g_x;\ x \in C\}$ and CM belong to V_R, the rest belongs to V_C. There is one cop, the "commander", whose prescribed starting position is the vertex HQ. From every vertex $w \in V \setminus (V_C \cup \{CM\} \cup \{g_x;\ x \in C\})$ we add the edge (w, HQ) to $\overrightarrow{G}$, thus the only time the commander can leave HQ is when the robber stands at CM. The normal scenario is as follows: If the robber moves to CM, the commander decides one variable x to be changed and moves to h_x, simultaneously the cop in the variable cell V_x starts its movement towards the opposite vertex. The commander temporarily guards the vertex f_x, which is otherwise guarded by the cop in the cell V_x. Then the robber moves (away from CM) and the commander has to return to HQ in the next move.

Lemma 3. *Let us consider the Commander gadget and the variable cells V_x for $x \in C$ with exactly one cop each, standing either on T_x or F_x. Let the robber be at the vertex CM and the cop at HQ, with the cop-player on move. Suppose no other cop can access the vertices in the Commander gadget. Then the normal scenario is that in at most one variable cell V_x, $x \in X$ the cop can start moving from T_x to F_x or vice versa.*

Proof. Only the vertex f_x is temporarily (for one move) guarded by the commander. If two variable cops starts moving, some f_y is unsecured and robber exploits it by moving to g_y in his next move. □

Note that the Manipulator allows the robber to "pass" changing of his variable by setting the current position of cop in some variable. Also note, that the robber may stay on the vertex CM, thus allowing the cop-player to change more than one of his variables. However, in any winning strategy of the robber-player this is not necessary and if the robber-player does not have a winning strategy, this trick does not help him as the cops may pass.

2.2 The Gates to V_C

For every clause ϕ of F_R, there is one Robber gate gadget R_ϕ. If ϕ is satisfied by the current setting of variables, R_ϕ allows the robber to enter V_C.

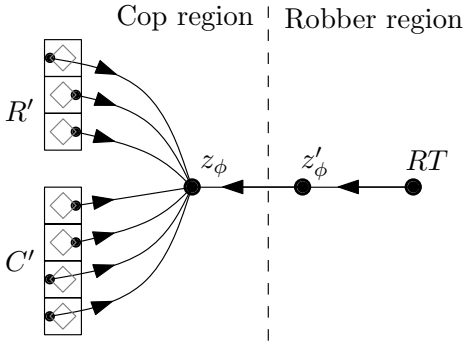


Fig. 5. The Robber Gate R_ϕ

The Robber gate R_ϕ consists of a directed path (RT, z'_ϕ, z_ϕ) and the following edges. Let $\phi = (\ell_1 \& \ldots \& \ell_{12})$ where each ℓ_i is a literal. If $\ell_i = x$ then we put the edge (F_x, z_ϕ) to $\overrightarrow{G}$. If $\ell_i = \neg x$ then we put the edge (T_x, z_ϕ) to $\overrightarrow{G}$. See Fig. 5 for illustration. The vertices $\{z'_\phi;\ \phi \in F_R\}$ and RT belong to V_R, the rest belongs to V_C.

Lemma 4. *Let ϕ be a clause of F_R, consider a Robber Gate R_ϕ. Let the robber stand at the vertex RT and let there be exactly one cop in each V_x, $x \in \phi$,*

standing either on T_x or F_x. Suppose no other cop can access R_ϕ in less than three moves. Then in the normal scenario robber can reach z_ϕ if and only if ϕ is satisfied under the current setting of variables (given by the positions of cops on variable cells).

Proof. If ϕ is satisfied, no cop at the variable cells can reach z_ϕ in two (or less) steps. Therefore, the robber may enter z_ϕ. On the other hand, if ϕ is not satisfied, at least one cop is one step from z_ϕ and the robber would be blocked forever if he moves to z'_ϕ. □

For every clause ψ of F_C, there is one *Cop Gate* gadget C_ψ (see Fig. 6). If ψ is satisfied, C_ψ allows cops to forever block the entrance to V_C, the vertices z_ϕ from each Robber Gate R_ϕ. The Cop Gate C_ψ consists of directed paths $(CT, b'_{\psi,x}, b_{\psi,x})$ for each variable x of the clause ψ, the directed cycle $(a_\psi, a'_\psi, a''_\psi, a'''_\psi)$ and edges $\{(a_\psi, b_{\psi,x}), (a''_\psi, b_{\psi,x});\ x \in \psi\}$ and $\{(a''_\psi, z_\phi);\ \phi \in F_R\}$.

Let $\psi = (\ell_1 \& \ldots \& \ell_{12})$ where each ℓ_i is a literal. If $\ell_i = x$ then we put the edge $(T_x, b_{\psi,x})$ to $\overrightarrow{G}$. If $\ell_i = \neg x$ then we put the edge $(F_x, b_{\psi,x})$ to $\overrightarrow{G}$. From the vertices a_ψ and a''_ψ there is an edge to every $b_{\psi,x}$ and from a''_ψ there is an edge to every z_ϕ (from each Robber Gate R_ϕ). There is a cop, we call him Arnold, and his prescribed starting position is a_ψ. Each C_ψ has its own Arnold, it would be therefore more correct to name him ψ-Arnold, however, we would use the shorter name if no confusion can occur. The vertices $\{b'_{\psi,x};\ \psi \in F_C,\ x \in \psi\}$ and CT belong to V_R, the rest belongs to V_C.

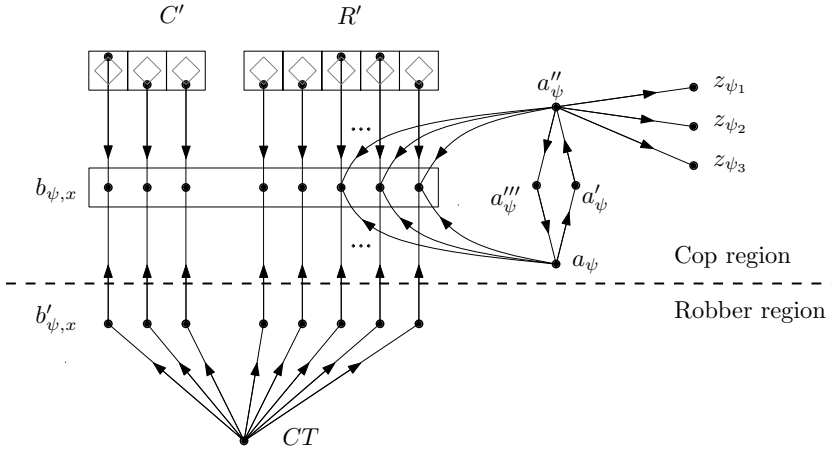


Fig. 6. The Cop Gate C_ψ

Lemma 5. *Let us consider a Cop Gate C_ψ. Let there be one cop at the vertex a_ψ (we call him Arnold) and let there be exactly one cop in each V_x, $x \in \psi$, standing on either T_x or F_x. Let the robber be at the vertex CT and no other cop can access C_ψ in less than three moves. Then in the normal scenario, Arnold is able to move to a''_ψ (and therefore block all the entrances z_ϕ forever) without*

permitting robber to enter V_C if and only if ψ is satisfied under the current setting of variables (given by the position of cops in the variable cells).

Proof. If ψ is satisfied, the vertices $b_{\psi,x}$, $x \in \psi$ are all guarded by the variable cops, therefore Arnold can start moving from a_ψ towards a''_ψ. If the robber meanwhile moves to some $b'_{\psi,x}$, the variable cop from V_x will intercept him by moving to $b_{\psi,x}$ and the robber loses the game. On the other hand, if ψ is not satisfied, there is some $b_{\psi,x}$ unguarded by the cop from V_x. Therefore, Arnold cannot leave a_ψ, because otherwise robber would reach $b_{\psi,x}$ before Arnold or the cop from V_x could block him. □

2.3 The Big Picture

We further need to assure that the cops cannot move arbitrarily. This means, that the following must be the normal scenario:

1. During the "Robber Move" phase, the only cop who can move is the cop in variable cell V_x chosen by the robber when he enters Manipulator M_x. All other variable cops must stand on either T_x of F_x vertices for some variable x. The cop in V_x must reach the vertex T_x from F_x (or vice versa) in two consecutive moves.
2. During the "Robber Test" phase, no cop can move.
3. During the "Cop Move" phase, only the commander and the cop in exactly one variable cell V_x can move. The cop in V_x must reach the vertex T_x from F_x (or vice versa) in two consecutive moves.
4. During the "Cop Test" phase, no other cop than Arnold may move. Arnold may move from vertex a_ψ to a''_ψ and he must do that in two consecutive steps (and of course Arnold may do that only if the clause ψ is satisfied).

We say that we *force* the vertex w by the vertex set S, when for every $v \in S$ we add the oriented path $P_{v,w} = (v, p_{vw}, p'_{vw}, w)$ of length 3 to the graph $\overrightarrow{G}$. The vertices p_{vw}, p'_{vw} belong to V_R. We say that we *block* the vertex w by the vertex set S, when for every $v \in S$ we add the Blocker gadget B_{wv}. The Blocker B_{wv} consists of vertices $p_1^v, p_2^v \in V_R$ and $q_1^v, q_2^v \in V_C$ and the edges $(v, p_i^v), (p_i^v, q_i^v), (w, q_i^v)$ for $i = 1, 2$.

A cop on a vertex w blocked by v cannot leave w even for one move when the robber is on v. Note also that if the cop on w enters q_i^v when it is not necessary to block p_i^v, then he is permanently disabled until the end of the game and the next time the robber visits v he may enter the cop-region through the other p_j^v.

Forcing serves as a tool to prevent moving of more than one variable cops (and Arnolds) however, because of the structure of variable cells, we cannot do it by simply blocking the vertices T_x, F_x and we have to develop the notation of forcing.

Case 1: For every variable $x \in C \cup R$ do the following construction. Let $S_x = \{RM, RT\} \cup \{V(M_y);\ y \in R, x \neq y\}$ where $V(M_y)$ are the vertices of Manipulator for variable y. We force the vertices T_x and F_x by the set S_x. Let $S_1 = \{RM\} \cup \{V(M_y);\ y \in R\}$. For each Cop Gate C_ψ, we force the vertex a_ψ by the set S_1. Finally, we block the vertex HQ by the set S_1. Observe that

whenever a cop from any other V_y than given by the Manipulator M_x is not on T_y or F_y, the robber can reach V_C faster than the variable cop can block him. On the other hand, if all variable cops are in the right places, the robber may never reach V_C unless being forever blocked. The same holds for Arnold on vertices a_ψ and a''_ψ. The commander cannot move because of the properties of the Blocker gadget. If the variable cop does not use his second turn to finish his movement, the robber will exploit this by reaching V_C by a path from the vertex RT.

Case 2: Let $S_2 = \{RT\} \cup \{z'_\phi;\ \phi \in F_R\}$ and let $F = \{T_x, F_x;\ x \in C \cup R\} \cup \{a_\psi;\ \psi \in F_C\}$. We force every $v \in F$ by the set S and we block the vertex HQ by S_2. Observe that in the normal scenario no cop may move.

Case 3: Let $S_3 = \{CM\}$ and let $F = \{T_x, F_x;\ x \in R\} \cup \{a_\psi;\ \psi \in F_C\}$. We force every $v \in F$ by S_3. Now, in normal scenario, no variable cop from V_x, $x \in R$ may move and by Lemma 3, only commander and exactly one variable cop from V_y, $y \in C$ may move.

Case 4: Let $S_4 = \{CT\}$ and let $F = \{T_x, F_x;\ x \in C \cup R\}$. We force every $v \in F$ by S_4 and we block the vertex HQ by S_4. Observe that in normal scenario no variable cop and the commander may move. The rest follows from Lemma 5 and the fact, that a''_ψ is forced by the vertex RM.

Finally, we connect the vertices in a directed cycle (RM, RT, CM, CT) and let the prescribed starting position r of the robber be the vertex RM. All the construction elements so far presented prove the following corollary.

Corollary 2. *For every game $\mathcal{F} = (\tau, F_C(C,R), F_R(C,R), \alpha)$ there exists a guarding game $\mathcal{G} = (\overrightarrow{G}, V_C, c, S, r)$, $\overrightarrow{G}$ directed, with a prescribed starting positions such that player I wins $\mathcal{F}$ if and only if the robber-player wins the game $\mathcal{G}$.*

Next we note, that we can modify our current construction so that it fully conforms to the definition of the guarding game on a directed graph.

Lemma 6. *Let $\mathcal{G} = (\overrightarrow{G}, V_C, c, S, r)$ be a guarding game with a prescribed starting positions. Let the position r has no in-going edge and let no two cops start at the same vertex. Then there exists a guarding game $\mathcal{G}' = (\overrightarrow{G}', V'_C, c')$, $\overrightarrow{G} \subseteq \overrightarrow{G}'$, $V_C \subseteq V'_C$ such that*

- *the robber-player wins $\mathcal{G}'$ if and only if the robber-player wins the game $\mathcal{G}$*
- *if the cop-player does not place the cops to completely cover S in his first move, he will lose*
- *if the robber-player does not place the robber on r in his first move, the cops win.*

Proof. Consider an edge $(u,v) \in E(\overrightarrow{G})$ such that $u \in V_R$ and $v \in V_C$ (a border edge). Observe, that the out-degree of each such vertex u in our construction is exactly 1. Let $m = |\{v \in V_C;\ (u,v) \in E(\overrightarrow{G}),\ u \in V_R\}|$ be the number of vertices from V_C directly threatened (i.e. in distance 1) from the robber region.

Let us define the graph $\overrightarrow{G}' = (V', E')$ such that $V' = V(\overrightarrow{G}) \cup \{r\} \cup T$ where $T = \{t_1, \ldots, t_m\}$ is the set of new vertices and $E' = E(\overrightarrow{G}) \cup \{(r,v);\ v \in T \cup S\}$. Consider the game $\mathcal{G}' = (\overrightarrow{G}', V'_C, c')$ where $V'_C = V_C \cup T$ and $c' = c + m$. See Fig. 7 for illustration.

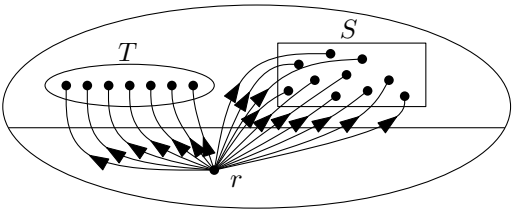


Fig. 7. Forcing starting positions

Suppose that the robber-player places the robber in the first move to some vertex $v \in V_R \setminus \{r\}$. Then there are m vertices in V_C directly threatened by edges going from V_R and because we have at least m cops available, the cops in the second move can occupy all these vertices and prevent the robber from entering V_C forever. So the robber must start at the vertex r. Then observe, that c cops must occupy the positions S and m cops must occupy the vertices T. If any cop does not start either on T or S, the robber wins in the next move. The cops on T remain there harmless to the end of the game. The cops cannot move until the robber decides to leave the vertex r. □

Let us have a guarding game $\mathcal{G} = (\overrightarrow{G}, V_C, c, S, r)$ with prescribed starting positions. Note that in our construction no two cops had the same starting position. We add new vertex r and edge (r, RM) to $\overrightarrow{G}$ and by the previous lemma there is an equivalent guarding game $\mathcal{G}'$, $\mathcal{G} \subseteq \mathcal{G}'$, without prescribed starting positions.

Theorem 1 is now proved.

3 Further Questions and Acknowledgements

For a guarding game $\mathcal{G} = (G, V_C, c)$, what happens if we restrict the size of strongly connected components of G? If the sizes are restricted by 1, we get DAG, for which the decision problem is PSPACE-complete. For unrestricted sizes we have shown that $\mathcal{G}$ is E-complete. Is there some threshold for $\mathcal{G}$ to become E-complete from being PSPACE-complete? We are also working on forcing the starting position in the guarding game on undirected graphs in a way similar to Theorem 1.

We would like to thank Peter Golovach for giving a nice talk about the problem, which inspired us to work on it. We would also like to thank Jarik Nešetřil for suggesting some of the previous open questions and to Honza Kratochvíl for fruitful discussion of the paper structure.

References

1. Fomin, F., Golovach, P., Hall, A., Mihalák, M., Vicari, E., Widmayer, P.: How to Guard a Graph? Algorithmica, doi:10.1007/s00453-009-9382-4
2. Stockmeyer, L., Chandra, A.: Provably Difficult Combinatorial Games. SIAM J. Comput. 8(2), 151–174 (1979)

3. Book, R.V.: Comparing complexity classes. J. of Computer and System Sciences 9(2), 213–229 (1974)
4. Alspach, B.: Searching and sweeping graphs: a brief survey. Matematiche (Catania) 59(1-2), 5–37 (2006)
5. Nowakowski, R., Winkler, P.: Vertex-to-vertex pursuit in a graph. Discrete Math. 43(2-3), 235–239 (1983)
6. Quilliot, A.: Some results about pursuit games on metric spaces obtained through graph theory techniques. European J. Combin. 7(1), 55–66 (1986)
7. Aigner, M., Fromme, M.: A game of cops and robbers. Discrete Appl. Math. 8(1), 1–11 (1984)
8. Nagamochi, H.: Cop-robber guarding game with cycle robber-region. Theoretical Computer Science 412, 383–390 (2011)
9. Thirumala Reddy, T.V., Sai Krishna, D., Pandu Rangan, C.: The Guarding Problem – Complexity and Approximation. In: Fiala, J., Kratochvíl, J., Miller, M. (eds.) IWOCA 2009. LNCS, vol. 5874, pp. 460–470. Springer, Heidelberg (2009)
10. Fomin, F.V., Thilikos, D.M.: An annotated bibliography on guaranteed graph searching. Theor. Comp. Sci. 399, 236–245 (2008)
11. Goldstein, A.S., Reingold, E.M.: The complexity of pursuit on a graph. Theor. Comp. Sci. 143, 93–112 (1995)
12. Fomin, F.V., Golovach, P.A., Kratochvíl, J., Nisse, N., Suchan, K.: Pursuing a fast robber on a graph. Theor. Comp. Sci. 411, 1167–1181 (2010)
13. Fomin, F.V., Golovach, P.A., Lokshtanov, D.: Guard games on graphs: Keep the intruder out! In: Bampis, E., Jansen, K. (eds.) WAOA 2009. LNCS, vol. 5893, pp. 147–158. Springer, Heidelberg (2010)
14. Fomin, F.V., Golovach, P.A., Kratochvíl, J.: On tractability Cops and Robbers Game. In: Proceedings of the 5th IFIP International Conference on Theoretical Computer Science (TCS 2008). IFIP, vol. 237, pp. 171–185. Springer, Heidelberg (2008)
15. Anderson, M., Barrientos, C., Brigham, R., Carrington, J., Vitray, R., Yellen, J.: Maximum demand graphs for eternal security. J. Combin. Math. Combin. Comput. 61, 111–128 (2007)
16. Burger, A.P., Cockayne, E.J., Grundlingh, W.R., Mynhardt, C.M., van Vuuren, J.H., Winterbach, W.: Infinite order domination in graphs. J. Combin. Math. Combin. Comput. 50, 179–194 (2004)
17. Goddard, W., Hedetniemi, S.M., Hedetniemi, S.T.: Eternal security in graphs. J. Combin. Math. Combin. Comput. 52, 169–180 (2005)
18. Goldwasser, J., Klostermeyer, W.F.: Tight bounds for eternal dominating sets in graphs. Discrete Math. 308, 2589–2593 (2008)
19. Klostermeyer, W.F.: Complexity of Eternal Security. J. Comb. Math. Comb. Comput. 61, 135–141 (2007)
20. Klostermeyer, W.F., MacGillivray, G.: Eternal security in graphs of fixed independence number. J. Combin. Math. Combin. Comput. 63, 97–101 (2007)
21. Klostermeyer, W.F., MacGillivray, G.: Eternally Secure Sets, Independence Sets, and Cliques. AKCE International Journal of Graphs and Combinatorics 2, 119–122 (2005)
22. Klostermeyer, W.F., MacGillivray, G.: Eternal dominating sets in graphs. J. Combin. Math. Combin. Comput. 68, 97–111 (2009)

Improved Steiner Tree Algorithms for Bounded Treewidth

Markus Chimani[1,⋆], Petra Mutzel[2], and Bernd Zey[2]

[1] Institute of Computer Science, Friedrich-Schiller-University of Jena
markus.chimani@uni-jena.de
[2] Department of Computer Science, TU Dortmund
{petra.mutzel,bernd.zey}@tu-dortmund.de

Abstract. We propose a new algorithm that solves the Steiner tree problem on graphs with vertex set V to optimality in $\mathcal{O}(B^2_{tw+2} \cdot tw \cdot |V|)$ time, where tw is the graph's treewidth and the *Bell number* B_k is the number of partitions of a k-element set. This is a linear time algorithm for graphs with fixed treewidth and a polynomial algorithm for $tw = \mathcal{O}(\log |V| / \log \log |V|)$.

While being faster than the previously known algorithms, our thereby used coloring scheme can be extended to give new, improved algorithms for the prize-collecting Steiner tree as well as the k-cardinality tree problems.

1 Introduction

In this paper we consider the well-known *Steiner tree problem* (STP), as well as the related problems *prize-collecting Steiner tree* (PCST) and *k-cardinality tree* (KCT), all defined on graphs. Our central results are new exact algorithms to solve these problems in the case of graphs with bounded treewidth: the *treewidth* tw of a graph (see below for a concise definition) can be seen as a measure of how similar the given graph is to a tree.

Let $G = (V, E)$ be a given edge-weighted graph and $T \subseteq V$ a set of *terminals*. The Steiner tree problem is to find a minimum-weight tree $\mathcal{S}$ in G which contains all terminals T and possibly also some non-terminal (*Steiner*) vertices of $V \setminus T$. Note that while often the edge weights are considered to be only positive, we do not require any such restriction. The corresponding decision problem is strongly $\mathcal{NP}$-complete, even when restricted to edge weights 1 and 2 [23], or when G is planar [18]. The traditional algorithm by Dreyfus and Wagner [17] solves the STP exactly in $\mathcal{O}(3^t \cdot |V|)$ time—recently improved to $\mathcal{O}(2^t \cdot |V|)$ [8]—where $t := |T|$ is the number of terminals.

Regarding G's treewidth tw, the oldest but yet strongest result is due to Korach and Solel [20]; yet this technical report has never been officially published and has been cited only rarely, e.g., in [7, 19, 22]. Their algorithm achieves a runtime of $\mathcal{O}(tw^{4tw} \cdot |V|)$ but the paper's description is very sketchy and leaves

⋆ Funded via a juniorprofessorship by the Carl-Zeiss-Foundation.

C.S. Iliopoulos and W.F. Smyth (Eds.): IWOCA 2011, LNCS 7056, pp. 374–386, 2011.
© Springer-Verlag Berlin Heidelberg 2011

many details unclear; it does not contain a formal proof of either the running time nor of its correctness. More recent publications, in particular those dealing with PTASes (see next paragraph) where the STP on bounded-treewidth-graphs arises as a subproblem, instead propose their own, yet weaker, results.

For the unweighted STP, i.e., the objective is to minimize the number of edges of $\mathcal{S}$, a very recent and surprising result by Cygan et al. [15] gives a Monte Carlo algorithm for the decision problem with a one-sided error—false negatives occur with probability of at most 1/2—requiring only $\mathcal{O}(3^{tw}|V|^{\mathcal{O}(1)})$ time. While the result is of course directly applicable to integer weighted STP where the maximum edge weight is bounded by a constant, we cannot see how to generalize the algorithm to arbitrary edge weights, and its derandomization is considered an open problem.

Recently, the STP and related problems for graphs with bounded treewidth achieved more attention due to their applicability to approximate network problems in planar graphs: In multiple papers [2,3,4,12,13,14], PTASes (polynomial time approximation schemes) are proposed which transform the given planar graph into a graph with bounded treewidth (via edge removals), solve the problem optimally (or within $1+\varepsilon$) on this modified graph, and then use this solution to construct a $(1+\varepsilon)$ solution to the original graph. Hence, the development of faster algorithms for the problem on bounded treewidth directly leads to faster PTASes for the corresponding problem on planar graphs.

For the STP, the approximation scheme of [13] uses an algorithm for solving the problem on graphs with bounded carving-width (a relative of treewidth) as a black box. Chekuri et al. [14] (later merged into [2]) give an algorithm for the prize-collecting Steiner tree problem (cf. Section 3) with running time $\mathcal{O}(B_k^3 \cdot s_k \cdot |V|)$, where $k := tw+1$, B_k is the number of partitions of a set with k elements (k-th *Bell number*), and s_k is the number of subgraphs of a k-vertex graph. Since $s_k = \mathcal{O}(2^{(k^2)})$, this leads to a running time of $\mathcal{O}(2^{(tw^2)} \cdot B_{tw+1}^3 \cdot |V|)$ for a graph with treewidth tw. This algorithm then allows PTASes for PCST and prize-collecting Steiner forest problems. Independently, Bateni et al. [4] (also later merged into [2]) describe PTASes for prize-collecting network design problems on planar graphs by using a similar approach. They investigate the PCST (the solution is a tree), prize-collecting TSP (the solution is a cycle), and the prize-collecting Stroll (the solution is a path). To this end they describe a $(1+\varepsilon)$-approximation for the PCST problem (that can be adapted to solve the other two considered problems as well) with a running time of order $\mathcal{O}(tw^{tw} \cdot 2^{tw} \cdot |V|)$.

Furthermore, Polzin and Daneshmand [22] introduced an algorithm with running time $\mathcal{O}(2^{3b} \cdot |V|)$ where b (the size of a "border" obtained throughout the algorithm) is a parameter similar to pathwidth. Yet note that even for simple trees—with natural treewidth 1—the pathwidth is unbounded.

Note that all these exact algorithms (not the approximations) fall into the category of FPT (fixed parameter tractable) algorithms w.r.t. the considered parameters (e.g., treewidth). An introduction to this research field can be found in [16, 21].

Our Contribution. Herein, we propose a new algorithm to solve the Steiner tree problem exactly in $\mathcal{O}(B_{tw+2}^2 \cdot tw \cdot |V|)$ time. The k-th *Bell number* B_k thereby is the number of partitions of a set with k elements, and can be recursively defined as $B_0 = 1$, $B_{k+1} = \sum_{i=0}^{k} \binom{k}{i} B_i$. We can bound $B_k < (0.792k/\ln(k+1))^k$ [5] and in particular $B_k < k! < k^k$ for $k \geq 3$. Our algorithm is hence linear for graphs with fixed treewidth and requires $\mathcal{O}(|V|^3 \log|V| / \log\log|V|)$ time for $tw \in \mathcal{O}(\log|V| / \log\log|V|)$. The algorithm guarantees a running time that is smaller than the currently best proposed running times, including the works of [20]. This paper therefore also closes the unclear situation regarding the latter. We will discuss our algorithm in Section 2.

To achieve this result, we use the well-known dynamic programming paradigm over the decomposition tree (see next section), coupled with a special numbering and coloring scheme. Furthermore, our new coloring scheme shows to be versatile enough to also allow new, faster algorithms to solve the prize-collecting Steiner tree problem in the same time complexity, as well as the k-cardinality tree problem in $\mathcal{O}(B_{tw+2}^2 \cdot (tw + k^2) \cdot |V|)$ time. We discuss these extensions in Sections 3 and 4, respectively.

Preliminaries: Tree Decompositions. The concept of treewidth was introduced by Robertson and Seymour [25] by the term tree decomposition. See [9,11] for an in-depth introduction to this topic:

Let $G = (V, E)$ be the given graph. Its tree decomposition $(\mathcal{T}, \mathcal{X})$ is a pair of a tree $\mathcal{T} = (I, F)$ and a collection $\mathcal{X} = \{X_i\}_{i \in I}$ of vertex subsets (called *bags*) with the following properties:

td/1: Every vertex $v \in V$ is contained in at least one bag X_i, $i \in I$. For every edge $(u, v) \in E$ there is at least one bag X_i, $i \in I$, containing both vertices u, v.

td/2: For every vertex $v \in V$, the nodes i with $v \in X_i$ form a subtree of $\mathcal{T}$.

To avoid confusion, we speak of *vertices* V in the graph G, and of *nodes* I in the tree $\mathcal{T}$. The width of a tree decomposition $(\mathcal{T}, \mathcal{X})$ is the size of the largest bag in $\mathcal{X}$ minus 1. The *treewidth* of a graph is the smallest width over all possible tree decompositions. Hence, the treewidth measures how similar the decomposed graph is to a tree: trees have treewidth 1, (generalized) series-parallel graphs have treewidth 2, etc. On the other side of the spectrum, complete graphs have treewidth $|V| - 1$, by putting all vertices in one bag. Determining whether a graph has treewidth k, for a given integer k, is $\mathcal{NP}$-complete [1] but polynomial (i.e., in FPT) for any constant k [10].

Most importantly, we note that the size of $(\mathcal{T}, \mathcal{X})$ is only linear, even when considering *nice* tree decompositions. Such tree decompositions always exist even for the optimal treewidth and have the following properties:

1. The tree $\mathcal{T}$ is considered to be rooted at some $r \in I$.
2. Each node is either a *leaf* (0 children), or has exactly 1 or 2 children.
3. Let $i \in I$ be a leaf, then $|X_i| = 1$.
4. Let $j \in I$ be the only child of a node $i \in I$, then either (a) X_j contains all vertices of X_i except for one ($X_j \subset X_i$, $|X_j|+1 = |X_i|$), or (b) X_j contains all

vertices of X_i plus one additional one ($X_i \subset X_j$, $|X_i|+1 = |X_j|$). Considering the tree in a bottom-up fashion, the node i is then called an *introduce* or *forget* node, respectively.

5. Let $j, j' \in I$ be the two children of a node $i \in I$, then all three corresponding bags are identical ($X_j = X_{j'} = X_i$), and i is called a *join* node.

Overall, given any tree decomposition, we can easily transform it into a nice tree decomposition where we pick the root r such that its bag X_r contains at least one terminal vertex. While the latter property is not ultimately necessary, it allows us to give a simpler description of our algorithm. We will discuss this in more detail at the end of Section 2.2.

2 Steiner Tree Algorithm

Our algorithm follows the classical bottom-up approach for algorithms based on tree decompositions: Starting from the leaves of a nice tree decomposition $(\mathcal{T} = (I, F), \mathcal{X})$, we enumerate a sufficient number of possible sub-solutions per tree node $i \in I$, using only the information previously computed for the children of i. Such information is stored in a table tab_i, for the node $i \in I$. The final optimal solution of the original problem can then be read from the table tab_r of $\mathcal{T}$'s root node r.

Since the tree traversal requires only $O(|V|)$ time, the algorithm's time complexity is mainly dependent on the amount of information to be stored per node (i.e., the size of tab_i which can be estimated by the number of sub-solutions times the size per sub-solution), as well as on the necessary effort to establish the sub-solutions at a node, based on its children's data.

In Section 2.1, we will concentrate on the first question, i.e., how to represent the necessary solutions efficiently. In fact, this modeling (based on coloring) is the main result of this paper, which subsequently allows us to obtain stronger memory and runtime bounds than the previous approaches. Section 2.2 then describes how to efficiently combine our coloring with the bottom-up traversal to solve the Steiner tree problem. Finally, Section 2.3 formally establishes the correctness and running time of our approach.

2.1 Representing Sub-solutions

The general idea of using the (rooted) tree decomposition is the following: Let i be any node in $\mathcal{T}$ with the corresponding bag X_i. We define X_i^+ to be the set of all vertices in X_j for all nodes $j \in I$ that are either i itself or any of its descendants. Then, let G_i (G_i^+) describe the subgraph of G induced by the vertices X_i (X_i^+, respectively). Let T_i (T_i^+) be the set of terminals in X_i (X_i^+, respectively).

When we consider any node $i \in I$, we observe, based on property td/2 of a tree decomposition, that no vertex of $X_i^+ \setminus X_i$ will appear in any other bag than the ones descending from node i. For our bottom-up approach this means that

these vertices are not considered in other parts of the algorithm and will never be considered again. Hence, the sub-solutions at node i have to ensure that all terminals $T_i^+ \setminus T_i$ are properly connected with other vertices to allow a feasible solution in the end. Consider the optimal Steiner tree $\mathcal{S}$ in G. The subgraph of $\mathcal{S}$ in G_i^+ then forms a forest, with the property that any terminal $T_i^+ \setminus T_i$ is connected to some vertex in X_i.

Our table tab_i hence stores multiple rows, each row representing a solution. Observe that we do not have to consider all possible subgraphs of a bag X_i but can use the fact that a forest in G_i contains at most $|X_i| - 1$ edges. It remains how to uniquely, succinctly, and compactly describe these forests (and allow for fast merging operations within the bottom-up approach). We show that it (coarsely) is sufficient to consider all possible *partitions* of the (at most $tw + 1$ many) vertices X_i by assigning colors to them. Each color then indicates the set of vertices that lie in a connected component (tree, in fact) in G_i^+. We will see that by careful enumeration we only require a table with at most B_{tw+2} different partitions, instead of the straight-forward $\mathcal{O}((tw+1)^{tw+1})$.

To obtain such a description scheme, we first consider some arbitrary but fixed total numbering $\Phi : V \xrightarrow{1:1} \{1, \ldots, |V|\}$ of all vertices of the given graph. Based thereon, we assign—locally for each bag X_i—the unique secondary index $\varphi_i : X_i \xrightarrow{1:1} \{1, \ldots, |X_i|\}$ which satisfies $\Phi(v) < \Phi(w) \Leftrightarrow \varphi_i(v) < \varphi_i(w)$ for all $v, w \in X_i$. We now introduce a coloring function $\gamma_i : X_i \to \{0, \ldots, |X_i|\}$; thereby any vertex $v \in X_i$ may only be colored by a color at most as large as its local index, i.e., $\gamma_i(v) \leq \phi_i(v)$. Our interpretation is that all vertices of color 0 are not contained in the represented sub-solution. All vertices with a common color > 0 are connected in the graph G_i^+. Note that these connections do not have to exist in G_i. Finally, in order to be a feasible coloring, we require all terminals T_i in X_i to be colored > 0.

Note that, by the above coloring properties, the color of a connected component C of the sub-solution is exactly the smallest secondary index of all vertices contained in C. We observe that a vertex v with $\varphi_i(v) = z$ has $z+1$ possible colors. Hence the number of possible colorings for a bag X_i (and therefore of rows in tab_i) can trivially be bounded by $\prod_{z=1}^{|X_i|}(z+1) = (|X_i|+1)! = \mathcal{O}((tw+2)!)$. This would already allow better overall bounds for the algorithm than previously known. Yet, we can observe that when we conceptually add an additional "ghost" element to an $|X_i|$-element set, and consider all possible partitions thereof, we can interpret these resulting partitions as all possible colorings: The partition that contains the "ghost" element is considered to be the partition with color 0. All other partitions get the color of the smallest secondary index among its elements. It is straight-forward to efficiently enumerate all $B_{|X_i|+1}$ possible partitions (hence rows in tab_i) of a $|X_i| + 1$-element set.

In each row, we store the unique corresponding coloring of the solution, i.e., a color index for each vertex of X_i, which we can trivially compute in $\mathcal{O}(tw)$ time. Additionally, we will store a solution value for each row, see below. Hence, the size of any table tab_i can be bounded by $\mathcal{O}(B_{tw+2} \cdot tw)$.

2.2 Processing the Decomposition Tree

Having our coloring concept at hand, we can now describe how to ensure its properties when computing the actual sub-solution tables in a bottom-up fashion. Our recursion can be described by distinguishing between the different currently considered nodes of $\mathcal{T}$. Recall that for each row, representing some coloring γ, we store the cost $val(\gamma)$ of the represented sub-solution.

Leaf Node. Let $i \in I$ be a leaf, and hence a (trivial) base case for our algorithm. The table tab_i contains only two rows corresponding to the two possible colors 0 and 1, respectively, for the only vertex $v \in X_i$. If $v \in T$ but is colored 0, the sub-solution's cost is $+\infty$; in all other cases the cost is 0.

Introduce Node. Let $i \in I$ be an introduce node, and $j \in I$ its only child. We have $X_j \subset X_i$, $|X_j| + 1 = |X_i|$, and let v be the additional vertex.

As a preprocessing, we initialize tab_i and modify tab_j as follows: We generate all $B_{|X_i|+1}$ possible rows of tab_i and set their value entries to $+\infty$. In tab_j we add an additional column for v (which remains uncolored, say color -1) and modify the other color numbers to match the coloring scheme of i, instead of j: By the fact that both secondary indices stem from a common primary index Φ, this means that precisely all colors $\geq \phi_i(v)$ have to be increased by one. We observe that this preprocessing takes only $\mathcal{O}(B_{tw+2} \cdot tw)$ time.

The cost for any coloring γ_i of X_i with $\gamma_i(v) = 0$ is straight-forward: Let γ_j be the unique coloring in tab_j that agrees with γ_i on all vertices except for v. If $v \in T$, i.e., v is a terminal vertex, $val(\gamma_i) = +\infty$, otherwise $val(\gamma_i) = val(\gamma_j)$.

Now, we consider all *compatible* combinations of rows of tab_j and tab_i with the intuition that several connected components of a solution at j may become connected via the newly inserted, > 0-colored vertex v. Therefore, a coloring γ_j of X_j is *compatible* with a coloring γ_i of bag X_i with $\gamma_i(v) \neq 0$ if and only if the color partitions agree for all colors except for the color to which v belongs. More formally, let c be the color of v in γ_i, then any vertex partition induced by some color in γ_j is either also a vertex partition with the same color in γ_i, or a (proper) subset of the vertex partition of color c in γ_i. Intuitively, the vertex v connects with some formerly separated color partitions, coloring them all with a common color.

We can compute the cost $val(\gamma_i)$ for this solution at the introduce node i by adding the costs of these new connections to the precomputed cost $val(\gamma_j)$ of γ_j. For the former, we simply have to find, for each *formerly* separate color partition W, the cheapest edge in G_i connecting v with any vertex in W, and sum over these costs. If no such edge exists, the corresponding connection cost is $+\infty$. If the so computed cost of γ_i is smaller than the current $val(\gamma_i)$ entry for this coloring in tab_i, we update $val(\gamma_i)$ accordingly. Hence, processing an introduce node takes $\mathcal{O}(B^2_{tw+2} \cdot tw)$ time.

Forget Node. Let $i \in I$ be a forget node, and $j \in I$ its only child. We have $X_i \subset X_j$, $|X_i| + 1 = |X_j|$, and let v be the additional (discarded, in fact) vertex.

As a preprocessing, we generate all rows of tab_i and set their solution costs to $+\infty$. We then look at the rows of tab_j one by one; let γ_j be the corresponding

coloring, and $c := \gamma_j(v)$. We say γ_j *induces* a coloring γ_i of the vertices X_i, by simply dropping the vertex v and shifting the color index by -1 for all colors $> \phi_j(v)$; the vertices colored with color $\phi_j(v)$ in γ_j obtain the color matching the smallest secondary index $\phi_i(.)$ among themselves. Note that we can look up the row of the induced coloring in tab_i in $\mathcal{O}(tw)$ by exploiting the enumeration scheme.

If $c > 0$ but there is no other vertex with color c, we cannot easily remove this vertex from the solution, as it represents a component (containing, in general, terminals) that has to be connected to the final Steiner tree $\mathcal{S}$ (recall that we can safely assume that the decomposition tree's root node contains a terminal). Hence we cannot use this sub-solution to improve the solution value of the induced coloring of X_i. Otherwise, we can safely drop the vertex and set $val(\gamma_i) := val(\gamma_j)$ if the current value of $val(\gamma_i)$ is not already smaller.

Join Node. Let $i \in I$ be a join node, and $j, j' \in I$ its two children. We have $X_j = X_{j'} = X_i$.

Again, we first construct all rows of tab_i and set the solution values to $+\infty$. Then we consider all possible combinations of solutions from X_j and $X_{j'}$. Let γ_j and $\gamma_{j'}$ be colorings (rows) of tab_j and $tab_{j'}$, respectively. We want to construct a merged solution γ_i that resembles the combined connectivities of both solutions, i.e., two vertices $v_s, v_t \in X_i$ should be in the same color partition if and only if there is a vertex sequence $\langle v_s := v_1, v_2, \ldots, v_\beta := v_t \rangle$ in X_i such that, for all $1 \leq \alpha < \beta$, the vertices $v_\alpha, v_{\alpha+1}$ have the same color in γ_j or $\gamma_{j'}$.

Note that, a priori, such a merge might lead to cycles in the solution: assume two vertices v_1, v_2 are colored with identical color c_j in γ_j. Furthermore, they have a (probably different but) common color $c_{j'}$ in $\gamma_{j'}$. Hence the vertices are connected in both sub-solutions, but the connection paths do not need to coincide. Even if the paths do coincide, we would have to identify them to not count their cost twice for the combined solution. Hence, we only want to combine solutions with the property that any pair of vertices has a common color > 0 in at most one of the two colorings $\gamma_j, \gamma_{j'}$. Then, the value of the combined solution can be given as $val(\gamma_i) := val(\gamma_j) + val(\gamma_{j'})$, which we can store into tab_i (unless the stored value for this solution is already smaller). Again, observe that we can identify the row index in tab_i of any given solution γ_i in $\mathcal{O}(tw)$ by exploiting the enumeration scheme.

It would be trivial to perform the check whether to merge, as well as the actual merge, in $\mathcal{O}(tw^2)$ time, for any given pair of sub-solutions. Yet, we can do better and perform the merge operation, including the check of the precondition, in linear time $\mathcal{O}(tw)$: Consider a helper array $recol : \{1, \ldots, |X_i|\} \to \{1, \ldots, |X_i|\}$ and construct a graph C with a vertex c_r per possible color r. Then, for each $v \in X_i$, add an edge $(c_{\gamma_j(v)}, c_{\gamma_{j'}(v)})$. Clearly, the graph has only $\mathcal{O}(tw)$ vertices and edges. Remove the vertex c_0 together with its incident edges, and mark all other vertices in C as unvisited. Then, for increasing $r \in \{1, \ldots, |X_i|\}$, start a depth-first search (DFS) in C at any unvisited c_r: set $recol(c_{r'}) := r$ for any vertex $c_{r'}$ visited in this DFS run. Hence, in the end, *recol* gives the new color for any color in either γ_j or $\gamma_{j'}$. Whenever a DFS run revisits an already visited

vertex (within the same run), we identified a cycle (including the special case of multiple edges), and the merge operation should be aborted. If no cycles are detected, we can finally again consider each $v \in X_i$ and set $\gamma_i(v) := 0$ if $\gamma_j(v) = \gamma_{j'}(v) = 0$, and $\gamma_i(v) := \mathit{recol}(\max\{\gamma_j(v), \gamma_{j'}(v)\})$ otherwise.

Remark. Note that if the given graph G has only positive edge weights, we do not need to actively identify cycles or multiedges: the merged solution's objective value will be greater than the alternative cycle/multiedge-free combination, which will, at some point, also be considered. Since we store only the best solution for any coloring in tab_i, the stored solutions will always be cycle- and multiedge-free.

Extracting the Solution at the Root Node. From the described construction process it is clear that each solution of a bag X_i describes the (minimum) costs of a forest where all terminals from X_i^+ are (probably indirectly) connected to some vertex of X_i. Also recall that it can be safely assumed that at least one terminal is contained in the root bag X_r of $\mathcal{T}$. Hence the optimum solution value for the whole graph can be found in the root bag X_r of $\mathcal{T}$, identifying a cheapest solution where all vertices with color $\neq 0$ are contained in the same connected component (i.e., have the same color).

Computing the optimum solution, i.e., the set of edges, is possible by backtracking or by storing the set of edges for each row and each bag. The latter increases the required memory but has no negative impact on the running time since these sets are simple linked lists that can be concatenated in $\mathcal{O}(1)$.

Remark. We can—with the same time complexity—also run the algorithm on a tree decomposition where the root node does not contain any terminal vertex. In this case, whenever we process a tree node i where $T \subseteq X_i^+$ (i.e., all terminals are within the subtree induced by i), we check for the best solution where all vertices with color $\neq 0$ belong to the same color partition, and store a reference to it. After processing the root node, this reference gives the optimal solution.

2.3 Analysis

In the following, we will discuss the algorithm's running time and prove that it correctly computes an optimal solution.

Lemma 1. *The above algorithm requires* $\mathcal{O}(B_{tw+2}^2 \cdot tw \cdot |V|)$ *time.*

Proof. The running time mainly depends on the size of the tables and the combination of tables during the bottom-up traversal of the decomposition tree. We already established that each table tab_i at some tree node i stores $\mathcal{O}(B_{tw+2})$ rows and requires overall $\mathcal{O}(B_{tw+2} \cdot tw)$ storage.

During the bottom-up traversal of $\mathcal{T}$ we consider all possible row combinations for two tables in the case of the introduce and the join node. For each such combination, we perform a merge operation in $\mathcal{O}(tw)$, and we hence require overall $\mathcal{O}(B_{tw+2}^2 \cdot tw)$ time. This bound dominates the time required for the other tree node types (forget and leaf nodes), as well as all other extra

effort—feasibility tests, shifting of indices, etc.—which is only linearly dependent on the treewidth. Due to the linear size of $\mathcal{T}$, we can deduce the overall running time. □

Lemma 2. *The above algorithm correctly computes an optimal solution to the given Steiner tree problem.*

Proof. The algorithm's correctness can be shown by a straight-forward inductive proof on the decomposition tree. Let $\Gamma_i^c := \{v \in X_i \mid \gamma_i(v) = c\}$ be the vertices colored c in a coloring γ_i. Our induction hypothesis (IH) states that, for each processed bag X_i, the cost of each solution γ_i corresponds to a minimum forest $F_i \subseteq G_i^+$ with the properties

- F_i consists of (pairwise disconnected) trees F_i^c, one for each color $c > 0$ with $\Gamma_i^c \neq \emptyset$, with $\Gamma_i^c \subseteq V(F_i^c)$, and $\Gamma_i^{c'} \cap V(F_i^c) = 0$ for all $c' \neq c$. I.e., each tree connects only vertices of the same color partition.
- F_i contains all terminals of G_i^+, i.e., $T_i^+ \subseteq V(F_i)$.

The base cases are leaf nodes where the hypothesis clearly holds. Now, let the induction hypothesis be true for all descendants of a bag X_i.

Forget node. Each coloring of a forget bag X_i is induced by $|X_i| + 1$ many colorings in the child table—one for each possible color of the forget vertex. Our algorithm picks the minimal among them that remains feasible after the removal of the forget vertex, and does not change its solution value.

Assume the minimum solution γ_i at X_i would be smaller then this identified sub-solution. Then we could add the forget vertex to the solution γ_i of X_i, coloring it as required by F_i. This is a feasible coloring for the child node, and stays feasible after removing the forget vertex. Hence, it would have been considered by our algorithm (without modifying its solution value).

Introduce node. For an introduce node i, the solution table contains a copy of its child table, when coloring the new vertex either 0 or with its own secondary index. Furthermore, the new vertex allows the connection of several components.

Assume some optimal solution at an introduce bag X_i would be smaller than the one obtained by the algorithm. If the introduced vertex v is colored 0 or has a unique color, the otherwise identical coloring (up to index shifting) was stored in the child table (IH). As the algorithm would not have changed the solution value, we arrive at a contradiction. Now assume v belongs to some color class Γ_i^c with more than one element, and let F_i^c be the corresponding solution tree. When we remove v from F_i^c, it decomposes into several components. Our algorithm considered all possible such components in the child table, including their optimal costs (IH), and attached them to v via the minimal edges. That means that our algorithm considered this solution and would have computed its costs correctly.

Join node. Similar to above, assume that we would have a solution γ_i at a join node i which is strictly smaller than the one computed by our algorithm, and consider the forest F_i. Observe that the vertex set X_i of i is identical to those of its children j, j'. We can partition F_i into two sub-forests: Let F_i^1 be the forest restricted to the edges of G_j^+, and let F_i^2 be the forest restricted to the edges $E(G_{j'}^+) \setminus E(G_{j'})$, i.e., it does not contain any edges already contained

in F_i^1. Observe that F_i^1 induces a feasible coloring solution at node j, and F_i^2 a feasible coloring solution at node j, and that both are disjoint. Hence, by (IH), our algorithm would have considered to merge the corresponding optimal sub-solutions to obtain γ_i, with the correct objective value based on summing the costs of F_i^1 and F_i^2. □

Finally, the following theorem summarizes the above lemmas.

Theorem 1. *Given a graph with vertex set V and a tree decomposition with treewidth tw, the Steiner tree problem can be solved to optimality in $\mathcal{O}(B_{tw+2}^2 \cdot tw \cdot |V|)$ time.*

3 Prize-Collecting Steiner Tree

The prize-collecting Steiner tree problem (PCSTP) is an extension of the STP. Thereby, instead of being required to connect all terminals, we get a (vertex-specific) *prize* for each vertex we connect. We are hence given a function $p : V \to \mathbb{R}_{>0}$ and want to find a tree $\mathcal{S} = (V_\mathcal{S}, E_\mathcal{S})$ that minimizes $\sum_{e \in E_\mathcal{S}} c(e) - \sum_{v \in \mathcal{S}} p(v)$, where c is the edge-cost function.[1]

Our algorithm for the STP can be adapted by introducing the profits into the cost calculations (at the introduce nodes) and removing the necessity that terminals are assigned a color $\neq 0$. Because terminals may be omitted, the optimum solution need not necessarily be captured by the table at the decomposition tree's root node. Hence, during the bottom-up traversal each row of each table is a potential global solution if the corresponding coloring induces a feasible tree. The remaining part of the algorithm remains identical and after the previous discussion on the running time and optimality we conclude the following theorem.

Theorem 2. *Given a graph with vertex set V and a tree decomposition with treewidth tw, the prize-collecting Steiner tree problem can be solved to optimality in $\mathcal{O}(B_{tw+2}^2 \cdot tw \cdot |V|)$ time.*

4 *k*-Cardinality Tree

The k-cardinality tree (KCT) problem is defined on an edge-weighted, undirected graph and asks for a minimum-cost tree containing exactly k edges. Betzler [7] introduced an FPT algorithm with parameter k and time complexity $\mathcal{O}(2^{\mathcal{O}(k)} k \cdot |E| \cdot \log |V|)$. Ravi et al. [24] sketched a general FPT strategy for any decomposable graph [6] (including graphs with bounded treewidth) with time complexity $\mathcal{O}(f(tw) \cdot k^2 \cdot |V|)$. As their general description considers any

[1] Sometimes, the objective function is also described as $\min \sum_{v \notin V_\mathcal{S}} p(v) + \sum_{e \in E_\mathcal{S}} c(e)$. From the point of view of optimal solutions, both problems are equivalent as $\sum_{v \in V} p(v)$ is a constant. We prefer the former definition to be able to locally evaluate the objective function at each node of the decomposition tree.

dependence on the decomposition's parameter (e.g., treewidth) a constant, there are no more details on the non-polynomial function $f(tw)$. In the following, we describe how to extend our previous algorithm for the STP to obtain an exact algorithm for the KCT problem with a running time that increases by less than a factor of k^2, compared to the STP. In fact, our extension follows the concept of [24], although in a less abstract way. Our obtained runtime bound is equivalent to their result, and, to our knowledge, constitutes the first published bound for $f(tw)$.

As for the (PC)STP, we enumerate all possible partitions of the vertices of each bag by assigning colors, and propagate optimal solutions to the root bag in a bottom-up traversal of $\mathcal{T}$. Yet, in contrast to the (PC)STP, holding a single solution per partition and choosing minima is not sufficient as we have to take the overall number of chosen edges into account.

Therefore the algorithm maintains, for each possible coloring at node i, a solution value of a minimal forest with exactly k' edges—establishing the color-induced partition—for each possible $0 \leq k' \leq k$. All vertices of such a k'-*forest* are either from X_i or are (indirectly) connected to vertices in X_i; all vertices of a tree of such a forest are colored identically. Clearly, for solutions with ℓ non-0-colored vertices and c different colors > 0, the solution value for $k' < \ell - c$ is $+\infty$, as any feasible forest requires more edges. The main observation is that these k'-forests are always disjoint from any solution considered at any node not in the subtree rooted at i, except for the vertices and edges in G_i. Overall, the size of each row at any table tab_i is $\mathcal{O}(tw + k)$.

Trivially, both possible colorings at a *leaf node* have cost 0 for the 0-forest and $+\infty$ otherwise. For a *forget node* the component of the forget vertex v has to be considered: if v is the only vertex with color > 0 in the child table tab_j, the attached size-k forest (tree, in fact) might be the optimum k-cardinality tree; hence, we compare and update the global optimum (similar as for the PCSTP). If v is the only vertex with color γ_j but there are also other vertices colored > 0, we cannot deduce a feasible solution and obtain forest values $+\infty$. Otherwise (i.e., v does not define its own color class, or is colored 0), v can be simply discarded without changing the costs of the k'-forests. Analogously to the STP—and in the following also for the cases of the other inner nodes—we always store the smallest solution value for each k' that is achievable by a reduction from any compatible coloring of the child bag.

The new vertex v in an *introduce node* might connect several connected components, say c many. Similar to the STP, the cheapest edges connecting v with each component are chosen; the cost of a k'-forest at the child node, together with the cost-sum of the new edges, gives the cost of a $(k' + c)$-forest for the considered coloring at node i. As we consider all compatible colorings at the child node and store the minimum per k'', we will, in the end, know the minimally achievable k''-forest for any possible cardinality $1 \leq k'' \leq k$ at i.

For a *join node* observe that k'-forests from two combined solutions are pairwise disjoint, as long as their coloring does not induce cycles or multiedges, as we discussed for the STP. Hence, as for the STP, we combine only two solutions with

this property, reusing our DFS sub-algorithm. To compute the new minimal k'-forest, for each k', we consider all combinations of a k_1-forest of the first, and a k_2-forest of the second solution with $k_1 + k_2 = k'$. These are $\mathcal{O}(k^2)$ computations.

After processing the root node, we may update the globally stored optimum by the k-forests (trees, in fact) arising from colorings with a single non-0 color.

Analyzing the running time, we again require tables with $\mathcal{O}(B_{tw+2})$ rows, each row of size $\mathcal{O}(tw + k)$. In case of a join and an introduce node, two tables are combined by considering all possible $\mathcal{O}(B^2_{tw+2})$ combinations; the largest effort of $\mathcal{O}(k^2 + tw)$ per combination arises at a join node. Due to space restrictions and obvious analogies to the STP we omit the correctness proof and close the discussion on the KCT problem with the following theorem.

Theorem 3. *Given a graph with vertex set V and a tree decomposition with treewidth tw, the k-cardinality tree problem can be solved optimally in $\mathcal{O}(B^2_{tw+2} \cdot (tw + k^2) \cdot |V|)$ time.*

5 Conclusions

We showed new, currently fastest treewidth-based exact algorithms for the STP, the PCSTP, and the KCT problem. For the former two problems, these algorithms also directly speed-up current PTASes for planar STP and PCSTP, as those use algorithms for bounded treewidth as their most time-consuming subroutines.

References

1. Arnborg, S., Corneil, D.G., Proskurowski, A.: Complexity of finding embeddings in a k-tree. SIAM J. Algebraic Discrete Methods 8(2), 277–284 (1987)
2. Bateni, M., Chekuri, C., Ene, A., Hajiaghayi, M., Korula, N., Marx, D.: Prize-collecting Steiner problems on planar graphs. In: SODA, pp. 1028–1049. SIAM (2011)
3. Bateni, M., Hajiaghayi, M., Marx, D.: Approximation schemes for Steiner forest on planar graphs and graphs of bounded treewidth. In: STOC, pp. 211–220. ACM (2010)
4. Bateni, M., Hajiaghayi, M., Marx, D.: Prize-collecting network design on planar graphs. CoRR, abs/1006.4339 (2010)
5. Berend, D., Tassa, T.: Improved bounds on Bell numbers and on moments of sums of random variables. Probability and Mathematical Statistics 30, 185–205 (2010)
6. Bern, M.W., Lawler, E.L., Wong, A.L.: Linear-time computation of optimal subgraphs of decomposable graphs. J. Algorithms 8, 216–235 (1987)
7. Betzler, N.: Steiner tree problems in the analysis of biological networks. Master's thesis, Universität Tübingen (2006)
8. Björklund, A., Husfeldt, T., Kaski, P., Koivisto, M.: Fourier meets Möbius: Fast subset convolution. In: STOC, pp. 67–74. ACM (2007)
9. Bodlaender, H.L.: A tourist guide through treewidth. Acta Cybernetica 11(1-2), 1–22 (1993)

10. Bodlaender, H.L.: A linear-time algorithm for finding tree-decompositions of small treewidth. SIAM J. Comput. 25(6), 1305–1317 (1996)
11. Bodlaender, H.L.: Treewidth: Structure and Algorithms. In: Prencipe, G., Zaks, S. (eds.) SIROCCO 2007. LNCS, vol. 4474, pp. 11–25. Springer, Heidelberg (2007)
12. Borradaile, G., Kenyon-Mathieu, C., Klein, P.: A polynomial-time approximation scheme for Steiner tree in planar graphs. In: SODA, pp. 1285–1294. SIAM (2007)
13. Borradaile, G., Klein, P., Mathieu, C.: An $O(n \log n)$ approximation scheme for Steiner tree in planar graphs. ACM Transactions on Algorithms 5, 1–31 (2009)
14. Chekuri, C., Ene, A., Korula, N.: Prize-collecting Steiner tree and forest in planar graphs. CoRR, abs/1006.4357 (2010)
15. Cygan, M., Nederlof, J., Pilipczuk, M., Pilipczuk, M., van Rooij, J., Wojtaszczyk, J.O.: Solving connectivity problems parameterized by treewidth in single exponential time. CoRR, abs/1103.0534 (2011)
16. Downey, R.G., Fellows, M.R.: Parameterized Complexity. Springer, Heidelberg (1999)
17. Dreyfus, S.E., Wagner, R.A.: The Steiner problem in graphs. Networks 1, 195–207 (1972)
18. Garey, M.R., Johnson, D.S.: The rectilinear Steiner tree problem is NP-complete. SIAM Journal on Applied Mathematics 32(4), 826–834 (1977)
19. Gassner, E.: The Steiner forest problem revisited. J. Discrete Algorithms 8(2), 154–163 (2010)
20. Korach, E., Solel, N.: Linear time algorithm for minimum weight Steiner tree in graphs with bounded treewidth. Technical Report 632, Israel Institute of Technology (1990)
21. Niedermeier, R.: Invitation to Fixed-Parameter Algorithms. Oxford University Press (2006)
22. Polzin, T., Daneshmand, S.: Practical Partitioning-Based Methods for the Steiner Problem. In: Àlvarez, C., Serna, M. (eds.) WEA 2006. LNCS, vol. 4007, pp. 241–252. Springer, Heidelberg (2006)
23. Prömel, H.J., Steger, A.: The Steiner Tree Problem. A Tour Through Graphs, Algorithms and Complexity. Vieweg Verlag (2002)
24. Ravi, R., Sundaram, R., Marathe, M.V., Rosenkrantz, D.J., Ravi, S.S.: Spanning trees short or small. In: SODA, pp. 546–555. SIAM (1994)
25. Robertson, N., Seymour, P.D.: Graph minors. II. Algorithmic aspects of tree-width. J. Algorithms 7(3), 309–322 (1986)

Author Index