Thierry Lecroq
Laurent Mouchard (Eds.)

LNCS 8288

Combinatorial Algorithms

24th International Workshop, IWOCA 2013
Rouen, France, July 2013
Revised Selected Papers

Springer

Lecture Notes in Computer Science 8288

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

Thierry Lecroq   Laurent Mouchard (Eds.)

Combinatorial Algorithms

24th International Workshop, IWOCA 2013
Rouen, France, July 10-12, 2013
Revised Selected Papers

Springer

Volume Editors

Thierry Lecroq
Université de Rouen
UFR des Sciences et Techniques, LITIS EA 4108
76821 Mont Saint Aignan Cedex, France
E-mail: thierry.lecroq@univ-rouen.fr

Laurent Mouchard
Université de Rouen
UFR des Sciences et Techniques, LITIS EA 4108
76821 Mont Saint Aignan Cedex, France
E-mail: laurent.mouchard@univ-rouen.fr

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-45277-2 e-ISBN 978-3-642-45278-9
DOI 10.1007/978-3-642-45278-9
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2013955008

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 2013

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in ist current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.
The use of general descriptive names, registered names, trademarks, service marks, 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.
While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

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 2013, the 24th International Workshop on Combinatorial Algorithms. The 24th IWOCA was held July 10–12, 2013, at the University of Rouen, France. The meeting was supported financially mainly by the University of Rouen and the LITIS EA 4108 Laboratory. The two co-chairs of both the Program and the Organizing Committees were Thierry Lecroq and Laurent Mouchard. 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 `http://www.iwoca.org/` where links to previous meetings can be found.

Using different e-mail lists, the IWOCA 2013 call for papers was distributed around the world, resulting in 91 submissions. The EasyChair system was used to facilitate management of submissions and refereeing, with three referees from the 47-member Program Committee assigned to each paper. A total of 34 papers (37%) were accepted, subject to revision, for presentation at the workshop, with an additional ten papers accepted for poster presentation. One full paper was later withdrawn by the authors.

Five invited talks were given:

- Ewan Birney (European Bioinformatics Institute, UK)
  "Annotating the Human Genome"
- Stefan Edelkamp (University of Bremen, Germany)
  "Weak Heaps and Friends - Show Me Your Bits"
- Jerrold R. Griggs (University of South Carolina, USA)
  "The Δ^2 Conjecture for Graph Labellings with Separation Conditions"
- Ralf Klasing (LaBRI and CNRS, France)
  "Efficient Exploration of Anonymous Undirected Graphs"
- Kunsoo Park (Seoul National University, Korea)
  "New Graph Model for Consistent Superstring Problems"

These proceedings contain all 33 presented papers, together with shortened versions of the ten poster papers and abstract or extended versions of the invited talks. The workshop also featured a problems session, chaired by Zsuzsanna Lipták (Università di Verona, Italy) and Hebert Pérez-Rosés (Universitat de Lleida, Spain). Nine open problems were presented. The IWOCA problem collection can be found on-line at `http://www.iwoca.org/mainproblems.php`.

The 72 first registered participants at IWOCA 2013 hold appointments at institutions in 24 different countries on five continents (Africa, America, Asia,

Europe and Oceania). The nations represented were:
Australia (4), Austria (1), Bangladesh (1), Belgium (1), Canada (6), Germany (4), Hungary (1), Finland (1), France (18), India (1), Israel (1), Italy (1), Japan (2), Korea (4), The Netherlands (1), New Zealand (1), Poland (1), Slovenia (1), South Africa (1), Spain (4), Sweden (2), Switzerland (1), Taiwan (3), UK (9), USA (2).

We thank the authors for their valuable combinatorial contributions and the referees for their thorough, constructive, and enlightening commentaries on the manuscripts and the two secretaries of the LITIS Laboratory, Fabienne Bocquet and Chantal Le Maistre, for their valuable help.

September 2013

Thierry Lecroq
Laurent Mouchard

Organization

Steering Committee

Costas S. Iliopoulos	King's College London, UK
Mirka Miller	University of Newcastle, Australia
William F. Smyth	McMaster University, Canada

Problem Session Co-chairs

Zsuzsanna Lipták	Università di Verona, Italy
Hebert Pérez-Rosés	Universitat de Lleida, Spain

Organizing Committee

Richard Groult	Université de Picardie Jules Verne, France
Thierry Lecroq (Co-chair)	Université de Rouen, France
Arnaud Lefebvre	Université de Rouen, France
Martine Léonard	Université de Rouen, France
Laurent Mouchard (Co-chair)	Université de Rouen, France
Élise Prieur-Gaston	Université de Rouen, France

Program Committee

Donald Adjeroh	West Virginia University
Amihood Amir	Bar-Ilan University and Johns Hopkins University
Dan Archdeacon	University of Vermont
Subramanian Arumugam	Kalasalingam University
Hideo Bannai	Kyushu University
Guillaume Blin	Université Marne-la-Vallée Paris-Est
Ljiljana Brankovic	University of Newcastle
Gerth Stølting Brodal	Aarhus University
Charles Colbourn	Arizona State University
Maxime Crochemore	King's College London
Diane Donovan	University of Queensland
Dalibor Froncek	University of Minnesota Duluth
Roberto Grossi	Universita di Pisa
Michel Habib	Université Paris 7
Sylvie Hamel	University of Montreal
Jan Holub	Czech Technical University in Prague
Seok-Hee Hong	University of Sydney

Costas Iliopoulos	King's College London
Ralf Klasing	LaBRI - CNRS
Rao Kosaraju	Johns Hopkins University
Marcin Kubica	Warsaw University
Gregory Kucherov	LIGM - CNRS
Thierry Lecroq	Université de Rouen
Zsuzsanna Liptak	Universita di Verona
Mirka Miller	University of Newcastle
Laurent Mouchard	Université de Rouen
Ian Munro	University of Waterloo
Rolf Niedermeier	TU Berlin
Pascal Ochem	LIRMM - CNRS
Kunsoo Park	Seoul National University
Hebert Perez-Roses	University of Newcastle
Solon Pissis	Heidelberg Institute for Theoretical Studies
Simon Puglisi	University of Helsinki
Sohel Rahman	Bangladesh University of Engineering and Technology
Rajeev Raman	University of Leicester
Vojtech Rodl	Emory University
Frank Ruskey	University of Victoria
William F. Smyth	McMaster University
Venkatesh Srinivasan	University of Victoria
Iain Stewart	Durham University
German Tischler	Wellcome Trust Sanger Institute
Alexander Tiskin	University of Warwick
Lynette van Zijl	Stellenbosch University
Ambat Vijayakumar	Cochin University of Science and Technology
Koichi Wada	Hosei university
Sue Whitesides	University of Victoria
Christos Zaroliagis	CTI and University of Patras

Additional Reviewers

Barton, Carl
Bhat, Vindya
Biedl, Therese
Blanchet-Sadri, Francine
Blondin Massé, Alexandre
Bong, Novi Herawati
Brlek, Srecko
Brodnik, Andrej
Chapelle, Mathieu
Chen, Jiehua
Christodoulakis, Manolis
Cicalese, Ferdinando
Cooper, Colin
Creignou, Nadia
Csürös, Miklós
Dondi, Riccardo
Erickson, Alejandro
Fertin, Guillaume
Fici, Gabriele
Fink, Martin
Fotakis, Dimitrios
Foucaud, Florent

Frati, Fabrizio
Froese, Vincent
Fuller, Jessica
Fàbrega, Josep
Gadouleau, Maximilien
Gagie, Travis
Gallopoulos, Efstratios
Gambette, Philippe
Giaquinta, Emanuele
Gil, Reynaldo
Hasan, Md. Mahbubul
Hossain, Md. Iqbal
Hsieh, Sun-Yuan
Hüffner, Falk
Islam, A.S.M. Sohidull
Izumi, Taisuke
Janodet, Jean-Christophe
Janson, Svante
Jerrum, Mark
Johanne, Cohen
Johnson, Matthew
Jørgensen, Allan Grønlund
Kamei, Sayaka
Karapetyan, Daniel
Kay, Bill
Kilic, Elgin
Kiniwa, Jun
Komusiewicz, Christian
Kravchenko, Svetlana
Kärkkäinen, Juha
Labarre, Anthony
Ladra, Susana
Langiu, Alessio
Larsen, Victor
Lefebvre, Arnaud
Long, Darell
Lui, Edward
Macgillivray, Gary
Madelaine, Florent
Manlove, David
Meister, Daniel
Milanic, Martin
Miller, Michael
Myoupo, Jean-Frederic
Mömke, Tobias
Newman, Alantha
Nicholson, Patrick K.
Nichterlein, André
Nielsen, Jesper Sindahl
Ono, Hirotaka
Ouangraoua, Aida
Pajak, Dominik
Phanalasy, Oudone
Pineda-Villavicencio, Guillermo
Proietti, Guido
R, J
Radoszewski, Jakub
Razi, Alim Al Islam
Righini, Giovanni
Rusu, Irena
Sacomoto, Gustavo
Saffidine, Abdallah
Sankar, Lalitha
Siantos, Yiannis
Sorge, Manuel
Tischler, German
Van 'T Hof, Pim
van Bevern, René
van Stee, Rob
Varbanov, Zlatko
Vialette, Stéphane
Vildhøj, Hjalte Wedel
Waleń, Tomasz
Wilkinson, Bryan T.

Sponsors

UNIVERSITÉ DE ROUEN

litis

UFR Sciences et Techniques

REGION HAUTE NORMANDIE

Rouen

gdr iM

Invited Papers

Annotating the Human Genome

Ewan Birney

The EMBL-European Bioinformatics Institute
Wellcome Trust Genome Campus
Hinxton, Cambridge
CB10 1SD, United Kingdom
birney@ebi.ac.uk

The human genome is the “hard disk” for human biology, encoding the instructions for each protein and RNA and the elements which regulate their expression. I will provide a perspective of the discovery, annotation and utility of these different components over the last two decades, from the annotation of the draft human genome through to the ENCODE project.

The forthcoming decade will see many more molecular tools being used in clinical research and, in some cases, in practicing medicine. There is a wealth of information and experience from existing use of genetic information in medicine as well as new opportunities available to researchers and practitioners. I will discuss some of the experience I have made in translating this information into the clinic setting.

Finally molecular biology is a leading example of a data intensive science, with both pragmatic and theoretical challenges being raised by data volumes and dimensionality of the data. This shift in modality is creating a wealth of new opportunities and has some accompanying challenges. In particular there is a continued need for a robust information infrastructure for molecular biology. I will briefly outline the challenges and the framework for the solution being the pan-European life sciences information infrastructure, Elixir.

The Δ^2 Conjecture for Graph Labellings with Separation Conditions

Jerrold R. Griggs

Department of Mathematics
University of South Carolina
Columbia, SC 29208 USA
griggs@math.sc.edu

In 1988 Roberts described a problem posed to him by Lanfear concerning the efficient assignment of channels to a network of transmitters in the plane. To understand this problem, Griggs and Yeh introduced the theory of integer vertex λ-labellings of a graph G. To prevent interference, labels for nearby vertices must be separated by specified amounts k_i depending on the distance i, $1 \leq i \leq p$. One seeks the minimum span of such a labelling. The $p = 2$ case with $k_1 = 2$ and $k_2 = 1$ has attracted the most attention, particularly the tantalizing conjecture that for such "$L(2,1)$"-labellings, if G has maximum degree $\Delta \geq 2$, then the minimum span is at most Δ^2. It has now been proven for all sufficiently large Δ, but remains open for small Δ, even for $\Delta = 3$. The theory has been expanded to accommodate real number labellings and separations k_i, with a given separation for each pair of vertices, not necessarily based on distance. Infinite graphs, such as regular lattices, are considered.

Weak Heaps and Friends: Recent Developments

Stefan Edelkamp[1], Amr Elmasry[2], Jyrki Katajainen[3], and Armin Weiß[4]

[1]Faculty 3—Mathematics and Computer Science, University of Bremen
P.O. Box 330 440, 28334 Bremen, Germany
[2]Department of Computer Engineering and Systems, Alexandria University
Alexandria 21544, Egypt
[3]Department of Computer Science, University of Copenhagen
Universitetsparken 5, 2100 Copenhagen East, Denmark
[4]Institute for Formal Methods in Computer Science, University of Stuttgart
Universitätstraße 38, 70569 Stuttgart, Germany

Abstract. A weak heap is a variant of a binary heap where, for each node, the heap ordering is enforced only for one of its two children. In 1993, Dutton showed that this data structure yields a simple worst-case-efficient sorting algorithm. In this paper we review the refinements proposed to the basic data structure that improve the efficiency even further. Ultimately, *minimum* and *insert* operations are supported in $O(1)$ worst-case time and *extract-min* operation in $O(\lg n)$ worst-case time involving at most $\lg n + O(1)$ element comparisons. In addition, we look at several applications of weak heaps. This encompasses the creation of a sorting index and the use of a weak heap as a tournament tree leading to a sorting algorithm that is close to optimal in terms of the number of element comparisons performed. By supporting *insert* operation in $O(1)$ amortized time, the weak-heap data structure becomes a valuable tool in adaptive sorting leading to an algorithm that is constant-factor optimal with respect to several measures of disorder. Also, a weak heap can be used as an intermediate step in an efficient construction of binary heaps. For graph search and network optimization, a weak-heap variant, which allows some of the nodes to violate the weak-heap ordering, is known to be provably better than a Fibonacci heap.

Efficient Exploration of Anonymous Undirected Graphs[*]

Ralf Klasing

CNRS - LaBRI - Université de Bordeaux
351 cours de la Libération, 33405 Talence, France
klasing@labri.fr

Abstract. We consider the problem of exploring an anonymous undirected graph using an oblivious robot. The studied exploration strategies are designed so that the next edge in the robot's walk is chosen using only local information. In this paper, we present some current developments in the area. In particular, we focus on recent work on *equitable strategies* and on the *multi-agent rotor-router*.

[*] The research was partially funded by the ANR project "DISPLEXITY".

New Graph Model for Consistent Superstring Problems

Kunsoo Park

Seoul National University, Korea
kpark@snu.ac.kr

The problems related to string inclusion and non-inclusion have been vigorously studied in such diverse fields as data compression, molecular biology, and computer security. Given a finite set P of positive strings and a finite set N of negative strings, a string A is a consistent superstring if every positive string is a substring of A and no negative string is a substring of A. The shortest (resp. longest) consistent superstring problem is finding a string A that is the shortest (resp. longest) among all the consistent superstrings of the given sets P and N. In this talk, I will present a new graph model for consistent superstrings of P and N. The new graph model is more intuitive than the previous one, and it leads to simpler and more efficient algorithms for consistent superstring problems.

Table of Contents

Weak Heaps and Friends: Recent Developments

Stefan Edelkamp[1], Amr Elmasry[2], Jyrki Katajainen[3], and Armin Weiß[4]

[1] Faculty 3—Mathematics and Computer Science, University of Bremen
P.O. Box 330 440, 28334 Bremen, Germany

[2] Department of Computer Engineering and Systems, Alexandria University
Alexandria 21544, Egypt

[3] Department of Computer Science, University of Copenhagen
Universitetsparken 5, 2100 Copenhagen East, Denmark

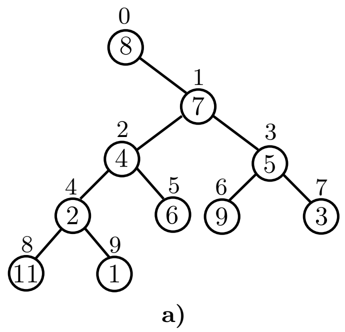
[4] Institute for Formal Methods in Computer Science, University of Stuttgart
Universitätstraße 38, 70569 Stuttgart, Germany

Abstract. A weak heap is a variant of a binary heap where, for each node, the heap ordering is enforced only for one of its two children. In 1993, Dutton showed that this data structure yields a simple worst-case-efficient sorting algorithm. In this paper we review the refinements proposed to the basic data structure that improve the efficiency even further. Ultimately, *minimum* and *insert* operations are supported in $O(1)$ worst-case time and *extract-min* operation in $O(\lg n)$ worst-case time involving at most $\lg n + O(1)$ element comparisons. In addition, we look at several applications of weak heaps. This encompasses the creation of a sorting index and the use of a weak heap as a tournament tree leading to a sorting algorithm that is close to optimal in terms of the number of element comparisons performed. By supporting *insert* operation in $O(1)$ amortized time, the weak-heap data structure becomes a valuable tool in adaptive sorting leading to an algorithm that is constant-factor optimal with respect to several measures of disorder. Also, a weak heap can be used as an intermediate step in an efficient construction of binary heaps. For graph search and network optimization, a weak-heap variant, which allows some of the nodes to violate the weak-heap ordering, is known to be provably better than a Fibonacci heap.

1 Weak Heaps

In its elementary form, a priority queue is a data structure that stores a collection of elements and supports the operations *construct*, *minimum*, *insert*, and *extract-min* [4]. In applications where this set of operations is sufficient, the priority queue that the users would select is a binary heap [30] or a weak heap [7]. Both of these data structures are known to perform well, and in typical cases the difference in performance is marginal. Most library implementations are based on a binary heap. However, one reason why a user might vote for a weak heap over a binary heap is that weak heaps are known to perform less element comparisons in the worst case: Comparing binary heaps vs. weak heaps for *construct* we have $2n$ vs. $n-1$ and for *extract-min* we have $2\lceil \lg n \rceil$ vs. $\lceil \lg n \rceil$

© Springer-Verlag Berlin Heidelberg 2013



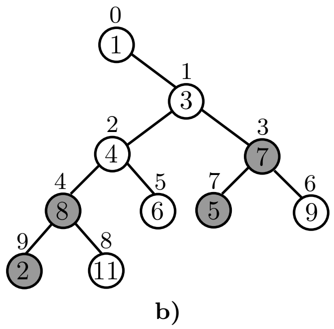


Fig. 1. a) An input of 10 integers and **b)** a weak heap constructed by the standard algorithm (reverse bits are set for grey nodes, small numbers above the nodes are the actual array indices) Source: [11, 12]

element comparisons, where n denotes the number of elements stored in the data structure prior to the operation. Moreover, *minimum* and *insert* have matching complexities 0 and $\lceil \lg n \rceil$ element comparisons, respectively.

More formally, a *weak heap* (see Fig. 1) is a binary tree that has the following properties:

1. The root of the entire tree has no left child.
2. Except for the root, the nodes that have at most one child are at the last two levels only. Leaves at the last level can be scattered, i.e. the last level is not necessarily filled from left to right.
3. Each node stores an element that is smaller than or equal to every element stored in its right subtree.

From the first two properties we can deduce that the height of a weak heap that has n elements is $\lceil \lg n \rceil + 1$. The third property is called *weak-heap ordering* or *half-tree ordering*. In particular, this property does not enforce any relation between an element in a node and those stored in the left subtree of that node. If perfectly balanced, weak heaps resemble heap-ordered binomial trees [28]. Binomial-tree parents are distinguished ancestors in the weak-heap setting.

In an array-based implementation, besides the element array a, an array r of *reverse bits* is used, i.e. $r_i \in \{0, 1\}$ for $i \in \{0, \ldots, n-1\}$. The array index of the left child of a_i is $2i + r_i$, the array index of the right child is $2i + 1 - r_i$, and the array index of the parent is $\lfloor i/2 \rfloor$ (assuming that $i \neq 0$). Using the fact that the indices of the left and right children of a_i are reversed when flipping r_i, subtrees can be swapped in constant time by setting $r_i \leftarrow 1 - r_i$. In a compact representation of a bit array on a w-bit computer $\lceil n/w \rceil$ words are used.

In a pointer-based implementation, the bits are no more needed, but the children and the parent of a node are accessed by following pointers, and the children are reversed by swapping pointers. Pointer-based weak heaps can be used to implement addressable priority queues, which also support *delete* and *decrease* operations [1, 9, 10].

2 Constant-Factor-Optimal Sorting

Dutton [7] showed that to sort n elements WeakHeapSort, a Heapsort variant that uses a weak heap, requires at most $n\lceil \lg n\rceil - 2^{\lceil \lg n\rceil} + n - 1 \leq n\lg n + 0.089n$ element comparisons in the worst case. Algorithms, for which the constant in the leading term in the bound expressing the number of element comparisons performed is the best possible, are called *constant-factor optimal.* Since the early attempts [24], many people have tried to close the gap to the lower bound and to derive constant-factor-optimal algorithms for which the number of primitive operations performed is in $O(n\lg n)$. Other members in the exclusive group of constant-factor-optimal Heapsort algorithms include UltimateHeapsort [22] and MDRHeapsort [26] (analysed by Wegener [29]); the former is fully in-place whereas the latter needs $2n$ extra bits, but for the in-place algorithm the constant α in the bound $n\lg n + \alpha n$ is larger. Knuth [24] showed that MergeInsertion is a sorting algorithm which performs at most $n\lg n - (3 - \lg 3)n + n(\phi + 1 - 2^{\phi}) + O(\lg n)$ element comparisons, where $3 - \lg 3 \approx 1.41$ and $0 \leq \phi \leq 1$. However, when implemented with an array, a quadratic number of element moves may be needed to accomplish the task.

Edelkamp and Wegener [15] gave the worst-case and best-case examples for WeakHeapSort, which match the proved upper bounds. Experimentally they showed that in the average case the number of element comparisons performed is about $n\lg n + \beta n$ with $\beta \in [-0.46, -0.42]$. Edelkamp and Stiegeler [14] showed that for sorting indices (as required in many database applications) WeakHeapSort can be implemented so that it performs at most $n\lg n - 0.91n$ element comparisons, which is only off by about $0.53n$ from the lower bound [24].

Recently, in [16], the idea of QuickHeapsort [2, 5] was generalized to the notion of QuickXsort: Given some black-box sorting algorithm X, QuickXsort can be used to speed X up provided that X satisfies certain natural conditions. QuickWeakHeapsort and QuickMergesort were described as two examples of this construction. QuickMergesort performs $n\lg n - 1.26n + o(n)$ element comparisons on the average and the worst case of $n\lg n + O(n)$ element comparisons can be achieved without affecting the average case. Furthermore, a weak-heap tournament tree yields an efficient implementation of MergeInsertion for small values of n. Taking it as a base case for QuickMergesort, a worst-case-efficient constant-factor-optimal sorting algorithm can be established, which performs $n\lg n - 1.3999n + o(n)$ element comparisons on an average. QuickMergesort with constant-size base cases showed the best performance [16]: When sorting integers it was only 15% slower than Introsort [27] taken from a C++ standard-library implementation.

In [8], two variations of weak heaps were described: The first one uses an array-based weak heap and the other, a *weak queue*, is a collection of pointer-based perfect weak heaps. 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, n being the number of elements stored. In both, the main idea is to temporarily store the inserted elements in a buffer and, once it becomes full, to move the buffer elements to the main queue using an efficient bulk-insertion procedure. By employing the new priority queues in AdaptiveHeapsort [25], the resulting algorithm was shown to be constant-factor optimal with respect to several measures of disorder. Unlike some previous constant-factor-optimal adaptive sorting algorithms [17–19], AdaptiveHeapsort relying on the developed priority queues is practically workable.

3 Relaxed Weak Heaps and Relaxed Weak Queues

In [1], experimental results on the practical efficiency of three addressable priority queues were reported. The data structures considered were a weak heap, a weak queue, and a *run-relaxed weak queue* that extends a weak queue by allowing some nodes to violate the half-heap ordering; a run-relaxed weak queue is a variant of a run-relaxed heap [6] that uses binary trees instead of multiary trees. All the studied data structures support *delete* and *extract-min* in logarithmic worst-case time. A weak queue reduces the worst-case running time of *insert* to $O(1)$, and a run-relaxed weak queue the complexity of both *insert* and *decrease* to $O(1)$. As competitors to these structures, a binary heap, a Fibonacci heap, and a pairing heap were considered. Generic programming techniques were heavily used in the code development. For benchmarking purposes several component frameworks were developed that could be instantiated with different policies.

In [9, 10], two new relaxed priority-queue structures, a *run-relaxed weak heap* and a *rank-relaxed weak heap*, were introduced. The relaxation idea originates from [6], but is applied in a single-tree context. In contrast to run relaxation, rank relaxation provides good amortized performance. Since rank-relaxed weak heaps are simpler and faster, they are better suited for network-optimization algorithms. For a request sequence of n *insert*, m *decrease*, and n *extract-min* operations, it can be shown that a rank-relaxed weak heap performs at most $2m + 1.5n\lceil \lg n \rceil$ element comparisons [9, 10]. When considering the same sequence of operations, this bound improves over the best bounds known for different variants of a Fibonacci heap, which may require $2m + 2.89n\lceil \lg n \rceil$ element comparisons in the worst case.

4 Heap Construction and Optimal In-Place Heaps

In [11, 12], different options for constructing a weak heap were studied. Starting from a straightforward algorithm, the authors ended up with a catalogue of algorithms that optimize the standard algorithm in different ways. As the optimization criteria, it was considered how to reduce the number of instructions, branch mispredictions, cache misses, and element moves. An approach relying on a non-standard memory layout was fastest, but the outcome is a weak heap where the element order is shuffled.

A binary heap can be built on top of a previously constructed navigation pile [23] with at most $0.625n$ element comparisons. In [3], it was shown how this transformation can be used to build binary heaps in-place by performing at most $1.625n + o(n)$ element comparisons. The construction of binary heaps via weak heaps is equally efficient, but this transformation requires a slightly higher number of element moves.

In contrast to binary heaps, n repeated *insert* operations (starting from an empty structure) can be shown to require at most $3.5n + O(\lg^2 n)$ element comparisons [12] (but $\Theta(n \lg n)$ time in the worst case). In addition, with constant memory overhead, $O(1)$ amortized time per *insert* can be improved to $O(1)$ worst-case time [12], while preserving $O(1)$ worst-case time for *minimum* and $O(\lg n)$ worst-case time with at most $\lg n + O(1)$ element comparisons for *extract-min*. This result was previously achieved only for more complicated structures like multipartite priority queues [19]. Still, none of the known constant-factor-optimal worst-case solutions can be claimed to be practical.

As a culmination, in [13], an in-place priority queue was introduced that supports *insert* in $O(1)$ worst-case time and *extract-min* in $O(\lg n)$ worst-case time involving at most $\lg n + O(1)$ element comparisons, n being the current size of the data structure. These upper bounds surpass the lower bounds known for a binary heap [21]. The designed priority queue is similar to a binary heap with two significant exceptions:

– To bypass the lower bound for *extract-min*, at the bottom levels a stronger invariant is enforced: For any node, the element at its left child should never be larger than the element at its right child.
– To bypass the lower bound for *insert*, $O(\lg^2 n)$ nodes are allowed to violate the binary-heap ordering in relation to their parents.

It is necessary to execute several background processes incrementally in order to achieve the optimal worst-case bounds on the number of element comparisons.

References

1. Bruun, A., Edelkamp, S., Katajainen, J., Rasmussen, J.: Policy-based benchmarking of weak heaps and their relatives. In: Festa, P. (ed.) SEA 2010. LNCS, vol. 6049, pp. 424–435. Springer, Heidelberg (2010)
2. Cantone, D., Cinotti, G.: QuickHeapsort, an efficient mix of classical sorting algorithms. Theoret. Comput. Sci. 285(1), 25–42 (2002)
3. Chen, J., Edelkamp, S., Elmasry, A., Katajainen, J.: In-place heap construction with optimized comparisons, moves, and cache misses. In: Rovan, B., Sassone, V., Widmayer, P. (eds.) MFCS 2012. LNCS, vol. 7464, pp. 259–270. Springer, Heidelberg (2012)
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. The MIT Press (2009)
5. Diekert, V., Weiß, A.: QuickHeapsort: Modifications and improved analysis. In: Bulatov, A.A., Shur, A.M. (eds.) CSR 2013. LNCS, vol. 7913, pp. 24–35. Springer, Heidelberg (2013)
6. Driscoll, J.R., Gabow, H.N., Shrairman, R., Tarjan, R.E.: Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. Commun. ACM 31(11), 1343–1354 (1988)
7. Dutton, R.D.: Weak-heap sort. BIT 33(3), 372–381 (1993)

8. Edelkamp, S., Elmasry, A., Katajainen, J.: Two constant-factor-optimal realizations of adaptive heapsort. In: Iliopoulos, C.S., Smyth, W.F. (eds.) IWOCA 2011. LNCS, vol. 7056, pp. 195–208. Springer, Heidelberg (2011)
9. Edelkamp, S., Elmasry, A., Katajainen, J.: The weak-heap family of priority queues in theory and praxis. In: Mestre, J. (ed.) CATS 2012, Conferences in Research and Practice in Information Technology, vol. 128, pp. 103–112. Australian Computer Society, Inc., Adelaide (2012)
10. Edelkamp, S., Elmasry, A., Katajainen, J.: The weak-heap data structure: Variants and applications. J. Discrete Algorithms 16, 187–205 (2012)
11. Edelkamp, S., Elmasry, A., Katajainen, J.: A catalogue of algorithms for building weak heaps. In: Smyth, B. (ed.) IWOCA 2012. LNCS, vol. 7643, pp. 249–262. Springer, Heidelberg (2012)
12. Edelkamp, S., Elmasry, A., Katajainen, J.: Weak heaps engineered. J. Discrete Algorithms (to appear)
13. Edelkamp, S., Elmasry, A., Katajainen, J.: Optimal in-place heaps (submitted)
14. Edelkamp, S., Stiegeler, P.: Implementing Heapsort with $n \log n - 0.9n$ and Quicksort with $n \log n + 0.2n$ comparisons. ACM J. Exp. Algorithmics 7, Article 5 (2002)
15. 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)
16. Edelkamp, S., Weiss, A.: QuickXsort: Efficient sorting with n log n − 1.399n+o(n) comparisons on average. E-print arXiv:1307.3033, arXiv.org, Ithaca (2013)
17. Elmasry, A., Fredman, M.L.: Adaptive sorting: An information theoretic perspective. Acta Inform. 45(1), 33–42 (2008)
18. Elmasry, A., Hammad, A.: Inversion-sensitive sorting algorithms in practice. ACM J. Exp. Algorithmics 13, Article 1.11 (2009)
19. Elmasry, A., Jensen, C., Katajainen, J.: Multipartite priority queues. ACM Trans. Algorithms 5(1), Article 14 (2008)
20. Elmasry, A., Katajainen, J.: Towards ultimate binary heaps. CPH STL Report 2013-3, Department of Computer Science, University of Copenhagen, Copenhagen (2013)
21. Gonnet, G.H., Munro, J.I.: Heaps on heaps. SIAM J. Comput. 15(4), 964–971 (1986)
22. Katajainen, J.: The ultimate heapsort. In: Lin, X. (ed.) CATS 2012, Australian Computer Science Communications, vol. 20, pp. 87–96. Springer-Verlag Singapore Pte. Ltd., Singapore (1998)
23. Katajainen, J., Vitale, F.: Navigation piles with applications to sorting, priority queues, and priority deques. Nordic J. Comput. 10(3), 238–262 (2003)
24. Knuth, D.E.: Sorting and Searching, The Art of Computer Programming, 2nd edn., vol. 3. Addison Wesley Longman, Reading (1998)
25. Levcopoulos, C., Petersson, O.: Adaptive heapsort. J. Algorithms 14(3), 395–413 (1993)
26. McDiarmid, C.J.H., Reed, B.A.: Building heaps fast. J. Algorithms 10(3), 352–365 (1989)
27. Musser, D.R.: Introspective sorting and selection algorithms. Software Pract. Exper. 27(8), 983–993 (1997)
28. Vuillemin, J.: A data structure for manipulating priority queues. Commun. ACM 21(4), 309–315 (1978)
29. Wegener, I.: Bottom-up-Heapsort, a new variant of Heapsort beating, on an average, Quicksort (if n is not very small). Theoret. Comput. Sci. 118(1), 81–98 (1993)
30. Williams, J.W.J.: Algorithm 232: Heapsort. Commun. ACM 7(6), 347–348 (1964)

Efficient Exploration of Anonymous Undirected Graphs*

Ralf Klasing

CNRS - LaBRI - Université de Bordeaux
351 cours de la Libération, 33405 Talence, France
klasing@labri.fr

Abstract. We consider the problem of exploring an anonymous undirected graph using an oblivious robot. The studied exploration strategies are designed so that the next edge in the robot's walk is chosen using only local information. In this paper, we present some current developments in the area. In particular, we focus on recent work on *equitable strategies* and on the *multi-agent rotor-router*.

1 Introduction

A widely studied problem concerns the exploration of an anonymous graph $G = (V, E)$, with the goal of visiting all its vertices and regularly traversing its edges. At each discrete moment of time, the robot is located at a node of the graph, and is provided with only a local view of the adjacent edges of the graph.

The *random walk* is an oblivious exploration strategy in which the edge used by the robot to exit its current location is chosen with equal probability from among all the edges adjacent to the current node; cf. e.g. [2, 21] for an extensive introduction to the topic. A recently proposed deterministic counterpart of the random walk is the *rotor-router* model [24] (a.k.a. the *Propp machine* or the *Edge Ant Walk* [26, 27]), which requires some (small) memory at the nodes of the graph. An alternative deterministic traversal method is the *basic walk*, in which a small amount of memory is provided to an agent and a certain type of graph preprocessing is permitted; cf. e.g. [10, 14, 20] for some recent papers on the latter topic. An extensive survey on graph exploration using the *random walk*, the *rotor-router* and the *basic walk* can be found in [15].

In this paper, we present some current developments in the area of memory efficient exploration of anonymous graphs. In particular, we focus on recent work on *equitable strategies*, in which the environment attempts to mimic the fairness properties of the random walk with respect to the use of edges, and on the *multi-agent rotor-router*, i.e., a rotor-router system in which more than one agent are deployed in the same environment.

* The research was partially funded by the ANR project "DISPLEXITY".

© Springer-Verlag Berlin Heidelberg 2013

2 Locally Fair Exploration Strategies

The exploration strategies studied in this section fall into the line of research devoted to derandomizing random walks in graphs [7,11,26,27].

Explorations achieved through random walks are on average good, in the sense that the following properties hold *in expectation*:

(1) Within polynomial time, the walk visits all of the vertices of the graph.
(2) Within polynomial time, the walk stabilizes to the steady state, and henceforth all edges are visited with the same frequency.

We focus on the problem of designing local exploration strategies which derandomize a random walk in a graph in an attempt to achieve the above stated properties in the deterministic sense of *worst-case performance*. The next vertex to be visited should depend only on the values of certain parameters associated with the edges adjacent to the current node. Such a problem naturally gives rise to the definition of *locally equitable strategies*, i.e. strategies, in which at each step the robot chooses from among the adjacent edges the edge which is in some sense the "poorest", in an effort to make the traversal fair. In this context, two natural notions of equity may be defined:

– An exploration is said to follow the *Oldest-First* (OF) strategy if it directs the robot to an unexplored neighboring edge, if one exists, and otherwise to the neighboring edge for which the most time has elapsed since its last traversal, i.e. the edge which has waited the longest.
– An exploration is said to follow the *Least-Used-First* (LUF) strategy if it directs the robot to a neighboring edge which has so far been visited by the robot the smallest number of times.

When the considered graph is *symmetric and directed*, and the above definitions are applied to directed edges, then the Oldest-First notion of equity is known to be strictly stronger than Least-Used-First, i.e. any exploration which follows the OF strategy also follows the LUF strategy [27]. Moreover, the Oldest-First strategy is in this context equivalent to the well-established efficient exploration model based on the rotor-router model. In the directed case, both of the described locally fair exploration stratagies are known to preserve properties (1) and (2) of the random walk. More precisely, for a symmetric directed graph of diameter D, any exploration which follows such a strategy achieves a *cover time* (i.e., the time until all nodes have been visited at least once) of $O(D\,|E|)$ and stabilizes to a globally fair traversal of all the edges. When the Oldest-First and Least-Used-First strategies are applied to the *undirected* edges of a graph, the results, and the used techniques, turn out to be surprisingly different. More precisely, [8] establishes the following properties of explorations which follow the Oldest-First or Least-Used-First strategies in undirected graphs.

*The Oldest-First (*OF*) strategy* in undirected graphs can be regarded as a natural analogue of the Oldest-First strategy (rotor-router model) for symmetric directed graphs. However, whereas the rotor-router model leads to explorations which traverse directed edges with equal frequency, and have a cover time bounded by $O(D\,|E|)$, this is not the case for Oldest-First explorations in undirected graphs. Indeed, [8] shows the following theorems.

– In some classes of undirected graphs, any exploration which follows the Oldest-First strategy is unfair, with an exponentially large ratio of visits between the most often and least often visited edges.
– There exist explorations following the Oldest-First strategy which have exponential cover time of $2^{\Omega(|V|)}$ in some graph classes.

*The Least-Used-First (*LUF*) strategy* in undirected graphs is fundamentally better than the Oldest-First strategy, which is contrary to the situation in symmetric directed graphs. In fact, [8] shows that, in undirected graphs, explorations which follow the LUF strategy are fair, efficient, and tolerant to perturbations of initial conditions, as expressed by the following theorems.

– Any exploration of an undirected graph which follows the Least-Used-First strategy is fair, achieving uniform distribution of visits to all edges.
– Any exploration of an undirected graph which follows the Least-Used-First strategy achieves a cover time of $O(D\,|E|)$, where D denotes the diameter. This bound is tight. When the exploration starts from a state with non-zero (corrupted) initial values of traversal counts on edges, the cover time is bounded by $O((|V| + p)|E|)$, where p is the maximal value of a counter in the initial state.

Strategies with local equity criteria have also been studied in the token circulation literature, in the context of strategies which are locally fair to vertices rather than edges. Two such strategies, named LF and LR, were proposed and analyzed in [22]. In the first of these, LF, the next vertex to be visited is always chosen as the least-often visited neighbor of the current vertex. In the second, LR, the next vertex to be visited is the neighbor which has not been visited for the longest time. The authors of [22] show that both of these strategies eventually visit all vertices, but in general do not satisfy any fairness criteria. Indeed, the time between successive visits to a vertex may be exponential in the order of the graph for LR, and unbounded for LF. In this sense, the results of [22] may be contrasted with the results of [8] for the LUF strategy.

3 The Multi-agent Rotor-Router

The study of deterministic exploration strategies in agent-based models of computation is largely inspired by considerations of random walk processes. For an undirected graph $G = (V, E)$, exploration with the random walk has many advantageous properties: the expected arrival time of the agent at the last unvisited node of the graph, known as the *cover time* $C(G)$, can in general be bounded as, e.g., $C(G) \in O(D|E|\log|V|)$, where D is the diameter of the graph. The random walk also has the property that in the limit it visits all of the edges of the graph with the same frequency, on average, traversing each once every

$|E|$ rounds. The rotor-router model, introduced by Priezzhev *et al.* [24] and further popularised by James Propp, provides a mechanism for the environment to control the movement of the agent deterministically, whilst retaining similar properties of exploration as the random walk.

In the rotor-router model, the agent has no operational memory and the whole routing mechanism is provided within the environment. The edges outgoing from each node v are arranged in a fixed cyclic order known as a *port ordering*, which does not change during the exploration. Each node v maintains a *pointer* which indicates the edge to be traversed by the agent during its next visit to v. If the agent has not visited node v yet, then the pointer points to an arbitrary edge adjacent to v. The next time when the agent enters node v, it is directed along the edge indicated by the pointer, which is then advanced to the next edge in the cyclic order of the edges adjacent to v.

The behavior of the rotor-router for a single agent is well understood. Yanovski *et al.* [27] showed that, regardless of the initialization of the system, the agent stabilizes to a traversal of a directed Eulerian cycle (containing all of the edges of the graph) within $2D|E|$ steps. A complementary lower bound was provided by Bampas *et al.* [4], who showed that for any graph there exists an initialization of the system for which covering all the nodes of the graph and entering the Eulerian cycle takes $\Theta(D|E|)$ steps. Robustness properties of the rotor-router were further studied in [5], who considered the time required for the rotor-router to stabilize to a (new) Eulerian cycle after an edge is added or removed from the graph.

[18, 27] consider the setting in which multiple, indistinguishable agents are deployed in parallel in the nodes of the graph, and move around the graph in synchronous rounds, interacting with a single rotor-router system. Yanovski *et al.* [27] showed that adding a new agent to the system cannot slow down exploration, and provided some experimental evidence showing a nearly-linear speed-up of cover time with respect to the number of agents in practical scenarios. They also show that the multi-agent rotor-router eventually visits all edges of the graph a similar number of times. In [18], new techniques are proposed which allow to perform a theoretical analysis of the multi-agent rotor-router model, and to compare it to the scenario of parallel independent random walks in a graph [3, 12, 13, 25]. The main results concern the n-node ring, and suggest a strong similarity between the performance characteristics of this deterministic model and random walks.

More precisely, it is shown that on the ring the rotor-router with k agents admits a cover time of between $\Theta(n^2/k^2)$ in the best case and $\Theta(n^2/\log k)$ in the worst case, depending on the initial locations of the agents, and that both these bounds are tight. The corresponding expected value of cover time for k random walks, depending on the initial locations of the walkers, is proven to belong to a similar range, namely between $\Theta(n^2/(k^2/\log^2 k))$ and $\Theta(n^2/\log k)$.

In addition, the limit behavior of the rotor-router system is studied. It is shown that, once the rotor-router system has stabilized, all the nodes of the ring are always visited by some agent every $\Theta(n/k)$ steps, regardless of how the system was initialized. This asymptotic bound corresponds to the expected time between successive visits to a node in the case of k random walks. All the results hold up to a polynomially large number of agents ($1 \leq k < n^{1/11}$).

A variant of the multi-agent rotor-router mechanism has been extensively studied in a different setting, in the context of balancing the workload in a network. The single agent is replaced with a number of agents, referred to as *tokens*. Cooper and Spencer [9] study d-dimensional grid graphs and show a constant bound on the difference between the number of tokens at a given node v in the rotor-router model and the expected number of tokens at v in the random-walk model. Subsequently, Doerr and Friedrich [11] analyse in more detail the distribution of tokens in the rotor-router mechanism on the 2-dimensional grid. Recent work on the multi-agent rotor-router mechanism for balancing the workload in a network comprises e.g. [1, 17].

4 Conclusion and Future Work

In Section 2, we have seen that locally fair strategies in undirected graphs can closely imitate random walks, allowing to obtain an exploration which is fair with respect to all edges, and efficient in terms of cover time. However, the fairness criterion has to be chosen much more carefully than for symmetric directed graphs: Least-Used-First works, but Oldest-First does not.

In future work it would be interesting to study modified notions of equity, which are inspired by random walks which select the next edge to be traversed with non-uniform probability. For example, it is possible to decrease the general-case bound on the cover time of a random walk to $O(|V|^2 \log |V|)$, by applying a probability distribution which reflects the degrees of the nearest neighbors of the current node [16, 19, 23]. It is an open question whether a similar bound can be obtained in the deterministic sense using a derandomized strategy. A somewhat different approach was adopted by Berenbrink *et al.* [6], who show that a random walk with the additional capability of marking one unvisited node in its neighborhood as visited can be used to speed up exploration. It is an open problem whether a similar speedup can be obtained using a derandomized strategy.

In Section 3, we have seen that the muliti-agent rotor-router and the parallel random walk have similar speed-up characteristics w.r.t. the number of deployed agents, at least in terms of cover time and return time on the ring. It is interesting to note that the worst-case speed-up on the ring is $\Theta(\log k)$ for both the k-agent random walk and the k-agent rotor-router, even though this speed up has a different explanation in both cases. For the random walk, it is a consequence of the properties of probability distributions of independent Markovian processes, while for the rotor-router, it results directly from the interactions between different agents and the pointers in the graph.

This work may also be seen as a step in the direction of understanding and characterizing the behavior of the multi-agent rotor-router in graphs different from the ring. Some of the techniques developed in [18], in particular analysis based on delayed deployments, are also applicable in the general case.

References

1. Akbari, H., Berenbrink, P.: Parallel rotor walks on finite graphs and applications in discrete load balancing. In: SPAA, pp. 186–195 (2013)
2. Aldous, D., Fill, J.: Reversible Markov Chains and Random Walks on Graphs (2001), `http://stat-www.berkeley.edu/users/aldous/RWG/book.html`
3. Alon, N., Avin, C., Koucký, M., Kozma, G., Lotker, Z., Tuttle, M.R.: Many random walks are faster than one. Combinatorics, Probability & Computing 20(4), 481–502 (2011)
4. Bampas, E., Gąsieniec, L., Hanusse, N., Ilcinkas, D., Klasing, R., Kosowski, A.: Euler tour lock-in problem in the rotor-router model. In: Keidar, I. (ed.) DISC 2009. LNCS, vol. 5805, pp. 423–435. Springer, Heidelberg (2009)
5. Bampas, E., Gasieniec, L., Klasing, R., Kosowski, A., Radzik, T.: Robustness of the rotor-router mechanism. In: Abdelzaher, T., Raynal, M., Santoro, N. (eds.) OPODIS 2009. LNCS, vol. 5923, pp. 345–358. Springer, Heidelberg (2009)
6. Berenbrink, P., Cooper, C., Elsässer, R., Radzik, T., Sauerwald, T.: Speeding up random walks with neighborhood exploration. In: SODA, pp. 1422–1435 (2010)
7. Bhatt, S.N., Even, S., Greenberg, D.S., Tayar, R.: Traversing directed eulerian mazes. Journal of Graph Algorithms and Applications 6(2), 157–173 (2002)
8. Cooper, C., Ilcinkas, D., Klasing, R., Kosowski, A.: Derandomizing random walks in undirected graphs using locally fair exploration strategies. Distributed Computing 24(2), 91–99 (2011)
9. Cooper, J.N., Spencer, J.: Simulating a random walk with constant error. Combinatorics, Probability & Computing 15(6), 815–822 (2006)
10. Czyzowicz, J., Dobrev, S., Gasieniec, L., Ilcinkas, D., Jansson, J., Klasing, R., Lignos, I., Martin, R., Sadakane, K., Sung, W.-K.: More efficient periodic traversal in anonymous undirected graphs. Theoretical Computer Science 444, 60–76 (2012)
11. Doerr, B., Friedrich, T.: Deterministic random walks on the two-dimensional grid. Combinatorics, Probability & Computing 18(1-2), 123–144 (2009)
12. Efremenko, K., Reingold, O.: How well do random walks parallelize? In: Dinur, I., Jansen, K., Naor, J., Rolim, J. (eds.) APPROX 2009. LNCS, vol. 5687, pp. 476–489. Springer, Heidelberg (2009)
13. Elsässer, R., Sauerwald, T.: Tight bounds for the cover time of multiple random walks. Theoretical Computer Science 412(24), 2623–2641 (2011)
14. Gasieniec, L., Klasing, R., Martin, R.A., Navarra, A., Zhang, X.: Fast periodic graph exploration with constant memory. Journal of Computer and System Sciences 74(5), 808–822 (2008)
15. Gąsieniec, L., Radzik, T.: Memory efficient anonymous graph exploration. In: Broersma, H., Erlebach, T., Friedetzky, T., Paulusma, D. (eds.) WG 2008. LNCS, vol. 5344, pp. 14–29. Springer, Heidelberg (2008)
16. Ikeda, S., Kubo, I., Yamashita, M.: The hitting and cover times of random walks on finite graphs using local degree information. Theoretical Computer Science 410(1), 94–100 (2009)
17. Kijima, S., Koga, K., Makino, K.: Deterministic random walks on finite graphs. In: ANALCO, pp. 18–27 (2012)
18. Klasing, R., Kosowski, A., Pajak, D., Sauerwald, T.: The multi-agent rotor-router on the ring: a deterministic alternative to parallel random walks. In: PODC, pp. 365–374 (2013)
19. Kosowski, A.: A $\tilde{o}\ (n^2)$ time-space trade-off for undirected s-t connectivity. In: SODA, pp. 1873–1883 (2013)

20. Kosowski, A., Navarra, A.: Graph decomposition for memoryless periodic exploration. Algorithmica 63(1-2), 26–38 (2012)
21. Lovász, L.: Random walks on graphs: A survey. Bolyai Society Mathematical Studies 2, 353–397 (1996)
22. Malpani, N., Chen, Y., Vaidya, N.H., Welch, J.L.: Distributed token circulation in mobile ad hoc networks. IEEE Transactions on Mobile Computing 4(2), 154–165 (2005)
23. Nonaka, Y., Ono, H., Sadakane, K., Yamashita, M.: The hitting and cover times of metropolis walks. Theoretical Computer Science 411(16-18), 1889–1894 (2010)
24. Priezzhev, V., Dhar, D., Dhar, A., Krishnamurthy, S.: Eulerian walkers as a model of self-organized criticality. Physical Review Letters 77(25), 5079–5082 (1996)
25. Sauerwald, T.: Expansion and the cover time of parallel random walks. In: PODC, pp. 315–324. ACM (2010)
26. Wagner, I.A., Lindenbaum, M., Bruckstein, A.M.: Distributed covering by ant-robots using evaporating traces. IEEE Transactions on Robotics and Automation 15(5), 918–933 (1999)
27. Yanovski, V., Wagner, I.A., Bruckstein, A.M.: A distributed ant algorithm for efficiently patrolling a network. Algorithmica 37(3), 165–186 (2003)

On Maximum Rank Aggregation Problems*

Christian Bachmaier, Franz Josef Brandenburg,
Andreas Gleißner, and Andreas Hofmeier

University of Passau
94030 Passau, Germany
{bachmaier,brandenb,gleissner,hofmeier}@fim.uni-passau.de

Abstract. The rank aggregation problem consists in finding a consensus ranking on a set of alternatives, based on the preferences of individual voters. These are expressed by permutations, whose distance can be measured in many ways.

In this work we study a collection of distances, including the Kendall tau, Spearman footrule, Spearman rho, Cayley, Hamming, Ulam, and Minkowski distances, and compute the consensus against the maximum, which attempts to minimize the discrimination against any voter.

We provide a general schema from which we can derive the **NP**-hardness of the maximum rank aggregation problems under the aforementioned distances. This reveals a dichotomy for rank aggregation problems under the Spearman footrule and Minkowski distances: the common sum version is solvable in polynomial time whereas the maximum version is **NP**-hard. Moreover, the maximum rank aggregation problems are proved to be 2-approximable under all pseudometrics and fixed-parameter tractable under the Kendall tau, Hamming, and Minkowski distances.

1 Introduction

The task of ranking a list of alternatives is encountered in many situations. One of the underlying goals is to find the best consensus. This task is known as the rank aggregation problem, and was widely studied in the past decade [1, 13]. The problem has numerous applications in sports, voting systems for elections, search engines and evaluation systems on the web.

From mathematical and computational perspectives, the rank aggregation problem is given by a set of m permutations on a set of size n, and the goal is to find a consensus permutation with minimum distance to the given permutations. There are many ways to measure the distance between two permutations and to aggregate the cost by an objective function. Kemeny [19] proposed to count the pairwise disagreements between the orderings of two items, which is commonly known as the Kendall tau distance. For permutations it is the 'bubble sort' distance, i. e., the number of pairwise adjacent transpositions needed to transform one permutation into the other, or the number of crossings in a two-layered

* Supported by the Deutsche Forschungsgemeinschaft (DFG), grant Br835/16-1.

© Springer-Verlag Berlin Heidelberg 2013

drawing [6]. Another popular measure is the Spearman footrule distance [11], which is the L_1-norm of two n-dimensional vectors.

The geometric median of the input permutations is commonly taken for the optimal aggregation, which means the *sum* of the cost of the comparison of each input permutation and the consensus. From the computational perspective this makes a difference between the Spearman footrule and the Kendall tau distance, since the further allows a polynomial time solution via weighted bipartite matching [13], whereas the latter leads to an **NP**-hard rank aggregation problem [3], even for four voters [6,13]. It has an expected $\frac{11}{7}$ randomization [2], a PTAS [20], and is fixed-parameter tractable [5, 18].

Here we study the *maximum version*, which attempts to avoid a discrimination of a single voter or permutation against the consensus. The objective is a minimum k such that all permutations are within distance k from the consensus. Biedl et al. [6] studied this version for the Kendall tau distance and showed that determining whether there is a permutation τ which is within distance at most k to all input permutations, is **NP**-hard, even for any $m \geq 4$ permutations.

There are other distance measures on permutations than the Kendall tau and the Spearman footrule distances. These can be derived from steps in sorting algorithms. In their fundamental study Diaconis and Graham [11] relate the Kendall tau and Spearman footrule distance, and the Spearman rho and Cayley distance. Critchlow [9] added the Hamming and edit distances.

Our main contribution is a general schema for the complexity analysis, which allows us to prove that the maximum rank aggregation problem is **NP**-hard and fixed-parameter tractable under any metric d which satisfies some requirements. These are granted by the aforementioned distances. For the **NP**-hardness results we provide a simpler reduction from the Closest Binary String problem and from the Hitting String problem. Previous reductions used the Feedback Arc Set problem (see [6,13]).

The paper is organized as follows. After some preliminaries in Sect. 2 we show in Sect. 3 that Maximum Ranking (MR) is tractable under the Maximum distance, whereas MR is intractable under many other distances as shown in Sect. 4. In Sect. 5 we establish that MR is 2-approximable for pseudometrics. Finally, in Sect. 6, we present fixed-parameter algorithms to solve MR under various distances.

2 Preliminaries

For a binary relation R on a domain $\mathcal{D}$ and for each $x, y \in \mathcal{D}$, we write $x <_R y$ if $(x, y) \in R$ and $x \not<_R y$ if $(x, y) \notin R$. A binary relation κ is a (strict) *partial order* if it is *irreflexive*, *asymmetric* and *transitive*, i. e., $x \not<_\kappa x$, $x <_\kappa y \Rightarrow y \not<_\kappa x$, and $x <_\kappa y \wedge y <_\kappa z \Rightarrow x <_\kappa z$ for all $x, y, z \in \mathcal{D}$. Candidates x and y are called *unrelated by* κ if $x \not<_\kappa y \wedge y \not<_\kappa x$, which we denote by $x \gtrless_\kappa y$. The intuition of $x <_\kappa y$ is that κ *ranks* x *before* y, which means a preference for x. If $x <_\kappa y$ or $y <_\kappa x$, we speak of a *constraint of* κ *on* x *and* y. For $\mathcal{X}, \mathcal{Y} \subseteq \mathcal{D}$ we denote $\mathcal{X} <_\kappa \mathcal{Y}$ if $\underset{x \in \mathcal{X}}{\forall} \underset{y \in \mathcal{Y}}{\forall} x <_\kappa y$ and define $x <_\kappa \mathcal{Y}$ and $\mathcal{X} <_\kappa y$ accordingly.

A *total order* is a *complete* partial order, i. e., $x <_\tau y \vee y <_\tau x$ for all $x, y \in \mathcal{D}$ with $x \neq y$. Let $n = |\mathcal{D}|$ and $\underline{n} = \{1, \dots, n\}$. For every total order τ there is a unique *permutation*, i. e., a bijection $\tau' : \mathcal{D} \to \underline{n}$ such that $x <_\tau y \Leftrightarrow \tau'(x) < \tau'(y)$. In the rest of the paper we identify total orders and their corresponding permutations, taking the view whichever comes in more handy. The set of all permutations on $\mathcal{D}$ is denoted by $\mathrm{Perm}(\mathcal{D})$. We denote the permutation $\{x_1, \dots, x_n\} \to \underline{n} : x_i \mapsto i$ by $[x_1 x_2 \dots x_n]$.

A total order $\tau \in \mathrm{Perm}(\mathcal{D})$ is a *total extension* of a partial order κ if τ does not contradict κ, i. e., $x <_\kappa y$ implies $x <_\tau y$ for all $x, y \in \mathcal{D}$. We denote the set of total extensions of a partial order κ by $\mathrm{Ext}(\kappa)$.

A *bucket order* is a partial order κ for which unrelatedness $\not\gtrless_\kappa$ is transitive. Then $\not\gtrless_\kappa$ is an equivalence relation whose equivalence classes are called *buckets*. In other words, κ induces a total order order on the buckets while candidates of the same bucket are unrelated, see [1,2,15].

A *transposition* is a permutation on $\underline{n}$ switching the positions of two candidates. Hence, for positions $i, j \in \underline{n}$, we define the transposition $T_{i,j} \in \mathrm{Perm}(\underline{n})$ by $T_{i,j}(i) = j$, $T_{i,j}(j) = i$ and $T_{i,j}(k) = k$ for $k \notin \{i, j\}$. Transpositions can also be considered as operations acting on permutations on $\mathcal{D}$. For $x, y \in \mathcal{D}$ and $\sigma \in \mathrm{Perm}(\mathcal{D})$ we say $T_{\sigma(x),\sigma(y)} \circ \sigma \in \mathrm{Perm}(\mathcal{D})$ is the *transposition of x and y in σ*. Transpositions $T_{i,j}$ of adjacent candidates with $|i - j| = 1$ are called *swaps*.

A binary function $d : \mathrm{Perm}(\mathcal{D}) \times \mathrm{Perm}(\mathcal{D}) \to \mathbb{R}$ is called a *pseudometric* if $d(\sigma, \tau) \geq 0$, $d(\sigma, \tau) = d(\tau, \sigma)$, $\sigma = \tau \Rightarrow d(\sigma, \tau) = 0$, and $d(\sigma, \tau) + d(\tau, \rho) \geq d(\sigma, \rho)$ for all $\sigma, \tau, \rho \in \mathrm{Perm}(\mathcal{D})$. It is a *metric* if, additionally, $\sigma = \tau \Leftrightarrow d(\sigma, \tau) = 0$.

Next we introduce the main concepts of this work: The maximum version of the rank aggregation problem under various distances [9,10].

Definition 1 (Maximum Ranking (MR)).
Instance: A set $\mathcal{D}$ of n candidates, m voters $\sigma_1, \dots, \sigma_m \in \mathrm{Perm}(\mathcal{D})$, $k \in \mathbb{N}$.
Question: Does there exist a permutation $\tau \in \mathcal{D}$ with $\max_{j=1}^{m} d(\sigma_j, \tau) \leq k$?

Then permutation τ is called *k-consensus*. Observe that this is equivalent to say that $d(\sigma_j, \tau) \leq k$ for all voters σ_j, $j \in \underline{m}$.

Let $\sigma, \tau \in \mathrm{Perm}(\mathcal{D})$. Define the set of *dirty* pairs $\mathcal{K}(\sigma, \tau) = \{\{x, y\} \subseteq \mathcal{D} : x <_\sigma y \wedge y <_\tau x\}$ as the set of pairs of candidates $x, y \in \mathcal{D}$ where σ and τ disagree on their order. Then the *Kendall tau distance* K is defined by $K(\sigma, \tau) = |\mathcal{K}(\sigma, \tau)|$. It coincides with the minimum number k of swaps $T_1, \dots, T_k$ such that $\tau = T_k \circ \dots \circ T_1 \circ \sigma$. If we also allow switching non-adjacent candidates, we obtain the *Cayley distance* $C(\sigma, \tau)$, which is the minimum number of transpositions $T_1, \dots, T_k$ such that $\tau = T_k \circ \dots \circ T_1 \circ \sigma$. A permutation on $\underline{n}$ can also be specified by its constituent cycles. A cycle $\mathcal{C} = (x_1 x_2 \dots x_{|\mathcal{C}|})$ of $\rho \in \mathrm{Perm}(\underline{n})$ is a (cyclic) sequence of distinct candidates such that $\rho(x_i) = x_{i+1}$ for $1 \leq i < |\mathcal{C}|$ and $\rho(x_{|\mathcal{C}|}) = x_1$. The cycles form a partition of $\underline{n}$. Denote by $\sharp\mathcal{C}(\rho)$ the number of cycles of ρ. The Cayley distance can be expressed as $C(\sigma, \tau) = n - \sharp\mathcal{C}(\tau \circ \sigma^{-1})$ [10].

Define the set of *displaced* candidates by $\mathcal{H}(\sigma, \tau) = \{x \in \mathcal{D} : \sigma(x) \neq \tau(x)\}$ as the set of candidates $x \in \mathcal{D}$ where σ and τ disagree on their position. The *Hamming distance* H is defined by $H(\sigma, \tau) = |\mathcal{H}(\sigma, \tau)|$, which is the number of positions $i \in \underline{n}$ where $\sigma^{-1}(i) \neq \tau^{-1}(i)$. This view is also taken by the

Hamming distance between strings $s, t \in \{0,1\}^n$, which is defined as $H(s,t) = |\{i \in \underline{n} : s(i) \neq t(i)\}|$ where $s(i)$ denotes the i-th character of s.

Let $\sigma, \tau \in \mathrm{Perm}(\mathcal{D})$. A tuple $(x_1, \ldots, x_l)$ with $x_i \in \mathcal{D}$ is a *common subsequence* of σ and τ if $i < j \Leftrightarrow x_i <_\sigma x_j \wedge x_i <_\tau x_j$. Let $\mathrm{lcs}(\sigma, \tau) = \max\{\, l\colon (x_1, \ldots, x_l)$ is a common subsequence of σ and $\tau \,\}$. Then the *Ulam distance* is $U(\sigma, \tau) = n - \mathrm{lcs}(\sigma, \tau)$.

Finally, the *Minkowski distance* F_p is defined as $F_p(\sigma, \tau) = \left(\sum_{x \in \mathcal{D}} |\sigma(x) - \tau(x)|^p\right)^{\frac{1}{p}}$ for $p \in \mathbb{N} \setminus \{0\}$. F_1 is also known as the Spearman Footrule distance or taxicab metric. F_2 is the Euclidean metric and also known as the Spearman rho distance [10]. To simplify proofs we introduce the notion of the *raised Minkowski distance* $\hat{F}_p$ defined by $\hat{F}_p(\tau, \sigma) = (F_p(\tau, \sigma))^p = \sum_{x \in \mathcal{D}} |\tau(x) - \sigma(x)|^p$.

One can also consider the limit for $p \to \infty$ and $p \to -\infty$. The Chebyshev or *Maximum distance* is $F_\infty(\sigma, \tau) = \max_{x \in \mathcal{D}} |\sigma(x) - \tau(x)|$. Define the *Minimum distance* $F_{-\infty}(\sigma, \tau) = \min_{x \in \mathcal{D}} |\sigma(x) - \tau(x)|$. Note that $F_{-\infty}$ is not a metric and satisfies only non-negativity and symmetry.

3 Efficient Algorithms

Theorem 1. *MR is efficiently solvable under the Maximum distance F_∞.*

Proof. To find a permutation τ satisfying $\max_{j=1}^m \max_{x \in \mathcal{D}} |\sigma_j(x) - \tau(x)| \leq k$, we solve a maximum matching problem in the bipartite graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with vertices $\mathcal{V} = \mathcal{D} \uplus \underline{n}$ and an edge $(x, i) \in \mathcal{E}$ if $\max_{j=1}^m |\sigma_j(x) - i| \leq k$. Every matching of size n corresponds to a k-consensus τ and vice versa. As $|\mathcal{E}| < n(2k+1)$, this can be done in $\mathcal{O}(n^2 \cdot k)$ time. For an improvement observe that the suitable positions for each candidate are consecutive, thus form an interval. Assign to each candidate $x \in \mathcal{D}$ the interval $I_x = \{i \in \underline{n} : \max_{j=1}^m |\sigma_j(x) - i| \leq k\}$. Then iterate over the positions $i \in \underline{n}$. In step i, select the candidate to place at position i. Choose from those candidates x with $i \in I_x$ and which have not been placed before. If there are multiple suitable candidates, prefer a candidate whose interval has the least upper endpoint. In the case that there are no suitable candidates, reject the instance. We use a heap to manage the intervals of unplaced candidates, inserting the interval once we reach its lower endpoint. Determining the endpoints of the intervals can be done in $\mathcal{O}(n \cdot m)$ and the iteration is done in $\mathcal{O}(n \log n)$, resulting in a total running time of $\mathcal{O}(n(\log n + m))$. □

4 Intractability Results

We show that MR is **NP**-complete under the Hamming, Minkowski, Kendall tau, Cayley, Ulam and the Minimum distances. As these distances can be efficiently computed between total orders [4,6,9,21], membership is in **NP**. For the **NP**-hardness proofs we develop a general schema. First we proof that the **NP**-complete Closest Binary String problem [16] can be reduced to a special case of MR under any metric subject to Requirements 1 and 2 defined below. Then we show that these requirements are satisfied by all of the aforementioned metrics except the Minimum distance, for which we provide a reduction from the **NP**-complete Hitting String problem [14].

Definition 2 (Closest Binary String [16]).
Instance: $k, n \in \mathbb{N}$, *a list* $s_1, \dots, s_m \in \{0,1\}^n$ *of* m *binary strings of length* n.
Question: Does there exist a string $t \in \{0,1\}^n$ *with* $\max_{j=1}^{m} H(s_j, t) \leq k$*?*

For the rest of this section, we introduce distinct elements a_i, b_i and sets $\mathcal{B}_i = \{a_i, b_i\}$ for $i \in \underline{n}$ and let $\mathcal{D} = \bigcup_{i=1}^{n} \mathcal{B}_i$. Let κ be the bucket order on $\mathcal{D}$ with buckets $\mathcal{B}_i$ ordered by $\mathcal{B}_1 <_\kappa \dots <_\kappa \mathcal{B}_n$. We call a permutation *local* if it is an extension of κ. We state the following properties to be met by a metric d in order to be applicable in the forthcoming reduction.

Requirement 1 (Optimality of local permutations). *Let* $\sigma_1, \dots, \sigma_m \in \mathrm{Ext}(\kappa)$ *and* $k \in \mathbb{N}$. *If there is a* k*-consensus* $\tau \in \mathrm{Perm}(\mathcal{D})$ *with* $\max_{j=1}^{m} d(\sigma_j, \tau) \leq k$*, then there also is a local permutation* $\tau' \in \mathrm{Ext}(\kappa)$ *with* $\max_{j=1}^{m} d(\sigma_j, \tau') \leq k$.

In other words, if all voters are local and our metric meets Requirement 1, then we can safely demand that the consensus is local, too, without impairing its chance to satisfy the upper bound k. Note that d satisfies Requirement 1 if for every $\sigma \in \mathrm{Ext}(\kappa)$ and $\tau \in \mathrm{Perm}(\mathcal{D})$ we can find $\tau' \in \mathrm{Ext}(\kappa)$ such that $d(\tau', \sigma) \leq d(\tau, \sigma)$. The second requirement puts tight constraints on the distance of local permutations.

Requirement 2 (Distance constraints). *There is a constant* $c > 0$ *such that for all local permutations* $\sigma, \tau \in \mathrm{Ext}(\kappa)$ *the distance is* $d(\sigma, \tau) = c \cdot |\mathcal{K}(\sigma, \tau)|$.

Note that all local permutations agree on the order of candidates from different buckets. Thus, a distance satisfying Requirement 2 is exactly a constant multiple of the number of buckets $\mathcal{B}_i$ where one permutation ranks a_i before b_i and the other ranks b_i before a_i.

Theorem 2. *MR under a metric* d *is* **NP***-hard if* d *satisfies Requirements 1 and 2.*

Proof. Consider an instance of Closest Binary String consisting in a list $s_1, \dots, s_m \in \{0,1\}^n$ of m binary strings of length n and an upper bound $k \in \mathbb{N}$ as in Definition 2. We choose the candidate set $\mathcal{D}$ as defined above. Consider the bijective mapping $f : \{0,1\}^n \to \mathrm{Ext}(\kappa)$, which encodes strings of length n as a local permutation where $a_i <_{f(s)} b_i$ if $s(i) = 0$ and $b_i <_{f(s)} a_i$ if $s(i) = 1$. More formally, $f(s)(a_i) = 2i - 1 + s(i)$ and $f(s)(b_i) = 2i - s(i)$ for all strings $s \in \{0,1\}^n$. For instance, $f(\texttt{"010"}) = [\underbrace{a_1 b_1}_{\mathcal{B}_1} \underbrace{b_2 a_2}_{\mathcal{B}_2} \underbrace{a_3 b_3}_{\mathcal{B}_3}]$. Observe that for strings $s, t \in \{0,1\}^n$ and $i \in \underline{n}$ we have $s(i) \neq t(i)$ if and only if $\{a_i, b_i\} \in \mathcal{K}(f(s), f(t))$. For each string s_j we introduce the voter $\sigma_j = f(s_j)$ and let $k' = c \cdot k$, where c is the constant from Requirement 2.

Suppose that a string $t^* \in \{0,1\}^n$ satisfies $\max_{j=1}^m H(s_j, t^*) \le k$. Let $j \in \underline{m}$. We have

$$\begin{aligned} k' = c \cdot k \ge c \cdot H(s_j, t^*) &= c \cdot |\{i \in \underline{n} : s_j(i) \ne t^*(i)\}| = c \cdot |\mathcal{K}(f(s_j), f(t^*))| \\ &= d(\sigma_j, f(t^*)) \end{aligned}$$

by Requirement 2. Therefore, $f(t^*)$ is a k'-consensus for the MR problem.

Conversely suppose that τ^* satisfies $\max_{j=1}^m d(\sigma_j, \tau^*) \le k'$. W.l.o.g. assume that τ^* is local by Requirement 1. Again, let $j \in \underline{m}$. By Requirement 2 we obtain

$$\begin{aligned} k = \frac{k'}{c} \ge \frac{1}{c} \cdot d(\sigma_j, \tau^*) = |\mathcal{K}(\sigma_j, \tau^*)| &= \left|\{i \in \underline{n} : f^{-1}(\sigma_j) \ne f^{-1}(\tau^*)\}\right| \\ &= H(s_j, f^{-1}(\tau^*)), \end{aligned}$$

i.e., the string $t^* = f^{-1}(\tau^*) \in \{0,1\}^n$ satisfies $\max_{j=1}^m H(s_j, t^*) \le k$. □

Lemma 1. *Let $\sigma, \tau \in \mathrm{Perm}(\mathcal{D})$ and $\{x,y\} \in \mathcal{K}(\sigma,\tau)$ be a dirty pair between σ and τ. Then the Kendall tau distance strictly decreases if we transpose x and y in τ, i.e., $K(\sigma, T_{\tau(x),\tau(y)} \circ \tau) < K(\sigma, \tau)$.*

Proof. Let $\tau' = T_{\tau(x),\tau(y)} \circ \tau$. W.l.o.g. assume $x <_\tau y$. We compare the set $\mathcal{K}^+ = \mathcal{K}(\tau', \sigma) \setminus \mathcal{K}(\tau, \sigma)$ with the set $\mathcal{K}^- = \mathcal{K}(\tau, \sigma) \setminus \mathcal{K}(\tau', \sigma)$. Then $K(\tau', \sigma) < K(\tau, \sigma)$ if $|\mathcal{K}^+| < |\mathcal{K}^-|$. Now, let $Z_<, Z_|$ and $Z_>$ be the candidates that are ranked by σ before, between, and after x and y, respectively. Formally, $Z_< = \{z \in \mathcal{D} : x <_\tau z <_\tau y \wedge z <_\sigma y <_\sigma x\}$, $Z_| = \{z \in \mathcal{D} : x <_\tau z <_\tau y \wedge y <_\sigma z <_\sigma x\}$, and $Z_> = \{z \in \mathcal{D} : x <_\tau z <_\tau y \wedge y <_\sigma x <_\sigma z\}$. By a simple but cumbersome distinction of cases we obtain

$$\begin{aligned} \mathcal{K}^+ &= \bigcup_{z \in Z_<} \{\{y,z\}\} \cup \bigcup_{z \in Z_>} \{\{x,z\}\}, \text{ and} \\ \mathcal{K}^- &= \bigcup_{z \in Z_<} \{\{x,z\}\} \cup \bigcup_{z \in Z_>} \{\{y,z\}\} \cup \bigcup_{z \in Z_|} \{\{x,z\},\{y,z\}\} \cup \{\{x,y\}\}. \end{aligned}$$

Hence, $K(\tau', \sigma) = K(\tau, \sigma) - \left|Z_|\right| - 1$. □

Next we show that Requirements 1 and 2 hold for the Kendall tau, Cayley, Hamming, Ulam, and Minkowski distances.

Lemma 2. *Let τ^* be an optimal consensus for the MR problem under the Kendall tau distance K with voters $\sigma_1, \ldots, \sigma_m$. Let $\mu = \bigcap_{j=1}^m \sigma_j$ be the partial order with $x <_\mu y$ if and only if $x <_{\sigma_j} y$ for all $j \in \underline{m}$. Then $\tau^* \in \mathrm{Ext}(\mu)$.*

Proof. Assume by contradiction that there are candidates $x, y \in \mathcal{D}$ with $x <_\mu y$ and $y <_{\tau^*} x$. Then $x <_{\sigma_j} y$ and $\{x,y\} \in \mathcal{K}(\sigma_j, \tau^*)$ for every $j \in \underline{m}$. Thus, $\max_{j=1}^m d(\sigma_j, T_{\tau^*(x),\tau^*(y)} \circ \tau^*) < \max_{j=1}^m d(\sigma_j, \tau^*)$ by Lemma 1, which is a contradiction to the optimality of τ^*. □

Corollary 1. *The Kendall tau distance K satisfies Requirements 1 and 2.*

Proof. Let $\sigma_1, \dots, \sigma_m \in \mathrm{Ext}(\kappa)$ be local permutations and $\mu = \bigcap_{j=1}^{m} \sigma_j$. Every extension of μ is also an extension of κ since $\kappa \subseteq \mu$. Hence, Requirement 1 follows immediately from Lemma 2. Let $c = 1$. Then Requirement 2 is just the definition of the Kendall tau distance restricted to local permutations. □

Lemma 3. *The Cayley distance C satisfies Requirement 2.*

Proof. Let $\sigma, \tau \in \mathrm{Ext}(\kappa)$ be local permutations. Since σ and τ agree on the order of candidates in different buckets, $\mathcal{K}(\sigma, \tau) \subseteq \{\, \mathcal{B}_i \colon i \in \underline{n}\,\}$. Consider a bucket $\mathcal{B}_i = \{a_i, b_i\}$. If $\mathcal{B}_i \in \mathcal{K}(\sigma, \tau)$, then a_i and b_i form a single cycle $(a_i b_i)$ of size 2 in $\tau \circ \sigma^{-1}$ as $\sigma(a_i) = \tau(b_i)$ and $\sigma(b_i) = \tau(a_i)$. If otherwise $\mathcal{B}_i \notin \mathcal{K}(\sigma, \tau)$, a_i and b_i each form a cycle of size 1. Thus, $C(\sigma, \tau) = 2n - \sharp\mathcal{C}(\tau \circ \sigma^{-1}) = 2n - |\mathcal{K}(\sigma, \tau)| - 2 \cdot |\{\, \mathcal{B}_i \colon i \in \underline{n}\,\} \setminus \mathcal{K}(\sigma, \tau)| = |\mathcal{K}(\sigma, \tau)| = K(\sigma, \tau)$. □

Lemma 4. *The Cayley distance C satisfies Requirement 1, i. e., $C(l(\tau), \sigma) \leq C(\tau, \sigma)$ for every $\sigma \in \mathrm{Ext}(\kappa)$ and $\tau \in \mathrm{Perm}(\mathcal{D})$.*

Proof. For $x \in \mathcal{D}$, denote by $\left|\mathcal{C}_x(\tau \circ \sigma^{-1})\right|$ the size of the cycle in $\tau \circ \sigma^{-1}$ containing x. If $\sigma(x) \neq \tau(x)$, then $\left|\mathcal{C}_x(\tau \circ \sigma^{-1})\right| \geq 2$, but $\left|\mathcal{C}_x(l(\tau) \circ \sigma^{-1})\right| \leq 2$ as shown in the proof of Lemma 3. If otherwise $\sigma(x) = \tau(x)$, then $\left|\mathcal{C}_x(\tau \circ \sigma^{-1})\right| = \left|\mathcal{C}_x(l(\tau) \circ \sigma^{-1})\right| = 1$. Observe that $\sum_{x \in \mathcal{D}} \frac{1}{|\mathcal{C}_x(\tau \circ \sigma^{-1})|} = \sharp\mathcal{C}(\tau \circ \sigma^{-1})$. Hence, $\sharp\mathcal{C}(\tau \circ \sigma^{-1}) \leq \sharp\mathcal{C}(l(\tau) \circ \sigma^{-1})$. □

Proposition 1. *Let $\sigma, \tau \in \mathrm{Perm}(\mathcal{D})$ and $x \in \mathcal{H}(\sigma, \tau)$ be a displaced candidate. Then $H(\sigma, T_{\sigma(x),\tau(x)} \circ \tau) < H(\sigma, \tau)$.*

Proof. Let $y \in \mathcal{D}$ such that $\tau(y) = \sigma(x)$. Note that $y \in \mathcal{H}(\sigma, \tau)$ and the transposition of x and y in τ does not affect other candidates. Thus, $\mathcal{H}(\sigma, T_{\sigma(x),\tau(x)} \circ \tau) = \mathcal{H}(\sigma, \tau) \setminus \{x\}$ or even $\mathcal{H}(\sigma, T_{\sigma(x),\tau(x)} \circ \tau) = \mathcal{H}(\sigma, \tau) \setminus \{x, y\}$ if $\tau(x) = \sigma(y)$. □

Lemma 5. *If $p \in \mathbb{N} \setminus \{0\}$, then the raised Minkowski distance $\hat{F}_p$ satisfies Requirement 1, i. e., $\hat{F}_p(l(\tau), \sigma) \leq \hat{F}_p(\tau, \sigma)$ for every $\sigma \in \mathrm{Ext}(\kappa)$ and $\tau \in \mathrm{Perm}(\mathcal{D})$.*

Proof. Let $x \in \mathcal{D}$. If $\tau(x) = \sigma(x)$ then $l(\tau)(x) = \sigma(x)$ since $x \notin \mathcal{A}_\tau$. Otherwise, $|\tau(x) - \sigma(x)| \geq 1$ implies $|\tau(x) - \sigma(x)|^p \geq 1$, but $|l(\tau)(x) - \sigma(x)| \leq 1$. In both cases $|l(\tau)(x) - \sigma(x)|^p \leq |\tau(x) - \sigma(x)|^p$. □

Proposition 2. *MR under the raised Minkowski distance $\hat{F}_p$ for $p \in \mathbb{N} \setminus \{0\}$ and under the Hamming distance H satisfies Requirement 2.*

Proof. Let $\sigma, \tau \in \mathrm{Ext}(\kappa)$ be local permutations. Recall that $\mathcal{K}(\sigma, \tau) \subseteq \{\, \mathcal{B}_i \colon i \in \underline{n}\,\}$ as σ and τ agree on the order of candidates in different buckets. Hence, $|\tau(x) - \sigma(x)| = |\tau(y) - \sigma(y)| = 1$ for every bucket $\{x, y\} \in \mathcal{K}(\sigma, \tau)$, i. e., both x and y contribute 1 to the distance. Members of the remaining buckets $\{x, y\} \in \{\, \mathcal{B}_i \colon i \in \underline{n}\,\} \setminus \mathcal{K}(\sigma, \tau)$ contribute 0. □

By a similar proof we obtain:

Lemma 6. *The Ulam distance U satisfies Requirement 2.*

For the proof of the following lemma we define the *refinement* of a bucket by a total order as in [7,15]. The *refinement* of a bucket order κ by a total order τ is the total order $\tau * \kappa$ such that $x <_{\tau * \kappa} y \Leftrightarrow x <_\kappa y \vee x \ngtr_\kappa y \wedge x <_\tau y$ holds for all $x, y \in \mathcal{D}$. Note that $\tau * \kappa \in \mathrm{Ext}(\kappa)$.

Lemma 7. *The Ulam distance U satisfies Requirement 1, i. e., $U(\tau * \kappa, \sigma) \leq U(\tau, \sigma)$ for every $\sigma \in \mathrm{Ext}(\kappa)$ and $\tau \in \mathrm{Perm}(\mathcal{D})$.*

Proof. Let $\sigma \in \mathrm{Ext}(\kappa)$, $\tau \in \mathrm{Perm}(\mathcal{D})$, and $(x_1, \ldots, x_l)$ be a *longest* common subsequence of τ and σ, i. e., $l = \mathrm{lcs}(\sigma, \tau)$. As $\sigma \in \mathrm{Ext}(\kappa)$, all elements $x_1, \ldots, x_l$ are ordered by both σ and τ according to κ. Hence, $(x_1, \ldots, x_l)$ is also a common subsequence for $\tau * \kappa$ and σ and thus, $\mathrm{lcs}(\tau * \kappa, \sigma) \geq \mathrm{lcs}(\tau, \sigma)$. □

Theorem 3. *Requirements 1 and 2 are satisfied by the Kendall tau, Cayley, Hamming, Ulam, and Minkowski distances F_p for $p \in \mathbb{N} \setminus \{0\}$. Thus, MR is* **NP***-complete under these distances.*

In consequence we have a dichotomy between the sum and the maximum versions of the rank aggregation problem, in particular for the Spearman footrule distance.

Corollary 2. *For the Minkowski distances F_p and $p \in \mathbb{N} \setminus \{0\}$ (i) the common rank aggregation problem taking the sum is efficiently solvable, and (ii) the maximum rank aggregation problem MR is* **NP***-complete.*

Proof. The common rank aggregation problem can be solved by weighted bipartite matching, where the weights $w_{x,i}$ express the cost of placing x at position i [13], and (ii) follows from Theorem 3. □

Since the Minimum distance does not satisfy Requirement 2, we provide a different reduction from HITTING STRING.

Definition 3 (Hitting String [14]).
Instance: $n \in \mathbb{N}$, a list $s_1, \ldots, s_m \in \{0, 1, *\}^n$ of m strings of length n.
Question: Does there exist a string $t \in \{0, 1\}^n$ such that every string s_j is hit *by t in at least one position, i. e., $\forall j \in \underline{m} : \exists i \in \underline{n} : s_j(i) = t(i)$.*

Theorem 4. *MR under the Minimum distance $F_{-\infty}$ is* **NP***-complete even for $k = 0$.*

Proof. There is a consensus $\tau \in \mathrm{Perm}(\mathcal{D})$ with $\max_{j=1}^{m} \min_{x \in \mathcal{D}} |\sigma_j(x) - \tau(x)| = 0$ if and only if for every σ_j there is a candidate x such that $\sigma_j(x) = \tau(x)$. Then *τ hits σ_j at position $\tau(x)$* and we call *τ hitting consensus.*

First we show how to construct an instance with $2n$ voters of length n which has no hitting consensus. Let $\mathcal{D} = \{u_1, \ldots, u_n\}$ and $\sigma_1 : \mathcal{D} \to \underline{n} : u_i \mapsto i$. We obtain n *primary voters* $\sigma_1, \ldots, \sigma_n$ by rotating σ_1, i. e., for every $j \in \underline{n}$ let $\sigma_j(u_i) = (i + j - 2) \mod n + 1$. Additionally, we introduce the *secondary voters* $\sigma'_1, \ldots, \sigma'_n$ defined by $\sigma'_j = T_{1,2} \circ \sigma_j$. For instance if $\mathcal{D} = \{a, b, c, d, e\}$, then the list of voters is

$$\begin{array}{ll} \sigma_1 = [abcde] & \sigma'_1 = [bacde] \\ \sigma_2 = [eabcd] & \sigma'_2 = [aebcd] \\ \sigma_3 = [deabc] & \sigma'_3 = [edabc] \\ \sigma_4 = [cdeab] & \sigma'_4 = [dceab] \\ \sigma_5 = [bcdea] & \sigma'_5 = [cbdea] \,. \end{array}$$

Assume for contradiction that this list of voters has a hitting consensus τ. Since there are n primary voters and no two primary voters place any candidate at the same position, every primary voter is hit at exactly one position and τ hits exactly one primary voter at position 1. Let σ be the primary voter hit at position 1 by a candidate x. Then τ cannot hit the secondary voter $\sigma' = T_{1,2} \circ \sigma$ at the positions 1 or 2 as $\tau(x) = \sigma(x) = 1 \neq 2 = \sigma'(x)$. Thus, it cannot hit σ' at all since σ and σ' agree in all other positions $\underline{n} \setminus \{1, 2\}$, a contradiction. We call the above list of voters the *n-anti-pattern*. With this in mind, we reduce from the **NP**-complete Hitting String to MR under the Minimum distance.

As in the proof of Theorem 2, let $\mathcal{D} = \bigcup_{i=1}^{n} \{a_i, b_i\}$ be the set of candidates and let $f : \{0,1\}^n \to \mathrm{Perm}(\mathcal{D})$ with $f(s)(a_i) = 2i - 1 + s(i)$ and $f(s)(b_i) = 2i - s(i)$. For every string s_j, $j \in \underline{m}$, we introduce a list of voters Σ_j in two steps. The instance of MR is then the concatenation of all $\Sigma_j, j \in \underline{m}$ and $k = 0$. In the first step we create a *template* $\rho_j : \mathcal{D} \to \underline{n} \cup \{*\}$ from which the actual list is obtained in the second step. Let $\rho_j(a_i) = f(s_j)(a_i)$ and $\rho(b_i) = f(s_j)(b_i)$ if $s(i) \in \{0, 1\}$ and $\rho_j(a_i) = \rho_j(b_j) = *$, otherwise. If none of the strings s_j did contain $*$, then we could establish a one-to-one correspondence between a hitting consensus for voters $\rho_1, \ldots, \rho_m$ and a hitting string for $s_1, \ldots, s_m$ as in Theorem 2 and would be done. Let $\mathcal{U}_j = \{x \in \mathcal{D} : \rho_j = *\}$ be the set of candidates which are not assigned a position by ρ_j. In Hitting String the $*$ marks a position where an input string cannot be hit however the hitting string looks alike. We reproduce this situation for MR by making $2\,|\mathcal{U}_j|$ copies $\sigma_j^{(1)}, \ldots, \sigma_j^{(2|\mathcal{U}_j|)}$ of ρ_j such that all copies agree on the candidates $\mathcal{D} \setminus \mathcal{U}_j$ but form a $|\mathcal{U}_j|$-anti-pattern if the candidate set is restricted to $\mathcal{U}_j$.

Suppose that t^* is a hitting string for $s_1, \ldots, s_m$. Then $f(t^*)$ is a hitting consensus since for every $j \in \underline{m}$ there is an $i \in \underline{n}$ with $t^*(i) = s_j(i)$, thus $f(t^*)(a_i) = \sigma_j(a_i)$. Conversely suppose that τ^* is a hitting consensus. Consider the string $t^* \in \{0,1\}^n$ defined by $t^*(i) = 0$ if $\tau^*(a_i) = 2i - 1 \vee \tau^*(b_i) = 2i$ and $t^*(i) = 1$, otherwise. For every $j \in \underline{m}$ there must be a candidate $x \notin \mathcal{U}_j$ with $\tau(x) = \sigma_j^{(r)}(x)$ for all $\sigma_j^{(r)} \in \Sigma_j$ since they form a $|\mathcal{U}_j|$-anti-pattern when restricted to $\mathcal{U}_j$. The position of $x \in \mathcal{D} \setminus \mathcal{U}_j$ in all $\sigma_j^{(r)} \in \Sigma_j$ is identical and determined by s_j. Therefore, $x = a_i$ or $x = b_i$ for a position i where $s_j(i) \neq *$, and thus, $t^*(i) = s_j(i)$. Hence, t^* is a hitting string. □

5 Approximability

We shortly discuss approximations.

Lemma 8. *The associated minimization problem of MR is 2-approximable for any pseudometric d.*

Proof. Let $\tau^* \in \mathrm{Perm}(\mathcal{D})$ be the optimal consensus for the MR problem under pseudometric d with voters $\sigma_1, \ldots, \sigma_m \in \mathcal{D}$. Then the *pick-a-perm* method [1] with $\tau = \sigma_j$ for $j \in \underline{m}$ yields a 2-approximation since for all $i \in \underline{m}$ we have

$$d(\sigma_i, \tau) \leq d(\sigma_i, \tau^*) + d(\tau^*, \tau) \leq 2 \cdot \max\{d(\sigma_i, \tau^*), d(\tau^*, \tau)\} \leq 2 \cdot \max_{j=1}^{m} d(\sigma_j, \tau^*) \square$$

Note that this approximation ratio for pick-a-perm is tight for all metrics satisfying Requirements 1 and 2. For instance, consider the voters $f(\texttt{"}1000\ldots\texttt{"})$, $f(\texttt{"}0100\ldots\texttt{"})$, $f(\texttt{"}0010\ldots\texttt{"})$ with f as defined in the last section. The distance between each pair of voters is $2c$, while the optimal consensus would be $f(\texttt{"}0000\ldots\texttt{"})$ with a distance of c.

6 Fixed-Parameter Tractability

The reduction in Sect. 4 demonstrates a close relationship between Closest Binary String and MR. We strengthen this observation by extending a fixed parameter algorithm for Closest Binary String [17, 22] such that it can be applied to MR under several metrics. For an introduction to fixed-parameter tractability see [12, 22].

The notion of the *modification set* $M(\tau, \sigma) \subseteq \mathrm{Perm}(\mathcal{D})$ is at the heart of our generalized algorithm. Intuitively, it captures the idea of going "one step" from τ to σ. The structure of the modification set must be chosen individually for each metric d. We state a sufficient condition, which we call the *δ-improving* of M, such that the algorithm actually finds the optimal consensus.

Requirement 3 (δ-improving). *Let $\delta \in \mathbb{N} \setminus \{0\}$. Let $\sigma, \tau, \tau^* \in \mathrm{Perm}(\mathcal{D})$ and $k \in \mathbb{N}$ such that $d(\tau^*, \sigma) \leq k$ and $d(\tau, \tau^*) \leq k$. If $k < d(\tau, \sigma) \leq 2k$, then there exists a $\tau' \in M(\tau, \sigma)$ such that $d(\tau', \tau^*) \leq d(\tau, \tau^*) - \delta$.*

```
   Input: Voters $\sigma_1, \ldots, \sigma_m \in \mathrm{Perm}(\mathcal{D})$, bound $k \in \mathbb{N}$.
   Output: $k$-consensus $\tau^* \in \mathrm{Perm}(\mathcal{D})$ or reject.
1  search$(\sigma_1, k)$;

2  function search$(\tau, \Delta k)$
3      if $\forall j \in \underline{m} : d(\tau, \sigma_j) \leq k$ then return $\{\tau\}$;
4      if $\exists j \in \underline{m} : d(\tau, \sigma_j) > k + \Delta k$ then return $\emptyset$;
5      if $\Delta k > 0$ then
6          let $j \in \underline{m}$ such that $k < d(\tau, \sigma_j) \leq k + \Delta k$;
7          foreach $\tau' \in M(\tau, \sigma_j)$ do
8              $R \leftarrow$ search$(\tau', \Delta k - \delta)$;
9              if $R \neq \emptyset$ then return $R$;
10     return $\emptyset$
```

Algorithm 1. Fixed-parameter algorithm for MR

Lemma 9. *Suppose that there is a k-consensus τ^*, i. e., $\max_{j=1}^{m} d(\tau^*, \sigma_j) \leq k$. If M is δ-improving, then at recursion depth i,* search *in Algorithm 1 has either already found a k-consensus, or is called at least once with a parameter τ such that $d(\tau, \tau^*) \leq k - \delta i$.*

Proof. We proof by induction on the recursion depth i. Induction basis: Since $\tau = \sigma_1$ in depth 0, we have $d(\tau, \tau^*) \leq k - 0$ by definition. Induction step: Suppose the program has not found the solution yet and that at recursion depth $i \leq \lceil \frac{k}{\delta} \rceil$ search is called with τ' having $d(\tau, \tau^*) \leq k - \delta i$. If $d(\tau, \sigma_j) \leq k$ we have found a k-consensus and are done. Otherwise, $d(\tau, \sigma_j) > k$. The break condition in line 4 does not hold since $d(\tau, \sigma_j) \leq d(\tau, \tau^*) + d(\tau^*, \sigma_j) \leq k - \delta i + k = \Delta k + k$. As τ' iterates over $M(\tau, \sigma_j)$, by Requirement 3 there is at least one iteration where search is called with a τ' where $d(\tau', \tau^*) \leq k - \delta i - \delta$. □

Theorem 5. *If M is δ-improving, then Algorithm 1 finds a k-consensus τ^* or correctly reports that no such consensus exists. Its running time is $\mathcal{O}((f(k))^{\lceil \frac{k}{\delta} \rceil} \cdot g(k, n))$, where $f(k)$ is the maximum size of the constructed modification sets and $g(k, n)$ is the time required for the construction of a modification set.*

Proof. The recursion depth is bounded by $\lceil \frac{k}{\delta} \rceil$ and the branching factor is limited by the maximum size of the modification set. The running time is worst if no k-consensus exists, in which case search returns the empty set. Otherwise, suppose that τ^* is a k-consensus. Then, by Lemma 9, search finds a different k-consensus or is eventually called with a τ such that $d(\tau, \tau^*) = 0$ which implies $\tau = \tau^*$. □

For fixed-parameter results it remains to construct a suitable modification set for each distance.

Lemma 10. *The modification set $M(\tau, \sigma) = \{T_{\tau(x), \tau(y)} \circ \tau : \{x, y\} \in \mathcal{K}(\tau, \sigma)\}$ is 1-improving under the Kendall tau distance K.*

Proof. Let $k \in \mathbb{N}$ and $\sigma, \tau, \tau^* \in \text{Perm}(\mathcal{D})$ such that $K(\tau^*, \sigma) \leq k < K(\tau, \sigma) \leq 2k$. We show that $\mathcal{K}(\tau, \sigma) \cap \mathcal{K}(\tau, \tau^*) \neq \emptyset$ since for any dirty pair $\{x, y\} \in \mathcal{K}(\tau, \sigma) \cap \mathcal{K}(\tau, \tau^*)$ we have $d(T_{\tau(x), \tau(y)} \circ \tau, \tau^*) < d(\tau, \tau^*)$ by Lemma 1. Assume for contradiction that $\mathcal{K}(\tau, \sigma)$ and $\mathcal{K}(\tau, \tau^*)$ are disjoint. Let $\{x, y\} \in \mathcal{K}(\tau, \sigma)$. As $\{x, y\} \notin \mathcal{K}(\tau, \tau^*)$, we know that τ and τ^* agree on the relative order of x and y, which implies that $\{x, y\} \in \mathcal{K}(\sigma, \tau^*)$. Hence, $\mathcal{K}(\tau, \sigma) \subseteq \mathcal{K}(\sigma, \tau^*)$. Now let $\{x, y\} \in \mathcal{K}(\tau, \tau^*)$. As $\{x, y\} \notin \mathcal{K}(\tau, \sigma)$, τ and σ agree on the relative order of x and y, implying $\{x, y\} \in \mathcal{K}(\sigma, \tau^*)$. Hence, $\mathcal{K}(\tau, \tau^*) \subseteq \mathcal{K}(\sigma, \tau^*)$. We conclude that $K(\sigma, \tau^*) = |\mathcal{K}(\sigma, \tau^*)| \geq |\mathcal{K}(\tau, \sigma)| + |\mathcal{K}(\tau, \tau^*)| \geq k + 1$, a contradiction. □

Corollary 3. *MR under the Kendall tau distance K can be computed in $\mathcal{O}((2k)^k \cdot (mn \log n + k)\})$ time.*

Proof. Consider the modification set of Lemma 10, whose size is $|M(\tau, \sigma)| = |\mathcal{K}(\tau, \sigma)| = K(\tau, \sigma) \leq 2k$. The distance of two permutations can be computed in $\mathcal{O}(n \log n)$ time [21]. Hence, lines 3, 4 and 6 of Algorithm 1 need $\mathcal{O}(mn \log n)$ time. For efficiency, we represent the modification set $M(\tau, \sigma)$ only implicitly by the set $\mathcal{K}(\tau,\sigma)$ of at most $2k$ dirty pairs, which can be computed in $\mathcal{O}(n \log n + k)$ time [6]. We iterate τ' over $M(\tau,\sigma)$ by transposing the next dirty pair in $\mathcal{K}(\tau,\sigma)$, descent recursively, and undo the transposition after the recursive call returns. Thus, excluding the recursion, the loop requires $\mathcal{O}(n \log n + k)$ time. □

Lemma 11. *The modification set* $M(\tau,\sigma) = \{T_{\tau(x),\sigma(x)} \circ \tau : x \in \mathcal{H}(\tau,\sigma)\}$ *is 1-improving under the Hamming distance* H*.*

Proof. Let $k \in \mathbb{N}$ and $\sigma, \tau, \tau^* \in \mathrm{Perm}(\mathcal{D})$ such that $H(\tau^*,\sigma) \le k < H(\tau,\sigma) \le 2k$. The size of the modification set is $|M(\tau,\sigma)| = |\mathcal{H}(\tau,\sigma)| = H(\tau,\sigma) \le 2k$. σ and τ^* agree in the position of at least $|D|-k$ candidates. As $H(\tau,\sigma) > k$, there is at least one candidate x with $\tau(x) \neq \sigma(x) = \tau^*(x)$. Hence, $H(T_{\tau(x),\sigma(x)} \circ \tau, \tau^*) < H(\tau,\tau^*)$ (see Proposition 1). □

Corollary 4. *MR under the Hamming distance* H *can be computed in* $\mathcal{O}((2k)^k \cdot mn)$ *time.*

Proof. Consider the modification set of Lemma 11. The Hamming distance between two permutations can be computed in linear time. Thus, lines 3, 4 and 6 of Algorithm 1 need $\mathcal{O}(mn)$ time. Similarly to the proof of Corollary 3, the iteration of τ' over the modification set $M(\tau,\sigma)$ with $|M(\tau,\sigma)| \le 2k$ is done in place and needs only $\mathcal{O}(k)$ time. □

Lemma 12. *The modification set* $M(\tau,\sigma) = \{T_{\tau(x),i} \circ \tau : x \in \mathcal{H}(\tau,\sigma) \wedge i \in \{\tau(x) + j \cdot \mathrm{sgn}(\sigma(x) - \tau(x)) : j \in \underline{\lceil k^{\frac{1}{p}} \rceil}\} \cap \underline{n}\}$ *is* $(p+1)$*-improving under the raised Minkowski distance* $\hat{F}_p$ *for* $p \in \mathbb{N} \setminus \{0\}$*.*

Proof. Let $k \in \mathbb{N}$ and $\sigma, \tau, \tau^* \in \mathrm{Perm}(\mathcal{D})$ such that $\hat{F}_p(\tau^*,\sigma) \le k < \hat{F}_p(\tau,\sigma) \le 2k$. We take every displaced candidate $x \in \mathcal{H}(\tau,\sigma)$ and try all possibilities to transpose it with candidates placed at most k positions to its right or left, depending on whether $\sigma(x) > \tau(x)$ or $\sigma(x) < \tau(x)$, respectively. Suppose we have a candidate $x \in \mathcal{H}(\tau,\sigma)$ with $|\sigma(x) - \tau^*(x)| < |\sigma(x) - \tau(x)|$. There must be at least one such candidate since $\hat{F}_p(\tau^*,\sigma) < \hat{F}_p(\tau,\sigma)$. W. l. o. g. assume $\sigma(x) > \tau(x)$. Otherwise, the following arguments apply symmetrically. Let $\mathcal{Y} = \{\tau^{*-1}(i) : i \le \tau(x)\}$ be the set of candidates which are placed in τ^* to the left of or on the same position where x is placed in τ. As $\tau^*(x) > \tau(x)$, $x \notin \mathcal{Y}$, so by a counting argument there must be some $y \in \mathcal{Y}$ with $\tau(y) > \tau(x)$. We know that $\tau' = T_{\tau(x),\tau(y)} \circ \tau$ is contained in the modification set because $\tau(y) - \tau^*(y) \le k^{\frac{1}{p}}$ due to $\hat{F}_p(\tau,\tau^*) \le k$ by Requirement 3. We distinguish two cases whether or not $\tau(y) \le \tau^*(x)$.

Case 1: $\boldsymbol{\tau^*(y) \le \tau(x) < \tau(y) \le \tau^*(x)}$. Then both $\tau^*(x) - \tau'(x) = \tau^*(x) - \tau(y) < \tau^*(x) - \tau(x)$ and $\tau'(y) - \tau^*(y) = \tau(x) - \tau^*(y) < \tau(y) - \tau^*(y)$. Hence, by the Binomial Theorem, $|\tau(x) - \tau^*(x)|^p - |\tau'(x) - \tau^*(x)|^p =$

$$\underbrace{(|\tau(x) - \tau^*(x)| - |\tau'(x) - \tau^*(x)|)}_{\ge 1} \cdot \sum_{i=0}^{p-1} \underbrace{|\tau(x) - \tau^*(x)|^i}_{\ge 1} \cdot \underbrace{|\tau'(x) - \tau^*(x)|^{p-i-1}}_{\ge 1}$$

and thus $|\tau'(x) - \tau^*(x)|^p \leq |\tau(x) - \tau^*(x)|^p - p$. We obtain $|\tau'(y) - \tau^*(y)|^p \leq |\tau(y) - \tau^*(y)|^p - p$ symmetrically. In sum $|\tau'(x) - \tau^*(x)|^p + |\tau'(y) - \tau^*(y)|^p \leq |\tau(x) - \tau^*(x)|^p + |\tau(y) - \tau^*(y)|^p - 2p$.

Case 2: $\boldsymbol{\tau^*(y) \leq \tau(x) < \tau^*(x) < \tau(y)}$. Then $\tau'(x) - \tau^*(x) + \tau'(y) - \tau^*(y) = \tau(y) - \tau^*(x) + \tau(x) - \tau^*(y) < \tau(y) - \tau^*(y)$. By the Binomial Theorem we derive

$$\begin{aligned}(\tau'(x) - \tau^*(x) + \tau'(y) - \tau^*(y))^p &\leq (\tau(y) - \tau^*(y))^p - p\\ |\tau'(x) - \tau^*(x)|^p + |\tau'(y) - \tau^*(y)|^p &\leq |\tau(y) - \tau^*(y)|^p - p + \underbrace{|\tau(x) - \tau^*(x)|^p}_{\geq 1} - 1\end{aligned}$$

Recall that the positions of candidates $\mathcal{D} \setminus \{x, y\}$ are unaffected. Hence, in both cases $\hat{F}_p(\tau', \tau^*) \leq \hat{F}_p(\tau, \tau^*) - (p + 1)$. □

Corollary 5. *MR under the Minkowski distance F_p for $p \in \mathbb{N} \setminus \{0\}$ can be computed in $\mathcal{O}((2k^{p+1})^{\lceil \frac{k^p}{p+1} \rceil} \cdot mn)$ time.*

Proof. Let $\hat{k} = k^p$. Finding a k-consensus for F_p is equivalent to finding a $\hat{k}$-consensus for $\hat{F}_p$. Consider the modification set of Lemma 12. Its size is $|M(\tau, \sigma)| \leq 2\hat{k}^{1+\frac{1}{p}}$ since there are most $2\hat{k}$ displaced candidates which are each tested on at most $\hat{k}^{\frac{1}{p}}$ positions. The Minkowski distance between two permutations can be computed in linear time. Thus, lines 3, 4 and 6 of Algorithm 1 need $\mathcal{O}(mn)$ time. Finding the up to $2\hat{k}$ displaced candidates to build the modification set needs $\mathcal{O}(n)$ time. Each displaced candidate is tested on $\hat{k}^{\frac{1}{p}}$ positions. Then the total running time is in $\mathcal{O}((2\hat{k}^{1+\frac{1}{p}})^{\lceil \frac{\hat{k}}{p+1} \rceil} \cdot mn) = \mathcal{O}((2k^{p+1})^{\lceil \frac{k^p}{p+1} \rceil} \cdot mn)$. □

There are tractable algorithms for Closest Binary String parameterizing the number of strings m [22]. However, parameterizing MR by the number of voters m does not lead to efficient algorithms since MR under the Kendall tau distance is **NP**-hard for $m = 4$ [6].

Note that the **NP**-hardness of MR under the Minimum distance even for $k = 0$ implies that this problem is not fixed-parameter tractable by k unless **P** = **NP**.

7 Conclusion

We explored the complexity of MR by stating sufficient conditions for metrics under which MR is **NP**-complete and fixed-parameter tractable. Considering **NP**-hardness, the Requirements 1 and 2 should also hold for other distances, e. g., Damerau-Levenshtein, Block-Transpositions, or Reversals [9, 10]. Finding a suitable modification set (Requirement 3) for Cayley and Ulam distances is still open. Another extension of MR is to allow the voters providing partial orders. The distance is then measured by the *Nearest Neighbor distance*, which we studied for Spearman footrule and Kendall tau in [7, 8].

References

1. Ailon, N.: Aggregation of partial rankings, p-ratings and top-m lists. Algorithmica 57(2), 284–300 (2010)
2. Ailon, N., Charikar, M., Newman, A.: Aggregating inconsistent information: Ranking and clustering. J. ACM 55(5), 23:1–23:27 (2008)
3. Bartholdi, J.J., Tovey, C.A., Trick, M.A.: Voting schemes for which it can be difficult to tell who won the election. Soc. Choice Welfare 6(2), 157–165 (1989)
4. Bergroth, L., Harri, H., Timo, R.: A survey of longest common subsequence algorithms. In: Proc. String Processing and Information Retrieval, SPIRE 2000, pp. 39–48. IEEE (2000)
5. Betzler, N., Fellows, M.R., Guo, J., Niedermeier, R., Rosamond, F.A.: Fixed-parameter algorithms for Kemeny rankings. Theor. Comput. Sci. 410(8), 4554–4570 (2009)
6. Biedl, T., Brandenburg, F.J., Deng, X.: On the complexity of crossings in permutations. Discrete Math. 309(7), 1813–1823 (2009)
7. Brandenburg, F.J., Gleiβner, A., Hofmeier, A.: The nearest neighbor Spearman footrule distance for bucket, interval, and partial orders. J. Comb. Optim., 1–23 (March 2012)
8. Brandenburg, F.J., Gleiβner, A., Hofmeier, A.: Comparing and aggregating partial orders with Kendall tau distances. Discrete Math. Algorithms Appl. 5(2) (2013)
9. Critchlow, D.E.: Metric Methods for Analyzing Partially Ranked Data. LNS, vol. 34. Springer (1985)
10. Deza, M., Huang, T.: Metrics on permutations, a survey. J. Combin. Inform. System Sci. 23, 173–185 (1998)
11. Diaconis, P., Graham, R.L.: Spearman's footrule as a measure of disarray. J. Roy. Statist. Soc. B 39(2), 262–268 (1977)
12. Downey, R.G., Fellows, M.R.: Parameterized Complexity. Monographs in Computer Science. Springer (1999)
13. Dwork, C., Kumar, R., Naor, M., Sivakumar, D.: Rank aggregation methods for the web. In: Proc. World Wide Web Conference, WWW 2010, pp. 613–622 (2001)
14. Fagin, R.: Generalized first-order spectra and polynomial-time recognizable sets. In: Karp, R. (ed.) Complexity of Computation, SIAM-AMS Proceedings, vol. 7, pp. 43–73. ACM (1974)
15. Fagin, R., Kumar, R., Mahdian, M., Sivakumar, D., Vee, E.: Comparing partial rankings. SIAM J. Discrete Math. 20(3), 628–648 (2006)
16. Frances, M., Litman, A.: On covering problems of codes. Theor. Comput. Syst. 30(2), 113–119 (1997)
17. Gramm, J., Niedermeier, R., Rossmanith, P.: Fixed-parameter algorithms for Closest String and related problems. Algorithmica 37(1), 25–42 (2003)
18. Karpinski, M., Schudy, W.: Faster algorithms for feedback arc set tournament, Kemeny rank aggregation and betweenness tournament. In: Cheong, O., Chwa, K.-Y., Park, K. (eds.) ISAAC 2010, Part I. LNCS, vol. 6506, pp. 3–14. Springer, Heidelberg (2010)
19. Kemeny, J.: Mathematics without numbers. Daedalus 88, 577–591 (1959)
20. Kenyon-Mathieu, C., Schudy, W.: How to rank with few errors. In: Proc. Symposium on Theory of Computing, STOC 2007, pp. 95–103. ACM (2007)
21. Knight, W.R.: A computer method for calculating Kendall's tau with ungrouped data. J. Amer. Statist. Assoc. 61(314), 436–439 (1966)
22. Niedermeier, R.: Invitation to Fixed-Parameter Algorithms. Oxford University Press (2006)

Deciding Representability of Sets of Words of Equal Length in Polynomial Time*

Francine Blanchet-Sadri[1] and Sinziana Munteanu[2]

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

[2] Department of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213–3891, USA
smuntean@cs.cmu.edu

Abstract. De Bruijn sequences of order n *represent* the set A^n of all words of length n over a given alphabet A in the sense that they contain occurrences of each of these words. Recently, the computational problem of representing subsets of A^n by *partial words*, which are sequences that may have holes that match each letter of A, was considered and shown to be in $\mathcal{NP}$. However, membership in $\mathcal{P}$ remained open. In this paper, we show that deciding if a subset is representable can be done in polynomial time. Our approach is graph theoretical.

1 Introduction

A De Bruijn sequence of order n is a word over an alphabet A where each of the words of length n over A occurs as a subword exactly once. These sequences can be efficiently constructed by taking an Eulerian cycle of a De Bruijn graph where every length $n-1$ word corresponds to a vertex and every length n word corresponds to an edge (for alternative constructions see, e.g., [6,10]). De Bruijn sequences are useful and appear in a variety of contexts, e.g., combinatorics on words [1], modern public-key cryptographic schemes, pseudo-random number generation [9], digital fault testing, position sensing schemes [12], non-linear shift registers [5], coding [8], data compression, etc. A vast literature on the De Bruijn sequences exists and generalizations have been explored (e.g., [4,7]).

Algorithmic combinatorics on *partial words* has been developing in the past several years (e.g., [2]). Partial words over an alphabet A are sequences from $A_\diamond = A \cup \{\diamond\}$, where $\diamond \notin A$ is the hole symbol which is *compatible* with every letter in A (*full words* are sequences without holes). If w is a partial word over A, then a *factor* of w is a block of consecutive symbols of w and a *subword* of w is a full word over A compatible with a factor of w. For instance, if we consider the partial word $01\diamond1000$ with one hole over $\{0,1\}$, the full words $101, 111$ are

* This material is based upon work supported by the National Science Foundation under Grant No. DMS–1060775.

© Springer-Verlag Berlin Heidelberg 2013

the subwords compatible with the factor $1\diamond 1$. For any partial word w and integer $n \geq 0$, denote by $\text{sub}_w(n)$ the set of length n subwords of w.

Let S be a set of length n full words and let $h \geq 0$ be an integer. A partial word w such that $\text{sub}_w(n) = S$ is a *representing word* for S and a partial word w with h holes such that $\text{sub}_w(n) = S$ is an *h-representing word* for S. The set S is *representable* if there exists a representing word for S and is *h-representable* if there exists an h-representing word for S. If we consider $S = \{000, 001, 010, 100, 101\}$, then S can be 0-represented by $w = 00010100$, 1-represented by $\diamond 00101$, and 2-represented by $0\diamond 0\diamond$. As we allow more holes, the representing word shrinks.

Let Rep be the problem of deciding whether S is representable and h-Rep be the problem of deciding whether S is h-representable. Blanchet-Sadri and Simmons [3] showed that Rep is in $\mathcal{NP}$. Moreover, they showed that a certain subproblem of Rep is in $\mathcal{P}$, namely the problem of deciding whether a set S of words of length n can be represented by a partial word w, such that every length $n-1$ subword of the words in S occurs exactly once in w or in other words, is compatible with exactly one factor of w. However, whether or not Rep is in $\mathcal{P}$ remained an open problem. They also gave a polynomial-time algorithm (polynomial in the input size $n|S|$) for deciding h-Rep, thus showed that h-Rep is in $\mathcal{P}$, and their algorithm actually constructs an h-representing word.

This paper continues investigating representability of sets of words of equal length. Its contents are as follows: In Section 2, we discuss our graph theoretical approach to Rep. Given a set S of words of equal length n, we describe a decomposition of the Rauzy graph of order $n-1$ associated with S into subgraphs, called blocks, that play a central role in our paper. In Section 3, we describe polynomial-time algorithms for generating the factor set, S^i, and its extension, $\text{Ext}(S^i)$, related to each block i. In Section 4, using the factor sets and their extensions, we give a polynomial-time algorithm for deciding Rep, settling the question "Is Rep in $\mathcal{P}$?". Our algorithm constructs a representing word if S is representable. Finally in Section 5, we conclude with some remarks.

We end this section with some basic concepts. A *partial word* w over a non-empty finite alphabet A is a sequence of symbols from $A \cup \{\diamond\}$, where $\diamond \notin A$ is the hole symbol (a *full word* over A is just a sequence of letters from A). The *length* of a partial word w over A, denoted by $|w|$, is the number of symbols in w. The symbol at position i is denoted by $w[i]$ and the factor $w[i] \cdots w[j-1]$ by $w[i..j)$. A position i is *defined* if $w[i] \in A$ and it is a *hole* if $w[i] = \diamond$. For two partial words w and w' of equal length, w' is *contained in* w, denoted by $w' \subset w$, or w *contains* w', denoted by $w \supset w'$, if $w[i] = w'[i]$ for all defined positions i in w'. Say that w and w' are *compatible*, denoted by $w \uparrow w'$, if $w[i] = w'[i]$ for all positions i that are defined in both w and w'. A *completion* $\hat{w}$ is a full word compatible with a given partial word w. Another way to define compatibility is that w and w' share a common completion, e.g., $01\diamond 0\diamond$ and $\diamond 1\diamond 01$ share the completion 01001 and are thus compatible.

2 Graph Theoretical Approach to Rep

For any graph G, we denote by $V(G)$ the set of vertices of G and by $E(G)$ the set of edges of G. For any $V' \subseteq V(G)$, we denote by $G[V']$ the subgraph of G induced by V'. A digraph is *strongly connected* if, for every pair of vertices u and v, there exists a path from u to v. Computing the strongly connected components of G can be done in $O(|V(G)| + |E(G)|)$ time by using Tarjan's algorithm [11].

Let $S = \{s_0, \ldots, s_{m-1}\}$ be a set of m full words of length n. Define the *Rauzy graph* G of order $n-1$, associated with S, with $V(G) = \bigcup_{i=0}^{m-1}\{s_i[0..n-1), s_i[1..n)\}$ and $E(G) = \bigcup_{i=0}^{m-1}\{(s_i[0..n-1), s_i[1..n)\ )\}$. For every $0 \le i < m$, we label edge $(s_i[0..n-1), s_i[1..n))$ by s_i. It turns out that a 0-representing word for S, if any exists, is a path in the Rauzy graph of order $n-1$ associated with S that includes every edge at least once. Fig. 1 depicts an example of a Rauzy graph that will provide us with examples for the different concepts we introduce.

Define the subgraphs $G^0, \ldots, G^p$ of G inductively:

- Let G^0 be the union of the strongly connected components $G_0^0, \ldots, G_{p_0}^0$ of G such that for every $0 \le j \le p_0$, there exists no edge $(u, v) \in E(G)$ with $u \notin G_j^0$ and $v \in G_j^0$.
- For $0 < i \le p$, let G^i be the union of the strongly connected components $G_0^i, \ldots, G_{p_i}^i$ of $G[V(G) \setminus \bigcup_{j=0}^{i-1} V(G^j)]$ such that for every $0 \le j \le p_i$, there exists no edge $(u, v) \in E(G[V(G) \setminus \bigcup_{j=0}^{i-1} V(G^j)])$ with $u \notin G_j^i$ and $v \in G_j^i$.

For $0 \le i \le p$, call G^i *block* i and $G^0, \ldots, G^p$ the *decomposition of* G *into blocks*. Since each block is a subgraph of G, specifying the vertices of each block uniquely determines its edges.

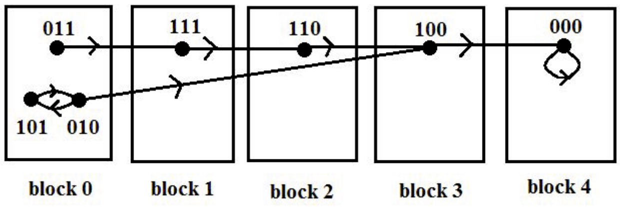


Fig. 1. The decomposition into blocks of the Rauzy graph G of order 3 associated with the set $S = \{0000, 0100, 0101, 0111, 1000, 1010, 1100, 1110\}$; for block 0, $V(G^0) = \{011, 101, 010\}$, $p_0 = 1$, $V(G_0^0) = \{010, 101\}$, and $V(G_1^0) = \{011\}$

Lemma 1. *For every* $0 \le i \le p, 0 \le j, j' \le p_i, j \ne j', u \in V(G_j^i)$ *and* $v \in V(G_{j'}^i)$, *there exist no paths in* G *from* u *to* v. *For every* $0 \le i < i' \le p, u \in V(G^{i'}), v \in V(G^i)$, *there exist no paths in* G *from* u *to* v. *For every* $0 \le i < p$, *there exists* $(u, v) \in E(G)$ *with* $u \in V(G^i)$ *and* $v \in V(G^{i+1})$.

Proof. Suppose the first statement is false. Then there exists an edge $(u', v') \in E(G)$ with $u' \in V(G^i_j)$ and $v' \in V(G^i_{j'})$, contradicting the definition of $G^i_{j'}$. The other two statements are proven with arguments similar to those in [3]. □

However, there can exist an edge $(u, v) \in E(G)$ with $u \in V(G^i), v \in V(G^{i'})$ and $0 \leq i < i' \leq p$. In Fig. 1, $(011, 111), (111, 110), (110, 100), (100, 000) \in E(G)$, so for every $0 \leq i < 4$, there exists $(u, v) \in E(G)$ with $u \in V(G^i)$ and $v \in V(G^{i+1})$. Moreover, $(010, 100) \in E(G)$ with $010 \in V(G^0), 100 \in V(G^3)$, and so there exists an edge with an endpoint in $V(G^i)$ and one in $V(G^{i'})$ with $0 \leq i < i' \leq 4$.

Let $(u, v) \in E(G)$, labelled by the subword $s \in S$. Let $0 \leq i \leq p$. If $u \in V(G^i)$, say edge (u, v) and subword s are *right incident* with G^i. If $v \in V(G^i)$, edge (u, v) and subword s are *left incident* with G^i. If $u \in V(G^i)$ or $v \in V(G^i)$, edge (u, v) and subword $s \in S$ are *incident* with G^i. For $0 \leq j \leq p_i$, we similarly define left incidence, right incidence, and incidence with G^i_j. In Fig. 1 for example, edge $(010, 100)$ is right incident with G^0 and left incident with G^3. Moreover, it is incident with G^0 and G^3.

For every $0 \leq i \leq p$, define the *type of block* i as follows: if $p_i \geq 1$, then block i is of type I; if $p_i = 0$ and there exists $(u, v) \in E(G), 0 \leq j < i < k \leq p$ with $u \in V(G^j)$ and $v \in V(G^k)$, block i is of type II; otherwise, block i is of type III. In Fig. 1, observe that block 0 is of type I, blocks 1 and 2 are of type II, and blocks 3 and 4 are of type III.

Let w be a partial word with $\text{sub}_w(n) = S$. For every $0 \leq i \leq p$, let w^i be the minimum length factor of w that contains all the occurrences of all the subwords incident with G^i; say w^i is the *factor of* w *containing block* i.

- Let $w^{i,left}$ be the minimum length prefix of w^i that contains all the occurrences of all the subwords left, but not necessarily right incident with G^i; say $w^{i,left}$ is the *factor of* w *containing the left part of block* i.
- Let $w^{i,right}$ be the minimum length suffix of w^i that contains all the occurrences of all the subwords right, but not necessarily left incident with G^i; say $w^{i,right}$ is the *factor of* w *containing the right part of block* i.

In Fig. 1, $w = 01\diamond10000$ is a representing word for S; $w^2 = 1\diamond100, w^{2,left} = 1\diamond10$ and $w^{2,right} = \diamond100$.

Lemma 2. *If block* i *is of type I or II, then* $|w^i| \leq 3n - 2$.

Proof. We prove for block i of type I. Let i_0 be the start index of w^i. By the definition of w^i, there exists s_0 incident with G^i occurring at i_0. Suppose without loss of generality that s_0 is incident with G^i_0.

Let i_1 be the largest index at which there exists an occurrence of a subword s_1 incident with G^i_j for some $0 < j \leq p_i$. Suppose $i_0 \leq i_1 - n$. Observe that $s_0 w[i_0 + n..i_1) s_1 \supset w[i_0..i_1 + n)$. If s_0 and s_1 are left incident with G^i, then there exists a path in G from $s_0[1..n)$ to $s_1[1..n)$ and hence there exists a path in G from a vertex in $V(G^i_0)$ to a vertex in $V(G^i_j)$. If s_0 or s_1 are right incident with G^i, we similarly obtain that there exists a path in G from a vertex in $V(G^i_0)$

to a vertex in $V(G^i_j)$ with $0 < j \leq p_i$. But this contradicts Lemma 1. Hence $i_0 > i_1 - n$.

Let i_2 be the largest index at which there exists an occurrence of a subword s_2 incident with G^i_0. Suppose $i_2 \geq i_1 + n$. Since $s_1 w[i_1 + n..i_2) s_2 \supset w[i_1..i_2 + n)$, there exists a path from a vertex in $V(G^i_j)$ to a vertex in $V(G^i_0)$, contradicting Lemma 1. Hence $i_2 < i_1 + n$.

If $i_2 \geq i_1$, then $|w^i| = i_2 + n - i_0 = n + (i_2 - i_1) + (i_1 - i_0) \leq n + (n-1) + (n-1) = 3n-2$. If $i_1 \geq i_2$, then $|w^i| = i_1 + n - i_0 = n + (i_1 - i_0) \leq n + n - 1 = 2n - 1 \leq 3n - 2$. Hence, $|w^i| \leq 3n - 2$. □

Lemma 3. *Both the inequalities $|w^{i,left}| \leq 2n - 1$ and $|w^{i,right}| \leq 2n - 1$ hold.*

Proof. Let i_0 be the start index of w^i. Let s_0 be a subword incident with G^i, occurring at index i_0. Let i_1 be the largest index at which there is an occurrence of a subword left, but not right incident with G^i. Let s_1 be this subword. By Lemma 1, s_1 is right incident with G^k for some $0 \leq k < i \leq p$.

Suppose $i_1 \geq i_0 + n$. Since $s_0 w[i_0 + n..i_1) s_1 \supset w[i_0..i_1 + n)$, there exists a path from a vertex in $V(G^i)$ to one in $V(G^k)$, contradicting Lemma 1. Thus, $i_1 < i_0 + n$.

Therefore, $|w^{i,left}| = i_1 + n - i_0 \leq 2n - 1$. We similarly obtain $|w^{i,right}| \leq 2n - 1$. □

Proposition 1. *There exists an algorithm that given as input a strongly connected digraph G and $v, v' \in V(G)$, outputs in $O(|E(G)|^2)$ time a path from v to v' that contains every edge of G at least once.*

Let block i be of type III (by definition, block i is strongly connected). Let $u, u' \in V(G^i)$ and $u = u_0, \ldots, u_k = u'$ be the vertices of the path output by the algorithm of Proposition 1 given G^i and u, u'. Define $\mathrm{P}(G^i, u, u') = u_0 u_1[n-2] \cdots u_k[n-2]$ to be *the word path* in G^i from u to u'. The set of all length n subwords of $\mathrm{P}(G^i, u, u')$ is the set of all subwords left and right incident with G^i.

Let $0 \leq i \leq p$. If block i is of type I or II, let $S^i = \{x \mid |x| \leq 3n-2, \mathrm{sub}_x(n) \subseteq S$ and all subwords incident with G^i are subwords of $x\}$. If block i is of type III, let $S^i = S^{i,0} \cup S^{i,1}$, where $S^{i,0}, S^{i,1}$ are as follows:

- $S^{i,0} = \{x \mid |x| \leq 4n - 3, \mathrm{sub}_x(n) \subseteq S$ and all subwords incident with G^i are subwords of $x\}$,
- $S^{i,left} = \{xy \mid |x| = 2n - 1, |y| = n - 1, y \in V(G^i)$, $\mathrm{sub}_{xy}(n) \subseteq S$ and all subwords left, but not right incident with G^i are subwords of $xy\}$, while $S^{i,right} = \{y'x' \mid |x'| = 2n - 1, |y'| = n - 1, y' \in V(G^i), \mathrm{sub}_{y'x'}(n) \subseteq S$ and all subwords right, but not left incident with G^i are subwords of $y'x'\}$,
- $S^{i,1} = \{x\mathrm{P}(G^i, y, y')x' \mid xy \in S^{i,left}, y'x' \in S^{i,right}$, $|y| = |y'| = n - 1\}$.

Say $S^{i,0}$ is the set of the *possible small factors containing block* i, $S^{i,1}$ is the set of the *possible large factors containing block* i, and S^i is the set of the *possible factors containing block* i. In Fig. 1, recall that block 3 is of type III. We have $\diamond 1000, 1\diamond 10000, 01\diamond 1000 \in S^{3,0}$. We can show that $S^{3,left} = \emptyset$ and $S^{3,right} = \{1000000000\}$ and hence $S^{3,1} = \emptyset$. Thus, $S^3 = S^{3,0}$.

Let $0 \leq i \leq p$, and let $\text{Ext}(S^i) = \{xyz \mid y \in S^i, |x| < n, |z| < n, \text{sub}_{xyz}(n) \subseteq S\}$, the set of the *extensions of the possible factors containing block* i. If block i is of type III, let $\text{Ext}(S^{i,0}) = \{xyz \mid y \in S^{i,0}, |x| < n, |z| < n, \text{sub}_{xyz}(n) \subseteq S\}$ and $\text{Ext}(S^{i,1}) = \{xyz \mid y \in S^{i,1}, |x| < n, |z| < n, \text{sub}_{xyz}(n) \subseteq S\}$. Say that $\text{Ext}(S^{i,0})$ is the set of the *extensions of the possible small factors containing block* i and $\text{Ext}(S^{i,1})$ is the set of the *extensions of the possible large factors containing block* i. Observe that $\text{Ext}(S^i) = \text{Ext}(S^{i,0}) \cup \text{Ext}(S^{i,1})$.

Returning to our example, we have $\diamond 1000 \in S^3$. All words in $\text{Ext}(S^3)$ of the form xyz with $y = \diamond 1000, |x| < 4, |z| < 4$ are

$$\diamond 1000, \diamond 10000, \diamond 100000, \diamond 1000000, 1\diamond 1000, 1\diamond 10000, 1\diamond 100000,$$
$$1\diamond 1000000, 01\diamond 1000, 01\diamond 10000, 01\diamond 100000, 01\diamond 1000000.$$

3 Generating the Factor Sets and Their Extensions

Given a set S of m words of length n, we give polynomial-time algorithms for generating S^i and $\text{Ext}(S^i)$. We recall from [3] the set $\text{Comp}(S)$ of partial words that have all their completions in S (referring to the set S from Fig. 1, $\diamond 100$, 1000, $01\diamond 1$, 0000, and $\diamond\diamond 00$ are some elements of $\text{Comp}(S)$).

Algorithm 1. Generating X

```
1: generate Comp($S$) using [3, Proposition 6]
2: $X \leftarrow \emptyset$
3: for all $x_0 \in$ Comp($S$) do
4:   for all $x_1 \in$ Comp($S$) do
5:     for $i = 1$ to $n$ do
6:       $X \leftarrow X \cup \{x_0 x_1[i..n)\}$
7:     for all $x_2 \in$ Comp($S$) do
8:       for $i = 1$ to $n$ do
9:         $X \leftarrow X \cup \{x_0 x_1 x_2[i..n)\}$
10:      for all $x_3 \in$ Comp($S$) do
11:        for $i = 3$ to $n$ do
12:          $X \leftarrow X \cup \{x_0 x_1 x_2 x_3[i..n)\}$
```

Algorithm 2. Generating $\text{sub}_x(n)$

```
1: $\text{sub}_x(n) \leftarrow \emptyset$
2: for $i = 0$ to $|x| - n$ do
3:   $y = x[i..i+n)$
4:   for all completions $\hat{y}$ of $y$ do
5:     $\text{sub}_x(n) \leftarrow \text{sub}_x(n) \cup \{\hat{y}\}$
```

Algorithm 3. Generating S^i when block i is of type I or II

```
1: generate $X$ using Algorithm 1
2: generate $G^0, \dots, G^p$
3: $S^i \leftarrow \emptyset$
4: for all $x \in X$ do
5:   if $|x| \leq 3n - 2$ then
6:     generate $\text{sub}_x(n)$ using Algorithm 2
7:     $valid \leftarrow true$
8:     for all $s \in \text{sub}_x(n)$ do
9:       if $s \notin S$ then
10:        $valid \leftarrow false$
11:    for all $s \in S$, $s$ incident with $G^i$, do
12:      if $s \notin \text{sub}_x(n)$ then
13:        $valid \leftarrow false$
14:    if $valid$ then
15:      $S^i \leftarrow S^i \cup \{x\}$
```

Lemma 4. *For any $0 \leq i \leq p$, Algorithm 6 generates S^i in polynomial time and Algorithm 7 generates Ext(S^i) in polynomial time.*

Proof. Suppose block i is of type I or II. We show that the output of Algorithm 3 is correct. We see that if x is added to S^i, then x satisfies all the defining properties of the words in S^i. Now we need to show that if x is a partial word that satisfies all the defining properties of the words in S^i, then x is added to S^i. If $n \leq |x| < 2n$, let $x_0 = x[0..n), x_1 = x[|x| - n..|x|)$. Then at some point, Line 6 of Algorithm 1 adds x to X and so there is an iteration in the loop at Line 4 of Algorithm 3 that corresponds to x. Since x satisfies all the defining properties of the elements in S^i, x is added to S^i in that iteration. So if $n \leq |x| < 2n$, the output of the algorithm is correct. Proceed similarly if $2n \leq |x| \leq 3n - 2$. In all cases, the output is correct.

Suppose block i is of type III. We show that the output of Algorithm 4 is correct. As in the case when block i is of type I or II, we see that the generated sets $S^{i,0}, S^{i,left}$ and $S^{i,right}$ have the desired properties. Moreover, Lines 27–30 generate the set $S^{i,1}$ with the desired properties. So the value of S^i at the end of the iteration is as desired.

From the definition of the type of a block, we see that Algorithm 5 correctly determines the type of block i. Since the outputs of Algorithms 3 and 4 are correct, the output of Algorithm 6 is correct. Similarly, Algorithm 7 correctly generates $\text{Ext}(S^i)$. □

Returning to our example, recall $\diamond 100, 1000 \in \text{Comp}(S)$ and hence $\diamond 1000 \in X$. Algorithm 6 checks the properties of the set S^3 and concludes that $\diamond 1000 \in S^3$. Since moreover, $01\diamond 1, 0000 \in \text{Comp}(S)$, we have $01\diamond 10000 \in X'$ (refer to Algorithm 7's pseudocode). Algorithm 7 checks the properties of the set $\text{Ext}(S^3)$ and concludes that $01\diamond 10000 \in \text{Ext}(S^3)$.

Algorithm 4. Generating S^i when block i is of type III

```
1: generate $X$ using Algorithm 1
2: generate $G^0, \dots, G^p$
3: $S^i \leftarrow \emptyset, S^{i,0} \leftarrow \emptyset, S^{i,1} \leftarrow \emptyset$
4: for all $x \in X$ do
5:    generate $\mathrm{sub}_x(n)$ using Algorithm 2
6:    $valid \leftarrow true$
7:    for all $s \in \mathrm{sub}_x(n)$ do
8:       if $s \notin S$ then
9:          $valid \leftarrow false$
10:   for all $s \in S$, $s$ incident with $G^i$, do
11:      if $s \notin \mathrm{sub}_x(n)$ then
12:         $valid \leftarrow false$
13:   if $valid$ then
14:      $S^{i,0} \leftarrow S^{i,0} \cup \{x\}$
15: $X^{left} \leftarrow \emptyset, X^{right} \leftarrow \emptyset$
16: for all $x \in X$ do
17:    if $|x| = 2n - 1$ then
18:       for all $y \in V(G^i)$ do
19:          $X^{left} \leftarrow X^{left} \cup \{xy\}$ and $X^{right} \leftarrow X^{right} \cup \{yx\}$
20: for all $x \in X^{left}$ $(X^{right})$ do
21:    repeat Lines 6–10 of Algorithm 3
22:    for all $s \in S$ left (right), but not right (left) incident with $G^i$, do
23:       if $s \notin \mathrm{sub}_x(n)$ then
24:          $valid \leftarrow false$
25:    if $valid$ then
26:       $S^{i,left} \leftarrow S^{i,left} \cup \{x\}$ $(S^{i,right} \leftarrow S^{i,right} \cup \{x\})$
27: for all $xy \in S^{i,left}$, with $|y| = n - 1$, do
28:    for all $y'x' \in S^{i,right}$ with $|y'| = n - 1$, do
29:       determine $\mathrm{P}(G^i, y, y')$
30:       $S^{i,1} \leftarrow S^{i,1} \cup \{x\mathrm{P}(G^i, y, y')x'\}$
31: $S^i \leftarrow S^{i,0} \cup S^{i,1}$
```

Algorithm 5. Determining the type of block i

```
1: generate $G^0, \dots, G^p$
2: generate $G^i_0, \dots, G^i_{p_i}$
3: if $p_i \geq 1$ then
4:    return block $i$ is of type I
5: else
6:    for all $(u, v) \in E(G)$ with $u \in V(G^j), v \in V(G^k)$ do
7:       if $j < i < k$ then
8:          return block $i$ is of type II
9: return block $i$ is of type III
```

Algorithm 6. Generating S^i

```
1: determine the type of block $i$ using Algorithm 5
2: if block $i$ is of type I or II (type III) then
3:    generate $S^i$ using Algorithm 3 (Algorithm 4)
```

Algorithm 7. Generating $\mathrm{Ext}(S^i)$

```
1: generate $S^i$ using Algorithm 6
2: $X' \leftarrow \emptyset$
3: **for all** $y \in S^i$ **do**
4:    **for all** $x \in \mathrm{Comp}(S)$ **do**
5:       **for all** $i = 0$ to $n - 1$ **do**
6:          **for all** $z \in \mathrm{Comp}(S)$ **do**
7:             **for all** $j = 1$ to $n$ **do**
8:                $X' \leftarrow X' \cup \{x[0..i)yz[j..n)\}$
9: $\mathrm{Ext}(S^i) \leftarrow \emptyset$
10: **for all** $x \in X'$ **do**
11:    $valid \leftarrow true$
12:    generate $\mathrm{sub}_x(n)$ using Algorithm 2
13:    **for all** $s \in \mathrm{sub}_x(n)$ **do**
14:       **if** $s \notin S$ **then**
15:          $valid \leftarrow false$
16:    **if** $valid$ **then**
17:       $\mathrm{Ext}(S^i) \leftarrow \mathrm{Ext}(S^i) \cup \{x\}$
18: **return** $\mathrm{Ext}(S^i)$
```

Algorithm 8. Deciding REP in polynomial time

```
1: generate $G^0, \ldots, G^p$
2: $V(G') \leftarrow \emptyset$ and $E(G') \leftarrow \emptyset$
3: **for** $i = 0$ to $p$ **do**
4:    generate $\mathrm{Ext}(S^i)$ using Algorithm 7
5:    $V(G') \leftarrow V(G') \cup \mathrm{Ext}(S^i)$
6: **for** $i = 0$ to $p - 1$ **do**
7:    **for all** $u \in \mathrm{Ext}(S^i)$ **do**
8:       **for all** $v \in \mathrm{Ext}(S^{i+1})$ **do**
9:          **for** $r = n$ to $\min(|u|, |v|)$ **do**
10:             **if** $u[|u| - r..|u|) = v[0..r)$ **then**
11:                $E(G') \leftarrow E(G') \cup \{(u, v)\}$
12: **for all** $u^0 \in \mathrm{Ext}(S^0)$ **do**
13:    perform a DFS or BFS traversal in $G'$ starting at $u^0$
14:    **if** there exists $u^p \in \mathrm{Ext}(S^p)$ visited by this traversal **then**
15:       let $u^0, \ldots, u^p$ be in order the vertices in the path from $u^0$ to $u^p$
16:       **for** $i = 1$ to $p$ **do**
17:          **for** $r = n$ to $\min(|u^{i-1}|, |u^i|)$ **do**
18:             **if** $u^{i-1}[|u^{i-1}| - r..|u^{i-1}|) = u^i[0..r)$ **then**
19:                $u^{0,\ldots,i} \leftarrow u^{0,\ldots,i-1}u^i[r..|u^i|)$
20:       **return** $S$ is representable and $u = u^{0,\ldots,p}$ is a representing word
21: **return** $S$ is not representable
```

4 Deciding Rep in Polynomial Time

Algorithm 8 decides Rep in polynomial time. Fig. 2 illustrates our algorithm.

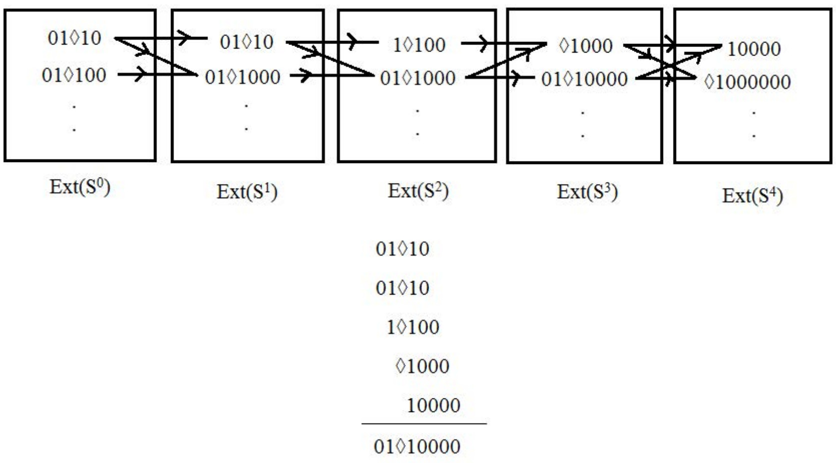


Fig. 2. The graph G' from Algorithm 8 for S in Fig. 1 and construction of the representing word 01⋄10000 for S from the path $01\diamond10 \in Ext(S^0), 01\diamond10 \in Ext(S^1), 1\diamond100 \in Ext(S^2), \diamond1000 \in Ext(S^3), 10000 \in Ext(S^4)$ in G'

Theorem 1. *Algorithm 8 decides if a given set of m words of equal length n is representable and if so, outputs a representing word in polynomial time in the size mn of the input. Therefore,* Rep *is in $\mathcal{P}$.*

Proof. Suppose the algorithm returns a word u. We show that $\text{sub}_u(n) = S$.

Let $s \in S$ and let $0 \leq i \leq p$ be such that s is incident with G^i. If block i is of type I or II, then from the definition of S^i, s is a subword of every word in S^i. If block i is of type III, then s is a subword of every word in $S^{i,0}$. If s is left, but not right incident with G^i, it is a subword of every word in $S^{i,left}$. If it is right, but not left incident with G^i, it is a subword of every word in $S^{i,right}$. If it is both right and left incident with G^i, it is a subword of any $\text{P}(G^i, y, y')$, for any $y, y' \in V(G^i)$. So s is a subword of every word in $S^{i,1}$. Thus, in all cases, s is a subword of every word in S^i. Since every word in $\text{Ext}(S^i)$ has a factor in S^i, s is also a subword of all the words in $\text{Ext}(S^i)$. In particular, $s \in \text{sub}_{u^i}(n)$.

We now prove by induction on i that u^i is a suffix of $u^{0,\ldots,i}$. This clearly holds for $i = 0$. Suppose it holds for $i-1$ and let r_i be the maximum integer such that $u^{i-1}[|u^{i-1}| - r_i..|u^{i-1}|) = u^i[0..r_i)$. Then,

$$\begin{aligned}
u^{0,\dots,i} &= u^{0,\dots,i-1}u^i[r_i..|u^i|)\\
&= u^{0,\dots,i-1}[0..|u^{0,\dots,i-1}|-|u^{i-1}|)u^{i-1}u^i[r_i..|u^i|)\\
&= u^{0,\dots,i-1}[0..|u^{0,\dots,i-1}|-|u^{i-1}|)u^{i-1}[0..|u^{i-1}|-r_i)\\
&\quad u^{i-1}[|u^{i-1}|-r_i..|u^{i-1}|)u^i[r_i..|u^i|)\\
&= u^{0,\dots,i-1}[0..|u^{0,\dots,i-1}|-|u^{i-1}|)u^{i-1}[0..|u^{i-1}|-r_i)u^i[0..r_i)u^i[r_i..|u^i|)\\
&= u^{0,\dots,i-1}[0..|u^{0,\dots,i-1}|-|u^{i-1}|)u^{i-1}[0..|u^{i-1}|-r_i)u^i.
\end{aligned}$$

Hence, u^i is a suffix of $u^{0,\dots,i}$. Since $u^{0,\dots,i}$ is a factor of u, u^i is also a factor of u and since $s \in \mathrm{sub}_{u^i}(n)$, we have that $s \in \mathrm{sub}_u(n)$. Therefore, $S \subseteq \mathrm{sub}_u(n)$.

Now let $s \in \mathrm{sub}_u(n)$ and let i be minimal such that $s \in \mathrm{sub}_{u^{0,\dots,i}}(n)$. Since $u^{0,\dots,i} = u^{0,\dots,i-1}u^i[r_i..|u^i|)$, we have that s is subword of

$$u^{0,\dots,i-1}[|u^{0,\dots,i-1}|-n..|u^{0,\dots,i-1}|)u^i[r_i..|u^i|).$$

By the previous claim, u^{i-1} is a suffix of $u^{0,\dots,i-1}$ and since $|u^{i-1}| \geq n$, the length n suffix of $u^{0,\dots,i-1}$ is the same as the length n suffix of u^{i-1}, i.e., $u^{0,\dots,i-1}[|u^{0,\dots,i-1}|-n..|u^{0,\dots,i-1}|) = u^{i-1}[|u^{i-1}|-n..|u^{i-1}|)$. Since $u^{i-1}[|u^{i-1}|-r_i..|u^{i-1}|) = u^i[0..r_i)$ and $r_i \geq n$, the length n suffix of u^{i-1} is the same as the length n suffix $u^i[0..r_i)$, i.e., $u^{i-1}[|u^{i-1}|-n..|u^{i-1}|) = u^i[r_i-n..r_i)$. So

$$\begin{aligned}
u^{0,\dots,i-1}[|u^{0,\dots,i-1}|-n..|u^{0,\dots,i-1}|) &= u^{i-1}[|u^{i-1}|-n..|u^{i-1}|)\\
&= u^i[r_i-n..r_i)
\end{aligned}$$

and hence

$$\begin{aligned}
u^{0,\dots,i-1}[|u^{0,\dots,i-1}|-n..|u^{0,\dots,i-1}|)u^i[r_i..|u^i|) &= u^i[r_i-n..r_i)u^i[r_i..|u^i|)\\
&= u^i[r_i-n..|u^i|).
\end{aligned}$$

So s is a subword of $u^i[r_i-n..|u^i|)$ and hence also of u^i. Since $u^i \in \mathrm{Ext}(S^i)$, all the length n subwords of u^i are in S. So $s \in S$ and $\mathrm{sub}_u(n) \subseteq S$. Therefore, $S = \mathrm{sub}_u(n)$.

Now suppose that there exists w with $\mathrm{sub}_w(n) = S$. We show that the algorithm decides that S is representable.

For $0 \leq i \leq p$, let w^i be the factor of w containing block i. For all $0 \leq i \leq p$ such that block i is of type III and $|w^i| > 4n-3$, let w^{left} be a completion of $w^i[2n-1..3n-2)$ and w^{right} be a completion of $w^i[|w^i|-3n+2..|w^i|-2n+1)$. Replace in w the factor $w^i[2n-1..|w^i|-2n+1)$ by $\mathrm{P}(G^i, w^{left}, w^{right})$. Observe that since $|w^i| \geq 4n-2$, $|w^i[2n-1..|w^i|-2n+1)| \geq 0$.

Suppose there exists j such that $0 \leq j \leq p, j > i$, block j is of type III, $|w^j| > 4n-3$ and the factors $w^i[2n-1..|w^i|-2n+1)$ and $w^j[2n-1..|w^j|-2n+1)$ overlap. By the definition of w^i and w^j, there exist a completion s_j of $w^j[0..n)$ incident with G^j and a completion s_i of $w^i[|w^i|-n..|w^i|)$ incident with G^i. Since $w^i[2n-1..|w^i|-2n+1)$ and $w^j[2n-1..|w^j|-2n+1)$ overlap, the start index of

$w^j[2n-1..|w^j|-2n+1)$ is smaller than the last index of $w^i[2n-1..|w^i|-2n+1)$. Thus, there exists a path in G from the vertex adjacent to s_j that is in $V(G^j)$ to the vertex adjacent to s_i that is in $V(G^i)$. Since $j > i$, by Lemma 1, we obtain a contradiction. Thus, there exists no such j. Hence, there are no two such transformations that interfere with each other. Let v be the word that results after making these transformations for all $0 \leq i \leq p$.

We show that $\mathrm{sub}_v(n) = S$. It is enough to show that if u is the word obtained after a single transformation, then $\mathrm{sub}_u(n) = S$. Let i_0 be the start index of $w^i[2n-1..|w^i|-2n+1)$ in w and the start index of $\mathrm{P}(G^i, w^{left}, w^{right})$ in u. Let i_1 be the last index of $w^i[2n-1..|w^i|-2n+1)$ in w and i_2 be the last index of $\mathrm{P}(G^i, w^{left}, w^{right})$ in u. Any length n subword of w is a subword of $w[0..i_0), w^i[n..|w^i|-n)$ or $w[i_1..|w|)$. Any length n subword of u is a subword of $u[0..i_0), u[i_0-n+1..i_2+n-1)$ or $u[i_2..|u|)$. Since $w[0..i_0) = u[0..i_0)$ and $w[i_1..|w|) = u[i_2..|u|)$, it is sufficient to show that if $s \in \mathrm{sub}_{w^i[n..|w^i|-n)}(n)$, then $s \in \mathrm{sub}_u(n)$ and if $s \in \mathrm{sub}_{u[i_0-n+1..i_2+n-1)}(n)$, then $s \in \mathrm{sub}_w(n)$.

Let $s \in \mathrm{sub}_{w^i[n..|w^i|-n)}(n)$. By the definition of w^i, there exists a completion s_0 of $w^i[0..n)$ and a completion s_1 of $w^i[|w^i|-n..|w^i|)$ that are incident with G^i. Since $s_0 w^i[n..|w^i|-n) s_1 \supset w^i$, there exists a path from a vertex in $V(G^i)$ to both vertices adjacent to s and from both vertices adjacent to s to a vertex in $V(G^i)$. By Lemma 1, $s \in E(G^i)$ and hence s is a length n subword of $\mathrm{P}(G^i, w^{left}, w^{right})$. So $s \in \mathrm{sub}_u(n)$. Now let $s \in \mathrm{sub}_{u[i_0-n+1..i_2+n-1)}(n)$. Then s is a subword of $u[i_0-n+1..i_0+n-1), u[i_2-n+1..i_2+n-1)$ or $\mathrm{P}(G^i, w^{left}, w^{right})$. If s is a subword of $\mathrm{P}(G^i, w^{left}, w^{right})$, then by construction $s \in E(G^i)$ and hence $s \in \mathrm{sub}_w(n)$.

Suppose $s \in \mathrm{sub}_{u[i_0-n+1..i_0+n-1)}(n)$. Since $u[i_0..i_0+n-1) = w^{left} \supset w[i_0..i_0+n-1)$, we have $u[i_0-n+1..i_0+n-1) \supset w[i_0-n+1..i_0+n-1)$ and hence s is also a subword of $w[i_0-n+1..i_0+n-1)$. Thus, $s \in \mathrm{sub}_w(n)$. Similarly if $s \in \mathrm{sub}_{u[i_2-n+1..i_2+n-1)}(n)$.

Therefore, $\mathrm{sub}_v(n) = S$. For $0 \leq i \leq p$, let v^i be the factor of v containing block i and let st_i and f_i be the start and last indices of v^i in v, respectively. Let $st_i^{ext} = \min(st_i, \ldots, st_p)$ and $f_i^{ext} = \max(f_1, \ldots, f_i)$. Let $\mathrm{Ext}(v^i) = v[st_i^{ext}..f_i^{ext})$.

Suppose $st_i \geq st_j + n$ for some $0 \leq i < j \leq p$. Then there is a length $n-1$ subword of $v[st_j..st_j+n)$ that is in $V(G^j)$ and there is a length $n-1$ subword of $v[st_i..st_i+n)$ that is in $V(G^i)$. Hence there is a path from a vertex in $V(G^j)$ to a vertex in $V(G^i)$, a contradiction. Hence, $st_i < st_j + n$, for all $0 \leq i < j \leq p$ and thus $st_i^{ext} > st_i - n$. Similarly, $f_i^{ext} < f_i + n$.

By the definition of v^i and S^i, we have $v^i \in S^i$. Moreover, since $st_i^{ext} > st_i - n$ and $f_i^{ext} < f_i + n$, we have $\mathrm{Ext}(v^i) \in \mathrm{Ext}(S^i)$ and hence $\mathrm{Ext}(v^i) \in V(G')$. By definition, $st_i^{ext} \leq st_{i+1}^{ext}$ and $f_i^{ext} \leq f_{i+1}^{ext}$ and hence there exists r_i with $\mathrm{Ext}(v^i)[|\mathrm{Ext}(v^i)| - r_i..|\mathrm{Ext}(v^i)|) = \mathrm{Ext}(v^{i+1})[0..r_i)$. By Lemma 1, there exists a subword s incident with G^i and G^{i+1}. Hence, all occurrences of s are in v^i and in v^{i+1}. Since v^i is a factor of $\mathrm{Ext}(v^i)$ and v^{i+1} is a factor of $\mathrm{Ext}(v^{i+1})$, all occurrences of s are also in $\mathrm{Ext}(v^i)$ and in $\mathrm{Ext}(v^{i+1})$ and hence in their intersection $\mathrm{Ext}(v^i)[|\mathrm{Ext}(v^i)| - r_i..|\mathrm{Ext}(v^i)|) = \mathrm{Ext}(v^{i+1})[0..r_i)$. Thus $r_i \geq n$

and $(\mathrm{Ext}(v^i), \mathrm{Ext}(v^{i+1})) \in E(G')$. Therefore, $\mathrm{Ext}(v^0), \dots, \mathrm{Ext}(v^p)$ is a path in G'. This path is found using the BFS traversal starting at $\mathrm{Ext}(v^0)$. Hence the algorithm decides that S is representable.

□

5 Conclusion

In this paper, we showed that deciding representability of a subset S of A^n, where A denotes an alphabet, can be done in polynomial time in the size of S, answering a problem left open in [3]. Motivation behind this computational problem is to represent such subsets by generalized De Bruijn sequences, i.e., sequences over A that only contain the words in S. By allowing sequences to have holes, we can actually find shorter representing words. A suggested topic for future research would be to investigate the problem of computing a shortest representing word, i.e., a minimal generating sequence or a sequence of minimal length that produces all words in S.

References

1. Berstel, J., Perrin, D.: The origins of combinatorics on words. European Journal of Combinatorics 28(3), 996–1022 (2007)
2. Blanchet-Sadri, F.: Algorithmic Combinatorics on Partial Words. Chapman & Hall/CRC Press, Boca Raton (2008)
3. Blanchet-Sadri, F., Simmons, S.: Deciding representability of sets of words of equal length. Theoretical Computer Science 475, 34–46 (2013)
4. Chung, F., Diaconis, P., Graham, R.: Universal cycles for combinatorial structures. Discrete Mathematics 110, 43–59 (1992)
5. Fredericksen, H.: A survey of full length nonlinear shift register cycle algorithms. SIAM Review 24, 195–221 (1982)
6. Fredricksen, H., Maiorana, J.: Necklaces of beads in k colors and k-ary de Bruijn sequences. Discrete Mathematics 23(3), 207–210 (1978)
7. Hurlbert, G., Isaak, G.: On the de Bruijn torus problem. Journal of Combinatorial Theory, Series A 64(1), 50–62 (1993)
8. Lind, D., Marcus, B.: Symbolic Dynamics and Codings. Cambridge University Press (1995)
9. van Lint, J.H., MacWilliams, F.J., Sloane, N.J.A.: On pseudo-random arrays. SIAM Journal on Applied Mathematics 36, 62–72 (1979)
10. Martin, M.H.: A problem in arrangements. Bulletin of the American Mathematical Society 40(12), 859–864 (1934)
11. Tarjan, R.: Depth-first search and linear graph algorithms. SIAM Journal on Computing 1(2), 146–160 (1972)
12. Tuliani, J.: De Bruijn sequences with efficient decoding algorithms. Discrete Mathematics 226(1-3), 313–336 (2001)

Prefix Table Construction and Conversion[⋆]

Widmer Bland[1], Gregory Kucherov[2], and W.F. Smyth[1,3]

[1] Algorithms Research Group, Department of Computing & Software
McMaster University, Hamilton, Ontario, Canada L8S 4K1
{blandwa,smyth}@mcmaster.ca

[2] Laboratoire d'Informatique Gaspard-Monge
Champs sur Marne 77454, Marne-la-Vallée, Cedex 2, France
Gregory.Kucherov@univ-mlv.fr

[3] School of Mathematics & Statistics
University of Western Australia, Crawley WA 6009, Australia

Abstract. The *prefix table* of a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is an array $\pi = \pi[1..n]$ such that $\pi[i]$ is the length of the longest substring beginning at i that equals a prefix of $\boldsymbol{x}$. In this paper we describe and evaluate algorithms for prefix table construction, some previously proposed, others designed by us. We also describe and evaluate new linear-time algorithms for transformations between π and the *border array.*

1 Introduction

This paper deals with two important data structures used in many algorithms on strings: the border array and the prefix table[1]. The *border array* $\beta[1..n]$ of a given string $\boldsymbol{x}$ is an array (string) of nonnegative integers such that, for every $i \in 1..n$, $\beta[i]$ is the length of the longest *border* of $\boldsymbol{x}[1..i]$, i.e. the longest proper suffix of $\boldsymbol{x}[1..i]$ that is also its prefix. β can be computed in $\Theta(n)$ time by a famous algorithm [1], and has the attractive property that if $\boldsymbol{x}[1..i]$ has a nonempty border $\boldsymbol{x}\big[1..\beta[i]\big]$, then $\boldsymbol{x}[1..i]$ also has a border of length $\beta\big[\beta[i]\big]$. Thus, for every $i \in 1..n$, the lengths of *all* the borders of $\boldsymbol{x}[1..i]$ are given by $\beta[i], \beta^2[i], \ldots, \beta^{k_i}[i] = 0$, for some integer $k_i \geq 1$.

An array $\pi[1..n]$ is the *prefix table* of $\boldsymbol{x}$ if for every $i \in 1..n$, $\boldsymbol{x}[i..i{+}\pi[i]{-}1]$ is the longest substring of $\boldsymbol{x}$ beginning at i that matches a prefix of $\boldsymbol{x}$. Though the prefix table is perhaps less commonly used than the border array, nevertheless its construction has been described as "the fundamental preprocessing" of a string [9]. It seems to have first been used some 30 years ago in the Main & Lorentz all-repetitions algorithm [11], where a $\Theta(n)$-time construction algorithm was described. The same algorithm is given in [6, Section 1.6], but then a modified

⋆ The work of the first and third authors was supported in part by the Natural Sciences & Engineering Research Council of Canada. The second author was supported by a Marie Curie Intra-European Fellowship for Career Development.

[1] The prefix table is also known as the prefix array. Here we use the former term to avoid possible confusion with the suffix array, a nonanalogous data structure.

© Springer-Verlag Berlin Heidelberg 2013

construction algorithm was proposed in [10, Section 8.3.1], later also in [7, Section 1.6]. Two other distinct algorithms on a "compressed" prefix table are described in [13]. Among its other virtues, the prefix table gives rise to an easy and efficient pattern-matching algorithm: given pattern $\boldsymbol{u}$ and text $\boldsymbol{v}$, form $\boldsymbol{x} = \boldsymbol{uv}$ and compute $\pi_{\boldsymbol{x}}$; then for $i \in |\boldsymbol{u}|{+}1..|\boldsymbol{x}|$, $\boldsymbol{x}[i..i{+}|\boldsymbol{u}|{-}1] = \boldsymbol{v}[i{-}|\boldsymbol{u}|..i{-}1]$ is an occurrence of $\boldsymbol{u}$ in $\boldsymbol{v}$ if and only if $\pi_{\boldsymbol{x}}[i] \geq |\boldsymbol{u}|$. The prefix table is also closely related to the *witness table* $\omega[1..n]$, which was conceived for parallel pattern matching [14] and later used for the two-dimensional versions of several problems: pattern matching [8], scaled dictionary matching [2], and compressed pattern matching [3]. For a string $\boldsymbol{x}[1..n]$, $\omega[i] = 0$ if $\boldsymbol{x}$ has period $i-1$; otherwise, $\omega[i] = k$ such that $\boldsymbol{x}[k]$ is a witness that $\boldsymbol{x}$ does not have period $i-1$ (i.e. $\boldsymbol{x}[k] \neq \boldsymbol{x}[k-i+1]$). Given π, the linear-time construction of an ω is direct: $\omega[i] = i + \pi[i]$ if $i + \pi[i] \leq n$; otherwise, $\omega[i] = 0$. A given ω does not necessarily correspond to a unique π unless ω contains only leftmost witnesses, in which case π and ω are equivalent.

Conversion of π into border array β and *vice versa* was also discussed in [7], but to our knowledge no linear-time conversion algorithms have been published, an oversight we remedy below. It is tempting to conclude therefore that the π and β data structures are in some sense equivalent, and in fact this is true for regular strings as defined above: both provide the same information with the same asymptotic complexity, both of time and space. However, as shown in [13], for *indeterminate strings*, defined not on elements but rather on *nonempty subsets* of Σ, the prefix table retains its utility, while the border array no longer exists in the form defined above. Similarly, a prefix table adapts easily to substrings compared under a given Hamming distance [4], while again the border array cannot be used. It is shown further in [5] that corresponding to every possible prefix table (such that $\pi[1] = n$ and $0 \leq \pi[i] \leq n{-}i{+}1$ for every $i \in 2..n$), there exists an indeterminate string. Since for every indeterminate string there also exists a prefix table, there is a many-to-many correspondence between prefix tables and indeterminate strings. As a data structure, the prefix table is more robust, thus worthy of closer study. Here is a simple example of β and π:

$$\begin{array}{rl} & \scriptstyle 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8 \\ \boldsymbol{x} = & a\ b\ a\ a\ b\ a\ b\ a \\ \beta = & 0\ 0\ 1\ 1\ 2\ 3\ 2\ 3 \\ \pi = & 8\ 0\ 1\ 3\ 0\ 3\ 0\ 1 \end{array}$$

Section 2 of this paper describes and compares prefix table construction algorithms, while Section 3 presents new algorithms for conversions between π and β. Section 4 summarizes test results for these algorithms on several classes of strings; Section 5 briefly discusses open problems.

2 Construction of the Prefix Table

When discussing prefix table algorithms, it is helpful to use the term *π-range of* ℓ to denote the interval $\ell..\ell + \pi[\ell] - 1$. We also use $\Pi[\ell]$ to denote the substring of $\boldsymbol{x}$ beginning at ℓ whose length is $\pi[\ell]$.

Algorithm ML

We begin with a brief review of the original prefix table construction algorithm, which we call ML, based on the write-up given in [12, pp. 340–347]. Algorithm ML computes $\pi[i]$ in order of increasing i without backtracking. Thus at each position i two cases arise: in Case 1, i lies outside all previous ranges $2..i-1$ already computed, and so $\pi[i]$ can be determined by a simple position-by-position scan; in Case 2, i falls within the π-range of at least one $\ell \in 2..i-1$, and so $\pi[i-\ell+1]$ provides useful information about $\boldsymbol{\pi}[i]$. In Case 2 it suffices to consider only the position $\ell < i$ and the corresponding substring $\Pi[\ell]$ for which $\ell+\pi[\ell]$ is a maximum. Then in Case 2, for such an ℓ, setting $\boldsymbol{w} = \boldsymbol{x}[i..\ell+\pi[\ell]-1]$, two subcases arise: (a) $\pi[i-\ell+1] < |\boldsymbol{w}|$, (b) $\pi[i-\ell+1] \geq |\boldsymbol{w}|$. The behaviour of Algorithm ML can then be outlined as follows:

1. $i \notin \ell..\ell+\pi[\ell]-1$
   - Determine $\pi[i]$ by letter comparisons starting at position i.
   - Set $\ell := i$.
2. $i \in \ell..\ell+\pi[\ell]-1$ and:
   (a) $\pi[i-\ell+1] < |\boldsymbol{w}|$
      - Set $\pi[i] = \pi[i-\ell+1]$.
      - (No letter comparisons needed. No change to ℓ needed.)

   (b) $\pi[i-\ell+1] \geq |\boldsymbol{w}|$
      - $\pi[i] \geq |\boldsymbol{w}|$, so determine $\pi[i]$ by letter comparisons starting at position $i+|\boldsymbol{w}|$.
      - Set $\ell := i$.

The pseudocode for the $\Theta(n)$-time algorithm ML is given in Figure 1.

Algorithm RML — Refined ML

As noted in the introduction, [10] proposes a refinement of Algorithm ML (also described in [7][2]), in which Case 2(b) is modified, now split into two subcases: (b′) $\pi[i-\ell+1] > |\boldsymbol{w}|$, (c′) $\pi[i-\ell+1] = |\boldsymbol{w}|$. For (b′), there is no change from ML; for (c′), $\boldsymbol{x}\big[\ell+\pi[\ell]\big]$ cannot be equal to $\boldsymbol{x}\big[\pi[\ell]+1\big]$, since if it were, $\Pi[\ell]$ would be longer. Hence $\pi[i] = |\boldsymbol{w}|$ and no letter comparisons are needed. The result is code for Algorithm RML that differs in just two lines (10 & 11) from ML, shown in Figure 2.

Algorithm PL1

Because $\pi[i] = 0$ for all i such that $\boldsymbol{x}[i] \neq \boldsymbol{x}[1]$, π is likely to be sparse. Even for strings over a binary alphabet, the expected number of nonzero elements is only half of the elements in π. A compressed representation of π consists of two arrays, $pos[1..m]$ and $len[1..m]$, where m is the number of nonzero elements in π (excluding $\pi[1]$). For all $j \in 1..m$, $pos[j] = i$ and $len[j] = \pi[i]$, where i is the position of the j^{th} nonzero element in π.

[2] The earlier, French version [6] presented unrefined ML.

```
ML($\boldsymbol{x}$)
 1  $n := |\boldsymbol{x}|$
 2  $\boldsymbol{x}[n+1] := \$$
 3  $\pi[1] := n$
 4  $\pi[2] := 0$
 5  while $\boldsymbol{x}[\pi[2]+1] == \boldsymbol{x}[\pi[2]+2]$
 6      $\pi[2] := \pi[2]+1$
 7  $\ell := 2$
 8  for $i := 3$ to $n$
 9      $w := \ell + \pi[\ell] - i$              // $w$ can be negative
10      if $\pi[i-\ell+1] < w$                  // Case 2(a)
11          $\pi[i] := \pi[i-\ell+1]$
12      else                                     // Cases 1 and 2(b)
13          $\pi[i] := \max(0, w)$
14          while $\boldsymbol{x}[\pi[i]+1] == \boldsymbol{x}[\pi[i]+i]$
15              $\pi[i] := \pi[i]+1$
16          $\ell := i$
17  return $\pi$
```

Fig. 1. Algorithm ML pseudocode

```
RML($\boldsymbol{x}$)

    ⋮ Algorithm MLfigure.caption.2
10  if $\pi[i-\ell+1] \neq w$ and $w > 0$        // changed from: $\pi[i-\ell+1] \leq w$
11      $\pi[i] := \min(\pi[i-\ell+1], w)$      // changed from: $\pi[i] := \pi[i-\ell+1]$
    ⋮
```

Fig. 2. Difference between RML and ML

Algorithm PL1 constructs π in *pos*/*len* form in $\Theta(n)$ time and $\Theta(m)$ space beyond that required for $\boldsymbol{x}$ [13]. The pseudocode is given in Figures 3 and 4. To make the similarities with Algorithm ML clear, we use $\bar{\pi}_i$ as the name of a variable that stores the value $\pi[i]$. Unlike ML, the values of $len[j]$ are not determined strictly left to right.

First, PL1 initializes *pos* and *len* by scanning over $\boldsymbol{x}$, locating all m positions i such that $\boldsymbol{x}[i] = \boldsymbol{x}[1]$. When the j^{th} such i is found, $pos[j]$ is assigned i and $len[j]$ is initialized to 1, the minimum possible length of the matching prefix.

PL1 uses an auxiliary function Copy which takes as arguments *pos*/*len* and integers j and r_{max} such that $pos[j]..r_{max}$ is a π-range. For every $i \in pos[j]..r_{max}$, Copy applies the case analysis of Algorithm ML. Only cases 2(a) and 2(b) are possible. In Case 2(b), $\pi[i] \geq w$ and letter comparisons are needed to determine $\pi[i]$. Copy updates the value of $len[j]$ to w, but it does not perform the letter comparisons.

In the variable r_{max}, PL1 maintains the position of the rightmost endpoint of any previously-computed π-range. For each $j \in 1..m$, the right endpoint r

```
PL1($\boldsymbol{x}$)
$n := |\boldsymbol{x}|$
$\boldsymbol{x}[n+1] := \$$
// Initialize $pos/len$ for $m$ positions $\boldsymbol{x}[i] = \boldsymbol{x}[1]$, $i \in 2..n$.
$m := 0$
$\lambda := \boldsymbol{x}[1]$
for $i := 2$ to $n$
    if $\boldsymbol{x}[i] == \lambda$
        $m := m+1$; $pos[m] := i$; $len[m] := 1$
$pos[m+1] := n+1$        // avoid having to do end-of-array tests
$len[m+1] := 0$
// For each $j \in 1..m$, determine longest match with a prefix of $\boldsymbol{x}$.
$r_{max} := 1$
$j := 1$
while $j \le m$
    $i := pos[j]$; $\bar{\pi}_i := len[j]$
    $r := i + \bar{\pi}_i - 1$
    if $r \ge r_{max}$
        while $\boldsymbol{x}[1+\bar{\pi}_i] == \boldsymbol{x}[i+\bar{\pi}_i]$
            $\bar{\pi}_i := \bar{\pi}_i + 1$
        $len[j] := \bar{\pi}_i$
        $r := i + \bar{\pi}_i - 1$
        if $r > r_{max}$
            $r_{max} := r$
            $pos, len :=$ COPY$(j+1, r_{max})$
    $j := j+1$
return $pos$, $len$, $m$
```

Fig. 3. Algorithm PL1 pseudocode

```
COPY($j, r_{max}, pos, len$)
$j' := 1$; $i := pos[j]$
while $i < r_{max}$
    $\bar{\pi}_{i-\ell+1} := len[j']$
    $w := r_{max} - i + 1$
    $\bar{\pi}_i := \min(\bar{\pi}_{i-\ell+1}, w)$
    $len[j] := \bar{\pi}_i$
    $j := j+1$; $j' := j'+1$; $i := pos[j]$
return $pos$, $len$
```

Fig. 4. Pseudocode for the COPY function used in Algorithm PL1

of the π-range at $pos[j]$ is calculated based on the current value of $len[j]$. If $r < r_{max}$, then i lies within a previously-computed π-range ending at r_{max}, and $len[j]$ has already been precisely determined according to Case 2(a). If $r \ge r_{max}$, then either Case 1 or Case 2(b) applies, and letter comparisons are needed to determine $len[j]$. After completing the letter comparisons, r is updated, then compared to r_{max}. If $r > r_{max}$, then i is the left end of a new rightmost π-range, and COPY is called, updating $len[j]$ for j such that $pos[j] \in i+1..r_{max}$.

Algorithm PL2

Algorithm PL2 also constructs π in *pos*/*len* form in $\Theta(n)$ time and $\Theta(m)$ additional space [13]. See Figures 5 and 6 for pseudocode. The main differences between PL1 and PL2 are that PL2 does not perform an initial scan of $\boldsymbol{x}$ to initialize *pos*/*len*, and that PL2's version of COPY terminates when it encounters an instance of Case 2(b). In the pseudocode below, the variable $p_{i-\ell+1}$ stores the value $i - \ell + 1$.

```
PL2($\boldsymbol{x}$)
 1  $n := |\boldsymbol{x}|;\ m := 0$
 2  $\ell := 1;\ \bar{\pi}_l := 0$
 3  while $\ell < n$
 4      if $\bar{\pi}_l == 0$
 5          $\ell := \ell + 1$
 6      while $s[1 + \bar{\pi}_\ell] == s[\ell + \bar{\pi}_\ell]$
 7          $\bar{\pi}_\ell := \bar{\pi}_\ell + 1$
 8      if $\bar{\pi}_\ell \neq 0$
 9          $m := m + 1;\ pos[m] := \ell;\ len[m] := \bar{\pi}_\ell$
10          $\ell,\ \bar{\pi}_\ell,\ pos,\ len,\ m :=$ COPY$(\ell, \bar{\pi}_\ell, pos, len, m)$
11  return $pos,\ len,\ m$
```

Fig. 5. Algorithm PL2 pseudocode

```
COPY$(\ell, \bar{\pi}_\ell, pos, len, m)$
 1  $j := 1;\ p_{i-\ell+1} := pos[j];\ i := p_{i-\ell+1} + \ell - 1$
 2  $i := p_{i-\ell+1} + \ell - 1$
 3  while $p_{i-\ell+1} \leq \bar{\pi}_\ell$
 4      $w := \bar{\pi}_\ell - p_{i-\ell+1} + 1$
 5      if $\bar{\pi}_{i-\ell+1} < w$
 6          $m := m + 1;\ pos[m] := i;\ len[m] := \bar{\pi}_{i-\ell+1}$
 7      elseif $\ell + \bar{\pi}_\ell \geq n$
 8          $m := m + 1;\ pos[m] := i;\ len[m] := w$
 9      else
10          $\bar{\pi}_\ell := -1$
11          $j := j - 1$
12      $j := j + 1;\ p_{i-\ell+1} := pos[j];\ \bar{\pi}_{i-\ell+1} := len[j]$
13      $i := p_{i-\ell+1} + \ell - 1$
14  if $\bar{\pi}_\ell == -1$
15      $\ell := i;\ \bar{\pi}_\ell := w$
16  else
17      $\ell := \ell + \bar{\pi}_\ell - 1;\ \bar{\pi}_\ell := 0$
18  return $\ell,\ \bar{\pi}_\ell,\ pos,\ len,\ m$
```

Fig. 6. Pseudocode for the COPY function used in Algorithm PL2

We have also designed and implemented compressed versions of algorithms ML and RML, but space restrictions prevent their description; in general, they are not competitive with the fastest π construction algorithms. The situation is different however with a "brute force" algorithm BF, which simply constructs π by computing $\pi[i]$ using letter comparisons independently for each i (Section 4).

3 Conversion between Prefix Table and Border Array

Prefix Table to Border Array

[7, Section 1.6] presents the following property that allows the conversion of a prefix table π into a border array β. For a given position j, for all ℓ such that j is a member of the π-range at ℓ, $\boldsymbol{x}[\ell..j] = \boldsymbol{x}[1..j-\ell+1]$ is a border of $\boldsymbol{x}[1..j]$. The longest border of $\boldsymbol{x}[1..j]$ corresponds to the minimum ℓ such that $j \in \ell..\ell{+}\pi[\ell]{-}1$. Formally,

$$\beta[j] = \begin{cases} 0 & \text{if } L = \emptyset \\ j - \min(L) + 1 & \text{otherwise} \end{cases} \tag{1}$$

where $L = \{\ell \mid \ell \leq j \leq \ell + \pi[\ell] - 1\}$. However, [7] does not specify the algorithm itself which we provide in Figure 7.

```
PREFIX-TO-BORDER($\pi$)
1  $n = |\pi|$
2  $\beta := 0^n$
3  $end := 2$
4  for $\ell := 2$ to $n$
5      if $end \leq \ell + \pi[\ell] - 1$
6          for $j := \max(end, \ell)$ to $\ell + \pi[\ell] - 1$
7              $\beta[j] := j - \ell + 1$
8          $end := \ell + \pi[\ell]$
9  return $\beta$
```

Fig. 7. PREFIX-TO-BORDER pseudocode

β is first initialized with zeros. The variable *end* stores the leftmost position in β that has not yet been determined; initially, $end = 2$ since by definition $\beta[1] = 0$. For each position ℓ from 2 to n, *end* is compared with $\ell + \pi[\ell] - 1$. If $end > \ell + \pi[\ell] - 1$, then the π-range at ℓ cannot help determine $\beta[end]$ (or any subsequent element of β). However, if $end \leq \ell + \pi[\ell] - 1$, then either $end < \ell$ or $end \in \ell..\ell{+}\pi[\ell]{-}1$. If $end < \ell$, then for all $j \in end..\ell - 1$, j is not part of any π-range, so $\beta[j] = 0$. If $end \in \ell..\ell + \pi[\ell] - 1$, then ℓ begins the leftmost π-range that contains positions $end..\ell + \pi[\ell] - 1$.

Note that since values are assigned to $\beta[j]$ in strictly increasing order of j, the time requirement of PREFIX-TO-BORDER is $\Theta(n)$.

Border Array to Prefix Table

More complex is the border to prefix conversion. We first summarize the main properties of π and β that will be used by our algorithms. Similar to π-ranges, we define a *β-range* to be an interval $j-\beta[j]+1..j$ for some position j.

Lemma 1.

(i) $\beta[j+1] \leq \beta[j]+1$,
(ii) *if* $\beta[j+1] \leq \beta[j]$, *then* $\pi[j-\beta[j]+1] = \beta[j]$,
(iii) *if* $\pi[\ell] > 1$, *then for* $i \in \ell+1..\ell+\pi[\ell]-1$
  (a) $\pi[i-\ell+1] < \pi[\ell]-(i-\ell) \implies \pi[i] = \pi[i-\ell+1]$,
  (b) $\pi[i-\ell+1] \geq \pi[\ell]-(i-\ell) \implies \pi[i] \geq \pi[\ell]-(i-\ell)$.

Proof. (i) simply states that if $\boldsymbol{x}[1..i]$ has a border of length k, then $\boldsymbol{x}[1..i-1]$ has a border of length $k-1$. (ii) holds because if the longest border of $\boldsymbol{x}[1,j+1]$ is no longer than the longest border of $\boldsymbol{x}[1,j]$, then $\boldsymbol{x}[j+1] \neq \boldsymbol{x}[\beta[j]+1]$. (iii) restates Case 2 of Algorithm ML (see Section 2). $\square$

Properties (i)-(iii) are used to compute the prefix table π from a border array β as follows. Consider a *right-maximal* β-range, i.e. $r-\beta[r]+1..r$ where $\beta[r+1] \leq \beta[r]$. For each such r, we set $\pi[r-\beta[r]+1] = \beta[r]$ according to property (ii).

Observe now that we only have to compute values of π inside right-maximal β-ranges. This is because if $\beta[j] = 0$, i.e. j is outside all β-ranges, then $\pi[j] = 0$ (except for the special case $j = 1$).

Consider a right-maximal β-range $\ell..r = r-\beta[r]+1..r$. To compute values $\pi[i]$ for $i \in \ell+1..r$, we use Lemma 1(iii). If $\pi[i-\ell+1] < \pi[\ell]-(i-\ell)$, then we set $\pi[i] = \pi[i-\ell+1]$ (Case (iii)(a)). If $\pi[i] \geq \pi[\ell]-(i-\ell)$ (Case (iii)(b)), we cannot immediately determine $\pi[i]$ (unless we reached the end of the string) and need to look at the next right-maximal β-range, which may (or may not) overlap the current one. This leads to a complication: it might be that we will have to look at a "cascade" of several (a non-constant number) of overlapping β-ranges. We need to take more care to avoid this look-ahead.

The key to deal with the case of Lemma 1(iii)(b) is to maintain the rightmost right-maximal β-range to which position i belongs. Formally, we maintain the following invariant: if we are about to compute $\pi[i]$ for a position i belonging to a right-maximal β-range, then $i \leq (r+1)-\beta[r+1]$. That is, the start of the next run of β-ranges is to the right of i. We show that in this case, $\pi[i]$ can be computed immediately.

The following lemma is straightforward.

Lemma 2. *A π-range cannot properly include a β-range other than as a prefix. That is, if $\pi[i] > 0$, then for every $j \in i..i+\pi[i]-1$, $\beta[j] \geq j-i+1$.*

Using Lemma 2, we can compute $\pi[i]$ immediately.

Lemma 3. *Assume a position i belongs to a right-maximal β-range $r-\beta[r]+1..r$, and assume that $i \leq (r+1)-\beta[r+1]$. If $\pi[i-r+\beta[r]] \geq r-i+1$, then $\pi[i] = r-i+1$.*

Proof. Lemma 3 refines condition (iii)(b) of Lemma 1, which states that if $\pi[i - r + \beta[r]] \geq r - i + 1$, then $\pi[i] \geq r - i + 1$. Lemma 3 states that if the condition $i \leq (r + 1) - \beta[r + 1]$ is verified in addition, the last inequality is actually an equality.

If $\beta[r + 1] = 0$, then $\pi[i] > r - i + 1$ is impossible as this would imply $\beta[r + 1] \geq r - i + 1 > 0$. If $\beta[r + 1] > 0$, then the π-range $i..i + \pi[i] - 1$ properly includes the β-range $(r + 1) - \beta[r + 1] + 1..(r + 1)$ since, by our assumption, $i \leq (r + 1) - \beta[r + 1]$. By Lemma 2, this is impossible. □

The pseudocode in Figure 8 specifies the algorithm. The running time of the algorithm is $\Theta(n)$, as the internal **while** loop increases r and therefore cannot iterate more than n times altogether.

```
Border-to-Prefix-A($\beta$)
$n := |\beta|$
$\pi[1] := n$
$\ell := 1$
$r := 1$
for $i := 2$ to $n$
    if $\beta[i] == 0$
        $\pi[i] := 0$
        $r := r + 1$
    elseif $r + 1 \leq n$ and $i == (r + 1) - \beta[r + 1] + 1$
        $\ell := i$                 // entered the next run of $\beta$-ranges
        $r := r + 1$
        while $r + 1 \leq n$ and $\ell == (r + 1) - \beta[r + 1] + 1$
            $r := r + 1$        // looking for the next right-maximal $\beta$-range
        $\pi[i] := r - i + 1$       // Lemma 1(ii)
    else
        $\pi[i] := \min(\pi[i - \ell + 1], r - i + 1)$  // Lemma 1(iii)(a), Lemma 3
return $\pi$
```

Fig. 8. Border-to-Prefix-A pseudocode

Our second algorithm for transforming β into π deals with the case of Lemma 1(iii)(b) in a conceptually simpler manner. After computing the right-maximal β-range, it assigns the π value to all positions in this range to be the minimum between $\pi[i - \ell + 1]$ and $\pi[\ell] - (i - \ell)$ (see Lemma 1(iii)). At this point, some of those values may be incorrect but they are later revisited when the next right-maximal β-range is processed, until eventually the correct values are assigned.

This algorithm runs in worst-case $O(n^2)$ time, which can be seen by considering strings of the form $a^k ba^{2k}$. Positions 1 to $k+1$ in π will each be set once. The next k positions will be set $1, 2, \ldots, k$ times, respectively. The final k positions will be set $k, k - 1, \ldots, 1$ times, respectively. The total number of assignment operations will be $(k+1) + k(k+1) = \frac{1}{9}n^2 + \frac{4}{9}n + \frac{4}{9} = O(n^2)$. Despite quadratic complexity, we find this algorithm is faster than our linear algorithm in practice

```
BORDER-TO-PREFIX-B($\beta$)
$n := |\beta|$
$\pi[1] := n$
$\ell := 2$
while $\ell \le n$
    if $\beta[\ell] == 0$
        $\pi[\ell] := 0$
        $\ell := \ell + 1$
    else
        $r := \ell$
        while $r + 1 \le n$ and $\beta[r+1] == \beta[r] + 1$
            $r := r + 1$
        $\ell := r - \beta[r] + 1$
        $\pi[\ell] := \beta[r]$
        for $i := \ell + 1$ to $r$
            $\pi[i] := \min(\pi[i - \ell + 1], r - i + 1)$
        $\ell := r + 1$
return $\pi$
```

Fig. 9. Algorithm BORDER-TO-PREFIX-B pseudocode

(see Section 4). Code profiling shows that this is because the linear algorithm spends more time executing its conditional statements.

4 Test Results

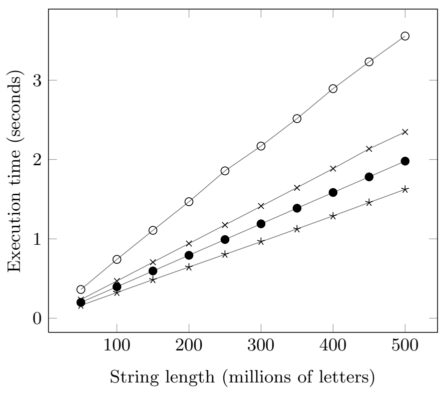
We implemented the algorithms in C++ and compiled the code using GCC 4.7.2 with the -O3 optimization option. We ran the tests on a MacBook Pro with a 2.3 GHz Intel Core i5 processor and 8 GB of memory running OS X 10.7.5. To measure execution time, we used the `clock()` function from `time.h` to obtain the CPU time taken to run ten identical trials, excluding input/output and memory allocation. The reported time is the average time for the ten trials.

Our test strings were prefixes of the Fibonacci string, a random binary string, and an English string. The English string was the text *english* from the Pizza & Chili corpus[3] (a concatenation of texts from Project Gutenberg). We generated the random binary string, choosing each letter independently with $p = 0.5$. From each of these three strings, we extracted ten prefixes ranging in length from 50 million to 500 million letters.

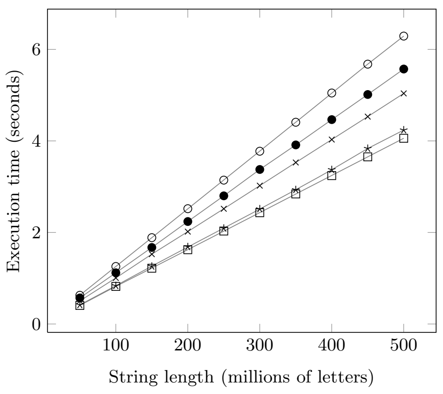
π Construction

Results for Fibonacci, random binary, and English strings are shown in Figures 10a, 10b and 10c, respectively. PL2 was the fastest of the four algorithms on Fibonacci and random binary strings (about 20% faster than ML), but PL1

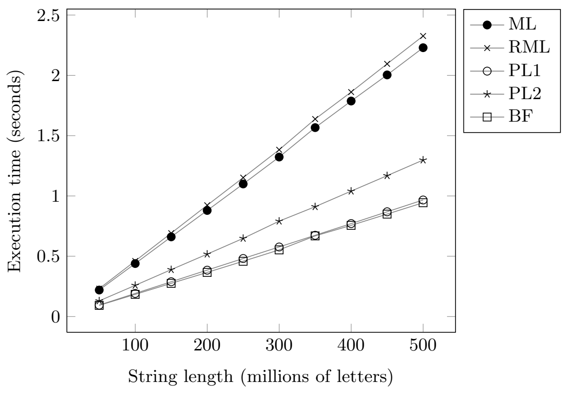
[3] `http://pizzachili.dcc.uchile.cl`



(a) Fibonacci strings



(b) Random binary strings



(c) English strings

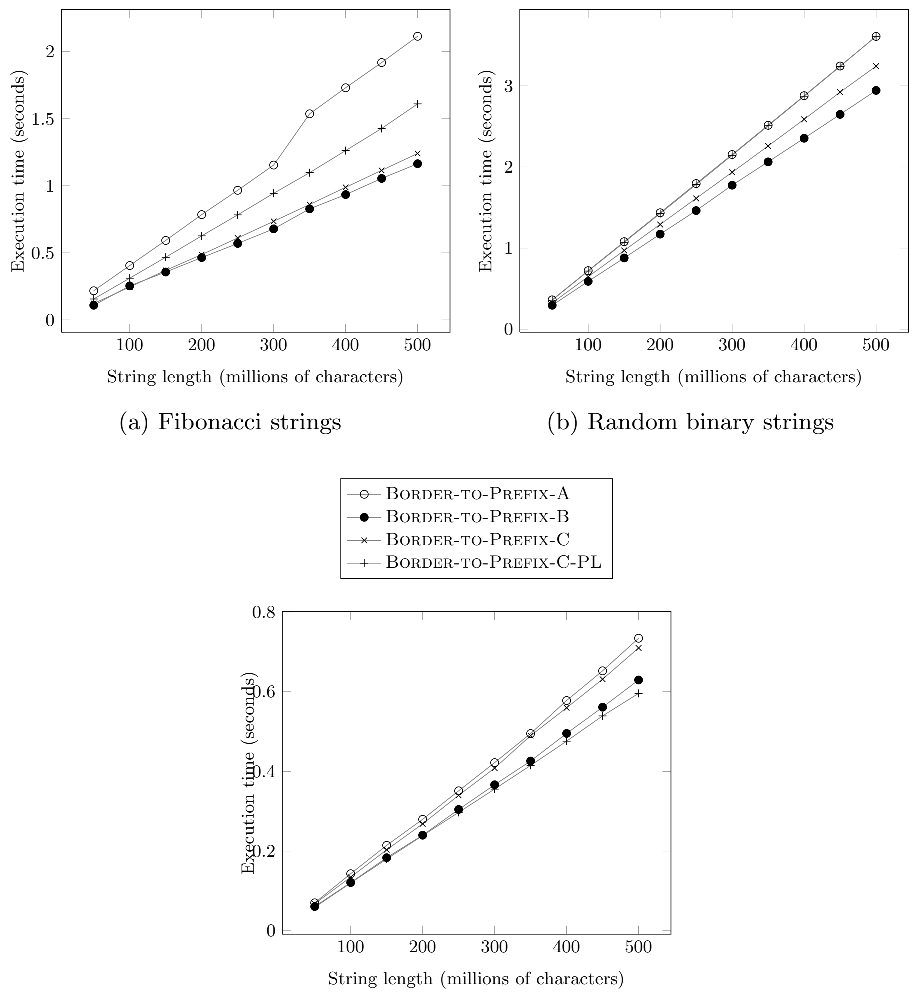
Fig. 10. π construction time

was about 30% faster on English strings. Though PL1 was the fastest of the four algorithms on English strings, it was the slowest on Fibonacci and random binary strings. ML and RML performed comparably, though ML was faster on Fibonacci and English strings, and RML was faster on random binary strings.

On Fibonacci strings, even the slowest algorithm was eight to ten times faster than BF. However, on English and random binary strings there is less periodicity to take advantage of, and BF was actually slightly faster than the fastest of the other algorithms.

$\boldsymbol{\pi \leftrightarrow \beta}$

We implemented a PL version of $\pi \rightarrow \beta$, using a strategy similar to that of PL1, which in some cases (English strings) seems to slightly reduce execution time. (We omit these plots for space.) For $\beta \rightarrow \pi$ we found that, despite the



(a) Fibonacci strings

(b) Random binary strings

(c) English strings

Fig. 11. $\beta \to \pi$ conversion time

backtracking, Border-to-Prefix-B was consistently faster than Border-to-Prefix-A on all strings tested (Figures 11a–11c); another approach, together with its PL variant, provided no clear advantage.

5 Future Directions

In this paper we have adopted a unified perspective on algorithms related to prefix tables, both those for prefix table construction and for conversion to border arrays. Some of these algorithms are available in the literature, others have been designed by us. All of the algorithms are linear in their time requirement, and so our purpose has been to provide guidelines for determining comparative efficiency in terms of the nature of the input files. The prefix table is a robust, perhaps somewhat neglected, data structure; we believe that study of its properties in the context of approximate string matching will yield useful results.

References

1. Aho, V.A., Hopcroft, J.E., Ullman, J.D.: The Design & Analysis of Computer Algorithms, p. 470. Addison-Wesley (1974)
2. Amir, A., Calinescu, G.: Alphabet-independent and scaled dictionary matching. J. Algorithms, Elsevier 34–62 (2000)
3. Amir, A., Landau, G.M., Sokol, D.: Inplace run-length 2d compressed search. In: Proc. 11th ACM-SIAM Symp. Discrete Algs., pp. 817–818. SIAM (2000)
4. Barton, C., Iliopoulos, C.S., Pissis, S., Smyth, W.F.: Prefix tables and border arrays with k-mismatches & applications (submitted for publication, 2013)
5. Christodoulakis, M., Ryan, P.J., Smyth, W.F., Wang, S.: Indeterminate strings, prefix arrays & undirected graphs (submitted for publication, 2013)
6. Crochemore, M., Hancart, C., Lecroq, T.: Algorithmique du Texte, p. 347, Vuibert (2001)
7. Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on Strings, p. 383. Cambridge University Press (2007)
8. Galil, Z., Park, K.: Truly alphabet-independent two-dimensional pattern matching. In: Proc. 33rd IEEE Symp. Found. Computer Science, pp. 247–256. IEEE (1992)
9. Gusfield, D.: Algorithms on Strings, Trees & Sequences, p. 534. Cambridge University Press (1997)
10. Lothaire, M.: Applied Combinatorics on Words, p. 610. Cambridge University Press (2005)
11. Main, M.G., Lorentz, R.J.: An $O(n \log n)$ algorithm for finding all repetitions in a string. J. Algorithms 5, 422–432 (1984)
12. Smyth, B.: Computing Patterns in Strings, p. 423. Pearson Addison-Wesley (2003)
13. Smyth, W.F., Wang, S.: New perspectives on the prefix array. In: Amir, A., Turpin, A., Moffat, A. (eds.) SPIRE 2008. LNCS, vol. 5280, pp. 133–143. Springer, Heidelberg (2008)
14. Vishkin, U.: Optimal parallel pattern matching in strings. Information and Control, Elsevier, 91–113 (1985)

On the Approximability of Splitting-SAT in 2-CNF Horn Formulas*

Hans-Joachim Böckenhauer and Lucia Keller

Department of Computer Science, ETH Zurich, Switzerland
{hjb,lucia.keller}@inf.ethz.ch

Abstract. Splitting a variable in a Boolean formula means to replace an arbitrary set of its occurrences by a new variable. In the minimum splitting SAT problem, we ask for a minimum-size set of variables to be split in order to make the formula satisfiable. This problem is known to be APX-hard, even for 2-CNF formulas. We consider the case of 2-CNF Horn formulas, i. e., 2-CNF formulas without positive 2-clauses, and prove that this problem is APX-hard as well. We also analyze subcases of 2-CNF Horn formulas, where additional clause types are forbidden. While excluding negative 2-clauses admits a polynomial-time algorithm based on network flows, the splitting problem stays APX-hard for formulas consisting of positive 1-clauses and negative 2-clauses only.

Instead of splitting as many variables as possible to make a formula satisfiable, one can also look at the dual problem of finding the maximum number of variables that can be assigned without violating a clause. We also study the approximability of this maximum assignment problem on 2-CNF Horn formulas. While the polynomially solvable subproblems are the same as for the splitting problem, the maximum assignment problem in general Horn formulas is as hard to approximate as the maximum independent set problem.

1 Introduction

Many problems arising from practical applications can be formulated using Boolean formulas in conjunctive normal form (CNF). Usually, the variables of the formula model some parameters of the problem, and the constraints of the problem are modeled by the clauses of the formula. The goal is to find out a valid parameter setting by computing a satisfiable assignment for the corresponding formula. But often the modeling of the practical situation is very complex, leading to some contradictory constraints in the model, and the corresponding formula turns out to be unsatisfiable. In this case, one often tries to find a maximum set of constraints that can be simultaneously satisfied. This leads to the well-known Max-SAT problem (see, e. g., [1] for an overview of the known results for Max-SAT). Another source of mistakes that might arise when modeling a real-world problem as a Boolean formula is a too coarse-grained choice of parameters, i. e., variables. If two different parameters are erroneously modeled

* This work was partially supported by SNF grant 200021/132510.

© Springer-Verlag Berlin Heidelberg 2013

by the same variable, this might also lead to an unsatisfiable formula. In other words, an unsatisfiable variable might contain one or more variables that should be *split* into two variables in order to make the formula satisfiable. The *minimum splitting SAT problem* formalizes this approach, the input is an (unsatisfiable) CNF formula and the goal is to find a minimum number of variables that have to be split into two to make the resulting formula satisfiable.

The splitting operation has not only been considered on formulas. For example, it arises in the context of vertex splitting in phylogenetic tree construction [9]. Splitting vertices in a graph was also considered for making a graph Hamiltonian [11]. To the best of our knowledge, splitting variables in a Boolean formula was introduced by Steinová [11], who showed that the minimum splitting SAT problem is APX-hard even for formulas in 2-CNF, i. e., when restricted to formulas in which each clause contains at most two literals.

In a 2-CNF formula, we can have the following five types of clauses:

P1: Positive 1-clauses (x) consisting of one positive literal,
N1: negative 1-clauses $(\overline{x})$ consisting of one negative literal,
M2: mixed 2-clauses $(x \vee \overline{y})$ consisting of one positive and one negative literal,
N2: negative 2-clauses $(\overline{x} \vee \overline{y})$ consisting of two negative literals, and
P2: positive 2-clauses $(x \vee y)$ consisting of two positive literals.

A 2-CNF formula without positive 2-clauses is called a 2-CNF *Horn formula.* We will restrict our attention to 2-CNF Horn formulas in the first part of the paper. We analyze which combinations of clause types make the minimum splitting SAT problem hard to approximate. An overview of the results is given in Figure 1. Observe that lower bounds carry over upwards and upper bounds downwards in the lattice of subsets. In particular, we show that the minimum splitting SAT problem remains exactly as hard to approximate as the vertex cover problem, when restricted to the special case of Horn formulas consisting of clauses of type P1 and N2 only. On the other hand, even when allowing additional clauses of type N1, the problem can be approximated exactly as good as the vertex cover problem, it becomes polynomially solvable when restricted to Horn formulas consisting of clauses of type P1, N1, and M2.

Another way to look at the splitting SAT problem is to ask for the maximum number of variables that can be assigned a truth value without evaluating any clause to 0, i. e., for the maximum number of variables that can be left unsplit. This is called the *maximum assignment SAT problem.* Obviously, the optimal solutions for minimum splitting SAT and maximum assignment SAT coincide, but we show that the approximability of the two problems essentially differs. The maximum assignment SAT problem on 2-CNF Horn formulas with clauses of type P1 and N2 turns out to be as hard to approximate as the maximum independent set problem, and, on arbitrary 2-CNF Horn formulas, it can be approximated as good as the maximum independent set problem. An overview of the results on the maximum assignment SAT problem is shown in Figure 2.

We complement our results with an approximation algorithm for the maximum assignment SAT problem on E2-CNF formulas, i. e., formulas containing

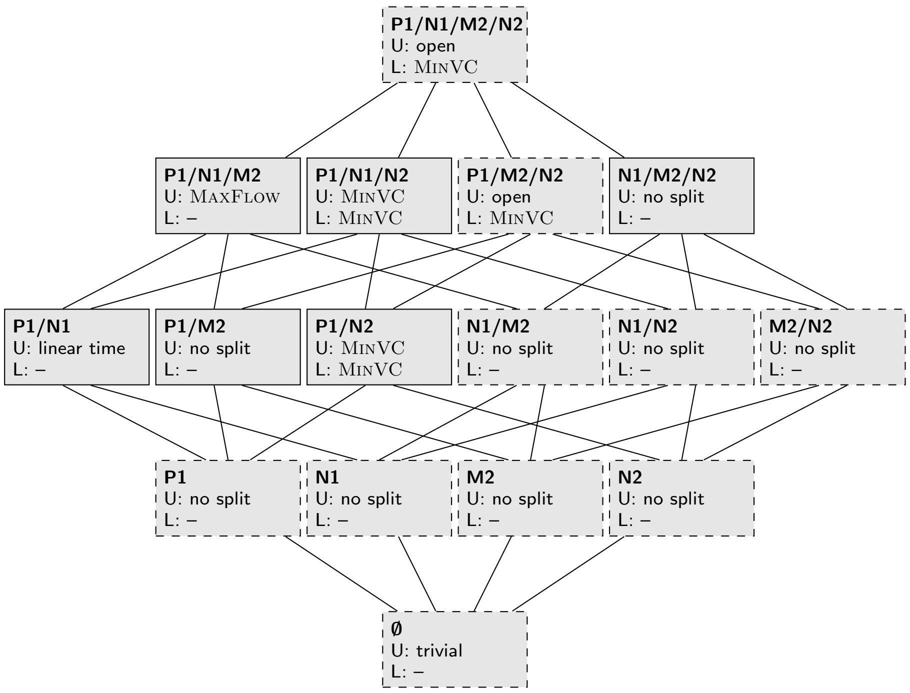


Fig. 1. Upper and lower bounds on the approximability of the minimum splitting SAT problem on 2-CNF Horn formulas for each set of allowed clause types

only clauses of the types M2, P2, and N2. The approximation hardness of this problem was shown by Steinová [11].

The paper is organized as follows: In Section 2, we fix our notation. Sections 3 and 4 are devoted to the analysis of minimum splitting SAT and maximum assignment SAT in Horn formulas, respectively. In Section 5, we discuss the case of E2-CNF formulas, and we conclude the paper in Section 6.

2 Basic Definitions

We start with formally defining the minimum splitting SAT problem and the maximum assignment SAT problem. We follow the definitions from [11].

Definition 1. *Let Φ be a Boolean formula over the variable set X and let $y, z \notin X$ be two new variables. We say that a variable $x \in X$ is* split *if each occurrence of x in Φ is replaced by either y or z and each occurrence of $\overline{x}$ is replaced by either $\overline{y}$ or $\overline{z}$. This operation is called a* splitting *of x. We call a set $X' \subseteq X$ such that splittting all variables from X' yields a satisfiable formula a* feasible splitting set *(or* splitting set *for short).*

Note that, when splitting a variable x into the two new variables y and z, we can replace all occurrences of the literal x by y and all occurrences of the literal

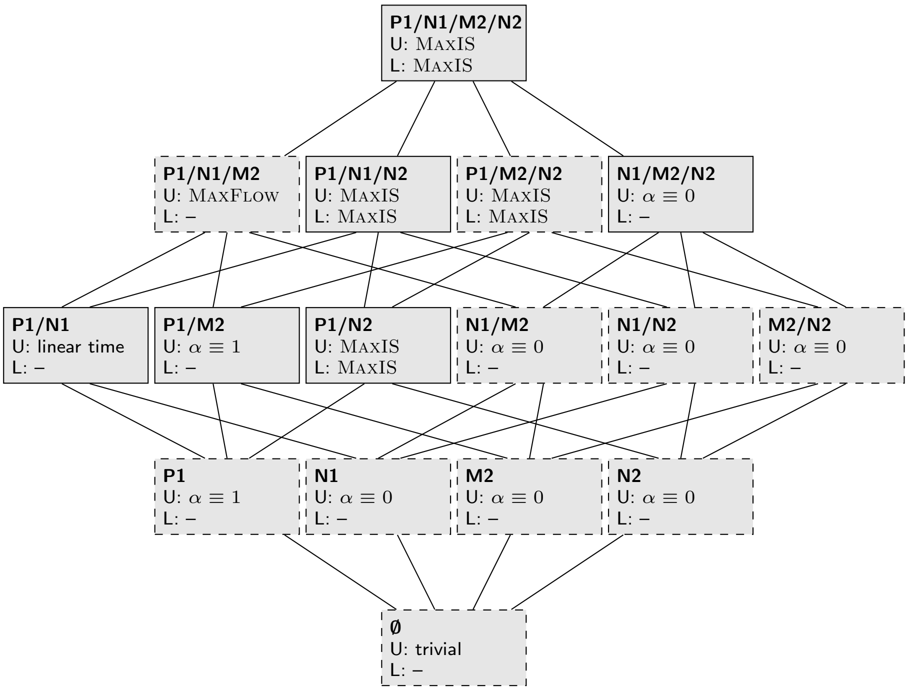


Fig. 2. Upper and lower bounds on the approximability of the maximum assignment SAT problem on 2-CNF Horn formulas for each set of allowed clause types

$\overline{x}$ by $\overline{z}$. Thus, the resulting formula is satisfiable if and only if the formula resulting from removing all clauses containing the variable x is satisfiable. Hence, we can think of a splitting operation as the removal of the split variable (together with all clauses it appears in) from the formula. Furthermore, note that the result of splitting multiple variables is independent from the order of applying the splitting operations.

Definition 2. *Let Φ be a Boolean formula over the variable set $X = \{x_1, \ldots, x_n\}$. We say that Φ is in* conjunctive normal form (CNF)*, if it is a conjunction of so-called* clauses*, which are disjunctions of* literals*, i. e., variables or negated variables. If the number of literals in a clause is bounded by some constant k, we say that Φ is in k-CNF.*

For a formula in 2-CNF and a set S of clause types from $\{\mathrm{P1}, \mathrm{N1}, \mathrm{M2}, \mathrm{N2}, \mathrm{P2}\}$ as defined above, we say that Φ is in S-2-CNF, if it contains only clauses of types from S. The $\{\mathrm{P1}, \mathrm{N1}, \mathrm{M2}, \mathrm{N2}\}$-2-CNF formulas are called Horn formulas.

Definition 3. *The* minimum splitting SAT problem*,* MinSplit-SAT *for short, is the following minimization problem: Given a Boolean formula in CNF over the variable set $X = \{x_1, \ldots, x_n\}$, find a minimum-size splitting set $X' \subseteq X$.*

If the input is restricted to k-CNF formulas, we call the resulting subproblem MinSplit-k-SAT*. When restricting the input to S-2-CNF formulas, for some*

$S \subseteq \{\text{P1}, \text{N1}, \text{M2}, \text{N2}, \text{P2}\}$*, the resulting subproblem is denoted as* MinSplit-S-2-SAT*. The problem* MinSplit-$\{\text{P1}, \text{N1}, \text{M2}, \text{N2}\}$-2-SAT *is called* MinSplit-Horn-2-SAT *and* MinSplit-$\{\text{P2}, \text{M2}, \text{N2}\}$-2-SAT *is called* MinSplit-E2-SAT.

In the following, we will only deal with MinSplit-2-SAT and its subproblems and we will assume without loss of generality that the input always is an unsatisfiable formula. Since 2-SAT, i.e., checking the satisfiability of a 2-CNF formula, is solvable in polynomial time, this is no severe restriction. This implies that the cost of any feasible solution is always strictly greater than zero, which allows us to consider the approximation ratio of an algorithm for MinSplit-2-SAT as the quotient of the cost of the computed solution and the optimal cost.

Definition 4. *The* maximum assignment SAT problem, MaxAssign-SAT *for short, is the following maximization problem: Given a Boolean formula in CNF over the variable set $X = \{x_1, \ldots, x_n\}$, find a maximum-size subset $X' \subseteq X$ such that there exists a partial assignment $\alpha \colon X' \to \{0, 1\}$ such that no clause is evaluated to* 0 *under this partial assignment.*

For the restrictions to special types of clauses, we use analogous notations as for the respective MinSplit-SAT *variants.*

Observation 1. *Let Φ be a formula in CNF over the set $X = \{x_1, \ldots, x_n\}$ of variables. A set $X' \subseteq X$ is an optimal* MinSplit-SAT *solution for Φ if and only if $X - X'$ is an optimal* MaxAssign-SAT *solution for Φ.*

Proof. Let X' be a feasible solution for MinSplit-SAT on Φ. Then all clauses are either satisfied by splitting one of the variables from X' or can be satisfied by some partial assignment α for the variables from $X - X'$ (otherwise another variable would need to be split). Thus, $X - X'$ is a feasible MaxAssign-SAT solution for Φ since all clauses not satisfied by α contain an unassigned variable from X' and thus are not evaluated to 0.

On the other hand, let $X - X'$ be a feasible MaxAssign-SAT solution for Φ. Then there exists a partial assignment α to the variables from $X - X'$, such that no clause evaluates to 0 under α. Thus, each clause is either satisfied by α or contains an unassigned variable from X'. Thus, splitting all variables from X' makes the formula satisfiable. $\square$

Several of our results are based on reductions from the minimum vertex cover problem and the maximum independent set problem. The *minimum vertex cover problem*, MinVC for short, is the following minimization problem: Given an undirected graph $G = (V, E)$, find a minimum-size vertex cover of G, i.e., a minimum-size subset $C \subseteq V$ such that $e \cap C \neq \emptyset$ for all $e \in E$. The *maximum independent set problem*, MaxIS for short, is the following maximization problem: Given an undirected graph $G = (V, E)$, find a maximum-size independent set of G, i.e., a maximum-size subset $I \subseteq V$ such that $\{x, y\} \notin E$ for all $x, y \in I$ with $x \neq y$.

MinVC is known to be approximable within a factor of $2 - (\log\log |V| / 2 \log |V|)$ [8], but it is APX-hard [10] and not approximable within a factor of $2 - \varepsilon$,

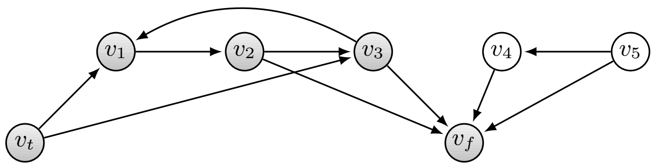


Fig. 3. The graph representation of the $\{\text{P1}, \text{N1}, \text{M2}\}$-2-CNF Horn formula $\Phi = (x_1) \wedge (\overline{x_2}) \wedge (x_3) \wedge (\overline{x_3}) \wedge (\overline{v_4}) \wedge (\overline{v_5}) \wedge (\overline{x_1} \vee x_2) \wedge (\overline{x_2} \vee x_3) \wedge (x_1 \vee \overline{x_3}) \wedge (v_4 \vee \overline{v_5})$

for any constant $\varepsilon > 0$, if the Unique Games Conjecture holds [7]. The MaxIS is approximable within $O(|V|/(\log |V|)^2)$ [2], but not approximable within $|V|^{1-\varepsilon}$, for any $\varepsilon > 0$, unless P=NP [5].

3 Splitting in 2-CNF Horn Formulas

In this section, we deal with the approximability of MinSplit-S-2-SAT, for all possible subsets $S \subseteq \{\text{P1}, \text{N1}, \text{M2}, \text{N2}\}$.

If all allowed clause types contain a positive literal or all contain a negative literal, setting all variables to 1 or 0, respectively, satisfies the formula and no splitting is needed.

Observation 2. *For $S = \{\text{P1}, \text{M2}\}$ or $S = \{\text{N1}, \text{M2}, \text{N2}\}$ or any subset thereof,* MinSplit-S-2-SAT *is solvable in constant time.* □

Similarly, as already observed in [11], for any formula consisting of 1-clauses only, MinSplit-S-2-SAT is easily solvable. This immediately leads to the following observation.

Observation 3. *For $S = \{\text{P1}, \text{N1}\}$,* MinSplit-$S$-2-SAT *is solvable in linear time.* □

Next, we prove that MinSplit-$\{\text{P1}, \text{N1}, \text{M2}\}$-2-SAT can be solved by computing the maximum flow in a given network. For this, we first define a representation of formulas of this type by directed graphs. This graph representation is a special case of the representation in [4].

Definition 5. *Given a $\{\text{P1}, \text{N1}, \text{M2}\}$-2-CNF Horn formula $\Phi = C_1 \wedge C_2 \wedge \ldots \wedge C_m$ over the variable set $X = \{x_1, x_2, \ldots, x_n\}$, $G_\Phi = (V_\Phi, E_\Phi)$ is a digraph with vertex set $V_\Phi = \{v_i \mid x_i \in X\} \cup \{v_t\} \cup \{v_f\}$ and arc set*

$$E_\Phi = \{(v_t, v_j) \mid (x_j) \text{ in } \Phi\} \cup \{(v_j, v_f) \mid (\overline{x_j}) \text{ in } \Phi\} \cup \{(v_i, v_j) \mid (\overline{x_i} \vee x_j) \text{ in } \Phi\}.$$

The arcs from v_t to vertices $v_i \in \{v_1, v_2, \ldots, v_n\}$ represent a satisfying assignment $\alpha(x_i) = 1$ for all clauses of type (x_i). The connecting arcs in the vertex set $\{v_1, v_2, \ldots, v_n\}$ indicate that, for an arc (v_i, v_j), $\alpha(x_i) = 0$ or $\alpha(x_j) = 1$ has to hold in order to satisfy the clauses of type $(\overline{x_i} \vee x_j)$. An arc from a vertex

$v_i \in \{v_1, v_2, \ldots, v_n\}$ to v_f represents a satisfying assignment $\alpha(x_i) = 0$ for the corresponding clause $(\overline{x_i})$. We see in Figure 3 that only the shaded vertices can be reached by a path from v_t. We call such a vertex a *v_t-pebbled* vertex or *pebbled vertex* for short.

This graph construction and the idea of a pebbling were used in a more general way by Dowling and Gallier [4]. They proved the following theorem.

Theorem 1 (Downing and Gallier [4]). *A Horn formula is satisfiable if and only if there is no directed path from v_t to v_f.* □

Furthermore, they made a statement about the assignment in the case of a satisfiable formula.

Theorem 2 (Downing and Gallier [4]). *Let Φ be a satisfiable Horn formula with variable set $X = \{x_1, x_2, \ldots, x_n\}$. The assignment $\alpha(x_i) = 1$ if and only if v_i is pebbled and $\alpha(x_i) = 0$ otherwise, is a satisfying assignment.* □

This means that we get a satisfying assignment if we set all variables corresponding to pebbled vertices to 1 and all other vertices to 0 in the case with no directed path from v_t to v_f.

Corollary 1. *Only pebbled vertices are candidates to split.*

Proof. There is no directed path from v_t to a non-pebbled vertex v_i and therefore no 1-clause (x_i). Furthermore, for all variables $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$ that correspond to vertices $v_{i_1}, v_{i_2}, \ldots, v_{i_k}$ for which a directed path from $v_{i_1}, v_{i_2}, \ldots, v_{i_k}$ to v_i exists, there are no 1-clauses $(x_{i_1}), (x_{i_2}), \ldots, (x_{i_k})$. Therefore, all variables that correspond to non-pebbled vertices can be set to 0 so that all clauses containing those vertices are satisfied by this assignment. □

If we remove all non-pebbled vertices from the graph, it remains connected with a directed path from v_t to v_f. To make the corresponding formula satisfiable, we have to delete some of the remaining vertices in order to disconnect v_t from v_f.

Lemma 1. *A splitting set of size k in a $\{\mathrm{P1}, \mathrm{N1}, \mathrm{M2}\}$-2-CNF Horn formula corresponds to a $v_t - v_f$ vertex cut of size k in the corresponding graph.*

Proof. After removing the split vertices from the formula Φ, the corresponding graph G_Φ has no directed path from v_t to v_f due to Theorem 1. Hence, the removed vertices form a $v_t - v_f$ vertex cut.

Conversely, removing the variables corresponding to a $v_t - v_f$ vertex cut in G_Φ from the formula Φ makes it satisfiable due to Theorem 1. □

The problem of finding a minimum $s - t$ vertex cut in a graph G equals the problem finding a minimum $s - t$ arc cut in a graph G' where we replace every vertex v of G, except the source s and the sink t, by two vertices v_1 and v_2 and an arc (v_1, v_2) in G'. The ingoing arcs of v are connected to v_1 in G' and the outgoing arcs of v are outgoing arcs of v_2 (see Figure 4). Additionally, the

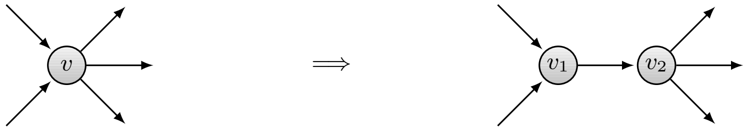


Fig. 4. In the transformation of an instance G for finding a minimum $s-t$ vertex cut into an instance G' for finding a minimum $s-t$ arc cut, every vertex v is replaced by two vertices v_1 and v_2 and the arcs are adjusted as shown above.

new arcs in G' get capacity 1 and the old ones capacity $2n+1$ such that in a minimum arc cut the new arcs will be chosen.

According to the well-known maxflow-mincut theorem [3], the problem of finding a minimum $s-t$ arc cut in a graph G equals the problem of finding a maximum value of a $s-t$ flow in G.

Corollary 2. *The problem of finding a splitting in a* $\{\mathrm{P1}, \mathrm{N1}, \mathrm{M2}\}$*-2-CNF Horn formula equals the problem of finding a maximum flow in a graph* G'_Φ.

Since the graph G'_Φ and its capacities are of polynomial size with respect to the formula size, the discussion above immediately yields the following theorem.

Theorem 3. MinSplit-$\{\mathrm{P1}, \mathrm{N1}, \mathrm{M2}\}$-2-SAT *is polynomial-time solvable.* □

Next, we show that MinSplit-$\{\mathrm{P1}, \mathrm{N2}\}$-2-SAT is as hard to approximate as MinVC. This result immediately implies that all remaining subcases of Horn formulas are also as hard to approximate as MinVC since they are generalizations of MinSplit-$\{\mathrm{P1}, \mathrm{N2}\}$-2-SAT.

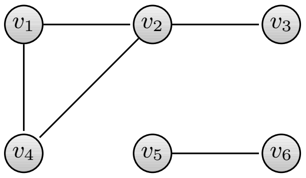
Theorem 4. MinVC $\leq_{\mathrm{AP}}$ MinSplit-$\{\mathrm{P1}, \mathrm{N2}\}$-2-SAT.

Proof. We present an AP-reduction from MinVC to MinSplit-$\{\mathrm{P1}, \mathrm{N2}\}$-2-SAT. For this, we give a polynomial-time function transforming any MinVC instance G into a MinSplit-$\{\mathrm{P1}, \mathrm{N2}\}$-2-SAT instance Φ_G such that any α-approximate feasible solution for Φ_G can be transformed in polynomial time into a feasible solution for G achieving the same approximation ratio. For more details on the general concept of AP-reductions, see [1,6].

Let G be a MinVC instance, where $G=(V,E)$ is an undirected graph. Let $V=\{v_1,\dots,v_n\}$. We construct a formula Φ_G from G as follows: Φ_G contains a positive 1-clause (x_i) for each non-isolated vertex $v_i \in V$ and a negative 2-clause $(\overline{x_i} \vee \overline{x_j})$ for each edge $\{v_i, v_j\} \in E$. An example of this construction is shown in Figure 5.

We now show that every vertex cover in G corresponds to a feasible set of split variables of the same size in Φ_G and vice versa.

Let $C=\{v_{i_1},\dots,v_{i_k}\}$ be a vertex cover of G of size k. We consider the corresponding variable set $X_C=\{x_{i_1},\dots,x_{i_k}\}$ in Φ_G. Following the construction, since C is a vertex cover, every 2-clause in Φ_G contains at least one variable from



$$\Longrightarrow \quad \begin{aligned} \Phi_G = {} & (x_1) \wedge (x_2) \wedge (x_3) \wedge (x_4) \wedge (x_5) \wedge (x_6) \\ & \wedge(\overline{x_1} \vee \overline{x_2}) \wedge (\overline{x_1} \vee \overline{x_4}) \wedge (\overline{x_2} \vee \overline{x_3}) \\ & \wedge(\overline{x_2} \vee \overline{x_4}) \wedge (\overline{x_5} \vee \overline{x_6}) \end{aligned}$$

Fig. 5. An example of the construction used in the proof of Theorem 4

X_C. Thus, splitting the variables from X_C removes all 2-clauses and the remaining formula consists of positive 1-clauses only and hence is obviously satisfiable.

Let, on the other hand, $X = \{x_{i_1}, \ldots, x_{i_k}\}$ be a set of variables whose splitting makes Φ_G satisfiable. Since there exists a positive 1-clause for each variable in Φ_G, every partial assignment setting any variable to 0 violates at least one of these 1-clauses. Thus, every variable that remains unsplit has to be assigned the value 1. This means that every 2-clause in Φ_G has to contain at least one variable from X. We consider the corresponding set $C_X = \{v_{i_1}, \ldots, v_{i_k}\}$ of vertices in G. Due to the construction, C_X is a vertex cover of G of size k.

Summing up, there is a one-to-one correspondence between vertex covers for G and feasible solutions for Φ_G of the same size proving our claim. □

We conclude this section with showing that MinSplit-{P1, N1, N2}-2-SAT can be approximated as good as MinVC. The upper bound on the approximability of MinSplit-{P1, M2, N2}-2-SAT and MinSplit-Horn-2-SAT remains open. For these cases, we only know about an $O(n/\log n)$-approximative algorithm due to Mömke that was mentioned in [11].

Theorem 5. *Any polynomial-time α-approximation algorithm for* MinVC *can be used to approximate* MinSplit-{P1, N1, N2}-2-SAT *within a factor of α in polynomial time.*

Proof. We first preprocess a {P1, N1, N2}-CNF formula Φ in order to get a {P1, N2}-CNF formula Φ'. We first remove all clauses containing variables x_i with (x_i) and $(\overline{x_i})$ in Φ and add those variables to the splitting set. All remaining variables x_i with $(\overline{x_i})$ in Φ can be set to 0 such that no clause is violated and all clauses containing the variable x_i are satisfied. Thus, we remove all clauses containing those variables x_i. After that, we set all variables x_i occurring only positively or only negatively in Φ to the value 1 or 0, respectively. The remaining formula Φ' contains only clauses of type P1 and N2 because of the construction, and every variable occurs in a positive 1-clause.

Now, we present a reduction from MinSplit-{P1, N2}-2-SAT to MinVC. Let Φ' be a {P1, N2}-CNF formula with variable set $X = \{x_1, \ldots, x_n\}$. Then, $G_{\Phi'}$ is a graph with vertex set $V_{\Phi'} = \{v_i \mid (x_i) \text{ in } \Phi'\}$ and edge set $E_{\Phi'} = \{\{v_i, v_j\} \mid (\overline{x_i} \vee \overline{x_j}) \text{ in } \Phi'\}$. Note that, since every possible positive 1-clause is present in Φ', this is exactly the reverse of the construction used in the proof of Theorem 4, where we have already proven the one-to-one correspondence between feasible splitting sets for Φ' and vertex covers for $G_{\Phi'}$. This proves our claim. □

4 Maximum Assignment in 2-CNF Horn Formulas

In this section, we deal with the approximability of MaxAssign-Horn-2-SAT and its subproblems. According to Observation 1, every poynomial-time algorithm for minimum splitting immediately yields a polynomial-time algorithm for maximum assignment. Hence, the results of Observation 3 and Theorem 3 directly carry over to MaxAssign-Horn-2-SAT.

Observation 4. *For* $S = \{\mathrm{P1}, \mathrm{M2}\}$ *or* $S = \{\mathrm{P1}, \mathrm{N1}\}$ *or* $S = \{\mathrm{N1}, \mathrm{M2}, \mathrm{N2}\}$ *or any subset thereof,* MaxAssign-S-2-SAT *is solvable in linear time.* □

Theorem 6. MaxAssign-$\{\mathrm{P1}, \mathrm{N1}, \mathrm{M2}\}$-2-SAT *is polynomial-time solvable.* □

It is well known that, if C is a vertex cover of size k in a graph $G = (V, E)$ with $|V| = n$, then $V - C$ is an independent set of size $n - k$ in G. This strong correspondence between MinVC and MaxIS resembles the correspondence between minimum splitting and maximum assignment. Thus, we can use similar ideas as in the previous section to prove that MaxAssign-$\{\mathrm{P1}, \mathrm{N2}\}$-2-SAT is as hard to approximate as MaxIS and that MaxAssign-Horn-2-SAT can be approximated using MaxIS algorithms.

Theorem 7. *Unless P=NP,* MaxAssign-$\{\mathrm{P1}, \mathrm{N2}\}$-2-SAT *cannot be better approximated than* MaxIS.

Proof sketch. We use the same reduction as in the proof of Theorem 4 to transform a given MaxIS instance G into a MaxAssign-$\{\mathrm{P1}, \mathrm{N2}\}$-2-SAT instance Φ_G. Following the discussion about the relation of MinVC and MaxIS above, it is easy to see that every independent set in G corresponds to a set of variables in Φ_G of the same size which can be assigned the truth value 1 without generating an unsatisfied clause. □

Theorem 8. *Any polynomial-time $f(n)$-approximation algorithm for* MaxIS *can be used to approximate* MaxAssign-Horn-2-SAT *within $f(2 \cdot n)$ in polynomial time, where n denotes the number of vertices or variables, respectively.*

Proof. Let Φ be an input instance for MaxAssign-Horn-2-SAT. We start with a preprocessing of Φ. If there exists a variable x that occurs both in a positive and a negative 1-clause, then x obviously cannot be assigned any truth value, thus, we delete all clauses containing x from the formula. If, for some variable x and some literal l, there exist two clauses $(\overline{x})$ and $(\overline{x} \vee l)$, we can remove the clause $(\overline{x} \vee l)$. This is due to the fact that any partial assignment that does not contradict $(\overline{x})$ also does not contradict $(\overline{x} \vee l)$. Analogously, for any two clauses (x) and $(x \vee \overline{y})$, we can remove the clause $(x \vee \overline{y})$. Finally, if some variable appears only positively or only negatively in the formula, we can assign the respective value to it and shrink the formula accordingly.

For the remainder of the proof, let $\Phi = C_1 \wedge \ldots \wedge C_m$ denote a 2-CNF Horn formula on the variable set $X = \{x_1, \ldots, x_n\}$ that cannot be further shrunk

$$\begin{aligned}\Phi = (x_1) \wedge (\overline{x_2}) \wedge (\overline{x_1} \vee x_2)\\ \wedge(\overline{x_1} \vee x_3) \wedge (\overline{x_1} \vee \overline{x_3})\\ \wedge(x_2 \vee \overline{x_3})\end{aligned} \quad \Longrightarrow \quad G_\Phi :$$

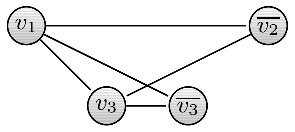


Fig. 6. An example of the construction in the proof of Theorem 8

by the preprocessing as described above. We construct a graph G_Φ from Φ as follows. For each variable x_i that appears in a positive 1-clause, we add a vertex v_i, for each variable x_i that appears in a negative 1-clause, we add a vertex $\overline{v_i}$, and for each variable x_k only appearing in 2-clauses of Φ, we add two vertices v_k and $\overline{v_k}$ and an edge between them. Additionally, we add an edge $\{\overline{v_i}, v_j\}$ for each clause $(x_i \vee \overline{x_j})$ and an edge $\{v_i, v_j\}$ for each clause $(\overline{x_i} \vee \overline{x_j})$. Note that, due to the preprocessing, this construction is well-defined. An example of the construction is shown in Figure 6.

We now show that any independent set in G_Φ directly translates into a set of variables in Φ that can be assigned without raising a contradiction. Let $V' \subseteq V_\Phi$ be an independent set in G_Φ. By the construction, at most one of the vertices v_i and $\overline{v_i}$ can be part of V', for every $1 \leq i \leq n$. Every variable x_i corresponding to a vertex $v_i \in V'$ can be set to the value 1 and every variable x_i corresponding to a vertex $\overline{v_i} \in V'$ can be set to the value 0. Since V' is an independent set, this leaves at least one endpoint of each edge in G_Φ unassigned, thus the corresponding 2-clause does not cause a contradiction. On the other hand, any partial assignment not causing any conflict in Φ directly translates into an independent set in G_Φ of the same size. Since the graph G_Φ has at most $2n$ vertices, we lose at most a factor of 2 on the approximation ratio. □

Corollary 3. *Since* MaxIS *can be approximated with at least a linear function, any polynomial-time $f(n)$-approximation algorithm for* MaxIS *can be used to approximate* MaxAssign-Horn-2-SAT *within $2 \cdot f(n)$ in polynomial time.*

5 Maximum Assignment in Exact-2-CNF Formulas

In this section, we deal with the case of E2-CNF formulas, i. e., formulas containing only clauses of types M2, P2, and N2. The approximation hardness of this problem was implicitly shown by Steinová [11], in her proof of the approximation hardness of the general MaxAssign-2-SAT, she constructs formulas consisting of 2-clauses only.

Theorem 9 (Steinová, 2012). *There exists an AP-reduction from* MaxIS *on undirected hypergraphs to* MaxAssign-E2-SAT. □

We complement this result by the following upper bound.

Theorem 10. MaxAssign-E2-SAT $\leq_{\text{AP}}$ MaxIS.

$$\begin{aligned}\Phi = {} & (x_1 \vee x_3) \wedge (x_1 \vee \overline{x_3}) \\ & \wedge (\overline{x_1} \vee \overline{x_2}) \wedge (x_2 \vee x_3) \\ & \wedge (x_2 \vee \overline{x_3}) \wedge (\overline{x_2} \vee \overline{x_3})\end{aligned} \quad \Longrightarrow \quad G_\Phi :$$

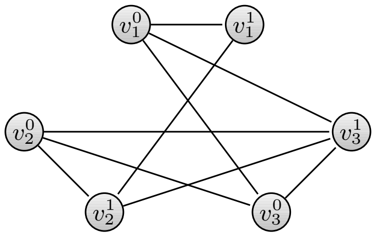


Fig. 7. An example of the construction in the proof of Theorem 10

Proof. To prove AP-reducability of MaxAssign-E2-SAT to MaxIS, we need the following polynomial-time function that transforms an E2-CNF formula Φ with variable set $X = \{x_1, x_2, \ldots, x_n\}$ into a graph instance G_Φ for MaxIS with vertex set $V_\Phi = \{v_i^0, v_i^1 \mid x_i \in X\}$ and ege set $E_\Phi = \{\{v_i^0, v_j^0\} \mid (x_i \vee x_j) \text{ in } \Phi\} \cup \{\{v_i^0, v_j^1\} \mid (x_i \vee \overline{x_j}) \text{ in } \Phi\} \cup \{\{v_i^1, v_j^1\} \mid (\overline{x_i} \vee \overline{x_j}) \text{ in } \Phi\} \cup \{\{v_i^0, v_i^1\} \mid 1 \leq i \leq n\}$. In other words, every variable x_i gives rise to two vertices v_i^0 an v_i^1 representing the assignment 0 or 1. Those two vertices are connected by an edge and there is also an edge for every assignment restriction given by the clauses (see Figure 7). Obviously, this transformation can be implemented in polynomial time.

We show that every feasible set of variables in Φ with a partial assignment not violating any clause corresponds to an independent set in G_Φ of the same size.

Let $X' = \{x_{i_1}, x_{i_2}, \ldots, x_{i_k}\} \subseteq X$ be a subset of variables and let $\alpha \colon X' \to \{0,1\}$ be a partial assignment such that no clause is evaluated to 0. We show that, for a variable $x_i \in X'$ and an assignment $\alpha(x_i) = b$ for some $b \in \{0,1\}$, the corresponding vertex v_i^b is part of an independent set $I_{X'} \subseteq V_\Phi$ in G_Φ. No two endpoints of edges of type $\{v_i^0, v_i^1\}$ will be part of $I_{X'}$ since a variable can be set either to 1 or to 0, but not to both values. Moreover, no two endpoints of the remaining edges will be both part of $I_{X'}$ since then the corresponding assignment would cause an empty clause. Therefore, $I_{X'}$ is an independent set in G_Φ.

Conversely, let $I \subseteq V_\Phi$ be an independent set in G_Φ. For every $v_i^b \in I$, let $\alpha(x_i) = b$ define the partial assignment for the vertex set $S_I \subseteq X$ in Φ. This assignment does not violate any clause since never both endpoints of an edge will be part of I and therefore, never both literals of a clause will be set to 0.

Hence, we have a one-to-one correspondence between a partial assignment in Φ not causing empty clauses and an independent set in G_Φ of the same size. □

6 Conclusion

We have explored the approximability of the minimum splitting problem and the maximum assignment problem in various special cases of 2-CNF formulas, including Horn formulas and E2-CNF formulas. The main open problem is to close the gap between the trivial upper bound and the lower bound for general 2-CNF Horn formulas and for those excluding negative 1-clauses in the splitting case. It would also be interesting to extend the results to other classes of non-Horn 2-CNF formulas besides the E2-CNF formulas.

References

1. Ausiello, G., Crescenzi, P., Gambosi, G., Kann, V., Marchetti-Spaccamela, A., Protasi, M.: Complexity and Approximation. Springer (1999)
2. Boppana, R., Halldórsson, M.: Approximating maximum independent sets by excluding subgraphs. BIT Numerical Mathematics 32, 180–196 (1992)
3. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press (2009)
4. Dowling, W.F., Gallier, J.H.: Linear-time algorithms for testing the satisfiability of propositional Horn formulae. Journal of Logic Programming 1(3), 267–284 (1984)
5. Håstad, J.: Clique is hard to approximate within $n^{1-\varepsilon}$. In: FOCS 1996: Proceedings of the 37th Annual Symposium on Foundations of Computer Science, p. 627. IEEE Computer Society (1996)
6. Hromkovič, J.: Algorithmics for Hard Problems. Introduction to Combinatorial Optimization, Randomization, Approximation, and Heuristics. Texts in Theoretical Computer Science. An EATCS Series. Springer (2003)
7. Khot, S., Regev, O.: Vertex cover might be hard to approximate to within 2-epsilon. Journal of Computer and System Sciences 74(3), 335–349 (2008)
8. Monien, B., Speckenmeyer, E.: Ramsey numbers and an approximation algorithm for the vertex cover problem. Acta Informatica 22, 115–123 (1985)
9. Ono, H.: Personal communication
10. Papadimitriou, C.H., Yannakakis, M.: Optimization, approximation, and complexity classes. Journal of Computer and System Sciences 43(3), 425–440 (1991)
11. Steinová, M.: Measuring Hardness of Complex Problems: Approximability and Exact Algorithms. PhD thesis, ETH Zurich (2012)

Boundary-to-Boundary Flows in Planar Graphs

Glencora Borradaile[1] and Anna Harutyunyan[2,⋆]

[1] Oregon State University
[2] Vrije Universiteit Brussel

Abstract. We give an iterative algorithm for finding the maximum flow between a set of sources and sinks that lie on the boundary of a planar graph. Our algorithm uses only $O(n)$ queries to simple data structures, achieving an $O(n \log n)$ running time that we expect to be practical given the use of simple primitives. The only existing algorithm for this problem uses divide and conquer and, in order to achieve an $O(n \log n)$ running time, requires the use of the (complicated) linear-time shortest-paths algorithm for planar graphs.

Keywords: maximum flow, multiple terminal, planar graphs.

1 Introduction

The problem of finding maximum flow in planar graphs has a long history, starting with the work of Ford and Fulkerson [7] in which the Max-flow, Min-cut Theorem was proved and the augmenting-paths algorithm was introduced. Since then, algorithms for maximum flow in planar graphs have fallen into one of three paradigms: augmenting paths, divide and conquer using small balanced planar separators, or via shortest paths in the dual. We note a subset of these results that are relevant to this paper. Borradaile and Klein gave an augmenting-paths algorithm for maximum st-flow in directed planar graphs that uses dynamic trees to achieve an $O(n \log n)$ running time [3]. For the special case when s and t are on the same face, an augmenting-paths algorithm can be simulated via Dijkstra's algorithm or, equivalently, determined from shortest-path distances in the dual graph [9] (details in Section 2). Borradaile et al. gave a rather complicated $O(n \log^3 n)$-time divide-and-conquer algorithm for when there are multiple sources and sinks (not necessarily on a common face) [4]. For the special case when these sources and sinks are all on a common face[1] (such as the boundary of the embedded graph), Miller and Naor gave a simpler divide-and-conquer algorithm [12].

In this work we give an iterative algorithm for this last *boundary-to-boundary* case. While our algorithm does not improve on the asymptotic running time of Miller and Naor's work, in order for Miller and Naor's algorithm to be implemented in $O(n \log n)$ time, one requires repeated applications of the linear-time

⋆ Work done while at Oregon State University.

[1] Note that there is no planarity-maintaining reduction from this case to the single-source, single-sink case.

© Springer-Verlag Berlin Heidelberg 2013

shortest-paths algorithm of Henzinger et al. [10]. This shortest-paths algorithm is arguably impractical: it is also a divide-and-conquer algorithm using small planar separators, involves 'large constants' and, to our knowledge, has not been implemented. Our algorithm, on the other hand, requires just $O(n)$ (with a small constant) queries to simple data structures: namely a priority queue and a linked list [6].

Our algorithm is an augmenting-paths algorithm that iterates over the source-sink pairs. We simulate finding the flow between a given source and sink using Hassin's method – via Dijkstra's algorithm in the dual graph. In order to prevent searching the same region of the graph multiple times, we search the graph in a biased way [8], such that we need only reuse the boundary of the searched region for augmenting further source-sink pairs. In order to reuse these boundaries efficiently, we use a simple generalization of priority queues in which queues are merged whose relative priorities differ by a constant or *offset*. These *offset queues* are implemented using edge weights to encode the offset in a tree implementation of the heap; doing so does not affect the asymptotic running time of the basic priority queue operations. Details are given in Appendix A.

We believe that the methods used in this paper may be applicable to other planar flow problems. For example, in a companion paper [2], we argue that the augmenting-paths algorithm of Borradaile and Klein for maximum *st*-flow in directed planar graphs can also be simulated by Dijkstra in the dual graph; the details of the implementation in this paper may lead to an $O(n \log n)$ algorithm for maximum *st*-flow in directed planar graphs that does not require the more cumbersome dynamic-trees data structure.

1.1 Definitions

We give a brief outline of definitions where we may stray from convention. For more complete and formal definitions, please refer to Borradaile's dissertation [5]. We extend any function or property on elements to sets of elements in the natural way.

Our algorithms are for directed graphs, but we consider the underlying undirected graph where each edge has two oppositely directed darts. Darts are oriented from *tail* to *head*. Capacities, $\boldsymbol{c}$, on the darts are positive and asymmetric, reflecting the original directed problem. Paths and cycles are sequences of darts and so are naturally directed; a path or a cycle may visit the same vertex multiple times; those that do not are *simple*; a path may be trivial, in which case it is a vertex. $X[a, b]$ denotes the a-to-b subpath of X where X is a path, cycle or tree; $\circ$ denotes the concatenation of paths (which may result in a cycle).

A flow $\boldsymbol{f}$ is an assignment of real numbers to darts that is antisymmetric (for a dart and its reverse), respects capacities and is balanced at all non-terminal (non-source, sink) vertices. The value $|\boldsymbol{f}|$ of a flow is the net flow entering the sinks. A flow is a circulation if there are no terminals. The residual capacities $\boldsymbol{c}_f$ of capacities $\boldsymbol{c}$ w.r.t. flow $\boldsymbol{f}$ are given by:

$$\boldsymbol{c}_{\boldsymbol{f}}[d] = \boldsymbol{c}[d] - \boldsymbol{f}[d], \ \forall \text{darts } d \tag{1}$$

A path or cycle X is residual if the residual capacity of every dart in X is strictly positive. A dart is saturated if its residual capacity is zero. Residuality is w.r.t. capacities (such as $\boldsymbol{c}$ or $\boldsymbol{c_f}$).

An xy-cut in G is a set of darts C, the removal of which leaves no x-to-y paths. The value of a cut is the total capacity of its darts. The value of the *minimum* xy-cut equals to that of the maximum xy-flow [7].

We use the usual definitions for planar graphs and their duals. We denote any path, cycle, vertex, face, dart in the dual graph with a $*$-superscript. If d is a dart in G, then d^* is the corresponding dual dart; if v is a vertex and f is a face in G, v^* is a *face* and f^* is a *vertex* in G^*. The boundary of the graph is denoted ∂G and is taken to be clockwise. We refer to simple cycles as being clockwise (c.w.) or counterclockwise (c.c.w.); c.w. and c.c.w. depend on the choice of infinite face, f_∞, which, throughout this paper, we will take to be the face common to all the sources and sinks.

For two non-crossing x-to-y paths P and Q, we say P is left of Q if $P \circ rev(Q)$ is c.w. A path is leftmost if there are no paths left of it. For an x-to-y path P that starts and ends on ∂G, we say a face, edge, path, etc. X is (strictly) left of P if X is (strictly) contained by the c.w. cycle $\partial G[x, y] \circ rev(P)$. We say that a planar flow $\boldsymbol{f}$ is *leftmost* if every c.w. cycle is non-residual w.r.t. $\boldsymbol{c_f}$. We say that capacities are c.w. acyclic if every c.w. cycle is non-residual w.r.t. the capacities.

2 Leftmost Maximum Flows and Shortest Paths

Khuller, Naor and Klein [11] showed that a flow that is derived from shortest-path distances in the dual is c.w. acyclic. Formally:

Theorem 1 (Clockwise acyclic flows). *Let $\boldsymbol{d}$ be the shortest-path distances in G^* from f^*_∞ interpreting capacities as lengths. Then every c.w. cycle is non-residual w.r.t. the flow*

$$\boldsymbol{f}[d] = \boldsymbol{d}[head(d^*)] - \boldsymbol{d}[tail(d^*)] \ \forall \ darts \ d \tag{2}$$

where $head(d^*)$ and $tail(d^*)$ are the head and tail vertices of d^* in G^*.

Earlier, Hassin had used this idea to find a maximum st-flow in an st-planar graph [9]. We can view his algorithm by turning it into a circulation problem: introduce a new infinite-capacity arc ts embedded so that every s-to-t residual path forms a c.w. cycle with ts and then saturate the c.w. cycles. We describe an equivalent formulation which we use in this paper. Split the dual vertex f^*_∞ into two vertices a^*_∞ and b^*_∞ such that all the darts in $\partial G[s,t]^*$ are incident to a^*_∞ and all the darts in $\partial G[t,s]^*$ are incident to b^*_∞; denote the resulting graph G^*_{st}. Let $\boldsymbol{d}[x^*]$ be the shortest-path distance from a^*_∞ to x^* in G^*_{st}, viewing capacities as lengths. Then the flow assignment $\boldsymbol{f_{st}}$ for G given as in Equation (2) is a maximum st-flow. It follows directly from Theorem 1 that $\boldsymbol{f_{st}}$ is the leftmost maximum st-flow.

Since simple cuts in the primal map to simple cycles in the dual (and vice versa) [13], the darts of an st-cut C form an a^*_∞-to-b^*_∞ path C^* in G^*_{st}. If C is a minimum cut, C^* is a shortest path.

Observation 1. *A leftmost flow w.r.t. c.w. acyclic residual capacities is acyclic.* [3]

Because of this acyclicity, one can easily show:

Observation 2. *Let $\boldsymbol{c}$ be c.w. acyclic capacities and let $\boldsymbol{f}$ the leftmost, max st-flow for s and t on f_∞. Then there is a decomposition of $\boldsymbol{f}$ into unique, non-crossing s-to-t paths $P_1, P_2, \ldots, P_\ell$ where P_i carries $f_i > 0$ units of flow and P_i is left of P_j $\forall i < j$. Further, an augmenting-paths algorithm that always saturates the leftmost path first saturates the paths $P_1, \ldots, P_\ell$ in order.*

Our algorithm requires c.w. acyclic capacities; the analysis will use this fact indirectly by invoking Observation 2. We will achieve this property in a pre-processing step and maintain this as an invariant throughout the algorithm. It follows from Equation (2) and Observation 2 that, for every primal face x (dual vertex x^*):

$$\boldsymbol{d}[x^*] = \begin{cases} \sum_{j=1}^{i} f_j & \text{if } x \text{ is right of } P_i \text{ and left of } P_{i+1} \\ \sum_{j=1}^{\ell} f_j = |\boldsymbol{f}| & \text{if } x \text{ is right of } P_\ell \end{cases} \tag{3}$$

2.1 *st*-Planar Flow via Biased Search

We describe how to find an *st*-planar flow via biased search (in the dual) that does not necessarily search the entire graph, assuming that the initial capacities are c.w. acyclic. We assume that there are no degree-2 vertices in the primal; any such vertex could be removed by merging the adjacent darts (in each direction) and keeping the minimum of the capacities. Parallel darts (not antiparallel) can be merged by taking the sum of their capacities. We additionally assume that the finite faces of the primal are triangulated (which can be achieved by the addition of 0-capacity edges).

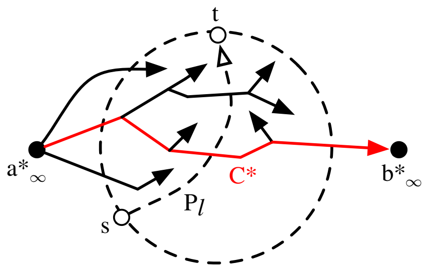


Fig. 1. ∂G is the dashed circle and the dashed s-to-t path is P_ℓ. In G^*_{st}, a^*_∞ is incident to the duals of all the arcs on the path of the circle c.w. from s to t and b^*_∞ is incident to the duals of all the arcs on the path of the circle c.w. from t to s. The solid tree is the search tree used in the biased search algorithm with the a^*_∞-to-b^*_∞ path representing the leftmost cut C^*.

We implicitly and iteratively build a decomposition as given in Observation 2 using Dijkstra's algorithm in the dual. Initially $P_1 = \partial G[s,t]$. In phase i, we have already found path P_i; we maintain that, at the start of phase i, the faces adjacent to and right of P_i are in the queue Q_i. (Keep in mind that faces are vertices in the dual, and we are really just finding shortest paths in the dual graph, applying the standard rules for Dijkstra's algorithm.) The priority of face x is the capacity of the minimum-capacity dart bounding x in P_i. Say the minimum priority in the queue is q; to find P_{i+1} we pop faces off the queue with priority q until the minimum priority in the queue is $> q$. Now we have popped off all the faces between P_i and P_{i+1} (by Equation (3)) and Q_{i+1} contains all the faces to the right of and adjacent to P_{i+1}.

So far, we have just described Hassin's algorithm, but have made explicit the augmenting paths that are implicit in his algorithm. We have also identified *phases.* In each phase, all the faces of a given distance label are explored via 0-length darts (in the dual).

We modify the algorithm so that we do not explore the entire graph. Note that all the faces to the right of P_ℓ (the last augmenting path), by Equation (3) have distance label $|\boldsymbol{f}|$. Rather than label all these faces, after getting to the start of phase ℓ, we wish to find the *leftmost cut.* Let C^* be the leftmost, shortest a^*_∞-to-b^*_∞ path in G^*_{st}; C is the leftmost cut. The part of C^* that is strictly to the right of P_ℓ consists of 0-length darts, since the sum of the capacities of the darts in C^* that are in $P_1, \ldots, P_\ell$ is $|\boldsymbol{f}|$ by Equation (3). In addition to identifying the leftmost cut, we wish to not explore any part of the graph strictly right of P_ℓ and C^*. (See Figure 1.)

We find the leftmost cut by at each phase additionally maintaining an ordering A_i of the faces in Q_i that reflects their order along P_i from t to s. We maintain and query the ordering using the order maintenance data structure DSOrder due to Dietz and Sleator [6] which is a circularly linked list with order information determined using 2's complement arithmetic. (See Appendix B for details. Each of the operations takes either $O(1)$ or $O(\log n)$ time per visited face.) During a phase, we:

(1) Start with faces that are closest to t in the ordering.

(2) Explore along 0-length darts in the dual in a depth-first *leftmost* fashion; this can be done by following the combinatorial embedding of the darts around a vertex in a c.w. order, using the parent dart in the search tree implicit to Dijkstra's algorithm [3].

(3) If we reach b^*_∞ during this search, we immediately stop the algorithm. (More details of this are given below.)

(4) At the end of this 0-length exploration, we remove from the queue and order any faces that we have reached in this exploration. Suppose T^* is the dual search tree we have explored that contains the shortest paths found by Dijkstra's algorithm, rooted at a face adjacent to P_i. We add the never-visited faces adjacent to T^* in their c.w. order around T^* (according to their shortest adjacency to T^*). This ordering is easily visualized by contracting the edges of T^* and considering the c.w. ordering of the darts around the new (dual) vertex.

At the start of each phase, the queue and the order contain the same set of elements. The leftmost-bias to the search additionally guarantees that the final dual search tree T^* contains leftmost shortest paths. This can be easily shown via induction. Since we stop as soon as we reach b^*_∞ and we search in a leftmost fashion, T^* does not contain any darts *strictly* right of both the last flow path P_ℓ found and $T^*[a^*_\infty, b^*_\infty]$. In this way, we also guarantee:

Observation 3. *At the end of this biased search, the queue and order contain the faces adjacent to and right of P_ℓ.*

In our multi-source, multi-sink algorithm, we will reuse this queue and order. To do so, we need to know the residual capacities of the darts in P_ℓ. If a face f in the queue has exactly one bounding arc in P_ℓ, then the priority of f reflects exactly the residual capacity of that dart. If f has two bounding darts d_1 and d_2 in P_ℓ (i.e., the head of d_1^* and d_2^* in G^* is f^*), then, *to the right of* P_ℓ, we can only push the minimum of these darts' residual capacities along this section of P_ℓ. (Put another way, if we remove everything strictly to the left of P_ℓ, d_1 and d_2 would be incident to a degree 2 vertex, which we would remove according to the rule at the start of this section.) We get:

Observation 4. *The priority of a face f in the queue reflects the residual capacity of the dart(s) bounding the face in P_ℓ; the residual capacity is the priority less $|\boldsymbol{f}|$.*

Subtracting $|\boldsymbol{f}|$ from the priorities in the ending queue can be done in $O(1)$ time using offset queues (Appendix A). Finally, the DSOrder data structure does not allow us to pull the *first* element of the order (having minimum priority in the queue) but does allow us to sort a subset of items. In doing so, we spend $O(\log n)$-amortized time per element. We do not wish to repeat this work. If we reach b_∞^* in the middle of a phase and have a subset of items X that we have sorted using DSOrder, we break the ties in the priorities of these items in the priority queue. When we return to use this queue/order, we will not need to resort these items.

3 Algorithm

For simplicity of presentation we will assume that the terminals are alternating sources and sinks along ∂G. This can be attained by taking a consecutive group of sources S, introducing a new source and connecting the new source to every source in S with an infinite capacity arc. We number the sources and sinks according to their c.w. ordering on ∂G, $s_1, t_1, s_2, t_2, \ldots, s_m, t_m$, starting with an arbitrary source. We return the difference between the original capacities and final residual capacities, which, by Equation (1), is the corresponding flow.

```
AbstractFlow ($G$, $\{s_1, t_1, s_2, t_2, \ldots, s_m, t_m\}$, $\boldsymbol{c}$)
    Saturate all $s_j$-to-$t_i$ residual paths $\forall i < j$ and all c.w. cycles.
    Let $\boldsymbol{c}_0$ be the resulting residual capacities.
    For $j = 1, 2, \ldots, m$:
        for $i = j, j-1, \ldots, 1$:
            let $\boldsymbol{c}'_{ij}$ be the current residual capacities.
            Find the leftmost $s_i$-to-$t_j$ flow $\boldsymbol{f}_{ij}$ w.r.t. $\boldsymbol{c}'_{ij}$.
            Let $\boldsymbol{c}_{ij}$ be the residual capacities of $\boldsymbol{c}'_{ij}$ w.r.t. $\boldsymbol{f}_{ij}$.
    Return $\boldsymbol{c}[d] - \boldsymbol{c}_{mm}[d]$ for all darts $d$.
```

The first step can be done with one shortest-path computation in the dual as follows (in $O(n \log n)$ time using Dijkstra's algorithm, for example); refer to

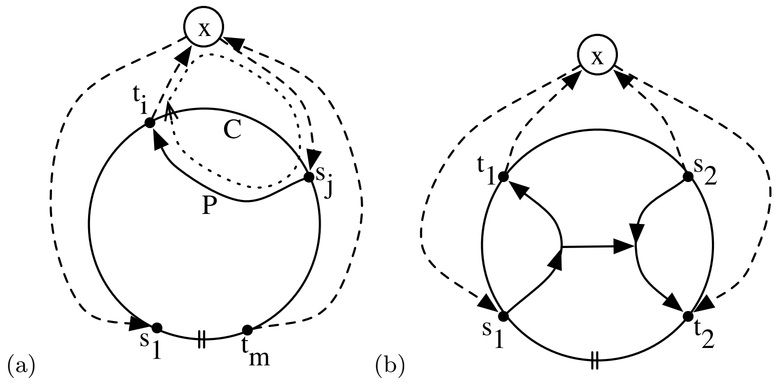


Fig. 2. (a) Illustrating the first step of the AbstractFlow. (b) A simple example illustrating why this first step cannot be repeated to find the overall maximum flow. The equivalent step would saturate all c.c.w. cycles. If the solid edges have equal capacity, this would saturate the s_1-to-t_2 path, since the method for saturating all c.c.w. cycles (like for c.w. cycles) saturates all largest such cycles. However, doing so would create a residual path from s_2 to t_1.

Figure 2(a). Embed a vertex x in f_∞. Connect x to every source and every sink with infinite-capacity arcs. Embed these arcs so that s_1, t_m and x are on the infinite face. Let $\boldsymbol{f}$ be the circulation that saturates all the c.w. residual cycles in this graph (Theorem 1). Let $\boldsymbol{c}_0$ be the residual capacities of the darts in G w.r.t. $\boldsymbol{f}$. Consider any simple path P from s_j to t_i in G. For $j > i$, $P \circ t_i x \circ x s_j$ is a c.w. cycle C. Therefore C must be non-residual w.r.t. $\boldsymbol{c}_0$ and, since the arcs $t_i x$ and $x s_j$ have infinite capacity, P must be non-residual w.r.t. $\boldsymbol{c}_0$.

Note that while the iterative part of the algorithm saturates all s_i-to-t_j paths $\forall i < j$, we cannot achieve this with a symmetric application of the first step. The simple example in Figure 2(b) illustrates why.

In the remainder of the paper we will give an efficient implementation of the double loop of AbstractFlow. We first show that the abstract algorithm guarantees several useful invariants that limit the region of the graph that is involved in each iteration. These invariants allow us to explore the graph in such a way that no region is explored multiple times. Correctness of AbstractFlow will also follow from these invariants. By iteration i, j, we will mean iteration i of the inner loop and iteration j of the outer loop.

3.1 Invariants

Since only leftmost flows are augmented we get (by definition and induction):

Invariant 1. *There are no clockwise residual cycles in G w.r.t. $\boldsymbol{c}_{ij}, \forall i \leq j$.*

Since the sink is in common to all the iterations of the inner loop, for a given iteration of the outer loop, we get:

Invariant 2. *There are no residual s_j-to-t_k paths w.r.t. $\boldsymbol{c}_{i,k}$ for $j > i$.*

More formally, this follows from the Sinks Lemma [4]. The following invariant shows that we do not undo the progress made by the first step of AbstractFlow.

Invariant 3. *There are no s_i-to-t_j residual paths s.t. $i > j$ w.r.t. $\boldsymbol{c}_0$ or $\boldsymbol{c}_{k\ell}$, $\forall k < \ell$.*

Proof. We prove this invariant by induction. It holds w.r.t. $\boldsymbol{c}_0$ as argued in Section 2. For a contradiction, let $\boldsymbol{c}_{k\ell}$ be the first residual capacities that introduce an s_i-to-t_j residual path R $(i < j)$. Then there must be an s_k-to-t_ℓ path A that is augmented in iteration k, ℓ and that uses a dart d in $rev(R)$.

Let x and y be the last and first, resp., vertices of R that are in A. A, $R[s_i, y]$ and $R[x, t_j]$ are residual w.r.t. $\boldsymbol{c}'_{k\ell}$ (the residual capacities at the start of iteration k, ℓ). It follows that $k \leq j$ and $\ell > i$, for otherwise we contradict the inductive hypothesis. However, iteration k, ℓ comes after i, ℓ in AbstractFlow. Invariant 2 tells us that there cannot be an s_i-to-t_ℓ path that is residual w.r.t. $\boldsymbol{c}'_{k\ell}$, contradicting the existence of $R[s_i, y] \circ A[y, t_\ell]$. □

The optimality of the flow found by AbstractFlow follows from the last invariant (along with Invariants 2 and 3):

Invariant 4. *There are no s_i-to-t_j residual paths w.r.t. $\boldsymbol{c}_{\ell k}$ for any ℓ and any $k > j$.*

Proof. We prove this invariant by induction. It holds w.r.t. $\boldsymbol{c}'_{1,j+1}$ by Invariant 2. For a contradiction, let $\boldsymbol{c}_{\ell k}$ be the first residual capacities that introduce an s_i-to-t_j residual path R. W.l.o.g. assume that $i \leq j$ as the case $i > j$ is handled by Invariant 3. Then there must be an s_ℓ-to-t_k path A that is augmented in iteration ℓ, k and that uses a dart d in $rev(R)$.

Let x and y be the first and last, resp., vertices of R that A shares. Since A and $R[y, t_j]$ are residual, $\ell \leq j$ by Invariant 3. However, by Invariant 2, there are no s_ℓ-to-t_j paths that are residual w.r.t. $\boldsymbol{c}_{1j}$, so $\ell > j$, a contradiction. □

3.2 Unusability Structures

We will illustrate our implementation of AbstractFlow with a recursive algorithm. To that end, we show that the cut and the flow found in iteration i, j separates the graph into two pieces that act independently for the remainder of the algorithm. Let P be the rightmost path in the path decomposition of $\boldsymbol{f}_{ij}$ given in Observation 2 (that has non-zero flow). The following lemma allows us to delete everything strictly to the left of P at the end of iteration i, j for future iterations without affecting optimality.

Lemma 1. *There are no paths from s_k to P that are residual w.r.t. $\boldsymbol{c}_{ij}$ for $k > i$.*

Proof. First we make an observation. Inner iterations $j, j-1, \ldots, i$ are equivalent to adding a new source s, connecting s to $s_j, s_{j-1}, \ldots, s_i$ by high-capacity arcs and saturating the leftmost max st_k-flow[2]. By Observation 2, this is done by saturating a set of non-crossing s-to-t_k paths $\mathcal{P} = P_1, P_2, \ldots$ ordered from left to right. In AbstractFlow, iteration ℓ, k will saturate a contiguous subset $\mathcal{P}_\ell$ of $\mathcal{P}$ for $i \le \ell \le j$. By saturating these paths in order, we first cut s_j from t_k by saturating $\mathcal{P}_j$, then cut s_{j+1} from t_k and so on.

For $i < k \le j$, the lemma follows from the fact that iteration k, j precedes i, j: a path Q, from s_k-to-P concatenated with the suffix of P, would be saturated before P. For $k > j$, Q would be residual w.r.t. capacities $\boldsymbol{c}'_{ij}$ since $\boldsymbol{f}_{ij}$ does not change the capacities of darts strictly to the right of P; Q violates Invariant 3. □

Let C be the leftmost minimum $s_i t_j$-cut. The next lemma shows that we can delete the darts in C (among others on the t_j side of the cut) without affecting optimality. In the biased-search algorithm (Section 2.1), the darts satisfying Lemma 2 are exactly those that are searched to the right of the last flow path (T^*) in finding the leftmost cut (C).

Lemma 2. *Let W^* be any from-a^*_∞ $|\boldsymbol{f}_{ij}|$-length path in $G^*_{s_i t_j}$ that is left of C^*. Then no s-to-t path that is residual w.r.t. $\boldsymbol{c}_{ij}$ uses a dart in W.*

Proof. For a contradiction, suppose there is a s_k-to-t_ℓ path R that is residual w.r.t. $\boldsymbol{c}_{ij}$ that uses a dart of W. Since, by Invariant 3, $\ell \ge k$, s_k must be on the t_j side of C for otherwise, R would have to cross back and forth across C, but the darts of C are only residual w.r.t. $\boldsymbol{c}_{ij}$ from the t_j side to the s_i side.

We have just finished iteration i, j, $k > j$, and so, by Invariant 3, there is an $s_k t_j$-cut K. Take K to be the *rightmost* of these cuts (defined analogously to leftmost). In $G^*_{s_i t_j}$, K^* is a c.c.w. cycle through b^*_∞; K^* is 0-length (or, equivalently, composed entirely of darts that are non-residual w.r.t. $\boldsymbol{c}_{ij}$).

K^* must be left of C^*, for otherwise, the leftmost-ness of C^* and the rightmostness of K^* would be violated. If R uses a dart d of W, then d must be on the s_k side of K. Then, in the dual, W^* must intersect K^* at a dual vertex x^*. But then $W^*[a^*_\infty, x^*] \circ K^*[x^*, b^*_\infty]$ is a a^*_∞-to-b^*_∞ path of length at most that of W^*; $W^*[a^*_\infty, x^*] \circ K^*[x^*, b^*_\infty]$ is left of C^*, contradicting that C is a leftmost cut. □

Lemmas 1 and 2 allow us to implement AbstractFlow recursively. That is, AbstractRecursiveFlow, below, finds the same (non-zero) flows $\boldsymbol{f}_{ij}$ in the same order as AbstractFlow. The recursive algorithm has a slightly different input, as there may be several consecutive sources for the recursive calls. We illustrate the algorithm without explicitly returning the flow. It is trivial to determine the flow from the residual capacities found throughout the algorithm.

[2] Note that in the implementation, we do not merge the sources in this way as doing so does not allow us to reuse the work done in previous iterations.

```
AbstractRecursiveFlow($G$, $\{s_1, t_1, \ldots, s_m, t_m\}$, $\boldsymbol{c}$)
    Saturate all $s_j$-to-$t_i$ residual paths $\forall i < j$ and all c.w. cycles.
    Let $\boldsymbol{c}_0$ be the resulting residual capacities.
    AbstractRecursiveFlowHelper ($G$, $\{\}$, $\{s_1, t_1, \ldots, s_m, t_m\}$, $\boldsymbol{c}_0$)

AbstractRecursiveFlowHelper($G$, $\{s_1, s_2, \ldots, s_{\ell-1}\}$, $\{s_\ell, t_\ell, s_{\ell+1}, t_{\ell+1}, \ldots, s_m, t_m\}$, $\boldsymbol{c}$)
    Find the leftmost $s_\ell$-to-$t_\ell$ flow $\boldsymbol{f}$ w.r.t. $\boldsymbol{c}$.
    Let $\boldsymbol{c}'$ be the residual capacities of $\boldsymbol{c}$ w.r.t. $\boldsymbol{f}$.
    Let $P$ be the rightmost path in the path-decomposition of $\boldsymbol{f}$ and let $C$ be the leftmost cut.
    Let $G_1$ and $G_2$ be the components resulting from deleting all the darts
            strictly to the left of $P$ and the darts of $C$ from $G$.
    If $t_\ell \in G_2$:
        Let $k$ be the greatest index s.t. $t_k \in G_2$.
        AbstractRecursiveFlowHelper($G_2$, $\{\}$, $\{s_{\ell+1}, t_{\ell+1}, \ldots, s_k, t_k\}$, $\boldsymbol{c}'$)
        Let $h$ be the smallest index $> \ell$ s.t. $t_h \in G_1$.
        Extend $Q$ and $A$ to contain all the faces in $G^*_{1,s_\ell t_h}$ that are incident to $a^*_\infty$
        AbstractRecursiveFlowHelper($G_1, \{s_1, s_2, \ldots, s_\ell\}, \{s_h, t_h, \ldots, s_m, t_m\}, \boldsymbol{c}'$)
    Else:
        Let $j$ be the greatest index $< \ell$ s.t. $s_j \in G_1$.
        AbstractRecursiveFlowHelper($G_1$, $\{s_1, s_2, \ldots, s_j\}$, $\{s_{\ell+1}, t_{\ell+1}, \ldots, s_m, t_m\}$, $\boldsymbol{c}'$)
```

Lemma 3. AbstractRecursiveFlow *implements* AbstractFlow*.*

Proof. Refer to Figure 3. By Lemmas 1 and 2, the deleted edges are *safe* to remove: solving the problem in the two subproblems will indeed find an optimal solution. The $s_\ell t_\ell$ augmentation performed by AbstractRecursiveFlowHelper corresponds to an iteration of AbstractFlow. If there are residual source-to-t_ℓ paths remaining after this augmentation, then there would necessarily be one such path from s_1, and $t_\ell \notin G_2$. AbstractRecursiveFlowHelper would continue to push flow from earlier sources to t_ℓ, just as AbstractFlow. Otherwise, both Abstract- and AbstractRecursive-Flow would move onto the next sink, in which case $t_\ell \in G_2$. □

3.3 Reusing Queues for an Efficient Implementation

We show how to implement AbstractRecursiveFlow using $O(n)$ queries to simple data structures: the priority queue and DSOrder data structure (which is at heart a linked list). The challenge in doing so can be illustrated by a simple example. Suppose s_1 has a high-capacity path P with many edges ending with a low-capacity star that connects to each of the sinks. In each iteration of the outer loop, we could require augmenting the flow along this long path. We overcome this barrier by reusing the work from earlier iterations in later iterations.

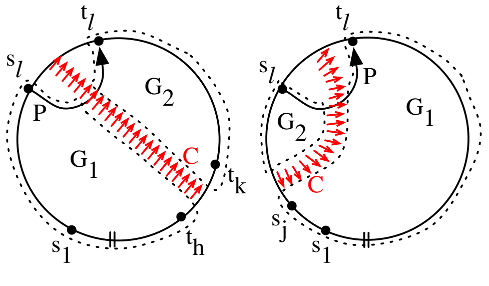


Fig. 3. The two cases for subproblems for (Abstract)RecursiveFlow. If $t_\ell \in G_2$ (left), there are 2 non-trivial subproblems.

We give an implementation (RecursiveFlow) of AbstractRecursiveFlowHelper. To implement the first step of AbstractRecursiveFlowHelper, we use the biased-search algorithm described in Section 2.1. Note that the subproblem corresponding to terminal sets $\{s_1, s_2, \ldots, s_{\ell-1}\}, \{s_\ell, t_\ell, s_{\ell+1}, t_{\ell+1}, \ldots, s_m, t_m\}$ results from having found maximum flows from $s_1, s_2, \ldots, s_{\ell-1}$ to t_ℓ. We keep the queue and order at the end of the biased-search algorithm used to find these flows.

Formally, we will pass to RecursiveFlow a queue and order for each source s_i, $i \leq \ell$. The queue Q_i and order A_i contains all the faces adjacent to and right of $\partial G[s_i, s_{i+1}]$ for $i < \ell$ and $\partial G[s_i, t_i]$ for $i = \ell$. The order reflects the c.c.w. ordering of the faces along ∂G. The priority of a face f in Q_i is the current residual capacity of the primal copy of the dart $f^*_\infty f^*$. Recall from Section 2.1 that the biased-search algorithm guarantees this at the end of the search.

```
RecursiveFlow($G$, $\{s_1, s_2, \ldots, s_{\ell-1}\}$, $\{s_\ell, t_\ell, s_{\ell+1}, t_{\ell+1}, \ldots, s_m, t_m\}$, $\{(Q_1, A_1), \ldots, (Q_\ell, A_\ell)\}$)
1   Find the leftmost $s_\ell t_\ell$-flow $\boldsymbol{f}$ via biased-search using $Q_\ell, A_\ell$ as the starting queue, order.
2   Let $P$ be the rightmost path in $\boldsymbol{f}$ and let $T^*$ be the search tree.
3   Let $Q, A$ be the queue and order at the end of this search.
4   Subtract $|\boldsymbol{f}|$ from the priorities in $Q$.
5   Delete everything to the left of $P$ in $G$.
6   Delete from $G$ the darts in $T^*$ that are left of $P$ creating components
            $G_1$ (that contains $s_1$) and $G_2$.
7   If $t_\ell \in G_2$:
8       Initialize the queue $Q_{\ell+1}$ and ordering $A_{\ell+1}$ of the dual vertices
                adjacent to $a^*_\infty$ in $G^*_{s_{\ell+1} t_{\ell+1}}$
9       Let $k$ be the greatest index s.t. $t_k \in G_2$.
10      RecursiveFlow($G_2$, $\{\}$, $\{s_{\ell+1}, t_{\ell+1}, \ldots, s_k, t_k\}$, $\{(Q_{\ell+1}, A_{\ell+1})\}$)
11      Let $h$ be the smallest index $> \ell$ s.t. $t_h \in G_1$.
12      Extend $Q$ and $A$ to contain all the faces in $G^*_{1, s_\ell t_h}$ that are incident to $a^*_\infty$
            not currently in $Q/A$ with the appropriate priority/order.
13      RecursiveFlow($G_1$, $\{s_1, s_2, \ldots, s_\ell\}$, $\{s_h, t_h, \ldots, s_m, t_m\}$, $\{(Q_1, A_1), \ldots, (Q_\ell, A_\ell), (Q, A)\}$)
14  Else:
15      Let $j$ be the greatest index $< \ell$ s.t. $s_j \in G_1$.
16      Extend $Q_j$ to $Q$ and $A_j$ to $A$, adding the missing faces in $G^*_{1, s_j t_{\ell+1}}$ that are incident to $a^*_\infty$.
17      RecursiveFlow($G_1$, $\{s_1, s_2, \ldots, s_j\}$, $\{s_{\ell+1}, t_{\ell+1}, \ldots, s_m, t_m\}$, $\{(Q_1, A_1), \ldots, (Q_{j-1}, A_{j-1}), (Q, A)\}$)
```

Running time and correctness of RecursiveFlow. By Observation 4, Step 4 results in the priorities reflecting exactly the residual capacities of the darts in P after saturating $\boldsymbol{f}$. G_1 and G_2 are the same as the subgraphs created in AbstractRecursiveFlow, as are the subproblems considered. The removed darts create a new boundary and so maintain triangulation of the finite faces. Step 12 can be done in $O(\log n)$ per new face added (Appendices A and B). Adding the faces can be achieved by a left-first search from Q (or from Q_j to Q); this creates the queue and order along the boundary of the graph. In order to combine the orders A_j and A in line 16, we observe that the order A_j is guaranteed to be right of the order A when they are joined together. The DSOrder data structure allows us to concatenate these orders efficiently (details in Appendix B).

Finally, we argue that the entire algorithm requires only $O(n)$ queries to priority queue and DSOrder data structure. The biased-search algorithm uses $O(k)$ priority-queue and DSOrder queries where k is the size of the search tree discovered (Section 2.1). This is in part due to the triangulation of the finite faces; the degree of the vertices from which we search during the biased-search algorithm have degree 3, so the 0-length darts leaving a vertex can be determined in constant time.

For the subproblem G_1, we start with queues that have already been initialized, so, as argued at the end of Section 2.1, we essentially pick up the search where we left off, not repeating any computation at the boundary where we left off (the rightmost path in a previous flow). For the subproblem G_2, P forms part of the boundary and so part of the queue/order ending at t_ℓ appear in this subgraph. However, by Lemma 1, no residual path intersects P. Since the finite faces are triangulated, no path can intersect a face adjacent to P without intersecting P. Therefore, none of the faces in the queue/order along P will be used in the subproblem corresponding to G_2. It follows that there are a constant number of data-structure queries per finite face of the original graph.

Acknowledgements. This material is based upon work supported by the National Science Foundation under Grant No. CCF-0963921.

References

1. Bender, M.A., Cole, R., Demaine, E.D., Farach-Colton, M., Zito, J.: Two Simplified Algorithms for Maintaining Order in a List. In: Möhring, R.H., Raman, R. (eds.) ESA 2002. LNCS, vol. 2461, pp. 152–164. Springer, Heidelberg (2002)
2. Borradaile, G., Harutyunyan, A.: Maximum st-flow in directed planar graphs via shortest paths. In: Lecroq, T., Mouchard, L. (eds.) IWOCA 2013. LNCS, vol. 8288, pp. 423–427. Springer, Heidelberg (2013)
3. Borradaile, G., Klein, P.: An $O(n \log n)$ algorithm for maximum st-flow in a directed planar graph. J. of the ACM 56(2), 1–30 (2009)
4. Borradaile, G., Klein, P., Mozes, S., Nussbaum, Y., Wulff-Nilsen, C.: Multiple-Source Multiple-Sink Maximum Flow in Directed Planar Graphs in Near-Linear Time. In: Proc. FOCS, pp. 170–179 (2011)
5. Borradaile, G.: Exploiting Planarity for Network Flow and Connectivity Problems. PhD thesis, Brown University (2008)
6. Dietz, P., Sleator, D.: Two algorithms for maintaining order in a list. In: Proc. STOC, pp. 365–372 (1987)
7. Ford, C., Fulkerson, D.: Maximal flow through a network. Canadian J. Math. 8, 399–404 (1956)
8. Goldberg, A., Harrelson, C.: Computing the shortest path: A search meets graph theory. In: Proc. SODA, pp. 156–165 (2005)
9. Hassin, R.: Maximum flow in (s, t) planar networks. IPL 13, 107 (1981)
10. Henzinger, M., Klein, P., Rao, S., Subramanian, S.: Faster shortest-path algorithms for planar graphs. JCSS 55(1), 3–23 (1997)
11. Khuller, S., Naor, J., Klein, P.: The lattice structure of flow in planar graphs. SIAM J. on Disc. Math. 6(3), 477–490 (1993)
12. Miller, G., Naor, J.: Flow in planar graphs with multiple sources and sinks. SIAM J. on Comp. 24(5), 1002–1017 (1995)
13. Whitney, H.: Planar Graphs. Fundamenta Mathematicae 21, 73–84 (1933)

A Priority Queues with Offsets

We show how to efficiently change all the priorities in a queue by a fixed amount. This will be used when we wish to merge two priority queues whose relative priorities differ by a constant. That is, we have two priority queues P and Q

that we want to merge, but the priorities of the items in P are *offset* from those in Q by some amount o. We illustrate this for a binomial-heap implementation of priority queues, but this technique is not limited to a specific implementation (although the details of handling the offsets will depend on the implementation).

For the purposes of this discussion the details of a binomial heap, beyond the fact that it is a set of rooted trees, are irrelevant. We refer the reader to any data structures textbook for details. We will argue that the standard operations (insert, find minimum, delete minimum, decrease key and merge) will have the same asymptotic running time with offsets as without. To do so, we annotate the edges of the trees in the heap with weights, initially zero. We give the roots of the trees a dummy parent edge so that every item in the queue (node in a tree) x has a unique parental edge weight $w(x)$. We say that node x has a local priority $p_\ell(x)$ and a global priority $p(x)$ where $p(x)$ is the sum of $p_\ell(x)$ plus the parental edge weights on the path to the root of the binomial tree containing x (including the weight of the dummy root edge). Initially the global priorities are the same as the local priorities. We will maintain that the heap property holds for the global priorities (ie. my children's global priorities are lower than mine).

We describe the modifications we make to the binomial-heap-based priority queue operations:

insert. Unchanged as insert reduces to merge.

find min. The minimum priority element is guaranteed to be a root of one of the trees. When comparing the roots of the trees, first sum the local priority and dummy root edge weight.

delete min. The standard operation is to delete the root that is the minimum priority element and then merge the resulting child trees with the remaining trees. We first add the weight of the dummy root edge to the weights of the child edges; these child edges become dummy parent edges of the trees before they are merged.

decrease key. The standard operation traverses the path from the node in question, x to the root and swaps nodes that violate the heap property. First compute the global priorities of the nodes on the x to root path. Then traverse to the to-root path: say x is a child of y such that $p(x) < p(y)$; let w be the weight of the edge xy. Swap x and y, add w to $p_\ell(x)$ and subtract w from $p_\ell(y)$.

merge. If we want to merge heap P with heap Q in such a way that the priorities in P are by an offset o higher than those in Q, o is added to the weight of the dummy root edges of P and in comparing the priorities of the roots of trees in P to those in Q, the global priorities are used. Merging binomial heaps is otherwise trivial.

We note that our modifications to not increase the asymptotic complexity of the operations. Although we do not need to maintain local priorities for our algorithm, we point out that local priorities can be retained. However, in the decrease-key operation, the weight of sibling edges would need to be modified as well, and, for binomial heaps, would require $O(\log^2 n)$ time.

B Maintaining Order

In order to maintain the left-to-right order of faces in the priority queue we refer to an order maintenance data structure DSORDER due to Dietz and Sleator [6]. DSORDER supports the following operations:

1. *Insert*$(X;Y)$: Insert a new element Y immediately after element X in the total order.
2. *Delete*(X): Remove an element X from the total order.
3. *Order*$(X;Y)$: Determine whether X precedes Y in the total order

While there are other data structures that are more efficient asymptotically [1], DSORDER is attractive for its simplicity, as it only relies on basic data structures. DSORDER is implemented as a circularly linked list that implicitly encodes the label bits to represent paths in a hypothetical $2-4$ tree and uses 2's complement arithmetic and a wrapping modulo to efficiently perform renumbering, giving:

Theorem 2. *[6] The amortized time to do* Insert *on a list containing n records is $O(\log n)$, and the amortized (and worst-case) time to do* Delete *or* Order *is $O(1)$.*

DSORDER generally draws its labels from integers in $\{0, \ldots, M-1\}$, where M is sufficiently large[3]. Since in our algorithm every face in a newly created order is *right of* the faces in the previous order, we modify this range as we move left-to-right to make simple concatenation possible. I.e. if n_i is the largest label in the order A_i, the labels for A_{i+1} are drawn from $\{n_i+1, \ldots, n_i+M\}$, where M is large w.r.t. the size of the graph. Then, an order B created after an order A, can be appended to A in constant time via standard linked list operations.

[3] $M > n^2$, where n is the size of the order.

Exact Algorithms for Weak Roman Domination*

Mathieu Chapelle[1], Manfred Cochefert[2], Jean-François Couturier[2], Dieter Kratsch[2], Mathieu Liedloff[3], and Anthony Perez[3]

[1] LIGM, Université Paris-Est Marne-La-Vallée
77454 Marne-La-Vallée Cedex 2, France
`mathieu.chapelle@univ-mlv.fr`
[2] LITA, Université de Lorraine
57045 Metz Cedex 01, France
`{couturier,cochefert,kratsch}@univ-metz.fr`
[3] LIFO, Université d'Orléans
45067 Orléans Cedex 2, France
`{mathieu.liedloff,anthony.perez}@univ-orleans.fr`

Abstract. We consider the WEAK ROMAN DOMINATION problem. Given an undirected graph $G = (V, E)$, the aim is to find a *weak roman domination* function (wrd-function for short) of minimum cost, *i.e.* a function $f : V \to \{0, 1, 2\}$ such that every vertex $v \in V$ is *defended* (*i.e.* there exists a neighbor u of v, possibly $u = v$, such that $f(u) \geqslant 1$) and for every vertex $v \in V$ with $f(v) = 0$ there exists a neighbor u of v such that $f(u) \geqslant 1$ and the function $f_{u\to v}$ defined by:

$$f_{u\to v}(x) = \begin{cases} 1 & \text{if } x = v \\ f(u) - 1 & \text{if } x = u \\ f(x) & \text{if } x \notin \{u, v\} \end{cases}$$

does not contain any undefended vertex. The *cost* of a wrd-function f is defined by $cost(f) = \sum_{v\in V} f(v)$. The trivial enumeration algorithm runs in time $\mathcal{O}^*(3^n)$ and polynomial space and is the best one known for the problem so far. We are breaking the trivial enumeration barrier by providing two faster algorithms: we first prove that the problem can be solved in $\mathcal{O}^*(2^n)$ time needing *exponential space*, and then describe an $\mathcal{O}^*(2.2279^n)$ algorithm using *polynomial space*. Our results rely on structural properties of a wrd-function, as well as on the best polynomial space algorithm for the RED-BLUE DOMINATING SET problem.

Keywords: exact algorithm, graph algorithm, roman domination.

1 Introduction

In this paper we investigate a domination-like problem from the exact exponential algorithms viewpoint. In the classical DOMINATING SET problem, one is given an undirected graph $G = (V, E)$, and asked to find a dominating set S, i.e. every vertex $v \in V$ either belongs to S or has a neighbor in S, of minimum

* This work was supported by the French AGAPE project (ANR-09-BLAN-0159).

© Springer-Verlag Berlin Heidelberg 2013

size. The Dominating Set problem ranges among one of the most famous NP-complete covering problems [8], and has received a lot of attention during the last decades. In particular, the trivial enumeration algorithm of runtime $\mathcal{O}^*(2^n)$ [1] has been improved by a sequence of papers [7,14,23]. The currently best known algorithms for the problem run in time $\mathcal{O}^*(1.4864^n)$ using polynomial space, and in time $\mathcal{O}^*(1.4689^n)$ needing exponential space [14].

Many variants of the Dominating Set problem have been introduced and studied extensively both from structural and algorithmic viewpoints. The number of papers on domination in graphs and its variants is in the thousands, and several well-known surveys and books are dedicated to the topic (see, *e.g.*, [12]). One of those variants called Roman Domination was introduced in [5] and motivated by the articles "Defend the Roman Empire!" of I. Stewart [21] and "Defendens Imperium Romanum: a classical problem in military strategy" of C.S. ReVelle and K.E. Rosing [20]. In general, the aim is to protect a set of locations (vertices of a graph) by using a smallest possible amount of legions (to be placed on those vertices). Motivated by a decree of the Emperor Constantine the Great in the fourth century A.D., Roman Domination uses the following rules for protecting a graph: a vertex can *protect* itself if it has one legion, and protect all its neighbors if it owns two legions, since Constantine decreed that two legions must be placed at a location before one may move to a nearby location (adjacent vertex) to defend it. The Roman Domination problem asks to minimize the number of legions used to defend all vertices.

Since then, numerous articles have been published around this problem, which has been studied from different viewpoints (see, *e.g.*, [1,2,4,6,17,18,24]). In particular, this NP-complete problem has been tackled using exact exponential algorithms. The first non-trivial one achieved had running time $\mathcal{O}^*(1.6183^n)$ and used polynomial space [15]. This result has recently been improved to $\mathcal{O}^*(1.5673^n)$ [22], which can be lowered to $\mathcal{O}^*(1.5014^n)$ at the cost of exponential space [22]. Moreover, the Roman Domination problem can be related to several other variants of *defense-like* domination, such as *secure domination* (see, *e.g.*, [3,4,11]), or *eternal domination* (see, *e.g.*, [9,10]).

We focus our attention on yet another variant of the Roman Domination problem. In 2003, Henning et al. [13] considered the following idea: location t can also be protected if one of its neighbors possesses one legion that can be moved to t in such a way that the whole collection of locations (set of vertices) remains protected. This variation adds some kind of dynamics to the problem and gives rise to the Weak Roman Domination problem. Formally, it can be defined as follows:

Weak Roman Domination:
Input: An undirected graph $G = (V, E)$.
Output: A weak roman domination function f of G of minimum cost.

A weak roman domination (wrd-function) is a function $f : V \to \{0, 1, 2\}$ such that every vertex $v \in V$ is *defended* (*i.e.* there exists a neighbor u of v, possibly

[1] The notation $\mathcal{O}^*(f(n))$ suppresses polynomial factors.

$u = v$, such that $f(u) \geqslant 1$) and for every vertex $v \in V$ with $f(v) = 0$ there exists a neighbor u of v such that $f(u) \geqslant 1$ and the function $f_{u \to v}$ defined by $f_{u \to v}(x) = 1$ if $x = v$, $f_{u \to v}(x) = f(x) - 1$ if $x = u$ and $f_{u \to v}(x) = f(x)$ otherwise does not contain any undefended vertex. The *cost* of a wrd-function f is defined by $cost(f) = \sum_{v \in V} f(v)$.

Our Contribution. While several structural results on Weak Roman Domination are known, see, *e.g.*, [3,4,13,19], its algorithmic aspects have not been considered so far. In this paper, we give the first algorithms tackling this problem faster than by the $\mathcal{O}^*(3^n)$ bruteforce algorithm obtained by enumerating all legion functions. Both our algorithms rely on structural properties for weak roman domination functions, described in Section 3. In Section 4, we first give an $\mathcal{O}^*(2^n)$ time and exponential space algorithm. We then show how the exponential space can be avoided by using an exponential algorithm for the Red-Blue Dominating Set problem [22], which leads to an $\mathcal{O}^*(2.2279^n)$ algorithm.

2 Preliminaries and Notations

We consider simple undirected graphs $G = (V, E)$ and assume that $n = |V|$. Given a vertex $v \in V$, we denote by $N(v)$ its *open neighborhood*, by $N[v]$ its *closed neighborhood* (*i.e.* $N[v] = N(v) \cup \{v\}$). For $X \subseteq V$, let $N[X] = \cup_{v \in X} N[v]$ and $N(X) = N[X] \setminus X$. Similarly, given $S \subseteq V$, we use $N_S(v)$ to denote the set $N(v) \cap S$. A subset of vertices $S \subseteq V$ is a *dominating set* of G if for every vertex $v \in V$ either $v \in S$ or $N_S(v) \neq \emptyset$. Furthermore, $Y \subseteq V$ dominates $X \subseteq V$ in $G = (V, E)$ if $X \subseteq N[Y]$. A subset of vertices $S' \subseteq V$ is an *independent set* in G if there is no edge in G between any pair of vertices in S'. Finally, a graph $G = (V, E)$ is *bipartite* whenever its vertex set can be partitioned into two independent sets V_1 and V_2.

Legion and wrd-Functions. A function $f : V \to \{0, 1, 2\}$ is called a *legion function.* With respect to f, a vertex $v \in V$ is said to be *secured* if $f(v) \geqslant 1$, and *unsecured* otherwise. Similarly, a vertex $v \in V$ is said to be *defended* if there exists $u \in N[v]$ such that $f(u) \geqslant 1$. Otherwise, v is said to be *undefended.* The function f is a *weak roman domination function* (wrd-function for short) if there is no undefended vertex with respect to f, and for every vertex $v \in V$ with $f(v) = 0$ there exists a secured vertex $u \in N(v)$ such that the function $f' : V \to \{0, 1, 2\}$ defined by:

$$f'(x) = \begin{cases} 1 & \text{if } x = v \\ f(u) - 1 & \text{if } x = u \\ f(x) & \text{if } x \notin \{u, v\} \end{cases}$$

has no undefended vertex (see Figure 1 (*a*)). In the following, given any legion function f and two vertices v and $u \in N(v)$ such that $f(v) = 0$ and $f(u) \geqslant 1$, we use $f_{u \to v}$ to denote the function f' as defined above. In other words, $f_{u \to v}$ denotes the legion function obtained by *moving* one legion from u to v.

Given a legion function f, we let V_f^1, V_f^2 denote the sets $\{v \in V \ : \ f(v) = 1\}$ and $\{v \in V \ : \ f(v) = 2\}$, respectively, and define its *underlying set* as $V_f = V_f^1 \cup V_f^2$. The *cost* of f is then defined by $cost(f) = \sum_{v \in V} f(v) = |V_f^1| + 2|V_f^2|$. Notice that when f is a wrd-function, the set V_f is a (not necessarily minimal) dominating set of G.

Safely-Defended Vertices. We now distinguish two types of *defended* vertices. Let $v \in V$ be any vertex and f be a legion function. We say that v is *safely defended by* f if one of the following holds:

- v is secured (i.e. $f(v) \geqslant 1$).
- there exists a neighbor u of v such that $f(u) = 2$.
- there exists a neighbor u of v such that $f(u) = 1$ and the vertices undefended by $f_{u \to v}$ are the same as the ones undefended by f, *i.e.*, $f_{u \to v}$ creates no new undefended vertex.

Otherwise, we say that v is *non-safely defended.* Notice that a legion function f is a wrd-function if and only if every vertex $v \in V$ is safely-defended by f.

Observe that for any non-safely defended vertex v, we have $f(v) = 0$, $f(u) = 1$ for every secured neighbor u of v and the legion function $f_{u \to v}$ previously defined contains (among possibly others) an undefended vertex $w \in N(u)$ for any such neighbor u. In the following, we will refer to w as *weakly defended by* u, *weakly defended due to* v, or simply *weakly defended* when the context is clear. Observe that a weakly defended vertex has exactly one secured neighbor. These notions are illustrated in Figure 1 (b).

3 Structure of a Weak Roman Domination Function

In this section, we prove several key structural properties of a wrd-function that will be used in our algorithms.

Given a graph $G = (V, E)$ and a subset of vertices $V' \subseteq V$, we define the legion function $\chi^{V'}$ as the indicator function of the subset V':

$$\chi^{V'}(x) = \begin{cases} 1 \text{ if } x \in V' \\ 0 \text{ otherwise.} \end{cases}$$

Lemma 1. *Let $G = (V, E)$ be a graph, f be a wrd-function of G of minimum cost, and V_f its underlying set. Then V_f^2 is a* minimum dominating set *of the vertices non-safely defended by χ^{V_f}.*

Proof. Let $u \in V \setminus V_f$ be a vertex non-safely defended by χ^{V_f}. Recall that u is non-safely defended by χ^{V_f} if for every $u' \in N_{V_f}(u)$ the legion function $\chi^{V_f}_{u' \to u}$ contains an undefended vertex. Hence, for every vertex $u' \in N_{V_f}(u)$, there exists a vertex u'' weakly defended due to u. In particular, this means that $u'u'' \in E$ and $uu'' \notin E$. We prove Lemma 1 through the following claims.

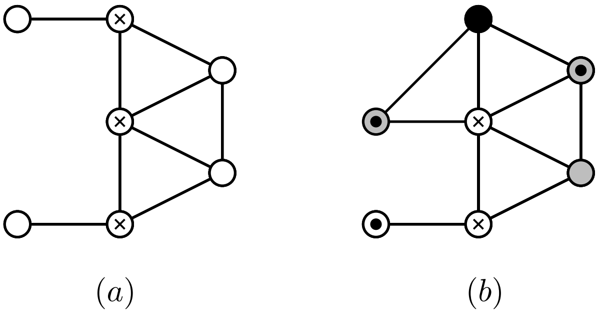


Fig. 1. (a) A graph $G = (V, E)$, and a wrd-function where each legion is represented by a cross. Any vertex is safely defended. (b) The black vertex is safely defended (one can safely move a legion on it without creating any undefended vertex), the gray vertices are non-safely defended (any move creates an undefended vertex) and the disked vertices are weakly defended.

Claim 1. *V_f^2 is a dominating set of the vertices non-safely defended by χ^{V_f}.*

Proof. Assume for a contradiction that there exists a vertex $u \in V \setminus V_f$ non-safely defended by χ^{V_f} such that $N_{V_f^2}(u) = \emptyset$. Let u'' be any vertex weakly defended due to u, and let u' be the common neighbor of u and u'' in V_f. Recall that $N(u'') \cap V_f = \{u'\}$, since otherwise u'' would be defended by $\chi^{V_f}_{u' \to u}$. Moreover, we know by assumption that $f(u') = 1$. Hence, the vertex u'' is undefended by $\chi^{V_f}_{u' \to u}$, which contradicts the fact that f is a wrd-function. $\diamond$

Claim 2. *V_f^2 is a minimal dominating set of the vertices non-safely defended by χ^{V_f}.*

Proof. Assume for a contradiction that there exists $u \in V_f^2$ such that $V_f^2 \setminus \{u\}$ is a dominating set of the vertices non-safely defended by χ^{V_f}. We claim that the legion function f_u defined as:

$$f_u(x) = \begin{cases} 1 \text{ if } x = u \\ f(v) \text{ otherwise} \end{cases}$$

is a wrd-function. To see this, observe that since $V_f^2 \setminus \{u\}$ is a dominating set of the vertices non-safely defended by χ^{V_f}, any vertex of $N_{V \setminus V_f}(u)$ is safely defended by f_u. It follows that f_u is a wrd-function with $cost(f_u) < cost(f)$, a contradiction. $\diamond$

Now, since f is a wrd-function of *minimum cost*, it follows from Claims 1 and 2 that V_f^2 is a *minimum* dominating set of the vertices non-safely defended by χ^{V_f}. This completes the proof of Lemma 1. $\square$

We conclude this section by showing that, given a dominating set V' of a graph $G = (V, E)$, a wrd-function can be obtained by computing a dominating set of the set $\overline{D}$ of all vertices non-safely defended by $\chi^{V'}$.

Lemma 2. *Let $V' \subseteq V$ be a dominating set of a graph $G = (V, E)$, and let S be a dominating set of all vertices $\overline{D}$ non-safely defended by $\chi^{V'}$. Then the function $f : V \to \{0, 1, 2\}$ defined by*

$$f(x) = \begin{cases} 2 \text{ if } x \in (V' \cap S) \\ 1 \text{ if } x \in (V' \cup S) \setminus (V' \cap S) \\ 0 \text{ otherwise} \end{cases}$$

is a wrd-function.

Proof. Let S be a dominating set of $\overline{D}$ in G. Observe first that since $V_f = V' \cup S$, and since V' is a dominating set, then so is V_f. We now show that the set $\overline{D}'$ of vertices non-safely defended by f is empty. Observe that since $V' \subseteq V_f$, we have $\overline{D}' \subseteq \overline{D} \setminus S$. Assume for a contradiction that $\overline{D}' \neq \emptyset$, and let $x \in \overline{D}'$. We distinguish two cases:

(i) If $N(x) \cap (V' \cap S) \neq \emptyset$ then x has a neighbor of f-value 2, and thus x is safely-defended, contradicting the choice of x.

(ii) Otherwise, by definition of V' and S, x has a neighbor y in S which does not belong to V'. We claim that the legion function $f_{y \to x}$ cannot contain any undefended vertex. Indeed, since y does not belong to the original dominating set V', all vertices are defended by V' in $f_{y \to x}$ (recall that any vertex v of V' satisfies $f(v) \geqslant 1$).

These two cases imply that $\overline{D}'$ is empty, and thus f is a wrd-function. □

4 Exact Algorithms for Weak Roman Domination

We now describe our exact algorithms solving the Weak Roman Domination problem. Observe that this problem can trivially be solved in $\mathcal{O}^*(3^n)$ time by enumerating all three-partitions of the set of vertices, which constitutes the best known bound for the problem so far. We first present an $\mathcal{O}^*(2^n)$ time and space algorithm, then an $\mathcal{O}^*(2.2279^n)$ time algorithm that only uses polynomial space.

4.1 Using Exponential Space

We first show that a wrd-function of minimum cost can be computed in $\mathcal{O}^*(2^n)$ time and space. Thanks to Lemma 1, a wrd-function f of minimum cost can be obtained by first guessing its underlying set V_f and then computing a minimum dominating set $V_f^2 \subseteq V_f$ of the vertices non-safely defended by χ^{V_f}. Finding such a set V_f^2 is done by a preprocessing step which involves a dynamic programming

Algorithm 1. The preprocessing step algorithm.

```
for $k = 0$ to $n$ do
    $\mathsf{DS}[\emptyset, k] = \emptyset$;
foreach $X \subseteq V$ s.t. $|X| \geq 1$ do
    $\mathsf{DS}[X, 0] = \{\infty\}$;
    // The set $\{\infty\}$ is a sentinel used to denote the non existence of
       a set $Y_k$ which dominates a nonempty set $X$; its cardinality is
       set to $\infty$.
foreach $X \subseteq V$ by increasing order of cardinality do
    for $k = 1$ to $n$ do
        $\mathsf{DS}[X, k] = \left\{ \begin{array}{l} \text{a set of minimum cardinality chosen amongst} \\ \mathsf{DS}[X, k-1] \text{ and } \{v_k\} \cup \mathsf{DS}[X \setminus N[v_k], k-1]. \end{array} \right\}$
```

inspired by the one given in [16]. This preprocessing step results in an exponential space complexity, which will be reduced to polynomial space in Section 4.2. However, instead of guaranteeing that indeed $V_f^2 \subseteq V_f$, the preprocessing step computes a minimum dominating set V_f^2 of the vertices non-safely defended by χ^{V_f} without constraint, *i.e.* $V_f^2 \subseteq V$ is allowed. We show in Lemma 3 the correctness of this approach. Let us first describe the preprocessing step; its correctness is shown after the description of the main algorithm.

Let $G = (V, E)$ be a graph of the Weak Roman Domination problem, and let $V = \{v_1, v_2, \ldots, v_n\}$. For each subset $X \subseteq V$ we start by computing a minimum dominating set Y of X in G, i.e. a subset $Y \subseteq V$ such that $X \subseteq N[Y]$. This is done by dynamic programming: for each subset X and each integer k ($1 \leqslant k \leqslant n$), $\mathsf{DS}[X, k]$ denotes a minimum dominating set Y_k of X such that $Y_k \subseteq \{v_1, v_2, \ldots, v_k\}$, if one exists. Algorithm 1 computes a corresponding table DS by dynamic programming.

Main Algorithm. The main steps of our exact algorithm are depicted in Algorithm 2. For each subset $V' \subseteq V$, we first verify whether $\chi^{V'}$ is (already) a wrd-function, *i.e.*, whether the set $\overline{D}$ of vertices non-safely defended by $\chi^{V'}$ is empty. Otherwise, we need to compute the set V_f^2. The preprocessing step then ensures that $S = \mathsf{DS}[\overline{D}, n]$ is a minimum dominating set of $\overline{D}$. If S is a subset of V', then a wrd-function f can be computed by Lemma 2; otherwise Lemma 3 ensures that there exists some other underlying set V'', being better than V'.

Lemma 3. *Let $V_1' \subseteq V$ be a dominating set of a graph $G = (V, E)$ and let S_1 be a minimum dominating set of the set $\overline{D_1}$ of all vertices non-safely defended by $\chi^{V_1'}$. Suppose that $S_1 \nsubseteq V_1'$. Then there exists a superset $V_2' \supset V_1'$ such that for any minimum dominating set S_2 of the set $\overline{D_2}$ of all vertices non-safely*

Algorithm 2. An $\mathcal{O}^*(2^n)$ exponential space algorithm for Weak Roman Domination.

```
foreach dominating set $V' \subseteq V$ do
    foreach $v \in V$ do
        Let $f(v) = 1$ if $v \in V'$, and $f(v) = 0$ otherwise;
    Compute the set $\overline{D}$ of vertices non-safely defended by $\chi^{V'}$;
    if $\overline{D} \neq \emptyset$ then
        $S = \mathsf{DS}[\overline{D}, n]$;
        if $S \subseteq V'$ then
            foreach $v \in S$ do
                Let $f(v) = f(v) + 1$;
return the computed wrd-function $f$ of minimum cost;
```

defended by $\chi^{V_2'}$*, it holds that cost*$(f_2) \leq$ *cost*(f_1)*, where* f_i *(*$i \in \{1, 2\}$*) is the legion function defined as:*

$$f_i(x) = \begin{cases} 2 \text{ if } x \in (V_i' \cap S_i) \\ 1 \text{ if } x \in (V_i' \cup S_i) \setminus (V_i' \cap S_i) \\ 0 \text{ otherwise} \end{cases}$$

Proof. Assume that there exist three sets V_1', S_1 and $\overline{D_1}$ as stated in the lemma and assume that $S_1 \nsubseteq V_1'$. Let $V_2' = V_1' \cup S_1$. Since $S_1 \nsubseteq V_1'$, it follows that $V_2' \supset V_1'$. Let $\overline{D_2}$ be the set of vertices non-safely defended by $\chi^{V_2'}$. Observe that $\overline{D_2} \subseteq \overline{D_1}$, since $V_1' \subset V_2'$. By Lemma 2, we know that the legion function f_1 is in fact a wrd-function. Hence, by Lemma 1, we also have that $(V_1' \cap S_1)$ is a dominating set of $\overline{D_1}$, and thus of $\overline{D_2}$.

Denote by S_2 a minimum dominating set of $\overline{D_2}$. Then $|S_2| \leqslant |V_1' \cap S_1|$. We now consider the legion function f_2 as defined in the lemma. By Lemma 2, we know that f_2 is a wrd-function. Finally, since $|V_2'| = |V_1'| + |S_1 \setminus V_1'|$ and $|S_2| \leq |V_1' \cap S_1|$, we conclude the proof by the relation *cost*$(f_1) = |V_1'| + |S_1| = |V_1'| + |S_1 \setminus V_1'| + |S_1 \cap V_1'| \geq |V_2'| + |S_2| =$ *cost*(f_2). □

Correctness. The correctness of the preprocessing step is based on arguments of [16]. If the set X is empty then the initialization $\mathsf{DS}[\emptyset, k] = \emptyset$, for any $0 \leq k \leq n$, is clearly correct. If the set X is non empty but no vertex can be used to dominate X (i.e. $k = 0$), then $\mathsf{DS}[X, 0]$ is set to $\{\infty\}$ as a sentinel, meaning that there is no set Y (with $Y = \emptyset$) that can dominate X. The cardinality of $\{\infty\}$ is set to ∞. Finally the computation of $\mathsf{DS}[X, k]$ is done via an induction formula: either $v_k \notin \mathsf{DS}[X, k]$ or $v_k \in \mathsf{DS}[X, k]$ and in that latter case, $N(v_k)$ is dominated by v_k. As the sets X are considered by increasing order as well as the values of k, we note that the values $\mathsf{DS}[X, k-1]$ and $\mathsf{DS}[X \setminus N[v_k], k-1]$ have already been computed when the computation of $\mathsf{DS}[X, k]$ is done.

Now we show the correctness of Algorithm 2. It enumerates all possible sets V' as being possible candidates for the underlying set V_f. In particular, we discard any subset V' that does not induce a dominating set. By Lemma 1, it is sufficient to compute a dominating set $S \subseteq V'$ of the set of vertices $\overline{D}$ being non-safely defended by $\chi^{V'}$. Lemma 3 shows that if S is not included in V', then there exists a proper superset of V' which gives a wrd-function of cost being no more than the one obtained from V' and S, by Lemma 2. Let $V'_0 = V'$ and $S_0 = S$. As the graph is finite and the superset given by Lemma 3 is proper, there exists a finite $\ell \leq n$ and a sequence $V'_0 \subset V'_1 \subset ... \subset V'_\ell \subseteq V$ such that $S_i \nsubseteq V'_i$, for all $0 \leq i < \ell$, and $S_\ell \subseteq V'_\ell$. Since the algorithm enumerates all supersets of V', it follows that the set V'_ℓ will be considered at some iteration of the for-loop. This shows the correctness of Algorithm 2.

Complexity. The preprocessing step needs to consider each subset X of V and each value of k, $1 \leq k \leq n$. For each such couple (X, k), it retrieves the values of $\mathsf{DS}[X, k-1]$ and $\mathsf{DS}[X \setminus N[v_k], k-1]$ previously computed, and stores the new value in DS. Thus the preprocessing step requires $\mathcal{O}^*(2^n)$ time and space. The main part of the algorithm considers each (dominating set) $V' \subseteq V$, and computes in polynomial-time the set $\overline{D}$ of vertices non-safely defended by $\chi^{V'}$. A dominating set S of $\overline{D}$ is then retrieved in the already computed table DS in polynomial-time.

Theorem 3. Weak Roman Domination *can be solved in* $\mathcal{O}^*(2^n)$ *time and space.*

4.2 Using Polynomial Space

In order to obtain an exact exponential algorithm using only polynomial space, we need to avoid any exponential space consuming *preprocessing step* such as the one in the previous section. For this purpose, we use instead an exact exponential algorithm for Red-Blue Dominating Set using polynomial space to decide which vertices will be valued 2 to dominate the non-safely defended vertices.

Red-Blue Dominating Set:
Input: A bipartite graph $G = (R \cup B, E)$.
Output: A subset $S \subseteq R$ of minimum size dominating B.

Theorem 4 ([22]). *The* Red-Blue Dominating Set *problem can be solved in* $\mathcal{O}^*(1.2279^{|R|+|B|})$ *time and polynomial space.*

Algorithm. We consider the algorithm depicted in Algorithm 3, which might be seen as some modification of the previous Algorithm 2.

Observe that before computing a minimum red-blue dominating set on the bipartite graph $(\overline{C} \cup \overline{D}, E)$, we may modify the sets $\overline{C}$ and $\overline{D}$ as follows: for every vertex $v \in \overline{C}$, if v has at least two weakly non-safely defended neighbors, then we set $f(v) = 2$, and remove v from $\overline{C}$ and $N_{\overline{D}}(v)$ from $\overline{D}$.

Algorithm 3. An $\mathcal{O}^*(2.2279^n)$ poly-space algorithm for the WEAK ROMAN DOMINATION problem.

```
foreach dominating set $V' \subseteq V$ do
    foreach $v \in V$ do
        Let $f(v) = 1$ if $v \in V'$, and $f(v) = 0$ otherwise;
    Compute the set $\overline{D}$ of vertices non-safely defended by $\chi^{V'}$;
    if $\overline{D} \neq \emptyset$ then
        Compute the set $\overline{C} \subseteq V'$ of secured vertices which have at least one
        neighbor in $\overline{D}$;
        /* Cleaning step */
        foreach $v \in \overline{C}$ with at least two weakly non-safely defended neighbors in
        $\overline{D}$ do
            Set $f(v) = 2$;
            Remove $N_{\overline{D}}(v)$ from $\overline{D}$;
            Remove $v$ from $\overline{C}$;
        Let $I = (\overline{C} \cup \overline{D}, E)$ be an instance of RED-BLUE DOMINATING SET;
        if $I$ admits a minimum red-blue dominating set $S \subseteq \overline{C}$ then
            Set $f(v) = 2$ for every $v \in S$;
        else
            The current function $f$ cannot yield a wrd-function;
return the computed wrd-function $f$ of minimum cost;
```

Proposition 1. *The cleaning step on $\overline{C}$ and $\overline{D}$ does not modify a solution for* RED-BLUE DOMINATING SET *on instance $I = (\overline{C} \cup \overline{D}, E)$.*

Proof. Let $v \in \overline{C}$ be a secured vertex with at least two weakly non-safely defended neighbors, say w_1 and w_2. Since w_1 and w_2 are weakly defended, their only secured neighbor is $v \in V'$; since they are non-safely defended, they need to be dominated by V_f^2 in order for f to be a wrd-function (Lemma 1). Thus we must set $f(v) = 2$. It follows that any minimum red-blue dominating set on instance $I = (\overline{C} \cup \overline{D}, E)$ must put $v \in \overline{C}$ into the red-blue dominating set in order to dominate all weakly non-safely defended neighbors of v in $\overline{D}$.

Now, observe that since all the neighbors of v are safely defended (because dominated by V_f^2), they can safely be removed from $\overline{D}$. Since v has no non-safely defended neighbor left, it can be removed from $\overline{C}$. □

Correctness. The correctness of the algorithm follows from Lemma 1 and the proof of correctness of Algorithm 2. The main difference lies in the computation of the dominating set of the vertices non-safely defended by $\chi^{V'}$. Indeed, in that case, we use Theorem 4 to find the vertices of V' that must have value 2 in order to dominate the vertices non-safely defended by $\chi^{V'}$. The correctness of this step follows from Lemma 1 and Proposition 1.

Complexity. Let us now give the time and space complexities of Algorithm 3. It is easy to see that for every subset $V' \subseteq V$, the initialisation of $f(x)$ for every $x \in V$ as well as the computation of the set $\overline{D}$ can be done in polynomial time and space, and that the cleaning step is also polynomial.

Regarding the legion function f being constructed, for any $V' \subseteq V$, our algorithm computes and reduces the set $\overline{D}$ of vertices non-safely defended by $\chi^{V'}$, and the set $\overline{C}$ of secured vertices which have at least one neighbor in $\overline{D}$. Those two sets are considered as an instance of Red-Blue Dominating Set to be solved in $\mathcal{O}^*(1.2279^{|\overline{D}|+|\overline{C}|})$ time and polynomial space using an algorithm from van Rooij [22]. To conclude our analysis, we need the following result.

Proposition 2. *For any $V' \subseteq V$, $|\overline{D}| + |\overline{C}| \leq |V| - |V'|$.*

Proof. For every vertex $v \in V'$, one of the following statements holds:

(i) v has no neighbor in $\overline{D}$, that is no neighbor non-safely defended by $\chi^{V'}$;
(ii) there exists at least one vertex $w \in V \setminus V'$ which is weakly defended by v.

First notice that (*i*) and (*ii*) are the only two possible cases. Indeed, if there exists $v \in V'$ such that $N_{\overline{D}}(v) \neq \emptyset$ but no vertex in $V \setminus V'$ is weakly defended by v, then the vertices in $N_{\overline{D}}(v)$ are safely defended, which is a contradiction.

If the first statement holds, then v is not included in $\overline{C}$. If the second statement holds, then either w is safely defended, or w is non-safely defended. If w is safely defended (that is no other neighbor of v is weakly defended by v), then w is not included in $\overline{D}$. If w is non-safely defended, then v has at least two weakly non-safely defended neighbors. Indeed, since w is weakly defended by v (as the second statement holds), v is the only neighbor of w in V'. Hence, there exists a nonempty set $\overline{D}_{v,w} = N_{\overline{D}}(v) \setminus N(w)$ such that w is weakly defended due to each vertex in $\overline{D}_{v,w}$. Now, since w is non-safely defended by v, there must exist a vertex $w' \in \overline{D}_{v,w}$ which is also weakly defended due to w. Then the cleaning step on $\overline{D}$ and $\overline{C}$ applies, which implies that v is removed from $\overline{C}$ and all neighbors of v (including w) are removed from $\overline{D}$. Altogether, for every vertex $v \in V'$, at least one vertex from V is not included in $\overline{C} \cup \overline{D}$, and hence at least $|V'|$ vertices from V are not included in $\overline{D} \cup \overline{C}$. ◇

The overall algorithm iteratively runs all the previously described computations for every subset $V' \subseteq V$, and stores the minimum wrd-function considered so far using polynomial space. We claim that its worst-case time complexity corresponds to the following:

$$\mathcal{O}^*\Big(\sum_{i=1}^{n}\binom{n}{i} \cdot T(n-i)\Big) = \mathcal{O}^*\big(\sum_{i=1}^{n}\binom{n}{i} \cdot 1.2279^{n-i}\big) = \mathcal{O}^*\left(2.2279^n\right)$$

where $T(p)$ stands for the time complexity needed to compute a minimum red-blue dominating set in a graph with p vertices (here we use the one of [22]). Indeed, for any subset $V' \subseteq V$ containing i vertices, we apply Theorem 4 on the bipartite graph induced by $\overline{C}$ and $\overline{D}$, which contain less than $|V| - |V'| = n - i$ vertices (Proposition 2).

Theorem 5. Weak Roman Domination *can be solved in* $\mathcal{O}^*(2.2279^n)$ *time and polynomial space.*

References

1. Chambers, E.W., Kinnersley, B., Prince, N., West, D.B.: Extremal problems for roman domination. SIAM J. Discret. Math. 23(3), 1575–1586 (2009)
2. Chellali, M., Rad, N.J., Volkmann, L.: Some results on roman domination edge critical graphs. AKCE Int. J. Graphs Comb. 9(2), 195–203 (2012)
3. Cockayne, E.J., Favaron, O., Mynhardt, C.M.: Secure domination, weak roman domination and forbidden subgraphs. Bull. Inst. Combin. Appl. 39, 87–100 (2003)
4. Cockayne, E.J., Grobler, P.J.P., Gründlingh, W.R., Munganga, J., van Vuuren, J.H.: Protection of a graph. Util. Math. 67, 19–32 (2005)
5. Cockayne, E.J., Dreyer Jr., P.A., Hedetniemi, S.M., Hedetniemi, S.T.: Roman domination in graphs. Discret. Math. 278(1-3), 11–22 (2004)
6. Favaron, O., Karami, K., Khoeilar, R., Sheikholeslami, S.M.: On the roman domination number of a graph. Discret. Math. 309, 3447–3451 (2009)
7. Fomin, F.V., Grandoni, F., Kratsch, D.: A measure & conquer approach for the analysis of exact algorithms. J. ACM 56(5) (2009)
8. Garey, M.R., Johnson, D.S.: Computers and intractability. Freeman (1979)
9. Goddard, W., Hedetniemi, S.M., Hedetniemi, S.T.: Eternal security in graphs. J. Combin. Math. Combin. Comput. 52, 160–180 (2005)
10. Goldwasser, J.L., Klostermeyer, W.F.: Tight bounds for eternal dominating sets in graphs. Discret. Math. 308, 2589–2593 (2008)
11. Grobler, P.J.P., Mynhardt, C.M.: Secure domination critical graphs. Discret. Math. 309, 5820–5827 (2009)
12. Haynes, T.W., Hedetniemi, S.T., Slater, P.J.: Domination in graphs: advanced topics. Pure and Applied Mathematics, vol. 209. Marcel Dekker Inc. (1998)
13. Henning, M.A., Hedetniemi, S.T.: Defending the Roman Empire: a new strategy. Discret. Math. 266(1-3), 239–251 (2003)
14. Iwata, Y.: A faster algorithm for dominating set analyzed by the potential method. In: Marx, D., Rossmanith, P. (eds.) IPEC 2011. LNCS, vol. 7112, pp. 41–54. Springer, Heidelberg (2012)
15. Liedloff, M.: Algorithmes exacts et exponentiels pour les problèmes NP-difficiles: domination, variantes et généralisations. Phd thesis, Laboratoire d'Informatique Théorique et Appliquée, Université Paul Verlaine, Metz (2007)
16. Liedloff, M.: Finding a dominating set on bipartite graphs. Inf. Proc. Lett. 107(5), 154–157 (2008)
17. Liedloff, M., Kloks, T., Liu, J., Peng, S.-L.: Efficient algorithms for roman domination on some classes of graphs. Discret. App. Math. 156, 3400–3415 (2008)
18. Liu, C.-H., Chang, G.J.: Roman domination on 2-connected graphs. SIAM J. Discret. Math. 26(1), 193–205 (2012)

19. Malini Mai, T.N.M., Roushini Leely Pushpam, P.: Weak roman domination in graphs. Discussiones Mathematicae Graph Theory 31(1), 161–170 (2011)
20. ReVelle, C.S., Rosing, K.E.: Defendens Imperium Romanum: a classical problem in military strategy. Math. Assoc. of America 107(7), 585–594 (2000)
21. Stewart, I.: Defend the Roman Empire!. Scientific American 281(6), 136–139 (1999)
22. van Rooij, J.M.M.: Exact exponential-time algorithms for domination problems in graphs. Phd thesis, Utrecht University, Netherlands (2011)
23. van Rooij, J.M.M., Bodlaender, H.L.: Exact algorithms for dominating set. Discret. App. Math. 159(17), 2147–2164 (2011)
24. Xing, H.-M., Chen, X., Chen, X.-G.: A note on roman domination in graphs. Discret. Math. 306, 3338–3340 (2006)

Verification Problem of Maximal Points under Uncertainty

George Charalambous and Michael Hoffmann

Department of Computer Science, University of Leicester
{gc100,mh55}@mcs.le.ac.uk

Abstract. The study of algorithms that handle imprecise input data for which precise data can be requested is an interesting area. In the *verification under uncertainty* setting, which is the focus of this paper, an algorithm is also given an assumed set of precise input data. The aim of the algorithm is to update the smallest set of input data such that if the updated input data is the same as the corresponding assumed input data, a solution can be calculated. We study this setting for the maximal point problem in two dimensions. Here there are three types of data, a set of points $P = \{p_1, \ldots, p_n\}$, the uncertainty areas information consisting of *areas of uncertainty* A_i for each $1 \leq i \leq n$, with $p_i \in A_i$, and the set of $P' = \{p'_1, ..., p'_k\}$ containing the assumed points, with $p'_i \in A_i$. An *update* of an area A_i reveals the actual location of p_i and *verifies* the assumed location if $p'_i = p_i$. The objective of an algorithm is to compute the smallest set of points with the property that, if the updates of these points verify the assumed data, the set of maximal points among P can be computed. We show that the maximal point verification problem is NP-hard, by a reduction from the minimum set cover problem.

1 Introduction

Nowadays more and more information is available. With a flood of sensors connected to a network, such as GPS-enabled mobile phones, up-to-date readings of these sensors are generally available. Algorithms that perform based on such information might not have the precise data available to them, as for example in some situations the traffic of collecting all such information would cause a problem on its own. In other situations, obtaining the up-date information of all sensors is costly in time and battery power or even other charges may occur. A practical solution is to work with slightly out of date data where possible and only request up to date information where needed. For example a sensor may automatically send its current measurement if this exceeds some predefined bounds of the latest send one. Hence based on the last send information a possible band or area is known where the current measurement of the sensor is within. If this rough information is not enough then the precise measurement can be obtained. Therefore an algorithm may have some precise data while at the same time some uncertain data, for which if needed an update request can be made and the precise measurement can be obtained. Problems under uncertainty capture this setting. The aim is to make the fewest update requests that

© Springer-Verlag Berlin Heidelberg 2013

allows the calculation to succeed. In the *verification under uncertainty* setting, which is the focus of this paper, an algorithm is also given an assumed set of precise input data. The aim of the algo'rithm is to update the smallest set of input data such that if the updated input data is the same as the corresponding assumed input data, a solution can be calculated.

While mobile devices moving in the plane is a classical example to motivate the study of geometric problems in the uncertainty setting, its applications are found in many different areas. For example data is collected in a large number of databases and distributed systems, such as prices and customer ratings of products. As the price and rating of a product may vary over time, an algorithm that has to identify all top items from a collection may work with uncertain data and only request more accurate prices and ratings if needed. With the idea of maximal points in mind, top items would be such that there is no other better in price and rating.

Work in computing under uncertainty falls in three main categories: In the *adaptive online* setting an algorithm initially knows only the uncertainty areas and performs updates one by one (determining the next update based on the information from previous updates) until it has obtained sufficient information to determine a solution. Algorithms are typically evaluated by competitive analysis, comparing the number of updates they make with the minimum number of updates that, in hindsight, would have been sufficient to determine a solution (referred to as the offline optimum). In the *non-adaptive online* setting an algorithm is also given only the uncertainty areas initially, but it must determine a set U of updates such that after performing all updates in U it is guaranteed to have sufficient information to determine a solution. Finally, there is the *verification* setting that was already described above. It is worth noting that the optimal update set of the verification setting is also the offline optimum of the adaptive online setting. Therefore, algorithms solving the verification problem are also useful for the experimental evaluation of algorithms for the adaptive online setting.

In this paper, we consider the Maximal Point Verification problem. The maximal point problem is a classical problem. Many aspects of the problem have sparked interesting research. It can be stated as follows: Given a set P of points and a partial order of the points, return all points where no point in P is higher. Typically the partial order is based on the coordinates of the points in the following way: a point p is higher than a point q if $p_x \geq q_x$ and $p_y \geq q_y$

and $p \neq q$. Such a partial order naturally extends to higher dimensions, but in this paper we only consider 2-dimensional points.

A formal definition of the Maximal Point Verification problem (MPV) is given in Definition 2.

Our main result is, as stated in Theorem 1, that by a reduction from the minimum set cover problem the MPV problem is also NP-hard. In our construction of the reduction each uncertain area contains either a single point (e.g the data is known precisely) or contains just two points. Hence, an MPV problem remains NP-hard even when restricted to areas of uncertainty that contain at most two points. It remains, however, open if the same holds when each uncertain area is connected.

The effect of our result is significant for experimental evaluation of algorithms in the online and verification setting of the maximal point problem under uncertainty. It strengthens the role of constant competitive online algorithms, as they also represent a constant approximation algorithm for the verification setting. Finally it gives rise to find new restrictions on the uncertainty areas, such that the verification problem becomes solvable in polynomial time, and captures a large variety of applications for maximal points under uncertainty.

Related Work

Kahan [6] presented a model for handling imprecise but updateable input data. He demonstrated his model on a set of real numbers where instead of the precise value of each number an interval was given. That interval when updated reveals that number. The aim is to determine the maximum, the median, or the minimal gap between any two numbers in the set, using as few updates as possible. His work included a competitive analysis for this type of online algorithm, where the number of updates is measured against the optimal number of updates. For the problems considered, he presented online algorithms with optimal competitive ratio. Feder et al. [4] studied the problem of computing the value of the median of an uncertain set of numbers up to a certain tolerance. Applications of uncertainty settings can be found in many different areas including structured data such as graphs, databases, and geometry. The work presented in this paper mainly concerns the latter two areas.

Bruce et al. [1] studied the geometric uncertainty problem in the plane. Here, the input consists of points in the plane and the uncertainty information is for each point of the input an area that contains that point. They gave a definition of the Maximal Point under Uncertainty as well as the Convex Hull under Uncertainty. They presented algorithms with optimal competitive ratio for both problems. The algorithms used are based on a more general technique called *witness set algorithm* that was introduced in their paper.

In [2], Erlebach et al. studied the adaptive online setting for minimum spanning tree (MST) under two types of uncertainty: the edge uncertainty setting, which is the same as the one considered by Feder et al. [3], and the vertex uncertainty setting. In the latter setting, all vertices are points in the plane and the graph is a complete graph with the weight of an edge being the distance between the vertices it connects. The uncertainty is given by areas for the location of each vertex. For both settings, Erlebach et al. presented algorithms with optimal competitive ratio for the MST under uncertainty. The competitive ratios are 2 for edge uncertainty and 4 for vertex uncertainty, and the uncertainty areas must satisfy certain restrictions (which are satisfied by, e.g., open and trivial areas in the edge uncertainty case). A variant of computing under uncertainty where updates yield more refined estimates instead of exact values was studied by Gupta et al. [5].

A different setting of the MST under vertex uncertainty was studied by Kamousi et al. [7]. They assume that point locations are known exactly, but each point i is present only with a certain probability p_i. They show that it is #P-hard to compute the expected length of an MST even in 2-dimensional Euclidean space, and provide a fully polynomial randomized approximation scheme for metric spaces.

Structure of the Paper. In Section 2 we give formal definitions and preliminaries. In Section 3 we present our construction of an MPV problem out of a minimum set cover problem. In Section 4 we demonstrate relation between the solutions of these two problems. In Section 5 we complete the proof of Theorem 1.

2 Preliminaries

The general setting of problems with areas of uncertainty can be described in the following way: Each problem instance $P = (C, \mathcal{A}, \phi)$ consists of following three components. The ordered set of data $C = \{c_1, \dots, c_n\}$ is also called a *configuration*. $\mathcal{A}$ is an ordered set of areas $\mathcal{A} = \{A_1, \dots, A_n\}$, such that $c_i \in C$ is an element of A_i for $1 \leq i \leq n$. The sets A_i are called *areas of uncertainty* or *uncertainty areas* for C. We say that an uncertainty area A_i that consists of a single element is *trivial*. ϕ is a function such that $\phi(C)$ is the set of solutions for P. (The function ϕ is the same for all instances of a problem and can thus be taken to represent the problem.) The aim is to calculate a solution in $\phi(C)$ based on the information of $\mathcal{A}$. If that is not possible, *updates* to elements of $\mathcal{A}$ can be made. These updates alter the set $\mathcal{A}$: After updating A_i, the new ordered set of areas of uncertainty for C is $\{A_1, \dots, A_{i-1}, \{c_i\}, A_{i+1}, \dots, A_n\}$. Hence the exact value of c_i is now revealed.

In the online setting, the set C is not known to the algorithm; the algorithm has to request updates until the set $\mathcal{A}$ is precise enough to allow the calculation of a solution in $\phi(C)$ based on $\mathcal{A}$. The *verification setting* is similar. The set C, however, is now also given to the algorithm. This additional information is not used to calculate $\phi(C)$ directly, but is used to determine which update requests should be made so that $\phi(C)$ can be calculated based on $\mathcal{A}$. Updating all non-trivial areas would reveal/verify the configuration C and would obviously allow us to calculate an element of $\phi(C)$ (under the natural assumption that ϕ is computable). A set of updates that reveal enough information of the configuration C such that an element of $\phi(C)$ can be calculated is an *update solution*. The aim of the algorithm is to use the smallest possible number of updates. For a given instance of a problem, we denote an update solution of minimal size also as an optimal update solution.

We use the uncertainty setting in the context of the Maximal Point problem. For this all points discussed in the paper are points in the 2D plane. So a point p may be written in coordinate form (p_x, p_y). We say a point $p = (p_x, p_y)$ is *higher* than a point $q = (q_x, q_y)$ if $p_x \geq q_x$ and $p_y \geq q_y$ and $p \neq q$. Note that this induces a partial order and leads to the following definition of a maximal point among a set of points.

Definition 1. *Let P be a set of points and p be a point in P. The point p is said to be maximal among P if there does not exist a point in P that is higher than p. Otherwise p is non-maximal among P.*

In the 'under Uncertainty' setting for the Maximal Point problem the set of points $P = \{p_1, \ldots, p_n\}$ is the configuration of the problem. The set of uncertainty areas consists of an area for each point in P. The solution $\phi(P)$ is the index set I such that p_i is maximal among P if and only if $i \in I$. Formally,

Definition 2. *A Maximal Point Verification problem, MPV for short, is a pair $(\mathcal{A}, P)$, where P is a set of points and $\mathcal{A}$ is a set of areas for P. The aim is to identify the smallest set of areas in $\mathcal{A}$, that when updated verifies the maximal points among P as maximal based on the information of $\mathcal{A}$ and the results of the updates.*

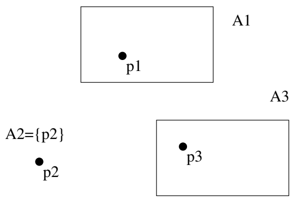


Fig. 1. Example of an MPV problem

In the example shown in figure 1, the problem consists of three points (p_1,p_2 and p_3) and three areas A_1, A_2 and A_3. The area A_2 consist only of the point p_2 and hence A_2 is a trivial area and the location of p_2 is already verified. For every point in the area A_1 there does not exist a point in A_2 or A_3 that is higher. Therefore, regardless of where p_1 lies in A_1 and where p_3 is located in A_3, the point p_1 will be maximal in P. Based only on the areas of uncertainty the point p_3 may or may not be maximal in P. So updates have to be requested to verify some points, and therefore to make the problem solvable based on the initial areas of uncertainty and the information retrieved by the updates. The set $\{A_1, A_3\}$ is clearly an update solution as after updating these two sets the location of p_1 and p_3 are verified and both are maximal points in P. However the set $\{A_1\}$ is also an update solution as after verifying the location of p_1, neither p_1 nor p_2 are higher than any point in A_3. Hence even without verifying the location of p_3 within the area A_3 both p_1 and p_3 must be both maximal in P. In this example the set $\{A_1\}$ is also an optimal update solution as without any update the maximal points cannot be calculated. We finish this example by noting that updating just A_3 is not an update solution. While this verifies the exact location of p_3, the area A_1 still contains some points that are higher than p_3 and some that are not. So without also verifying the location of p_1 it is not clear whether p_3 is a maximal point among P or not.

We will use the following notation:

An area A is said to be *maximal* among a set of areas $\mathcal{A}$ if there does not exist a point in area of $\mathcal{A} - A$ that is higher than a point in A.

Similarly, an area A is said to be *dominated* among a set of areas $\mathcal{A}$ if for every point $p \in A$ there is an area in $\mathcal{A} - A$ with every point higher than p.

We also note that an area might be neither maximal nor dominated among a set of areas, whereas a point is either maximal or non-maximal among a set of points as defined earlier. If this is the case then the set of maximal points cannot be calculated. In other words a problem is *solved* if and only if all areas in $\mathcal{A}$ are either maximal or dominated among $\mathcal{A}$.

For further convenience we say an area A is *potentially higher* than an area B if there exists a point in A higher than a point in B.

In the last part of this section we recall the Minimum Set Cover problem. The Minimum Set Cover (MSC) problem consists of a universe U and a family $\mathcal{S}$ of subsets of U. The aim is to find a family of sets in $\mathcal{S}$ of minimal size that covers U. It was shown by [8] that the problem is NP-Hard. Without loss of generality we assume that every element in U is found in at least one set of $\mathcal{S}$ and that all sets in $\mathcal{S}$ have size of at least 2.

A reduction fo the MSC problem will lead to the following main result.

Theorem 1. *Solving the Maximal Point Verification problem is NP-hard.*

3 MP-Construction

In this section we give the construction of an MPV problem out of an MSC problem. We call the instance of the MSC problem MC $= (U, \mathcal{S})$ with $U = \{1, \ldots, n\}$ and $\mathcal{S} = \{S_1, \ldots, S_k\}$. The instance of the MPV will be denoted by MP.

The idea behind the construction is to have different types of areas in MP representing different aspects of MC. A set of areas (B's) will correspond to elements of U and another set of areas (A's) will correspond to elements of each $S_j \in \mathcal{S}$. The areas are positioned such that for each area corresponding to an element of U, at least one area corresponding to the occurrence of i in the set S_j must be included in any update solution. With the help of another set of areas (D's), the areas corresponding to elements of a set S_j are linked together. So, if an update solution contains one area corresponding to an element of a set S_j the update solution can be modified to include all areas that correspond to elements of S_j without increasing the size of the update solution.

The construction is done by using three different types of gadgets, and each gadjet is placed in its own rectangular region. These regions are located in the plane in such a way that no point in one gadget is higher than any point in another gadget. This can be achieved by placing all regions for the gadgets diagonally top-left to bottom-right in the plane, see figure 2.

Type 1 Gadget. For each $i \in U$ there exists one gadget of type 1. This contains the point b_i, which is the lower left corner of the gadget, and multiple distinct

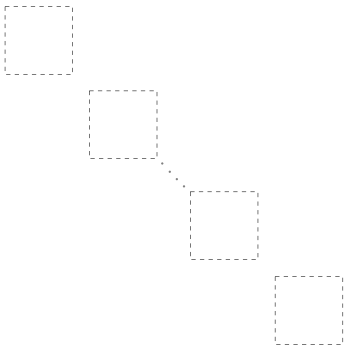


Fig. 2. Placement of gadgets

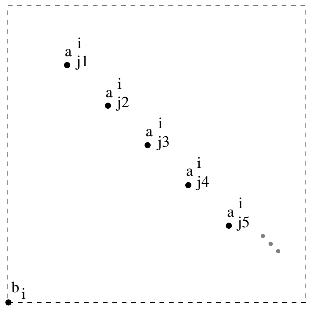


Fig. 3. Type 1 gadget

points along the diagonal of the gadget. For each set $S_j \in \mathcal{S}$ that contains i, a point a^i_j is placed on the diagonal. See figure 3.

Type 2 Gadget. For each set $S_j \in S$ there exists one gadget of type 2. This contains for every $i \in S_j$ a point c^i_j along the diagonal of the gadget such that all points are pairwise distinct. In addition points $d^1_j, \ldots, d^t_j$ with $t = |S_j| - 1$ are placed in such a way that for each d^r_j with $1 \leq r \leq t$ there exist exactly two points c^i_j and $c^{i'}_j$ that are higher. Furthermore any two neighbouring points c^i_j and $c^{i'}_j$ are higher than exactly one point d^r_j. This can be done easily by placing the points $d^1_j, \ldots, d^t_j$ along a line that is parallel to the diagonal, and closer to the bottom-left corner of the gadget than the diagonal. See figure 4.

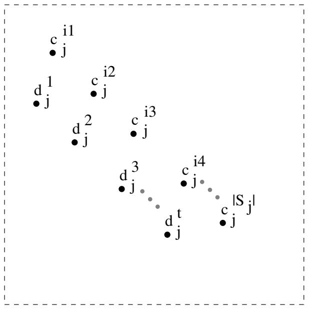


Fig. 4. Type 2 gadget

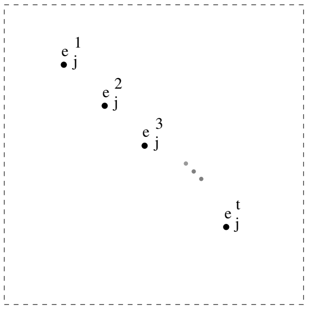


Fig. 5. Type 3 gadget

Type 3 Gadget. For each set $S_j \in \mathcal{S}$ there exists one gadget of type 3. This gadget just consists of $|S_j|-1$ distinct points $e^1_j, \ldots, e^t_j$ placed along the diagonal. See figure 5.

The various points placed in the three gadgets, are now used to define the areas of uncertainty $\mathcal{A}$, and the set of precise points P for MP.

Out of the points from the different gadgets we build the following sets where each set corresponds to an area for MP. For all $i \in U$ let B_i be the set containing only b_i. For all $i \in U$ and $S_j \in \mathcal{S}$ with $i \in S_j$ let A_j^i be the set containing the two points a_j^i and c_j^i. For all $S_j \in \mathcal{S}$ and $1 \leq r \leq |S_j| - 1$ let D_j^r be the set containing the two points d_j^r and e_j^r.

To handle these sets better in the remaining part of the paper we group some of these areas together. We say $A_j = \{A_j^i \mid i \in S_j\}$ and $D_j = \{D_j^1, \ldots, D_j^t\}$ with $t = |S_j| - 1$. We also note that $|A_j| = |D_j| + 1 = |S_j|$.

Further we say B is the set of all areas that correspond to an element of U (or formally $B = \{B_1, \ldots, B_n\}$), A is the set of all areas that correspond to an element of any set $S_j \in \mathcal{S}$ (or formally $A = \cup_{S_j \in \mathcal{S}} A_j$) and D is the set of all areas in any D_j (or formally $D = \cup_{S_j \in \mathcal{S}} D_j$).

This allows us to define our instance of the MPV in the following way: MP $= (\mathcal{A}, P)$ with $\mathcal{A} = B \cup A \cup D$ and $P = \{b_1, \ldots, b_n\} \cup \{a_j^i \mid i \in S_j\} \cup \{e_j^r \mid 1 \leq r \leq |S_j| - 1\}$.

We are now analysing the constructed problem MP and highlight properties that are needed in the further section.

Size of MP. There exist exactly n type 1 gadgets where each contains one point b_i with some $i \in U$. Each type 1 gadget contains at most further k points $\{a_j^i \mid i \in S_j\}$. There exist exactly k type 2 gadgets. Each contains at most $2n-1$ points $c_j^1, \ldots, c_j^{|S_j|}$ and $d_j^1, \ldots, d_j^{|S_j|-1}$, since $|S_j| \leq n$. There exist exactly k type 3 gadgets. Each contains at most $n-1$ points $e_j^1, \ldots, e_j^{|S_j|-1}$ since $|S_j| \leq n$.

Hence for the MP constructed we have $n + 2k$ gadgets and at most $n * (1 + k) + k * (2n - 1) + k * (n - 1) = n + 4nk - 2k$ points. As each point only lies in one area of uncertainty also $|\mathcal{A}|$ is at most $n + 4nk - 2k$ and so the input size of MP is polynomial in the size of MC.

Maximal Points among P. A point a_j^i for some j and i is part of a type 1 gadget and is clearly maximal among all points placed in the gadget. As two different gadgets are located so that no point of one is higher than a point of another, all points a_j^i are maximal in P. The same follows for the all points e_j^r in type 3 gadgets and therefore all such points are also maximal among P.

As for every $i \in U$ there must exists at least one $S_j \in \mathcal{S}$ with $i \in S_j$, by the construction of the type 1 gadget for i, also the point a_j^i was added to that gadget. As all such points are higher than b_i the point b_i is non-maximal among P.

Maximal Areas among $\mathcal{A}$. Each area in A consists of two points a_j^i and c_j^i. One is located inside a type 1 gadget and the other inside a type 2 gadget. For both points there is no area in $\mathcal{A}$ with a higher point, and therefore even without any updates all areas in A are maximal.

For each area $B_i \in B$ there exist some areas in A with a point above B_i and one point not above B_i. So among $\mathcal{A}$ the area B_i is neither maximal nor dominated and further updates are needed.

Each area in D has two points. One is located in a type 3 gadget which is clearly maximal; and one located in a type 2 gadget where there are two areas in A that contain points that are higher. So among $\mathcal{A}$ it is neither maximal nor dominated and further updates are needed.

Update Solutions for MP. Following from the above analysis of maximal areas among $\mathcal{A}$ we have the following remark:

Remark 1. A set of areas is an update solution if and only if it contains for each i an area $A_j^i \in A$ for some j, and also for each area in D either this area or the two areas in A that are potentially higher.

Following from this only updates of areas in A_j and D_j will help to identify areas of D_j as maximal. Based on the construction of type 2 gadgets, updating k areas of A_j can at most identify $k-1$ areas of D_j as maximal. Hence the smallest update set that identifies all areas of D_j as maximal is D_j itself. Any other set of updates must be bigger. Formally:

Remark 2. Let R be an update solution. Then for j the set R must contain either D_j or it must contain at least $|D_j| + 1$ areas of $D_j \cup A_j$.

This leads to the following Lemma:

Lemma 1. *Let R be an update solution for MP and let $A_i^j \in R$ for some j and i be an area. Then $R' = R - D_j + A_j$ is also an update solution and $|R'| \leq |R|$.*

Proof. Since R is an update solution, by Remark 1 for every $i \in U$ the set R must contain an area $A_{j'}^i$ for some j'. As R' in constructed by potentially removing areas of D and adding areas of A the set R' must also contain the area $A_{j'}^i$.

Let $D_{j''}^r \in D$. Again by Remark 1 either $D_{j''}^r \in R$ or the two areas in A that are potentially higher than $D_{j''}^r$ are in R. If R contains the two areas in A that are potentially higher than $D_{j''}^r$ then also R' must contain these areas as no area in A was removed when creating R'. If $D_{j''}^r \in R$ also R' must contain $D_{j''}^r$ unless $j'' = j$. In that case as all areas in A_j were added to R', it must also contain the two areas in A_j that are potentially higher than $D_{j''}^r$. Hence by Remark 1 also R' is an update solution.

We now show that $|R'| \leq |R|$. As $A_j^r \in R$, by Remark 2 R must contain at least $|D_j| + 1$ areas out of $D_j \cup A_j$. We noted in the construction of MP that $|A_j| = |D_j| + 1 = |S_j|$. So R' includes exactly $|D_j| + 1$ areas out of $D_j \cup A_j$. As R and R' only differ in selection of areas of D_j and A_j we have that $|R'| \leq |R|$.

4 Relating Update Solutions to Covers

In this section we show how to construct a cover of MC out of an update solution of MP and vice versa. We will also note how the size of the update solutions and covers relate to each other.

From Update Solution to Cover. Let R be an update solution for MP.

Before creating the cover we create a different update solution R'. The set R' is based on R but for all j such that there exists an i with $A_j^i \in R$ all potential areas of D_j are removed from R and all areas of A_j are added. By Lemma 1, we have that R' is also an update solution with no greater size than R. Furthermore by doing so, the update solution R' contains for every index j either the set A_j or D_j but never a mixture.

The cover $\mathcal{C}$ is constructed based on R' in the following way. For each index j such that $A_j \subseteq R'$ we choose the set $S_j \in \mathcal{S}$ to be included in $\mathcal{C}$ and otherwise not.

This is denoted as:

$$\mathcal{C} = \{S_j \in \mathcal{S} \mid A_j \subseteq R'\}$$

We now show that $\mathcal{C}$ is a cover, in other words that every element of U is found in at least one set of $\mathcal{C}$.

Let some $i \in U$ for the MC. Then in MP there exists the area B_i. By remark 1 there exists an index j with $A_j^i \in R'$. Since this area A_j^i was constructed in the creation of MP we have that $i \in S_j$. As A_j^i is also in R' the set A_j must be a subset of R and $S_j \in \mathcal{C}$.

We note that the construction of $\mathcal{C}$ is done in polynomial time and the sizes of R, R' and C relate to each other in the following way.

By the construction of R' we have:

$$|R'| = \sum_{A_j \subseteq R'} |A_j| \ + \sum_{A_j \not\subseteq R'} |D_j|$$

As $|A_j| = |D_j| + 1 = |S_j|$ for all j we get by the construction of $\mathcal{C}$ that:

$$|R'| = \sum_{S_j \in \mathcal{C}} |S_j| \ + \sum_{S_j \in \mathcal{S}-\mathcal{C}} (|S_j| - 1)$$

$$= |\mathcal{C}| + \sum_{S_j \in \mathcal{C}} (|S_j| - 1) \ + \sum_{S_j \in \mathcal{S}-\mathcal{C}} (|S_j| - 1)$$

$$= |\mathcal{C}| + \sum_{S_j \in \mathcal{S}} (|S_j| - 1)$$

Since by Lemma 1 we get $|R| \geq |R'|$ we have:

$$|R| \geq |\mathcal{C}| \ + \sum_{S_j \in \mathcal{S}} (|S_j| - 1)$$

We summarise our results on the construction of $\mathcal{C}$ in the following Lemma:

Lemma 2. *Let R be an update solution for MP. Then a cover of MC can be constructed in polynomial time with* $|R| \geq |\mathcal{C}| + \sum_{S_j \in \mathcal{S}} (|S_j| - 1)$.

From Cover to Update Solution. Similarly to the construction of a cover for MC out of a given update solution of MP, we now show how to construct an update solution for MP out of a given cover for MC.

Let $\mathcal{C}$ be a cover for MC.

The set R of areas in MP is based on $\mathcal{C}$ as follows: For each index j such that $S_j \in \mathcal{C}$ we choose the set A_j to be included in R. For each index j such that $S_j \in (\mathcal{S} - \mathcal{C})$ we choose the set D_j to be included in R.

This is denoted as:

$$R = (\bigcup_{S_j \in \mathcal{C}} A_j) \cup (\bigcup_{S_j \in (\mathcal{S}-\mathcal{C})} D_j)$$

We note that the following: Firstly, let $i \in U$. Since $\mathcal{C}$ is a cover there exists an index j such that $S_j \in \mathcal{C}$ and $i \in S_j$. Hence, by the construction of MP the area A_j^i exists in MP. As $S_j \in \mathcal{C}$ we have that $A_j \subseteq R$ and in particular $A_j^i \in R$.

Secondly, let $D_j^r \in D$. If $S_j \notin \mathcal{C}$ the set R contains D_j and therefore also D_j^r. Otherwise R contains A_j and therefore also the two areas in A_j that are potentially higher than D_j^r.

So R satisfies both condition of remark 1 and is hence an update solution for MP.

We recall from the MP-construction that for every set S_j there is a set A_j and a set D_j such that $|A_j| = |D_j| + 1 = |S_j|$. So,

$$\begin{aligned} |R| &= \sum_{S_j \in \mathcal{C}} |S_j| \ + \sum_{S_j \in \mathcal{S}-\mathcal{C}} (|S_j| - 1) \\ &= |\mathcal{C}| + \sum_{S_j \in \mathcal{C}} (|S_j| - 1) \ + \sum_{S_j \in \mathcal{S}-\mathcal{C}} (|S_j| - 1) \\ &= |\mathcal{C}| + \sum_{S_j \in \mathcal{S}} (|S_j| - 1) \end{aligned}$$

This leads to the following lemma:

Lemma 3. *Let $\mathcal{C}$ be a cover of MC. Then there exists an update solution R for MP with $|\mathcal{C}| \leq |R| + \sum_{S_j \in \mathcal{S}} (|S_j| - 1)$.*

5 NP-Hardness Proof

We have shown so far how an instance MP of the Maximal Point Verification problem can be constructed out of an instance MC of the Minimum Set Cover problem, how one can build a solution for one of these two problem instances based on the solution of the other, and how the sizes of the solutions are related. We now argue that an optimal update solution corresponds to a minimal cover.

Lemma 4. *Let R be an optimal update solution for MP. Then the cover $\mathcal{C}$ constructed out of R is a minimal cover for MC.*

Proof. Let's assume there exists a cover $\overline{\mathcal{C}}$ for MC such that $|\overline{\mathcal{C}}| < |\mathcal{C}|$.

Let $\overline{R}$ be the update solution for MP constructed from $\overline{\mathcal{C}}$ as shown in Section 4. Then by Lemmas 2 and 3 we have that

$$|\mathcal{C}| \leq |R| - \sum_{S_j \in \mathcal{S}} (|S_j| - 1)$$

and

$$|\overline{\mathcal{C}}| = |\overline{R}| - \sum_{S_j \in \mathcal{S}} (|S_j| - 1).$$

Since $|\overline{\mathcal{C}}| < \mathcal{C}$ so must $|\overline{R}| < |R|$. This is a contradiction as R was a minimal update solution. So, $\mathcal{C}$ must be a minimal cover of MC.

We are using the established results to prove theorem 1.

Proof. In Section 3 we have presented the construction of a MPV problem for a given MSC problem. As noted in Section 3 the size of the MPV problem is polynomial in the size of the MSC problem and the construction can be done in polynomial time.

By Lemma 4 a solution of the MPV can be used to construct a solution of the MSC problem. As remarked in Section 4 that construction is polynomial in the size of the MPV problem.

Hence, if the MPV problem is solvable in polynomial time, then this must also be the case for the MSC problem. By [8], the MSC problem is shown to be NP-hard. So, also the MPV problem is NP-hard.

References

1. Bruce, R., Hoffmann, M., Krizanc, D., Raman, R.: Efficient update strategies for geometric computing with uncertainty. Theory of Computing Systems 38(4), 411–423 (2005)
2. Erlebach, T., Hoffmann, M., Krizanc, D., Mihalák, M., Raman, R.: Computing minimum spanning trees with uncertainty. In: Albers, S., Weil, P. (eds.) STACS. LIPIcs, vol. 1, pp. 277–288. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany (2008)
3. Feder, T., Motwani, R., O'Callaghan, L., Olston, C., Panigrahy, R.: Computing shortest paths with uncertainty. Journal of Algorithms 62(1), 1–18 (2007)
4. Feder, T., Motwani, R., Panigrahy, R., Olston, C., Widom, J.: Computing the median with uncertainty. SIAM Journal on Computing 32(2), 538–547 (2003)
5. Gupta, M., Sabharwal, Y., Sen, S.: The update complexity of selection and related problems. In: IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2011). LIPIcs, vol. 13, pp. 325–338. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2011)
6. Kahan, S.: A model for data in motion. In: Proceedings of the 23rd Annual ACM Symposium on Theory of Computing (STOC 1991), pp. 267–277 (1991)
7. Kamousi, P., Chan, T.M., Suri, S.: Stochastic minimum spanning trees in Euclidean spaces. In: Proceedings of the 27th Annual ACM Symposium on Computational Geometry (SoCG 2011), pp. 65–74. ACM (2011)
8. Karp, R.M.: Reducibility among combinatorial problems. In: Complexity of Computer Computations, pp. 85–103 (1972)

Incidence Coloring Game and Arboricity of Graphs

Clément Charpentier[1,2] and Éric Sopena[1,2]

[1] Univ. Bordeaux, LaBRI, UMR5800, F-33400 Talence
[2] CNRS, LaBRI, UMR5800, F-33400 Talence

Abstract. An incidence of a graph G is a pair (v, e) where v is a vertex of G and e an edge incident to v. Two incidences (v, e) and (w, f) are adjacent whenever $v = w$, or $e = f$, or $vw = e$ or f. The incidence coloring game [S.D. Andres, The incidence game chromatic number, Discrete Appl. Math. 157 (2009), 1980–1987] is a variation of the ordinary coloring game where the two players, Alice and Bob, alternately color the incidences of a graph, using a given number of colors, in such a way that adjacent incidences get distinct colors. If the whole graph is colored then Alice wins the game otherwise Bob wins the game. The incidence game chromatic number $i_g(G)$ of a graph G is the minimum number of colors for which Alice has a winning strategy when playing the incidence coloring game on G.

Andres proved that $i_g(G) \leq 2\Delta(G) + 4k - 2$ for every k-degenerate graph G. We show in this paper that $i_g(G) \leq \lfloor \frac{3\Delta(G)-a(G)}{2} \rfloor + 8a(G) - 2$ for every graph G, where $a(G)$ stands for the arboricity of G, thus improving the bound given by Andres since $a(G) \leq k$ for every k-degenerate graph G. Since there exists graphs with $i_g(G) \geq \lceil \frac{3\Delta(G)}{2} \rceil$, the multiplicative constant of our bound is best possible.

Keywords: Arboricity, Incidence coloring, Incidence coloring game, Incidence game chromatic number.

1 Introduction

All the graphs we consider are finite and undirected. For a graph G, we denote by $V(G)$, $E(G)$ and $\Delta(G)$ its vertex set, edge set and maximum degree, respectively. Recall that a graph is *k-denegerate* if all of its subgraphs have minimum degree at most k.

The *graph coloring game* on a graph G is a two-player game introduced by Brams [7] and rediscovered ten years after by Bodlaender [3]. Given a set of k colors, Alice and Bob take turns coloring properly an uncolored vertex of G, Alice having the first move. Alice wins the game if all the vertices of G are eventually colored, while Bob wins the game whenever, at some step of the game, all the colors appear in the neighborhood of some uncolored vertex. The *game chromatic number* $\chi_g(G)$ of G is then the smallest k for which Alice has a winning strategy when playing the graph coloring game on G with k colors.

© Springer-Verlag Berlin Heidelberg 2013

The problem of determining the game chromatic number of planar graphs has attracted great interest in recent years. Kierstead and Trotter proved in 1994 that every planar graph has game chromatic number at most 33 [11]. This bound was decreased to 30 by Dinski and Zhu [6], then to 19 by Zhu [16], to 18 by Kierstead [10] and to 17, again by Zhu [17], in 2008. Some other classes of graphs have also been considered (see [2] for a comprehensive survey).

An *incidence* of a graph G is a pair (v, e) where v is a vertex of G and e an edge incident to v. We denote by $I(G)$ the set of incidences of G. Two incidences (v, e) and (w, f) are *adjacent* if either (1) $v = w$, (2) $e = f$ or (3) $vw = e$ or f. An *incidence coloring* of G is a coloring of its incidences in such a way that adjacent incidences get distinct colors. The smallest number of colors required for an incidence coloring of G is the *incidence chromatic number* of G, denoted by $\chi_i(G)$. Let G be a graph and $S(G)$ be the *full subdivision* of G, obtained from G by subdividing every edge of G (that is, by replacing each edge uv by a path $ux_{uv}v$, where x_{uv} is a new vertex of degree 2). It is then easy to observe that every incidence coloring of G corresponds to a *strong edge coloring* of $S(G)$, that is a proper edge coloring of $S(G)$ such that every two edges with the same color are at distance at least 3 from each other [4,14].

Incidence colorings have been introduced by Brualdi and Massey [4] in 1993, motivated by the study of the strong chromatic index of bipartite graphs. Upper bounds on the incidence chromatic number have been proven for various classes of graphs such as k-degenerate graphs and planar graphs [8,9], graphs with maximum degree three [13], and exact values are known for instance for forests [4], K_4-minor-free graphs [9], or Halin graphs with maximum degree at least 5 [15] (see [14] for an on-line survey).

In [1], Andres introduced the *incidence coloring game*, as the incidence version of the graph coloring game, each player, on his turn, coloring an uncolored incidence of G in a proper way. The *incidence game chromatic number* $i_g(G)$ of a graph G is then defined as the smallest k for which Alice has a winning strategy when playing the incidence coloring game on G with k colors. Upper bounds on the incidence game chromatic number have been proven for k-degenerate graphs [1] and exact values are known for cycles, stars [1], paths and wheels [12].

Andres observed that the inequalities $\lceil \frac{3}{2}\Delta(G) \rceil \leq i_g(F) \leq 3\Delta(G) - 1$ hold for every graph G [1]. For k-degenerate graphs, he proved the following:

Theorem 1 (Andres, [1]). *Let G be a k-degenerated graph. Then we have:*

(i) $i_g(G) \leq 2\Delta(G) + 4k - 2$,
(ii) $i_g(G) \leq 2\Delta(G) + 3k - 1$ *if* $\Delta(G) \geq 5k - 1$,
(iii) $i_g(G) \leq \Delta(G) + 8k - 2$ *if* $\Delta(G) \leq 5k - 1$.

Since forests, outerplanar graphs and planar graphs are respectively 1-, 2- and 5-degenerate, we get that $i_g(G) \leq 2\Delta(G) + 2$, $i_g(G) \leq 2\Delta(G) + 6$ and $i_g(G) \leq 2\Delta(G) + 18$ whenever G is a forest, an outerplanar graph or a planar graph, respectively.

Recall that the arboricity $a(G)$ of a graph G is the minimum number of forests into which its set of edges can be partitioned. In this paper, we will prove the following:

Theorem 2. *For every graph G, $i_g(G) \leq \lfloor \frac{3\Delta(G)-a(G)}{2} \rfloor + 8a(G) - 1$.*

Recall that $i_g(G) \geq 3\Delta(G)/2$ for every graph G so that the difference between the upper and the lower bound on $i_c(G)$ only depends on the arboricity of G.

It is not difficult to observe that $a(G) \leq k$ whenever G is a k-degenerate graph. Hence we get the following corollary, which improves Andres' Theorem and answers in the negative a question posed in [1]:

Corollary 3. *If G is a k-degenerate graph, then $i_g(G) \leq \lfloor \frac{3\Delta(G)-k}{2} \rfloor + 8k - 1$.*

Since outerplanar graphs and planar graphs have arboricity at most 2 and 3, respectively, we get as a corollary of Theorem 2 the following:

Corollary 4.

(i) $i_g(G) \leq \lceil \frac{3\Delta(G)}{2} \rceil + 6$ *for every forest G,*
(ii) $i_g(G) \leq \lfloor \frac{3\Delta(G)}{2} \rfloor + 14$ *for every outerplanar graph G,*
(iii) $i_g(G) \leq \lceil \frac{3\Delta(G)}{2} \rceil + 21$ *for every planar graph G.*

In a companion paper [5], we prove that $i_g(G) \leq \lceil \frac{3\Delta(G)}{2} \rceil + 4$ for every forest G, using a refinement of the strategy introduced in this paper.

This paper is organised as follows: we detail Alice's strategy in Section 2 and prove Theorem 2 in Section 3.

2 Alice's Strategy

We will give a strategy for Alice which allows her to win the incidence coloring game on a graph G with arboricity $a(G)$ whenever the number of available colors is at least $\lfloor \frac{3}{2}\Delta(G) \rfloor + 8a(G) - 1$. This strategy will use the concept of *activation strategy* [2], often used in the context of the ordinary graph coloring game.

Let G be a graph with arboricity $a(G) = a$. We partition the edges of G into a forests F_1, ..., F_a, each forest containing a certain number of trees. For each tree T, we choose an arbitrary vertex of T, say r_T, to be the *root* of T.

Notation. Each edge with endvertices u and v in a tree T will be denoted by uv if $dist_T(u, r_T) < dist_T(v, r_T)$, and by vu if $dist_T(v, r_T) < dist_T(u, r_T)$, where $dist_T$ stands for the distance within the tree T (in other words, we define an orientation of the graph G in such a way that all the edges of a tree T are oriented from the root towards the leaves).

We now give some notation and definitions we will use in the sequel (these definitions are illustrated in Fig. 1).

- For every edge uv belonging to some tree T, we say the incidence (u, uv) is a *top incidence* whereas the incidence (v, uv) is a *down incidence*. We then let $t(uv) = t(v, vu) = (u, uv)$ and $d(uv) = d(u, uv) = (v, uv)$.
  Note that each vertex in a forest F_i is incident to at most one down incidence belonging to F_i, so that each vertex in G is incident to at most a down incidences.

- For every incidence i belonging to some edge $uv \in E(G)$, let $tF(i) = \{t(wu),\ wu \in E(G)\}$ be the set of *top-fathers* of i, $dF(i) = \{d(wu),\ wu \in E(G)\}$ be the set of *down-fathers* of i and $F(i) = tF(i) \cup dF(i)$ be the set of *fathers* of i.

  Note that each incidence has at most a top-fathers and at most a down-fathers.
- For every incidence i belonging to some edge $uv \in E(G)$, let $tS(i) = \{t(vw),\ vw \in E(G)\}$ be the set of *top-sons* of i, $dS(i) = \{d(vw),\ vw \in E(G)\}$ be the set of *down-sons* of i and $S(i) = tS(i) \cup dS(i)$ be the set of *sons* of i.

  Note that each incidence has at most $\Delta(G) - 1$ top-sons and at most $\Delta(G) - 1$ down-sons.
- For every incidence i belonging to some edge $uv \in E(G)$, let $tB(i) = \{t(uw),\ uw \in E(G)\} - \{i\}$ be the set of *top-brothers* of i, $dB(i) = \{d(uw),\ uw \in E(G)\} - \{i\}$ be the set of *down-brothers* of i and $B(i) = tB(i) \cup dB(i)$ be the set of *brothers* of i.

  Note that each top incidence i has at most $\Delta(G) - |tF(i)| - 1$ top-brothers and $\Delta(G) - |tF(i)|$ down-brothers while each down incidence j has at most $\Delta(G) - |tF(j)|$ top-brothers and $\Delta(G) - |tF(j)| - 1$ down-brothers.

  Note also that any two brother incidences have exactly the same set of fathers.
- Finally, for every incidence i belonging to some edge $uv \in E(G)$, let $tU(i) = \{t(wv),\ wv \in E(G)\}$ be the set of *top-uncles* of i, $dU(i) = \{d(wv),\ wv \in E(G)\}$ be the set of *down-uncles* of i and $U(i) = tU(i) \cup dU(i)$ be the set of *uncles* of i (the term "uncle" is not metaphorically correct since the uncle of an incidence i is another father of the sons of i rather than a brother of a father of i).

  Note that each incidence has at most $a - 1$ top-uncles and at most $a - 1$ down-uncles. Moreover, we have $|dU(i)| + |tS(i)| \leq \Delta(G) - 1$ for every incidence $i \in I(G)$.

Fig. 1 illustrates the above defined sets of incidences. Each edge is drawn in such a way that its top incidence is located above its down incidence. Incidence i is drawn as a white box, top incidences are drawn as grey boxes and down incidences (except i) are drawn as black boxes.

We now turn to the description of Alice's strategy. For each set I of incidences, we will denote by I_c the set of colored incidences of I. We will use an *activation strategy*. During the game, each uncolored incidence may be either *active* (if Alice activated it) or *inactive*. When the game starts, every incidence is inactive. When an active incidence is colored, it is no longer considered as active. For each set I of incidences, we will denote by I_a the set of active incidences of I (I_a and I_c are therefore disjoint for every set of incidences I).

We denote by Φ the set of colors used for the game, by $\phi(i)$ the color of an incidence i and, for each set I of incidences, we let $\phi(I) = \bigcup_{i \in I} \phi(i)$. As shown by Fig. 1, the set of *forbidden* colors for an uncolored incidence i is given by:

- $\phi(F(i) \cup B(i) \cup tS(i) \cup dU(i))$ if i is a top incidence,
- $\phi(dF(i) \cup tB(i) \cup S(i) \cup U(i))$ if i is a down incidence.

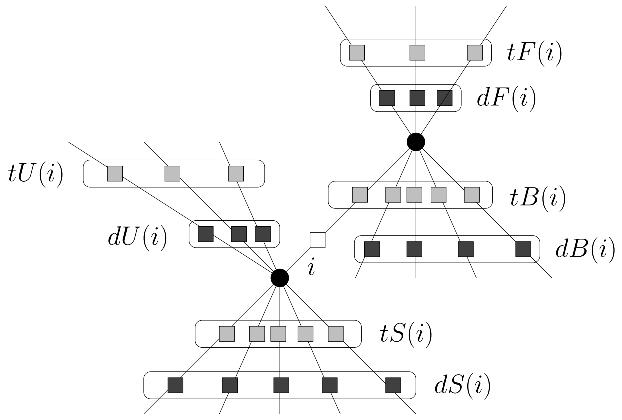


Fig. 1. Incidences surrounding the incidence i

Our objective is therefore to bound the cardinality of these sets. We now define the subset I_n of *neutral incidences* of $I(G)$, which contains all the incidences j such that:

(i) j is not colored, and
(ii) all the incidences of $F(j)$ are colored.

We also describe what we call a *neutral move* for Alice, that is a move Alice makes only if there is no neutral incidence and no activated incidence in the game. Let i_0 be any uncolored incidence of $I(G)$. Since there is no neutral incidence, either there is an uncolored incidence i_1 in $dF(i_0)$, or all the incidences of $dF(i_0)$ are colored and there is an uncolored incidence i_1 in $tF(i_0)$. We define in the same way incidences i_2 from i_1, i_3 from i_2, and so on, until we reach an incidence that has been already encountered. We then have $i_k = i_\ell$ for some integers k and ℓ, with $k \leq \ell$. The neutral move of Alice then consists in activating all the incidences within the loop and coloring any one of them.

Alice's strategy uses four rules. The first three rules, (R1), (R2) and (R3) below, determine which incidence Alice colors at each move. The fourth rule explains which color will be used by Alice when she colors an incidence.

(R1) On her first move,
- If there is a neutral incidence (i.e., in this case, an incidence without fathers), then Alice colors it.
- Otherwise, Alice makes a neutral move.

(R2) If Bob, in his turn, colors a down incidence i with no uncolored incidence in $dF(i)$, then
(R2.2.1) If there are uncolored incidences in $dB(i)$, then Alice colors one of them,
(R2.2.2) Otherwise,

  - If there is a neutral incidence or an activated incidence in $I(G)$, then Alice colors it,
  - If not, Otherwise, Alice makes a neutral move.

(R3) If Bob colors another incidence, then Alice *climbs* it. Climbing an incidence i is a recursive procedure, described as follows:
  - (R3.1) If i is active, then Alice colors i.
  - (R3.2) Otherwise, if i is not colored then Alice activates i, and:
    - If there are uncolored incidences in $dF(i)$, then Alice climbs one of them.
    - If all the incidences of $dF(i)$ are colored, and if there are uncolored incidences in $tF(i)$, then Alice climbs one of them.
    - If all the incidences of $F(i)$ are colored, then:
      - if there is a neutral incidence or an activated incidence in $I(G)$, then Alice colors it,
      - otherwise, Alice makes a neutral move.

(R4) When Alice has to color an incidence i, she proceeds as follows: if i is a down incidence with $|\phi(dB(i))| \geq 4a - 1$, she uses any available color in $\phi(dB(i))$; in all other cases, she chooses any available color.

Observe that, in a neutral move, all the incidences $i_k, i_{k+1}, \ldots, i_\ell$ form a *loop* where each incidence can be reached by climbing the previous one. We consider that, when Alice does a neutral move, all the incidences are climbed at least one.

Then we have:

Observation 5. *When an inactive incidence is climbed, it is activated. When an active incidence is climbed, it is colored. Therefore, every incidence is climbed at most twice.*

Observation 6. *Alice only colors neutral incidences or active incidences (typically, incidences colored by Rule (R2.2.1) are neutral incidences), except when she makes a neutral move.*

3 Proof of Theorem 2

We now prove a series of lemmas from which the proof of Theorem 2 will follow.

Lemma 7. *When Alice or Bob colors a down incidence i, we have*

$$|S_c(i)| + |U_c(i)| \leq 4a - 2.$$

When Alice or Bob colors a top incidence i, we have

$$|tS_c(i)| + |dU_c(i)| \leq 5a - 1.$$

Proof. Let first i be a down incidence that has just been colored by Bob or Alice. If $|S_c(i)| = 0$, then $|S_c(i)| + |U_c(i)| = |U_c(i)| \leq |U(i)| \leq 2a - 2$. Otherwise, let j be an incidence from $S(i)$ which was colored before i.

– If j was colored by Bob, then Alice has climbed i or some other incidence from $dU(i)$ in her next move by Rule (R2.1).
– If j was colored by Alice, then
  - either j was an active incidence and, when j has been activated, Alice has climbed either $d(i)$, or i, or some other incidence from $U(i)$,
  - or Alice has made a neutral move and, in the same move, has activated either $d(i)$, or i, or some other incidence from $U(i)$.

By Observation 5 every incidence is climbed at most twice, and thus $|S_c(i)| \leq 2 \times (|dU(i)| + 1)$. Since $|dU(i)| \leq a - 1$, we have $|S_c(i)| \leq 2a$. Moreover, since $|U_c(i)| \leq |U(i)| \leq 2a - 2$, we get $|S_c(i)| + |U_c(i)| \leq 4a - 2$ as required.

Let now i be a top incidence that has just been colored by Bob or Alice. If $|tS_c(i)| = 0$, then $|tS_c(i)| + |dU_c(i)| = |dU_c(i)| \leq |dU(i)| \leq a - 1$. Otherwise, let j be an incidence from $tS(i)$ which was colored before i.

– If j was colored by Bob then, in her next move, Alice either has climbed $d(i)$ or some other incidence from $dU(i)$ by Rule (R2.1), or i or some other incidence from $tU(i)$ by Rule (R2.3).
– If j was colored by Alice, then
  - either j was an active incidence and, when j has been activated, Alice has climbed either $d(i)$, or i, or some other incidence from $U(i)$,
  - or Alice has made a neutral move and, in the same move, has activated either $d(i)$, or i, or some other incidence from $U(i)$.

By Observation 5 every incidence is climbed at most twice, and thus $|tS_c(i)| \leq 2 \times (|U(i)| + 2)$. Since $|U(i)| \leq 2a - 2$, we have $|tS_c(i)| \leq 4a$. Moreover, since $|dU_c(i)| \leq |dU(i)| \leq a - 1$, we get $|tS_c(i)| + |dU_c(i)| \leq 5a - 1$ as required. □

Lemma 8. *Whenever Alice or Bob colors a down incidence i, there is always an available color for i if $|\Phi| \geq \Delta(G) + 5a - 2$. Moreover, if $|\phi(dB(i))| \geq 4a - 1$, then there is always an available color in $\phi(dB(i))$ for coloring i.*

Proof. When Alice or Bob colors a down incidence i, the forbidden colors for i are the colors of $tB(i)$, $dF(i)$, $S(i)$ and $U(i)$.

Observe that $|dF(i)| + |tB(i)| \leq \Delta(G) - 1$ for each down incidence i, so $|\phi(dF(i))| + |\phi(tB(i))| \leq \Delta(G) - 1$.

Now, since $|\phi(S(i))| + |\phi(U(i))| \leq |S_c(i)| + |U_c(i)| \leq 4a - 2$ by Lemma 7, we get that there are at most $\Delta(G) + 5a - 3$ forbidden colors, and therefore an available color for i whenever $|\Phi| \geq \Delta(G) + 5a - 2$.

Moreover, since the colors of $\phi(dF(i))$ and $\phi(tB(i))$ are all distinct from those of $\phi(dB(i))$, there are at most $|S_c(i)| + |U_c(i)| \leq 4a - 2$ colors of $\phi(dB(i))$ that are forbidden for i, and therefore an available color for i whenever $|\phi(dB(i))| \geq 4a - 1$. □

Lemma 9. *For every incidence i, $|\phi(dB(i))| \leq \lfloor \frac{|dB(i)|}{2} \rfloor + 2a$.*

Proof. For every incidence i, as soon as $|\phi(dB(i))| = 4a - 1$, there are at least $4a - 1$ colored incidences in $dB(i)$. If $dF(i)$ is not empty, then every incidence in $dF(i)$ has thus at least $4a - 1$ colored sons so that, by Lemma 7, every such incidence is already colored. During the rest of the game, each time Bob will color an incidence of $dB(i)$, if there are still some uncolored incidences in $dB(i)$, then Alice will answer by coloring one of them by Rule (R2.2.1). Hence, Bob will color at most $\lceil \frac{|dB(i)-(4a-1)|}{2} \rceil$ of these incidences. Since, by Rule (R3), Alice uses colors already in $\phi(dB(i))$ for the incidences she colors, we get $|\phi(dB(i))| \leq 4a - 1 + \lceil \frac{|dB(i)-(4a-1)|}{2} \rceil \leq \lfloor \frac{|dB(i)|}{2} \rfloor + 2a$ as required. □

Lemma 10. *When Alice or Bob colors a top incidence i, there is always an available color for i whenever $|\Phi| \geq \lfloor \frac{3\Delta(G)-a}{2} \rfloor + 8a - 1$.*

Proof. Let i be any uncolored top incidence. The forbidden colors for i are the colors of $tF(i)$, $dF(i)$, $tB(i)$, $dB(i)$, $dU(i)$ and $tS(i)$. We have:

- $|\phi(tF(i))| + |\phi(tB(i))| \leq |tF(i)| + |tB(i)| \leq \Delta(G) - 1$,
- $|\phi(dF(i))| \leq |dF(i)| \leq a$ and, by Lemma 9, $|\phi(dB(i))| \leq \lfloor \frac{|dB(i)|}{2} \rfloor + 2a$; since $|dF(i)| + |dB(i)| \leq \Delta(G)$, we get

$$\begin{aligned} |\phi(dF(i))| + |\phi(dB(i))| &\leq |dF(i)| + \lfloor \tfrac{\Delta(G)-|dF(i)|}{2} \rfloor + 2a \\ &= \lceil \tfrac{3|dF(i)|}{2} \rceil + \lfloor \tfrac{\Delta(G)}{2} \rfloor + 2a \\ &\leq \lceil \tfrac{3a}{2} \rceil + \lfloor \tfrac{\Delta(G)}{2} \rfloor + 2a \\ &= \lfloor \tfrac{\Delta(G)-a}{2} \rfloor + 3a, \end{aligned}$$

- $|\phi(tS(i))| + |\phi(dU(i))| \leq 5a - 1$ by Lemma 7.

So there are at most $\lfloor \frac{3\Delta(G)-a}{2} \rfloor + 8a - 2$ forbidden colors for i and the result follows. □

We are now able to prove our main result:

Proof (of Theorem 2). When Alice applies the above described strategy, we know by Lemma 10 that every top incidence can be colored, provided $|\Phi| \geq \lfloor \frac{3\Delta(G)-a(G)}{2} \rfloor + 8a(G) - 1$, and by Lemma 8 that this is also the case for every down incidence.

References

1. Andres, S.: The incidence game chromatic number. Discrete Appl. Math. 157, 1980–1987 (2009)
2. Bartnicki, T., Grytczuk, J., Kierstead, H.A., Zhu, X.: The map coloring game. Amer. Math. Monthly (November 2007)
3. Bodlaender, H.: On the complexity of some coloring games. Int. J. Found. Comput. Sci. 2, 133–147 (1991)
4. Brualdi, R., Massey, J.: Incidence and strong edge colorings of graphs. Discrete Math. 122, 51–58 (1993)

5. Charpentier, C., Sopena, E.: The incidence game chromatic number of forests (preprint, 2013)
6. Dinski, T., Zhu, X.: Game chromatic number of graphs. Discrete Math. 196, 109–115 (1999)
7. Gardner, M.: Mathematical game. Scientific American 23 (1981)
8. Hosseini Dolama, M., Sopena, E.: On the maximum average degree and the incidence chromatic number of a graph. Discrete Math. and Theoret. Comput. Sci. 7(1), 203–216 (2005)
9. Hosseini Dolama, M., Sopena, E., Zhu, X.: Incidence coloring of k-degenerated graphs. Discrete Math. 283, 121–128 (2004)
10. Kierstead, H.: A simple competitive graph coloring algorithm. J. Combin. Theory Ser. B 78(1), 57–68 (2000)
11. Kierstead, H., Trotter, W.: Planar graph coloring with an uncooperative partner. J. Graph Theory 18, 569–584 (1994)
12. Kim, J.: The incidence game chromatic number of paths and subgraphs of wheels. Discrete Appl. Math. 159, 683–694 (2011)
13. Maydansky, M.: The incidence coloring conjecture for graphs of maximum degree three. Discrete Math. 292, 131–141 (2005)
14. Sopena, E.: `http://www.labri.fr/perso/sopena/TheIncidenceColoringPage`
15. Wang, S., Chen, D., Pang, S.: The incidence coloring number of Halin graphs and outerplanar graphs. Discrete Math. 256, 397–405 (2002)
16. Zhu, X.: The game coloring number of planar graphs. J. Combin. Theory Ser. B 75(2), 245–258 (1999)
17. Zhu, X.: Refined activation strategy for the marking game. J. Combin. Theory Ser. B 98(1), 1–18 (2008)

Linear-Time Self-stabilizing Algorithms for Minimal Domination in Graphs[⋆]

Well Y. Chiu and Chiuyuan Chen[⋆⋆]

Department of Applied Mathematics, National Chiao Tung University,
Hsinchu 30010, Taiwan
weeeeeeeeell@gmail.com, cychen@mail.nctu.edu.tw

Abstract. A distributed system is self-stabilizing if, regardless of the initial state, the system is guaranteed to reach a legitimate (correct) state in finite time. In 2007, Turau proposed the first linear-time self-stabilizing algorithm for the minimal dominating set (MDS) problem under an unfair distributed daemon [6]; this algorithm stabilizes in at most $9n$ moves, where n is the number of nodes. In 2008, Goddard et al. [2] proposed a $5n$-move algorithm. In this paper, we present a $4n$-move algorithm. We also prove that if an MDS-silent algorithm is preferred, then distance-1 knowledge is insufficient, where a self-stabilizing MDS algorithm is MDS-silent if it will not make any move when the starting configuration of the system is already an MDS.

1 Introduction

Self-stabilization is a concept of designing a distributed system for transient fault toleration and was introduced by Dijkstra in 1974 [1]. Such a system has to be able to reach a legal configuration in finite time, given an illegal starting configuration; and once in a legal configuration, the system may only move to other legal configurations (in the absence of external interference). The concept of self-stabilization has been developed for different inter-node communication styles. In the *message-passing model* of communication networks, nodes communicate by exchanging messages with their neighbors. Each node performs a sequence of steps. There are different ways to define a step. An algorithm uses *composite atomicity*, which allows every atomic step to contain a read operation and a write operation. It is assumed that our algorithms use composite atomicity and the message-passing model of communication.

A self-stabilizing algorithm executed in every node comprises a collection of rules of the form:

$$\langle \text{precondition} \rangle \;\rightarrow\; \langle \text{statement} \rangle .$$

The precondition is a Boolean expression involving the states of the node and its neighbors. The statement updates the state of the node. A rule is *enabled* if its

⋆ This research was partially supported by the National Science Council of the Republic of China under the grants grant NSC100-2115-M-009-004-MY2.

⋆⋆ Corresponding author.

© Springer-Verlag Berlin Heidelberg 2013

precondition evaluates to be true. A node is *privileged* if at least one of its rules is enabled. The execution of a statement is called a *move*. It is assumed that the computation of a precondition and the move are performed in one atomic step, i.e., rules are atomically executed.

A self-stabilizing algorithm operates in rounds. In every round, every node checks the preconditions of its rules. Various execution models have been used and these models are encapsulated within the notion of a scheduler (or daemon). There are three types of daemons: in every round, the *central daemon* selects only one privileged node to make a move; under the *synchronous daemon*, all privileged nodes move simultaneously; and the *distributed daemon* selects a nonempty subset of all privileged nodes to move. A scheduler may be *fair* or *unfair*; the former guarantees that every node is eventually selected for making a move and the latter only guarantees global system progress, i.e. there is at least one move in each round. As a matter of fact, an unfair distributed scheduler is more practical for implementations than the other types of schedulers.

In this paper, we consider a distributed system whose topology is represented by an undirected, simple graph $G = (V, E)$, whose V represents the set of processes and E represents the set of edges (i.e., interconnections between processes). Let $n = |V|$. If two vertices are connected by an edge, they are called *neighbors*. A subset S of the vertex set V of a graph G is a *dominating set* (DS) if each vertex v is either a member of S or adjacent to a vertex in S. A dominating set of G is a *minimal dominating set* (MDS) if none of its proper subsets is a dominating set of G. A minimal dominating set has an application of clustering in wireless networks and is maintained for minimizing the number of required resource centers; see [4]. The MDS problem is that of finding a minimal dominating set in any given graph G. A survey for the self-stabilizing algorithms of DS, MDS, and other related problems can be found in [3].

The time complexity of proposed algorithms of this paper is measured by the numbers of moves they perform. The number of moves is well-concerned in wireless networks with bounded resources. After taking a move, the state of a node is changed, and then the node informs every adjacent nodes of its new state. Since communication is the chief action to consume energy, a reduction of the number of moves prolongs the lifetime of a network. For convenience, if a distributed algorithm takes at most t moves to stabilize, then we say it is a t-move algorithm, where t can be a function of n and other parameters of the graph.

It is challenging to develop an algorithm that takes fewer moves than the best known result: $5n$ moves under an unfair distributed daemon. The main contribution of this paper is to propose a $4n$-move, self-stabilizing algorithm for the MDS problem under the unfair distributed daemon. Our algorithm requires the local distinct identification property, that is, two adjacent nodes must have distinct identifiers. In particular, it is desired that an MDS algorithm is MDS-silent (meaning that if the original configuration is already an MDS, then the algorithm should not take any move). Unfortunately, it is impossible for an algorithm to be MDS-silent if only distance-1 knowledge is accessible. Hence, in this paper, we propose a $2n$-move algorithm by using distance-2 knowledge. We summarize all the known results in Table 1.

Table 1. Self-Stabilizing algorithms for the minimal dominating set problem

	stabilization time	scheduler type
Hedetniemi et al. [5]	$(2n+1)n$ moves	central
Xu et al. [7]	$4n$ rounds	synchronous
Turau [6]	$9n$ moves	distributed
Goddard et al. [2]	$5n$ moves	distributed
this paper (in Sec. 3)	$4n$ moves	distributed
this paper (in Sec. 4)	$2n$ moves	distributed

This paper is organized as follow. In Section 2, we formally give definitions of the graph model and self-stabilization, and describe the previous results. In Section 3, we propose a self-stabilizing algorithm for the MDS problem. In Section 4, we discuss the design of a self-stabilizing MDS-silent algorithm by using distance-2 knowledge. Concluding remarks are given in Section 5.

2 Preliminary

Let $V = \{i \mid 1 \leq i \leq n\}$ be a set of processes and graph $G = (V, E)$ be represented as the topology of the distributed system. Let v be a node. A node u is a neighbor of v if they are adjacent. We assume that the graph G is undirected and simple. Note that we use the terms nodes and processes interchangeably.

2.1 Self-stabilizing

Let Ω_i be the set of all possible combinations of local variables of the processer $i \in V$. Each element Q_i in Ω_i denotes a *state* of process i. A tuple of states of processes $(Q_1, Q_2, \ldots, Q_n)$ forms a *configuration* of a distributed system. Let Γ be the set of configurations of G. For any configuration γ_t at time t, let γ_{t+1} be a configuration that follows γ_t. Denote the transition relation by $\gamma_t \to \gamma_{t+1}$. A *computation sequence* starting from γ_0 is an infinite sequence of configurations $\gamma_0, \gamma_1, \ldots$ such that $\gamma_t \to \gamma_{t+1}$ for each $t \geq 0$.

Let Γ be the set of configurations in a distributed system S. The system S is *self-stabilizing* with respect to $\Lambda \subseteq \Gamma$ if the following conditions hold: (Convergence) starting from an arbitrary configuration $\gamma_0 \in \Gamma$, there exists an integer t such that $\gamma_t \in \Lambda$ in any computation sequence; (Closure) for every configuration $\lambda \in \Lambda$, any configuration that follows λ is also in Λ. Each $\lambda \in \Lambda$ is called a *legitimate* configuration.

In a graph algorithm such as the MDS problem, the states of processes are categorized into two cases. The first case is state `in`, with which the process is considered in the MDS set S with the desired property. On the other hand, if the process is considered not in S, therefore we say it has state `out`. The nodes are referred to as `in` nodes and `out` nodes according to their states.

2.2 Previous Results

In [5], Hedetniemi et al. proposed the first MDS self-stabilizing algorithms. Under a central daemon, the configuration of a graph stables in $O(n^2)$ moves. In their MDS algorithm, every node has two variables: a Boolean indicating its state is `in` or `out`, and a pointer pointing one of its neighbors. For convenience, the neighbors of state `in` are called `in` neighbors. A node will point to the unique `in` neighbor, otherwise it will point to null. A node is allowed to enter the MDS if it has no `in` neighbor. In contrast, a node will leave the MDS if it has at least one `in` neighbor and there is no neighbor pointing to it.

In [7], Xu et al. presented a self-stabilizing algorithm for the MDS problem using unique identifiers under the synchronous daemon. The stabilization time is $O(n)$. Like Hedetniemi's algorithm, every node has two variables: a Boolean variable and a pointer. A node will point (i) to the unique `in` neighbor, (ii) to itself if it has no `in` neighbor, or (iii) to null if it has more than one `in` neighbor. A node will enter the MDS if it has no `in` neighbor and it has the smallest identifier within its closed neighborhood. A node will leave the MDS under the same condition as in Hedetniemi's algorithm.

In [6], Turau proposed a linear-time self-stabilizing algorithm (called `Turau3n`) for MIS problem with the unique identifier assumption. Every node has a variable that may have one of three different values: `in` (in the set), `out` (out of the set), or `wait` (an `out` node with no `in` neighbor, waiting to join the set). So, `Turau3n` goes as follows: an `out` node that has no neighbor in the MIS will first change its variable to `wait`. After doing so, the node may change its variable to `in` if it has no neighbor with a lower identifier in the `wait` variable. Also, an `in` node may leave the MIS and change its variable to `out` if it has an `in` neighbor.

Based on `Turau3n`, Turau extended the rules to design the first self-stabilizing MDS algorithm `Turau9n`. Each node has two variables. The first three-valued variable s is defined as the one in `Turau3n`. The second is a pointer variable p. An `in` node changes p to null. An `out` node changes p to null if there is more than one `in` neighbor, or to the only one `in` neighbor. The entering rule is the same as in `Turau3n`. Besides, the rule for leaving is modified by adding the precondition "there is no `in` neighbor pointing to it". Turau's MDS algorithm `Turau9n` is a $9n$-move algorithm.

In [2], Goddard et al. proposed a $5n$-move algorithm `Goddard5n` for the MDS problem with nodes having locally distinct identifiers under a distributed daemon. In detail, each node has a Boolean variable s and a three-valued variable c indicating whether it belongs to MDS and counting the number of `in` neighbors.

A node is allowed to join the MDS if it has no `in` neighbor, its counter c equals to 0, and it has no lower identifier neighbor having $c = 0$. On the other hand, a node is allowed to leave the MDS if there is an `in` neighbor and every `out` neighbor has $c = 2$, which means they all have more than one `in` neighbor.

3 Main Result

The purpose of this section is to present our main result: `Well4n`, a $4n$-move self-stabilizing algorithm for the MDS problem under an unfair distributed daemon. We assume that each node has a locally distinct identifier. `Well4n` uses four states, which is defined by the four-valued variable *state*. The range of values of *state* is: `IN`, `OUT1`, `OUT2`, and `OUT0`. A node with $state = \texttt{IN}$ will be referred to as an `in` node. Let $S = \{v : v.state = \texttt{IN}\}$; i.e., S is the set of `in` nodes. A node with $state = \texttt{OUT1}$ or $state = \texttt{OUT2}$ or $state = \texttt{OUT0}$ will be referred to as an `out` node. A neighbor is an `in` (resp., `out`) neighbor if it is an `in` (resp., `out`) node.

The values of *state* have the following meanings. The value `IN` indicates that the node is in the MDS. The value `OUT1` means that the node is not in the MDS and it has a unique `in` neighbor. The value `OUT2` indicates that the node is not in the MDS and it has at least two `in` neighbors. The value `OUT0` means that the node is not in the MDS and it does not have any `in` neighbor.

3.1 The First Algorithm `Well4n`

Let $N(v)$ denote the set of neighbors of node v, $N[v]$ denote $N(v) \cup \{v\}$, and $v.id$ denote the identifier of v. To formally define the rules of `Well4n`, the following predicates defined for each node v are needed:

- $noInNbr \equiv \nexists\, w \in N(v) : w.state = \texttt{IN}$.
- $oneInNbr \equiv \exists\, unique\ w \in N(v) : w.state = \texttt{IN}$.
- $twoInNbr \equiv \exists\, at\ least\ two\ w \in N(v) : w.state = \texttt{IN}$.
- $noBtNbr \equiv \nexists\, w \in N(v) : w.state = \texttt{OUT0} \wedge w.id < v.id$.
- $noDpNbr \equiv \nexists\, w \in N(v) : w.state = \texttt{OUT1}$.

The meaning of the predicates *noInNbr* and *oneInNbr* is straightforward; *twoInNbr* indicates whether v has two or more `in` neighbors. To make `Well4n` easier to understand, we will not replace *oneInNbr* with $\neg noInNbr \wedge \neg twoInNbr$. We now explain the meaning of *noBtNbr* and *noDpNbr*. When two or more neighboring nodes want to enter the MDS simultaneously, our algorithm chooses the one with the smaller (smallest) *id*. According to this, if v is an `out` node and has a neighbor w such that $w.state = \texttt{OUT0}$ and $w.id < v.id$, then w is called a *better neighbor*. The predicate *noBtNbr* indicates that v has no better neighbor. Also, if v is an `in` node and has a neighbor w with $w.state = \texttt{OUT1}$, then w is called a *dependent neighbor* since w depends on its unique neighbor in the MDS (the neighbor is v). The predicate *noDpNbr* indicates that v has no dependent neighbor.

`Well4n` uses the following six rules and its state diagram is given in Figure 1.

R1. $state = \texttt{OUT0} \ \wedge \ noInNbr \ \wedge \ noBtNbr \rightarrow state := \texttt{IN}$.
R2. $state = \texttt{IN} \ \wedge \ oneInNbr \ \wedge \ noDpNbr \rightarrow state := \texttt{OUT1}$.
R3. $state = \texttt{IN} \ \wedge \ twoInNbr \ \wedge \ noDpNbr \rightarrow state := \texttt{OUT2}$.
R4. $state = \texttt{OUT0} \ \wedge \ oneInNbr \rightarrow state := \texttt{OUT1}$.
R5. $(state = \texttt{OUT1} \ \vee \ state = \texttt{OUT0}) \ \wedge \ twoInNbr \rightarrow state := \texttt{OUT2}$.
R6. $(state = \texttt{OUT1} \ \vee \ state = \texttt{OUT2}) \ \wedge \ noInNbr \rightarrow state := \texttt{OUT0}$.

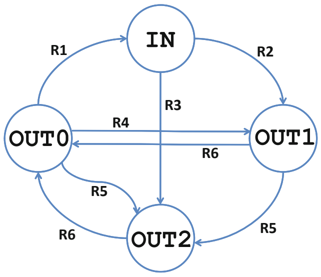


Fig. 1. The state diagram of `Well4n`

3.2 Correctness and Convergence

We now prove the correctness of `Well4n`.

Lemma 1. *In any configuration in which no node is privileged, the set S is a minimal dominating set for G.*

Proof. Suppose to the contrary that S is not a minimal dominating set for G. Then either (i) S is not a dominating set or (ii) S is a dominating set but not minimal. First consider (i). Since S is not a dominating set, there exists at least one node $u \notin S$ which has no `in` neighbor; let S' be the set of all such nodes. Since rule 6 is not enabled, every node in S' has $state = \texttt{OUT0}$. Let u_0 be the node in S' with minimum id. Then u_0 satisfies all the constraints of rule 1. Hence, rule 1 is enabled and this contradicts to the assumption that no node is privileged. Now consider (ii). Since S is a dominating set but not minimal, there must exist at least one node $u \in S$ such that $S \backslash \{u\}$ is also a dominating set for G. Then $|N(u) \cap S| \geq 1$ and for all u' in $N(u) \backslash S$, we have $|N(u') \cap S| \geq 2$. Thus, every node u' in $N(u) \backslash S$ has $twoInNbr = true$. Hence, every node u' in $N(u) \backslash S$ must have $u'.state = \texttt{OUT2}$; otherwise rule 5 is enabled on u'. Consequently, node u has $noDpNbr = true$ and either $oneInNbr = true$ (if $|N(u) \cap S| = 1$) or $twoInNbr = true$ (if $|N(u) \cap S| > 1$). Hence, either rule 2 or rule 3 is enabled on node u, which is a contradiction. □

For convenience, use `Goddard5n` to denote the $5n$-move algorithm given in [2]. We now show that `Well4n` converges faster than `Goddard5n`. In particular, we will show that the number of moves of `Well4n` is at most $4n$. Let k be a nonnegative integer and $\langle r_1, r_2, \ldots, r_k \rangle$ be a sequence of rules (r_i's are not necessarily distinct). The sequence $\langle r_1, r_2, \ldots, r_k \rangle$ is called a *move sequence* if a node can execute rule r_1, then rule r_2, ..., then rule r_k. The following two lemmas show that in any possible move sequence of a specific node, rule 1 and rule 6 will appear at most once.

Lemma 2. *If a node executes rule 1, then it will not execute any other rule. Consequently, if a node enters the set S, then it will never leave S.*

Proof. Let v be a node which executes rule 1. Then $v.state$ is set to `IN` and v enters S. By the precondition of rule 1, v has no `in` neighbor and no better neighbor; therefore no neighbor of v enters S at the same time. Thus, v is the only node in $N[v]$ that enters S and therefore both $oneInNbr$ and $twoInNbr$ are $false$. After executing rule 1, $v.state$ is `IN` and the possible rule that v can execute is either rule 2 or rule 3. Rule 2 is impossible since it requires $oneInNbr = true$; rule 3 is also impossible since it requires $twoInNbr = true$. Therefore, v will not execute any other rule. The second statement of this lemma now follows. □

Lemma 3. *A node can execute rule 6 at most once, or equivalently, a node can set its state to* `OUT0` *at most once.*

Proof. Let v be a node which executes rule 6. By the precondition of rule 6, v has no `in` neighbor. After executing rule 6, $v.state$ is set to `OUT0` and the possible rule that v can execute is rule 1 or rule 4 or rule 5. If v executes rule 1, then by Lemma 2, v will not execute any other rule and we have this lemma. If v executes rule 4, then $oneInNbr$ must be $true$ before rule 4 is enabled, meaning that v has a neighbor (say, u) which has executed rule 1; by Lemma 2, u will never leave S and therefore it is impossible to have $noInNbr = true$, which means that v could not execute rule 6 again. Similarly, if v executes rule 5, then $twoInNbr$ must be $true$ before rule 5 is enabled, meaning that v has two neighbors (say, u and w) which have executed rule 1; by Lemma 2, both u and w will never leave S and therefore it is impossible to have $noInNbr = true$, which means that v could not execute rule 6 again. □

Theorem 1. *The proposed algorithm* `Well4n` *is self-stabilizing under an unfair distributed daemon and it stabilizes after at most $4n - 2$ moves with a minimal dominating set, where n is the number of nodes. Moreover, the bound $4n - 2$ is tight.*

Proof. By Lemma 1, `Well4n` is correct. It suffices to show that any move sequence of a node is of length at most 4 under an unfair distributed daemon. Let v be an arbitrary node in G. By Lemma 3, v can execute rule 6 at most once. Thus, there are two cases: v never executes rule 6 and v executes rule 6 once.

First consider the case that v never executes rule 6. Then $v.state$ never changes to `OUT0`. Thus, the move sequence of v is either $\langle 1 \rangle$ or $\langle 2, 5 \rangle$ or $\langle 4, 5 \rangle$. It follows that any move sequence of v is of length at most 2.

Now consider the case that v executes rule 6 once. In this case, regard a move sequence of v as the concatenation of a prefix and a suffix. By Lemma 2, the prefix of any move sequence of v cannot contain 1 since if v executes rule 1 then v will not execute any other rule, including rule 6. Hence, the possible prefix of any move sequence of v is either $\langle 2, 6\rangle$ or $\langle 3, 6\rangle$ or $\langle 4, 6\rangle$ or $\langle 5, 6\rangle$. After v executes rule 6, $v.state$ changes to `OUTO`. Thus, the possible suffix of any move sequence of v is either $\langle 6, 1\rangle$ or $\langle 6, 4, 5\rangle$ or $\langle 6, 5\rangle$. Concatenating the prefix and suffix, we conclude that any move sequence of v is of length at most 4.

Thus algorithm `Well4n` stabilizes after at most $4n$ moves with a minimal dominating set. We now prove that the bound can be strengthen to $4n-2$. The cases of $n = 1$ and $n = 2$ are trivial. Suppose $n \geq 3$ and one of the processes makes four moves. By the above argument, this process has two neighbors executing rule 1. Thus, at least two processes in G make less than 4 moves. Hence, the upper bound of the number of moves is $4n - 2$.

We now prove that the upper bound $4n - 2$ is tight. Consider the complete bipartite graph $K_{2,n-2}$, $n \geq 3$. Let the two nodes in the partite of cardinality two have the maximum and the minimum identifiers among the n nodes. If initially all nodes are in state `IN`, then there is a way that all the rest of the nodes executes $< 3, 6, 4, 5 >$ but nodes with maximum and minimum identifiers execute $< 3, 6, 1 >$. All together $4n - 2$ moves are made. □

4 An MDS-Silent Algorithm

The purpose of this section is to discuss the development of a self-stabilizing MDS-silent algorithm. We begin with several definitions. A subset $S \subseteq V$ of nodes in a graph G is called *independent* if no two nodes in S are adjacent. An independent set S is a *maximal independent set* (MIS) if it is not a proper subset of any independent set. Let k be a positive integer. A distributed system is called *distance-k information system* if every process can access its k-neighborhood knowledge, i.e., the state information of its distance-1 neighbors, distance-2 neighbors, ..., distance-k neighbors [9]. Clearly, a distance-$(k + 1)$ system is also a distance-k system.

4.1 MDS-Silent Algorithms in Distance-1 Information Systems

It is well known that a maximal independent set is also a minimal dominating set. Moreover, the stabilizing time of a self-stabilizing MIS algorithm is usually less than that of a self-stabilizing MDS algorithm. Thus, why bother to develop a self-stabilizing MDS algorithm? In [6], Turau mentioned:

> Since it is desirable that a self-stabilizing algorithm initialized with a minimal dominating set does not make any moves, MIS-algorithms are not suitable solutions for the MDS problem.

This characterizes an important feature of a self-stabilizing MDS algorithm: not to make any move if the system is initialized with a minimal dominating set.

We call this important feature *MDS-silent*. The notion of MIS-silent can be defined similarly. We have a lemma.

Lemma 4. *Any self-stabilizing MIS algorithm is a self-stabilizing MDS algorithm, but not necessarily* MDS-silent.

Proof. It is clear that any self-stabilizing MIS algorithm is a self-stabilizing MDS algorithm. Let G be a path of four nodes v_1, v_2, v_3, v_4 and edges (v_1, v_2), (v_2, v_3), (v_3, v_4). Suppose initially v_2, v_3 are `in` nodes and v_1, v_4 are `out` nodes. Since $\{v_2, v_3\}$ is not an MIS, any self-stabilizing MIS algorithm will make a move. However, $\{v_2, v_3\}$ is an MDS; thus if the algorithm is MDS-silent, then it should not make any move. Therefore, a self-stabilizing MIS algorithm may not be MDS-silent. □

Unfortunately, unlike self-stabilizing MIS algorithms in [2], which is MIS-silent, none of the self-stabilizing MDS algorithms `Turau9n`, `Goddard5n`, and `Well4n` is MDS-silent. All of `Turau9n`, `Goddard5n`, and `Well4n` assume distance-1 knowledge. We now show that a distance-1 information system could not have a self-stabilizing MDS-silent algorithm.

Lemma 5. *If a distributed system is not a distance-*2 *information system, then there exists no self-stabilizing* MDS-silent *algorithm even if every node has a unique identifier.*

Proof. Suppose this lemma is not true and there exists a self-stabilizing MDS-silent algorithm $\mathcal{A}$ for a distance-1 information system. For convenience, let I and O denote the state `in` and `out`, respectively. Run algorithm $\mathcal{A}$ on the following three graphs:
G_1: a path of 4 nodes with initial configuration $OIIO$,
G_2: a path of 4 nodes with initial configuration $OIOI$,
G_3: a path of 5 nodes with initial configuration $OIIOI$.

Since the initial configuration of G_1 is legitimate, $\mathcal{A}$ will not make any move on G_1. Since the initial configuration of G_2 is also legitimate, $\mathcal{A}$ will not make any move on G_2. Now consider G_3. Since nodes in the system have distance-1 knowledge only, $\mathcal{A}$ cannot distinguish between the first three nodes of G_3 and the first three nodes of G_1, and $\mathcal{A}$ cannot distinguish between the last two nodes of G_3 and the last two nodes of G_2. Hence, for G_3, algorithm $\mathcal{A}$ stabilizes with the initial configuration $OIIOI$, which is not an MDS, a contradiction. □

4.2 Message Passing Model of Distance-2 Information Systems

A distance-2 information system can be implemented on an ad hoc network by using the beacon messages and the neighbor list messages of a node to inform neighbors of its continued presence and the change of the local state of its neighborhood.

A distance-2 information system assumes the following about the system. A link-layer protocol at each node v maintains the identifiers and states of its neighbors in the neighbor list $nbl(v)$. Furthermore, after exchange the neighbor lists, each node v constructs the 2-neighbor list 2-$nbl(v)$ which contains the identifies and states of 2-neighbors of v. In detail, each node periodically broadcasts a *beacon* message. When a neighboring node u sends a beacon message, it includes the state of the node u as used in the algorithm. A beacon message provides information about its neighbor nodes synchronously, and a node takes action after receiving beacon messages from all neighboring nodes. When node v receives the beacon signal from a neighbor u which is not in $nbl(v)$, it adds u to its neighbor list to establishing link (u, v). After nodes v updating its neighbor list, node v broadcasts $nbl(v)$. All nodes in the neighborhood $N(v)$ know the existence of link (u, v) according the $nbl(v)$ message and update their 2-neighbor lists. In the meantime, node v updates 2-$nbl(v)$ according the $nbl(u)$ message.

If node v does not receive the beacon signal from u within a fixed period, it assumes the link (u, v) is no longer available and removes u from both $nbl(v)$ and 2-$nbl(v)$. A node takes action only after receiving beacon messages or neighbor list messages from its neighboring nodes.

4.3 An MDS-Silent Algorithm in Distance-2 Information Systems

Now we propose a self-stabilizing MDS-silent algorithm `Well2n` under an unfair distributed daemon in a distance-2 information system. We assume that each node has a unique identifier. `Well2n` uses a two-valued variable *state*, which has values `in` and `out`. We use the same terminology as in Section 3 except as indicated.

To formally define the rules of `Well2n`, three predicates defined for each node v are needed:

- $noInNbr \equiv \nexists w \in N(v) : w.state = \texttt{in}$.
- $noBtNbr_2 \equiv \nexists w \in N(v) : w.state = \texttt{out} \wedge w.id < v.id \wedge w$ has no `in` neighbor.
- $noDpNbr_2 \equiv \nexists w \in N(v) : w.state = \texttt{out} \wedge\ w$ has exactly one `in` neighbor.

Notice that $noBtNbr_2$ and $noDpNbr_2$ require distance-2 knowledge since v has to check its neighbor's neighbors' *state*. `Well2n` uses only two rules:

R1. $state = \texttt{out} \ \wedge\ noInNbr \ \wedge\ noBtNbr_2 \rightarrow state := \texttt{in}$.
R2. $state = \texttt{in} \ \wedge\ \neg noInNbr \ \wedge\ noDpNbr_2 \rightarrow state := \texttt{out}$.

Rule 1 is regarded as the entering rule and rule 2, the leaving rule. An MIS algorithm usually applies the following entering rule and leaving rule: “A node having no neighbor in S joins S and a node having a neighbor in S leaves S.” `Well2n` follows the entering rule but modifies the leaving rule to be: “A node leaves S if (i) it has a neighbor in S and (ii) every neighbor is either in S or has at least two neighbors in S.” We now prove the correctness of `Well2n`.

Lemma 6. *In any configuration in which no node is privileged, the set S is a minimal dominating set of G.*

Proof. Suppose to the contrary that S is not a minimal dominating set for G. Then either (i) S is not a dominating set or (ii) S is a dominating set but not minimal. Consider (i). Since S is not a dominating set, there exists an `out` node having no `in` neighbor. Let u be such a node with the minimum *id*. Since u has no `in` neighbor and no better neighbor, rule 1 is enabled, which is a contradiction. Now consider (ii). Since S is a dominating set but not minimal, there must exist at least one node $u \in S$ such that $S \backslash \{u\}$ is also a dominating set for G. Then $|N(u) \cap S| \geq 1$ and for all u' in $N(u) \backslash S$, $|N(u') \cap S| \geq 2$. Since both $\neg noInNbr$ and $noDpNbr$ are $true$ for u, rule 2 is enabled on node u, which is a contradiction. □

The lemma below shows that `Well2n` is MDS-silent.

Lemma 7. `Well2n` *will not make any move if the initial configuration is an MDS.*

Proof. Suppose the initial configuration is an MDS. Let S be the set of nodes with $state$ = `in`. First consider an arbitrary node u in S. The only rule can be enabled on u is rule 2. Since S is an MDS, it is impossible that u has an `in` neighbor but has no dependent neighbor; otherwise $S \backslash \{u\}$ is also a dominating set for G. Thus, at least one of $\neg noInNbr$ and $noDpNbr$ is $false$ and rule 2 cannot be enabled on node u. Next consider an arbitrary node u' not in S. The only rule that can be enabled on u' is rule 1. Since S is an MDS, it is impossible that u' has no `in` neighbor. Thus, $noInNbr$ is $false$ and rule 1 cannot be enabled on node u'. We have this lemma. □

Lemma 8. *If a node executes rule 1, then it enters S and will never leave S afterward. Furthermore, neighbors of this node will not enter S.*

Proof. Let i be a node that executes rule 1. At this moment all neighbors of i have $state$ = `out` and are with a higher *id* than i. After i enters S, its neighbors have at least one `in` neighbor and therefore cannot execute rule 1. The neighbors of i also cannot execute rule 2 since they have $state$ = `out`. The only rule that i can execute next is rule 2, but in order to do so, one of i's neighbors must have $state$ = `in`. As long as i has $state$ = `in`, this is impossible. We have this lemma. □

Theorem 2. *The algorithm* `Well2n` *is self-stabilizing and* MDS-silent *under an unfair distributed daemon. It stabilizes after at most $2n-1$ moves with a minimal dominating set, where n is the number of nodes.*

Proof. The theorem follows from Lemmas 6, 7, and 8. The "minus one" part comes from the fact that in a connected graph of order larger than 1, the size of any MDS is at most $n - 1$. □

To see $2n - 1$ in Theorem 2 is tight, consider the star graph of order n with initial state of each node to be `in`.

5 Concluding Remarks

It is challenging to design a self-stabilizing MDS using a distributed daemon that makes fewer than 5n moves. The previous best known algorithm under an unfair distributed daemon is $5n$-move [2]. In this paper we present an $4n$-move algorithm; there exists an example such that `Well4n` takes $4n - 2$ moves. In [9] and [8], the authors considered distance-2 information systems. In particular, [9] proposed a $n(k+1)$-move self-stabilizing minimal $\{k\}$-dominating set algorithm; when $k = 1$, the algorithm finds an MDS using at most $2n$ moves. The paper [8] presented a $2n$-move self-stabilizing minimal k-dominating set algorithm; when $k = 1$, this algorithm finds an MDS. The algorithms in both [9] and [8] operate with a central daemon. In this paper, we present an algorithm, which is also $2n$-move but under an unfair distributed daemon and hence is more practical. It is open to design self-stabilizing minimal $\{k\}$-dominating or k-dominating set algorithms under an unfair distributed daemon for $k > 2$. We also conjecture that if we relax the MDS-silent property to allow the execution of non-critical moves, then there will exist an MDS algorithm having the relaxed MDS-silent property.

References

1. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Communication of the ACM 17(11), 643–644 (1974)
2. Goddard, W., Hedetniemi, S.T., Jacobs, D.P., Srimani, P.K., Xu, Z.: Self-stabilizing graph protocols. Parallel Processing Letters 18(1), 189–199 (2008)
3. Guellati, N., Kheddouci, H.: A survey on self-stabilizing algorithms for independent, domination, coloring, and matching in graphs. Journal Parallel and Distributed Computing 70(4), 406–415 (2010)
4. Haynes, T.W., Hedetniemi, S.T., Slater, P.J.: Fundamentals of Domination in Graphs. Marcel Dekker (1998)
5. Hedetniemi, S.M., Hedetniemi, S.T., Jacobs, D.P., Srimani, P.K.: Self-stabilizing algorithms for minimal dominating sets and maximal independent sets. Computer Mathematics and Applications 46(5–6), 805–811 (2003)
6. Turau, V.: Linear self-stabilizing algorithms for the independent and dominating set problems using an unfair distributed scheduler. Information Processing Letters 103(3), 88–93 (2007)
7. Xu, Z., Hedetniemi, S.T., Goddard, W., Srimani, P.K.: A synchronous self-stabilizing minimal domination protocol in an arbitrary network graph. In: Das, S.R., Das, S.K. (eds.) IWDC 2003. LNCS, vol. 2918, pp. 26–32. Springer, Heidelberg (2003)
8. Turau, V.: Efficient transformation of distance-2 self-stabilizing algorithms. Journal of Parallel and Distributed Computing 72, 603–612 (2012)
9. Gairing, M., Goddard, W., Hedetniemi, S.T., Kristiansen, P., McRae, A.A.: Distance-two information in self-stabilizing algorithms. Parallel Processing Letters 14(3–4), 387–398 (2004)

Phase Transition of Random Non-uniform Hypergraphs

Élie de Panafieu*

Univ Paris Diderot, Sorbonne Paris Cité,
LIAFA, UMR 7089,
75013, Paris, France

Abstract. Non-uniform hypergraphs appear in several domains of computer science as in the satisfiability problems and in data analysis. We analyze their typical structure before and near the birth of the *complex* component, that is the first connected component with more than one cycle. The model of non-uniform hypergraph studied is a natural generalization of the *multigraph process* defined in the "giant paper" [1]. This paper follows the same general approach based on analytic combinatorics. We study the evolution of hypergraphs as their complexity, defined as the *excess*, increases. Although less natural than the number of edges, this parameter allows a precise description of the structure of hypergraphs. Finally, we compute some statistics of the hypergraphs with a given excess, including the expected number of edges.

Keywords: Hypergraph, phase transition, analytic combinatorics.

1 Introduction

In the seminal article [2], Erdös and Rényi discovered an abrupt change of the structure of a random graph when the number of edges reaches half the number of vertices. It corresponds to the emergence of the first connected component with more than one cycle, immediately followed by components with even more cycles. The combinatorial analysis of those components improves the understanding of the objects modeled by graphs and has application in the analysis and the conception of graph algorithm. The same motivation holds for hypergraphs which appear for example for the modelisation of databases and xor-formulas.

Much of the literature on hypergraphs is restricted to the uniform case, where all the edges contain the same number of vertices. In particular, the analysis of the birth of the complex component can be found in [3] and [4].

There is no canonical choice for the size of a random edge in a hypergraph; thus several models have been proposed. One is developed in [5], where the size of the largest connected component is obtained using probabilistic methods. In [6], Darling and Norris define the important Poisson random hypergraphs model and analyze its structure via fluid limits of pure jump-type Markov processes.

* This work was partially founded by the ANR Boole and the ANR Magnum.

© Springer-Verlag Berlin Heidelberg 2013

We have not found in the literature much use of the generating function of non-uniform hypergraphs to investigate their structure, and we intend to fill this gap. However, similar generating functions have been derived in [7] for a different purpose: Gessel and Kalikow use it to give a combinatorial interpretation for a functional equation of Bouwkamp and de Bruijn. The underlying hypergraph model is a natural generalization of the *multigraph process* defined in [1].

In section 2 we introduce the hypergraph models, the probability distribution and the corresponding generating functions. The important notion of *excess* is also defined. Section 3 is dedicated to the asymptotic number of hypergraphs with n vertices and excess k. Some statistics on the random hypergraphs are derived, including the expected number of edges. The critical excess at which the first complex component appears is obtained in section 4. For a range of excess near and before this critical value, we compute the probability that a random hypergraph contains no complex component. The classical notion of *kernel* is introduced for hypergraphs in section 5. It is then used to derive the asymptotic of connected hypergraphs with n vertices and excess k up to a multiplicative factor independent of n. Finally, we present in section 6 a surprising result: although the critical excess is generally different for graphs and hypergraphs, both models share the same structure at their respective critical excess.

2 Definitions

In this paper, a hypergraph G is a multiset $E(G)$ of $m(G)$ edges. Each edge e is a multiset of $|e|$ vertices in $V(G)$, where $|e| \geq 2$. The vertices of the hypergraph are labelled from 1 to $n(G)$. We also set $l(G)$ for the *size* of G, defined by

$$l(G) = \sum_{e \in E(G)} |e| = \sum_{v \in V(G)} \deg(v).$$

The notion of *excess* was first used for graphs in [8], then named in [1], and finally extended to hypergraphs in [9]. The excess of a connected component C expresses how far from a tree it is: C is a tree if and only if its excess is -1 and is said to be *complex* if its excess is strictly positive. Intuitively, a connected component with high excess is "hard" to treat for a backtracking algorithm. The excess $k(G)$ of a hypergraph G is defined by

$$k(G) = l(G) - n(G) - m(G).$$

A hypergraph may contain several copies of the same edge and a vertex may appear more than once in an edge; thus we are considering multihypergraphs. A hypergraph with no loop nor multiple edge is said to be *simple*. Let us recall that a sequence is by definition an ordered multiset. We define $\mathrm{NumbSeq}(G)$ as the number of sequences of nonempty sequences of vertices that lead to G. For example, the sequences $(1,2),(2,3)$ and $(3,2),(1,2)$ represent the same hypergraph, but not $(2,1),(1,3)$. If G is simple, then $\mathrm{NumbSeq}(G)$ is equal to $m(G)! \prod_{e \in E(G)} |e|!$, otherwise it is smaller. We associate to any family $\mathcal{F}$ of hypergraphs the generating function

$$F(z,w,x) = \sum_{G \in \mathcal{F}} \frac{\text{NumbSeq}(G)}{m(G)!} \Big(\prod_{e \in E(G)} \frac{\omega_{|e|}}{|e|!} \Big) w^{m(G)} x^{l(G)} \frac{z^{n(G)}}{n(G)!} \qquad (1)$$

where ω_t marks the edges of size t, w the edges, x the size of the graph and z the vertices. Therefore, we count hypergraphs with a *weight* κ

$$\kappa(G) = \frac{\text{NumbSeq}(G)}{m(G)!} \prod_{e \in E(G)} \frac{\omega_{|e|}}{|e|!} \qquad (2)$$

that is the extension to hypergraphs of the *compensation factor* defined in section 1 of [1]. If $\mathcal{F}$ is a family of simple hypergraphs, then we obtain the simpler and natural expression

$$F(z,w,x) = \sum_{G \in \mathcal{F}} \Big(\prod_{e \in E(G)} \omega_{|e|} \Big) w^{m(G)} x^{l(G)} \frac{z^{n(G)}}{n(G)!}. \qquad (3)$$

Remark that the generating function of the subfamily of hypergraphs of excess k is $[y^k] F(z/y, w/y, xy)$, where $[x^n] \sum_k a_k x^k$ denotes the coefficient a_n.

We define the exponential generating function of the edges as

$$\Omega(z) := \sum_{t \geq 1} \omega_t \frac{z^t}{t!}.$$

For now on, the (ω_t) are considered as a bounded sequence of nonnegative real numbers with $\omega_0 = \omega_1 = 0$. The value ω_t represents how likely an edge of size t is to appear. Thus, for graphs we get $\Omega(z) = z^2/2$, for d-uniform hypergraphs $\Omega(z) = z^d/d!$, and for hypergraphs with weight 1 for all size of edge $\Omega(z) = e^z$. To simplify the saddle point proofs, we also suppose that $\Omega(z)/z$ cannot be written as $f(z^d)$ for an integer $d > 1$ and a power serie f with a non-zero radius of convergence. This implies that $e^{\Omega(z)/z}$ is aperiodic. Therefore, we do not treat the important, but already studied case of uniform hypergraphs.

The generating function of all hypergraphs is

$$\text{hg}(z,w,x) = \sum_n e^{w\,\Omega(nx)} \frac{z^n}{n!}. \qquad (4)$$

This expression can be derived from (1) or using the symbolic method presented in [10]. Indeed, $\Omega(nx)$ represents an edge of size marked by x and n possible types of vertices, and $e^{w\,\Omega(nx)}$ a set of edges. For the family of simple hypergraphs,

$$\text{shg}(z,w,x) = \sum_n \left(\prod_t (1 + \omega_t x^t w)^{\binom{n}{t}} \right) \frac{z^n}{n!}. \qquad (5)$$

Similar expressions have been derived in [7]. The authors use them to give a combinatorial interpretation of a functional equation of Bouwkamp and de Bruijn.

Comparing (1) with (3), simple hypergraphs may appear more natural than hypergraphs. But their generating function is more intricated and the asymptotics results on hypergraphs can often be extended to simple hypergraphs. This is another reason not to confine our study to simple hypergraphs.

So far, we have adopted an enumerative approach of the model, but there is a corresponding probabilistic description. We define $\mathrm{HG}_{n,k}$ (resp. $\mathrm{SHG}_{n,k}$) as the set of hypergraphs (resp. simple hypergraphs) with n vertices and excess k, and equippe it with the probability distribution induced by the weights (2). Therefore, the hypergraph G occurs with probability $\kappa(G)/\sum_{H\in\mathrm{HG}_{n,k}}\kappa(H)$.

3 Hypergraphs with n Vertices and Excess k

In this section, we derive the asymptotic of hypergraphs and simple hypergraphs with n vertices and global excess k. This result is interesting by itself and is a first step to find the excess k at which the first component with strictly positive excess is likely to appear.

Theorem 1. *Let λ be a strictly positive real value and $k=(\lambda-1)n$, then the sum of the weights of the hypergraphs in* $\mathrm{HG}_{n,k}$ *is*

$$\mathrm{hg}_{n,k}\sim\frac{n^{n+k}}{\sqrt{2\pi n}}\frac{e^{\frac{\Omega(\zeta)}{\zeta}n}}{\zeta^{n+k}}\frac{1}{\sqrt{\zeta\,\Omega''(\zeta)-\lambda}}$$

where $\Psi(z)$ denotes the function $\Omega'(z)-\frac{\Omega(z)}{z}$ and ζ is defined by $\Psi(\zeta)=\lambda$. A similar result holds for simple hypergraphs:

$$\mathrm{shg}_{n,k}\sim\frac{n^{n+k}}{\sqrt{2\pi n}}\frac{e^{\frac{\Omega(\zeta)}{\zeta}n}}{\zeta^{n+k}}\frac{\exp\left(-\frac{\omega_2^2\zeta^2}{4}-\frac{\zeta\,\Omega''(\zeta)}{2}\right)}{\sqrt{\zeta\,\Omega''(\zeta)-\lambda}}.$$

More precisely, if $k=(\lambda-1)n+xn^{2/3}$ where x is bounded, then the two previous asymptotics are multiplied by a factor $\exp\left(\frac{-x^2}{2(\tau\,\Omega''(\tau)-\lambda)}n^{1/3}\right)$.

Proof. With the convention (1), the sum of the weights of the hypergraphs with n vertices and excess k is

$$n![z^ny^k]\,\mathrm{hg}(z/y,1/y,y)=n![z^ny^k]\sum_n e^{\frac{\Omega(ny)}{y}}\frac{(z/y)^n}{n!}=n^{n+k}[y^{n+k}]e^{\frac{\Omega(y)}{y}n}.$$

The asymptotic is then extracted using the *large power* scheme presented in [10]. Remark that $\Psi(z)=\sum_t\omega_t(t-1)\frac{z^{t-1}}{t!}$ has nonnegative coefficients, so there is a unique solution of $\Psi(\zeta)=\lambda$, and that $\Psi(\zeta)=\lambda$ implies $\zeta\,\Omega''(\zeta)-\lambda>0$. For simple hypergraphs, the coefficient we want to extract from (5) is now

$$[y^{n+k}]\prod_t(1+\omega_t y^{t-1})^{\binom{n}{t}}=\frac{n^{n+k}}{2i\pi}\oint\exp\left(\sum_t\binom{n}{t}\log\left(1+\omega_t\left(\frac{y}{n}\right)^{t-1}\right)\right)\frac{dy}{y^{n+k+1}}.$$

The sum in the exponential can be rewritten

$$\frac{\Omega(y)}{y}n+\sum_t \binom{n}{t}\left(\log(1+\omega_t\left(\frac{y}{n}\right)^{t-1})-\omega_t\left(\frac{y}{n}\right)^{t-1}\right)-\left(\frac{n^t}{t!}-\binom{n}{t}\right)\omega_t\left(\frac{y}{n}\right)^{t-1}$$

which is $\frac{\Omega(y)}{y}n - \frac{\omega_2^2 y^2}{4} - \frac{y\,\Omega''(y)}{2} + \mathcal{O}(1/n)$ when y is bounded (we use here the hypothesis that $\omega_0 = \omega_1 = 0$). In the saddle point method, y is close to ζ, which in our case is fixed with respect to n. Therefore,

$$n![z^n y^k]\,\mathrm{shg}\left(\frac{z}{y},\frac{1}{y},y\right) \sim \exp\left(-\frac{\omega_2^2\zeta^2}{4}-\frac{\zeta\,\Omega''(\zeta)}{2}\right)\mathrm{hg}_{n,k}\,.$$

The factor $\exp\left(-\frac{\omega_2^2\zeta^2}{4}-\frac{\zeta\,\Omega''(\zeta)}{2}\right)$ is the asymptotic probability for a hypergraph in $\mathrm{HG}_{n,k}$ to be simple. For graphs, with $\Omega(z)=z^2/2$ and $\lambda=1/2$, we obtain the same factor $e^{-3/4}$ as in [1].

We study the evolution of hypergraphs as their excess increases. This choice of parameter is less natural than the number of edges, but it significantly simplifies the equations. On the other hand, we can compute statistics on the number of edges of hypergraphs with n edges and excess k.

Corollary 1. *Let λ be a positive value and G a random hypergraph in $\mathrm{HG}_{n,k}$ or in $\mathrm{SHG}_{n,k}$ with $k=(\lambda-1)n$, then the asymptotic expectations and factorial moments of the number m of edges and size l of G are*

$$\mathbb{E}_{n,k}(m) \sim \tfrac{\Omega(\zeta)}{\zeta}n,$$
$$\forall t\geq 0,\ \mathbb{E}_{n,k}\big(m(m-1)\dots(m-t)\big) \sim \left(\tfrac{\Omega(\zeta)}{\zeta}n\right)^{t+1},$$
$$\mathbb{E}_{n,k}(l) \sim \Omega'(\zeta)n.$$

where $\Psi(z)$ denotes the function $\Omega'(z)-\frac{\Omega(z)}{z}$ and ζ is the solution of $\Psi(\zeta)=\lambda$.

Reversely, the expectation and variance of the excess k of a random hypergraph with n vertices and m edges are

$$\mathbb{E}_{n,m}(k) = nm\frac{\Omega'(n)}{\Omega(n)} - n - m,$$
$$\mathbb{V}_{n,m}(k) = \frac{nm}{\Omega(n)}\left(n\,\Omega''(n) - n\frac{\Omega'(n)^2}{\Omega(n)} + \Omega'(n)\right).$$

Proof. Let us recall that if p_t denotes the probability that a discret random variable X takes the value t and $f(z)=\sum_n p_n z^n$, then the expectation of X is $f'(1)$ and its kth factorial moment is $\mathbb{E}(X(X-1)\dots(X-k)) = \partial^{t+1}f(1)$. By extraction from (4), the generating functions of the hypergraphs with n vertices and excess k (resp. m edges) and of the simple hypergraphs in $\mathrm{SHG}_{n,k}$ are

$$\mathrm{hg}_{n,k}(w) = n^{n+k}[y^{n+k}]e^{w\frac{\Omega(y)}{y}n},$$
$$\mathrm{hg}_{n,m}(y) = \frac{\Omega(ny)^m}{y^{n+m}m!},$$
$$\mathrm{shg}_{n,k}(w) = n^{n+k}[y^{n+k}]e^{w\frac{\Omega(y)}{y}n}e^{-\frac{y\,\Omega''(y)}{2}w-\frac{\omega_2^2y^2}{4}w^2+\mathcal{O}(1/n)}$$

where w and y mark respectively the number of edges and the excess. Therefore, the probability generating function corresponding to the distribution of m is $\mathrm{hg}_{n,k}(w)/\,\mathrm{hg}_{n,k}(1)$, and similarly for k. The asymptotics are then derived as in the proof of theorem 1.

The variance of $m(G)$ for G in $\mathrm{HG}_{n,k}$ cannot be straightforward derived from this corollary, because the asymptotic approximations of the factorial moments cannot be summed. If more terms of the asymptotic expansion of the factorial moments are derived, this variance can be bounded. However, it varies greatly with the parameters (ω_t). For example, the variance for graphs is 0, since all the graphs with n vertices and excess k have exactly $k+n$ edges.

4 Birth of the Complex Component

Let us recall that a connected hypergraph is *complex* if its excess is strictly positive. In order to locate the global excess k at which the first complex component appears, we compare the asymptotic numbers of hypergraphs and hypergraphs with no complex component.

We follow the conventions established in [11]: a *walk* of a hypergraph G is a sequence $v_0, e_1, v_1, \dots, v_{t-1}, e_t, v_t$ where for all i, $v_i \in V(G)$, $e_i \in E(G)$ and $\{v_{i-1}, v_i\} \subset e_i$. A *path* is a walk in which all v_i and e_i are distinct. A walk is a cycle if all v_i and e_i are distinct, except $v_0 = v_t$. Connectivity, trees and rooted trees are then defined in the usual way.

A *unicycle component* is a connected hypergraph that contains exactly one cycle. We also define a *path of trees* as a path that contains no cycle, plus a rooted tree hooked to each vertex, except to the two ends of the path. It can equivalently be defined as an unrooted tree with two distinct marked leaves.

Lemma 1. *Let T, U, V and P denote the generating functions of rooted trees, unrooted trees, unicycle components and paths of trees, using the variable z to mark the number of vertices, then*

$$T(z) = z e^{\Omega'(T(z))}, \tag{6}$$

$$U(z) = T(z) + \Omega(T(z)) - T(z)\,\Omega'(T(z)), \tag{7}$$

$$V(z) = \frac{1}{2}\log\frac{1}{1 - T(z)\,\Omega''(T(z))}, \tag{8}$$

$$P(z) = \frac{\Omega''(T(z))}{1 - T(z)\,\Omega''(T(z))}. \tag{9}$$

Proof. Those expressions can be derived from the tools presented in [10]. Equation (6) means that a rooted tree is a vertex (the root) and a set of edges from which a vertex has been removed and the other vertices replaced by rooted trees. Equation (7) is a classical consequence of the dissymmetry theorem described in [12] and studied in [13]. It can be checked that $z\partial_z U = T$. Unicycle components are cycles of rooted trees, which implies (8).

Theorem 3 counts the hypergraphs with no complex component. A phase transition occurs when $\frac{k}{n}$ reaches the critical value $\varLambda - 1$, defined in the next theorem, which corresponds to the coalescence of two saddle points. To extract the asymptotics, we need the following general theorem, borrowed from [14] and adapted for our purpose (in the original theorem, $\mu = 0$). It is also close to the lemma 3 of [1].

Theorem 2. *We consider a generating function $H(z)$ with nonnegative coefficients and a unique isolated singularity at its radius of convergence ρ. We also assume that it is continuable in $\varDelta := \{z \mid |z| < R, z \notin [\rho, R]\}$ and there is a $\lambda \in]1;2[$ such that $H(z) = \sigma - h_1(1 - z/\rho) + h_\lambda(1 - z/\rho)^\lambda + \mathcal{O}((1 - z/\rho)^2)$ as $z \to \rho$ in $\varDelta$. Let $k = \frac{\sigma}{h_1} n + x n^{1/\lambda}$ with x bounded, then for any real constant μ*

$$[z^n]\frac{H^k(z)}{(1-z/\rho)^\mu} \sim \sigma^k \rho^{-n} \frac{1}{n^{(1-\mu)/\lambda}} (h_1/h_\lambda)^{(1-\mu)/\lambda} G\left(\lambda, \mu; \frac{h_1^{1+1/\lambda}}{\sigma h_\lambda^{1/\lambda}} x\right) \tag{10}$$

where $G(\lambda, \mu; x) = \frac{1}{\lambda\pi} \sum_{k\geq 0} \frac{(-x)^k}{k!} \sin\left(\pi \frac{1-\mu+k}{\lambda}\right) \varGamma\left(\frac{1-\mu+k}{\lambda}\right)$.

Proof. In the Cauchy integral that represents $[z^n]\frac{H^k(z)}{(1-z/\rho)^\mu}$ we choose for the contour of integration a positively oriented loop, made of two rays of angle $\pm\pi/(2\lambda)$ that intersect on the real axis at $\rho - n^{-1/\lambda}$, we set $z = \rho(1 - t n^{-1/\lambda})$

$$[z^n]\frac{H^k(z)}{(1-z/\rho)^\mu} \sim \frac{-\sigma^k \rho^{-n}}{2i\pi n^{(1-\mu)/\lambda}} \int t^{-\mu} e^{\frac{h_\lambda}{h_1} t^\lambda} e^{-x \frac{h_1}{\sigma} t} dt$$

The contour of integration comprises now two rays of angle $\pm\pi/\lambda$ intersecting at -1. Setting $u = t^\lambda h_\lambda/h_1$, the contour transforms into a classical Hankel contour, starting from $-\infty$ over the real axis, winding about the origin and returning to $-\infty$.

$$\frac{-\sigma^k \rho^{-n}}{2i\pi n^{(1-\mu)/\lambda}} \frac{1}{\lambda} (h_1/h_\lambda)^{(1-\mu)/\lambda} \int_{-\infty}^{(0)} e^u e^{-x u^{1/\lambda} h_1^{1+1/\lambda}/(\sigma h_\lambda^{1/\lambda})} u^{\frac{1-\mu}{\lambda}-1} du$$

Expanding the exponential, integrating termwise, and appealing to the complement formula for the Gamma function finally reduces this last form to (10).

Theorem 3. *Let $\mathrm{thg}_{n,k}$ denote the sum of the weights of the hypergraphs with no complex component, n vertices and global excess k. Let $\varPsi(z)$ denote the function $\Omega'(z) - \frac{\Omega(z)}{z}$, τ be implicitely defined by $\tau\,\Omega''(\tau) = 1$ and $\varLambda = \varPsi(\tau)$.*

If $k = (\lambda - 1)n + \mathcal{O}(n^{1/3})$ with $0 < \lambda < \varLambda$, and $\varPsi(\zeta) = \lambda$, then

$$\mathrm{thg}_{n,k} \sim \frac{n^{n+k}}{\sqrt{2\pi n}} \frac{e^{\frac{\Omega(\zeta)}{\zeta} n}}{\zeta^{n+k}} \frac{1}{\sqrt{\zeta\,\Omega''(\zeta) - \lambda}}. \tag{11}$$

If $k = (\varLambda - 1)n + x n^{2/3}$ where x is bounded, then $\mathrm{thg}_{n,k}$ is equivalent to

$$\frac{n^{n+k}}{\sqrt{2\pi n}} \frac{e^{\frac{\Omega(\tau)}{\tau} n}}{\tau^{n+k}} \frac{1}{\sqrt{1-\varLambda}} \sqrt{\frac{3\pi}{2}} e^{-\frac{x^2}{2(1-\varLambda)} n^{1/3} - \frac{x^3}{3(1-\varLambda)^2}} G\left(\frac{3}{2}, \frac{1}{4}; -\frac{3^{2/3}\gamma^{1/3} x}{2(1-\varLambda)}\right) \tag{12}$$

where $G(\lambda, \mu; x)$ *is defined in theorem 2 and* $\gamma = 1 + \tau^2\,\Omega'''(\tau)$. *For simple hypergraphs, there is an additional factor* $\exp\left(-\frac{\zeta\,\Omega''(\zeta)}{2} - \frac{\omega_2^2\zeta^2}{4}\right)$ *for the first asymptotic, and* $\exp\left(-\frac{1}{2} - \frac{\omega_2^2\tau^2}{4}\right)$ *for the second one.*

Proof. A hypergraph G with no complex component is a forest of trees and unicycle components. The excess of a tree is -1, the excess of a unicycle component is 0. Since the excess of a hypergraph is the sum of the excesses of its components, the excess of G is the opposite of the number of trees. The sum of the weights of the hypergraphs with no complex component, n vertices and excess k (which is negative) is

$$n![z^n]\frac{U^{-k}}{(-k)!}e^V = \frac{n!}{(-k)!}[z^n]\frac{U^{-k}}{\sqrt{1-T\,\Omega''(T)}} = \frac{n!}{(-k)!}\frac{1}{2i\pi}\oint\frac{U^{-k}}{\sqrt{1-T\,\Omega''(T)}}\frac{dz}{z^{n+1}}.$$

For $k = (\lambda - 1)n$, there are two saddle points: one implicitly defined by $\Psi(\zeta) = \lambda$ and the other at τ. Those two saddle points coalesce when $\lambda = \Psi(\tau)$. For smaller values of λ, the first saddle point dominates and an application of the large power theorem of [10] leads to (11). When k is around its critical value $(\Lambda - 1)n$, we apply theorem 2. The Newton-Puiseux expansions of T, e^V and U can be derived from lemma 1

$$\begin{aligned}
&T(z) \sim \tau - \tau\sqrt{\frac{2}{\gamma}}\sqrt{1-z/\rho},\\
&e^{V(z)} \sim (2\gamma)^{-1/4}(1-z/\rho)^{-1/4},\\
&U(z) = \tau(1-\Psi(\tau)) - \tau(1-z/\rho) + \tau\frac{2}{3}\sqrt{\frac{2}{\gamma}}(1-z/\rho)^{3/2} + \mathcal{O}(1-z/\rho)^2,
\end{aligned}$$

where $\Psi(z) = \Omega'(z) - \frac{\Omega(z)}{z}$ and $\gamma = 1 + \tau^2\,\Omega'''(\tau)$. Using Theorem 2, we obtain $\mathrm{thg}_{n,k} \sim \frac{n!}{(-k)!}\frac{\sqrt{3}}{2}\frac{(\tau(1-\Lambda))^{-k}}{\rho^n\sqrt{n}}G\left(\frac{3}{2}, \frac{1}{4}; -\frac{3^{2/3}\gamma^{1/3}x}{2(1-\Lambda)}\right)$ which reduces to (12).

In the analysis of simple hypergraphs, the generating function $V(z)$ is replaced by $V(z) - \frac{T\,\Omega''(T)}{2} - \frac{\omega_2^2T^2}{4}$ to avoid loops and multiple edges (in unicycle components, those can only be two edges of size 2).

Combining theorems 1 and 3, we deduce that when $k = (\lambda - 1)n + \mathcal{O}(n^{1/3})$ with $\lambda < \Lambda$, the probability that a random hypergraph in $\mathrm{HG}_{n,k}$ has no complex component approaches 1 as n tends towards infinity. When $k = (\Lambda - 1)n + \mathcal{O}(n^{1/3})$, this limit becomes $\sqrt{2/3}$ because $G(2/3, 1/4; 0)$ is equal to $2/(3\sqrt{\pi})$. It is remarkable that this value does not depend on Ω, therefore it is the same as in [15] for graphs. However, the evolution of this probability between the subcritical and the critical ranges of excess depends on the (ω_t).

Corollary 2. *For* $k = (\Lambda - 1)n + xn^{2/3}$ *with* x *bounded, the probability that a hypergraph in* $\mathrm{HG}_{n,k}$ *or in* $\mathrm{SHG}_{n,k}$ *has no complex component is*

$$\sqrt{\frac{3\pi}{2}}\exp\left(\frac{-x^3}{3(1-\Lambda)^2}\right)G\left(\frac{3}{2}, \frac{1}{4}; -\frac{3^{2/3}\gamma^{1/3}x}{2(1-\Lambda)}\right).$$

Theorem 2 does not apply when $H(z)$ is periodic. This is why we restricted $\omega(y)/y$ not to be of the form $f(z^d)$ where $d > 1$ and $f(z)$ is a power serie with a strictly positive radius of convergence. An unfortunate consequence is that theorems 1 and 3 do not apply to d-uniform hypergraphs. However, the expression of the critical excess is still valid. For the d-uniform hypergraphs, $\Omega(z) = \frac{z^d}{d!}$, $\Psi(z) = \frac{(d-1)}{d!} z^{d-1}$ and $\tau^{d-1} = (d-2)!$, so we obtain $k = \frac{1-d}{d} n$ for the critical excess, which corresponds to a number of edges $m = \frac{n}{d(d-1)}$, a result already derived in [5].

5 Kernels

In the seminal articles [8] and [16], Wright establishes the connection between the asymptotic of connected graphs with n vertices and excess k and the enumeration of the connected *kernels*, which are multigraphs with no vertex of degree less than 3. This relation was then extensively studied in [1] and the notions of excess and kernels were extended to hypergraphs in [9].

A kernel is a hypergraph with additional constraints that ensure that:

- each hypergraph can be reduced to a unique kernel,
- the excesses of a hypergraph and its kernel are equal,
- for any integer k, there is a finite number of kernels of excess k,
- the generating function of hypergraphs of excess k can be derived from the generating function of kernels of excess k.

Following [9], we define the *kernel* of a hypergraph G as the result of the repeated execution of the following operations:

1. delete all the vertices of degree ≤ 1,
2. delete all the edges of size ≤ 1,
3. if two edges (a, v) and (v, b) of size 2 have one common vertex v of degree 2, delete v and replace those edges by (a, b),
4. delete the connected components that consist of one vertex v of degree 2 and one edge (v, v) of size 2.

The following theorem has already been derived for uniform hypergraphs in [9]. We give a new proof and an expression for the generating function of the *clean* kernels.

Theorem 4. *The number of kernels of excess k is finite and each of them contains at most $3k$ edges of size 2. We say that a kernel is* clean *if this bound is reached. The generating functions of connected clean kernels of excess k is*

$$c_k (1 + \omega_3 z^2)^{2k} \omega_2^{3k} z^{2k} \tag{13}$$

where $c_k = [z^{2k}] \log \sum_k \frac{(6k)!}{(3!)^{2k} 2^{3k} (3k)!} \frac{z^{2k}}{(2k)!}$ and the variables w and x have been omitted.

Proof. By definition, $k+n+m = \sum_{e \in E} |e| = \sum_{v \in V} \deg(v)$. By construction, the vertices (resp. edges) of a kernel have degree (resp. size) at least 2, so

$$k+n+m \geq 3m - m_2, \tag{14}$$

$$k+n+m \geq 3n - n_2, \tag{15}$$

where n_2 (resp. m_2) is the number of vertices of degree 2 (resp. edges of size 2). Furthermore, each vertex of degree 2 belongs to an edge of size at least 3, so

$$k+n+m \geq 2m_2 + n_2. \tag{16}$$

Summing those three inequalities, we obtain $3k \geq m_2$.

This bound is reached if and only if (14), (15) and (16) are in fact equalities. Therefore, the vertices (resp. edges) of a clean kernel have degree (resp. size) 2 or 3, each vertex of degree 2 belongs to exactly one edge of size 3 and all the vertices of degree 3 belongs to edges of size 2. Consequently, any connected clean kernel can be obtained from a connected cubic multigraph with $2k$ vertices through substitutions of vertices of degree 3 by groups of three vertices of degree 2 that belong to a common edge of size 3. This means that if $f(z)$ represent the cubic multigraphs where z marks the vertices, then the generating function of clean kernels is $f(z + \omega_3 z^3)$. The generating function of cubic multigraphs of excess k is $\frac{(6k)!}{(3!)^{2k} 2^{3k} (3k)!} \frac{z^{2k}}{(2k!)}$, and a cubic multigraph is a set of connected cubic multigraphs, so the value $(2k)! c_k$ defined in the theorem is the sum of the weights of the connected cubic multigraphs.

To prove that the total number of kernels of excess k is bounded, we introduce the *dualized* kernels, which are kernels where each edge of size 2 contains a vertex of degree at least 3. This implies the dual inequality of (16) $k+n+m \geq 2n_2 + m_2$ that leads to $7k \geq n+m$. Finally, each dualized kernel matches a finite number of normal kernels by substitution of an arbitrary set of vertices of degree 2 by edges of size 2.

The previous theorem implies that the generating function of the connected kernels of excess k is a multivariate polynomial of degree $3k$ in ω_2. One can develop a kernel into a hypergraph by adding rooted trees to its vertices, replacing its edges of size 2 by paths of trees and adding rooted trees into the edges of size greater than 2. This matches the following substitutions in the generating functions: $z \leftarrow T(z)$, $w_2 \leftarrow \frac{\Omega''(T)}{1 - T\,\Omega''(T)}$ and $w_t \leftarrow \Omega^{(t)}(T)$ for all $t > 2$. Therefore, there exists a polynomial $P_k(X)$ in $\mathbb{Q}[X, \Omega(X), \Omega'(X), \ldots]$ such that the generating function of connected hypergraphs of excess k is expressed as

$$\mathrm{chg}_k(z) = \frac{P_k(T)}{(1 - T\,\Omega''(T))^{3k}}. \tag{17}$$

From there, a singularity analysis gives the asymptotics of connected hypergraphs in $\mathrm{HG}_{n,k}$

$$n![z^n]\,\mathrm{chg}_k(z) \sim \sqrt{2\pi} \frac{P_k(\tau)(2 + 2\tau^2\,\Omega'''(\tau))^{-\frac{3k}{2}}}{\Gamma\left(\frac{3k}{2}\right)} \frac{e^{(\Omega'(\tau)-1)n}}{\tau^n} n^{n + \frac{3k-1}{2}}$$

where τ, value of T at its dominant singularity, is characterized by $\tau\,\Omega''(\tau) = 1$.

This formula gives the asymptotic number of connected hypergraphs with respect to n, up to a constant factor $P_k(\tau)$, which is computable through the enumeration of the connected kernels of excess k. This is however unsatisfactory, because the complexity of this computation is too high. We believe that the approach developed in [17] for graphs may be the solution. It starts by considering hypergraphs as sets of trees, unicycle components and connected components of higher order

$$\mathrm{hg}(z,y) = \sum_n e^{\frac{\Omega(yn)}{y}} \frac{(z/y)^n}{n!} = \exp\left(y^{-1}U + V + \sum_k \frac{P_k(T)}{(1 - T\,\Omega''(T))^{3k}} \right),$$

from which it may be possible to extract informations on the values $P_k(\tau)$.

6 Hypergraphs with Complex Components of Fixed Excess

The next theorem describes the structure of critical hypergraphs. It generalizes theorem 5 of [1] about graphs. Interestingly, the result does not depend on the (ω_t).

Theorem 5. *Let $r_1, \dots r_q$ denote a finite sequence of integers and $r = \sum_{t=1}^q t r_t$, then the limit of the probability for a hypergraph or simple hypergraph with n vertices and global excess $k = (\Lambda - 1)n + \mathcal{O}(n^{1/3})$ to have exactly r_t components of excess t for t from 1 to q is*

$$\left(\frac{4}{3}\right)^r \frac{r!}{(2r)!} \sqrt{\frac{2}{3}} \frac{c_1^{r_1}}{r_1!} \frac{c_2^{r_2}}{r_2!} \cdots \frac{c_q^{r_q}}{r_q!}. \tag{18}$$

where the (c_i) are defined as in Theorem 4. For $k = (\Lambda - 1)n + xn^{2/3}$ and x bounded, the limit of this probability is

$$3^{-r} \frac{c_1^{r_1}}{r_1!} \frac{c_2^{r_2}}{r_2!} \cdots \frac{c_q^{r_q}}{r_q!} \sqrt{\frac{3\pi}{2}} \exp\left(\frac{-x^3}{3(1-\Lambda)^2} \right) G\left(\frac{3}{2}, \frac{1}{4} + \frac{3r}{2}; -\frac{3^{2/3}\gamma^{1/3}x}{2(1-\Lambda)} \right).$$

Proof. Let $C_k(z)$ denote the generating function of connected hypergraphs of excess k. Those can be obtained by expansion of the connected kernels of excess k, so $C_k(z) = c_k (1 + T^2\,\Omega'''(T))^{2k} \frac{\Omega''(T)^{3k}}{(1 - T\,\Omega''(T))^{3k}} T^{2k} + \dots$ plus terms with a denominator $(1 - T\,\Omega''(T))$ of smaller order. Therefore, when z tends towards the dominant singularity ρ of $T(z)$, $C_k(z) \sim c_k \left(\frac{\sqrt{\gamma}}{2^{3/2}\tau} \right)^k (1 - z/\rho)^{-3k/2}$. The sum of the weights of hypergraphs with global excess k and r_t components of excess t is $n![z^n] \frac{U^{K-k}}{(K-k)!} e^V \frac{C_1(z)^{r_1}}{r_1!} \frac{C_2(z)^{r_2}}{r_2!} \cdots \frac{C_q(z)^{r_q}}{r_q!}$ and an application of Theorem 2 ends the proof, with $G(3/2, 1/4 + 3r/2; 0) = \frac{2}{3\sqrt{\pi}} \frac{4^r r!}{(2r)!}$. Those computations are the same as in Theorem 3.

Remark. We have seen in the proof that around the critical value of the excess $k = (\Lambda - 1)n$, the kernel of a hypergraph is clean with high probability. In [1], the authors remark that the theorem holds true when q is unbounded, because the sum of the probabilities (18) over all finite sequences (r_t) is 1.

7 Future Directions

Much more information can be extracted from the generating functions (4) and (5), as the number of edges at which the first cycle appears [15], more statistics on the parameters n, m, k and l for random hypergraphs and more error terms on the asymptotics presented. In particular, connected non-uniform hypergraphs deserve a dedicated paper, with an expression for the constants $P_k(\tau)$ defined in (17).

In the present paper, for the sake of the simplicity of the proofs, we restrained our work to the case where $e^{\Omega(z)/z}$ is aperiodic. This technical condition can be waived in the same way Theorem VIII.8 of [10] can be extended to periodic functions.

In the model we presented, the weight ω_t of an edge only depends on its size t. For some applications, one may need weights that also vary with the number of vertices n. It would be interesting to measure the impact of this modification on the phase transition properties described in this paper.

More generally, the study of the relation to other models, as the one presented in [6] and [18], could lead to new developments and applications.

References

1. Janson, S., Knuth, D.E., Luczak, T., Pittel, B.: The birth of the giant component. Random Struct. Algorithms 4(3), 233–359 (1993)
2. Erdös, P., Rényi, A.: On the evolution of random graphs. Publ. Math. Inst. Hung. Acad. Sci. 5, 17 (1960)
3. Karoǹski, M., Łuczak, T.: The phase transition in a random hypergraph. Journal of Computational and Applied Mathematics 42(1), 125–135 (2002); Probabilistic Methods in Combinatorics and Combinatorial Optimization
4. Ravelomanana, V.: Birth and growth of multicyclic components in random hypergraphs. Theor. Comput. Sci. 411(43), 3801–3813 (2010)
5. Schmidt-Pruzan, J., Shamir, E.: Component structure in the evolution of random hypergraphs. Combinatorica 5(1), 81–94 (1985)
6. Darling, R.W.R., Norris, J.R.: Structure of large random hypergraphs. Ann. Appl. Probab., 125–152 (2004)
7. Gessel, I.M., Kalikow, L.H.: Hypergraphs and a functional equation of Bouwkamp and de Bruijn. J. Comb. Theory, Ser. A 110(2), 275–289 (2005)
8. Wright, E.M.: The number of connected sparsely edged graphs. Journal of Graph Theory 1, 317–330 (1977)
9. Karoǹski, M., Łuczak, T.: The number of connected sparsely edged uniform hypergraphs. Discrete Mathematics 171(1-3), 153–167 (1997)
10. Flajolet, P., Sedgewick, R.: Analytic Combinatorics. Cambridge University Press (2009)

11. Berge, C.: Graphs and Hypergraphs. Elsevier Science Ltd. (1985)
12. Bergeron, F., Labelle, G., Leroux, P.: Combinatorial Species and Tree-like Structures. Cambridge University Press (1997)
13. Oger, B.: Decorated hypertrees. CoRR abs/1209.0941 (2012)
14. Banderier, C., Flajolet, P., Schaeffer, G., Soria, M.: Random maps, coalescing saddles, singularity analysis, and airy phenomena. Random Struct. Algorithms 19(3-4), 194–246 (2001)
15. Flajolet, P., Knuth, D.E., Pittel, B.: The first cycles in an evolving graph. Discrete Mathematics 75(1-3), 167–215 (1989)
16. Wright, E.M.: The number of connected sparsely edged graphs III: Asymptotic results. Journal of Graph Theory 4(4), 393–407 (1980)
17. Flajolet, P., Salvy, B., Schaeffer, G.: Airy phenomena and analytic combinatorics of connected graphs. The Electronic Journal of Combinatorics 11, 34 (2004)
18. Bollobás, B., Janson, S., Riordan, O.: Sparse random graphs with clustering, cite arxiv:0807 (2008)

Domino Tatami Covering Is NP-Complete

Alejandro Erickson and Frank Ruskey[⋆]

Department of Computer Science, University of Victoria, V8W 3P6, Canada

Abstract. A covering with dominoes of a rectilinear region is called *tatami* if no four dominoes meet at any point. We describe a reduction from planar `3SAT` to Domino Tatami Covering. As a consequence it is therefore NP-complete to decide whether there is a perfect matching of a graph that meets every 4-cycle, even if the graph is restricted to be an induced subgraph of the grid-graph. The gadgets used in the reduction were discovered with the help of a `SAT`-solver.

1 Introduction

Imagine that you want to "pave" a rectilinear driveway on the integer lattice using 1 by 2 bricks. Sometimes this will be possible, but sometimes not, depending on the shape of the driveway. Abstractly, a rectilinear driveway D is a connected finite induced subgraph of the infinite planar grid-graph, and a paving with bricks corresponds to a perfect matching. Since D is bipartite, various network flow algorithms can be used to determine whether there is a paving in low-order polynomial time.

However, an examination of typical paving patterns reveals that another constraint is often enforced/desired, probably for both aesthetic reasons and engineering reasons. The constraint is that no four bricks meet at a point. In some recent papers, this restriction has come to be known as the tatami constraint, because Japanese tatami mat layouts often adhere to it. The question that we wish to address in this paper is: What is the complexity of determining whether D has a paving also satisfying the tatami constraint? We will show that the problem is NP-complete.

A *rectilinear region*, R, is a finite subset of the integer grid. We say R *is covered by dominoes* if it is covered exactly by non-overlapping dominoes. We describe a polynomial reduction from the NP-complete problem planar `3SAT` to Domino Tatami Covering (`DTC`). The gadgets used in the reduction were discovered with the help of a `SAT`-solver.

Definition 1 (Domino Tatami Covering (`DTC`)).

INSTANCE: *A rectilinear region, R, represented as n grid squares.*
QUESTION: *Can R be covered exactly by non-overlapping dominoes such that no four of them meet at any one point?*

[⋆] This research was supported in part by an NSERC Discovery Grant

© Springer-Verlag Berlin Heidelberg 2013

Domino tatami coverings have an interesting combinatorial structure, elucidated for rectangles in [12] and further in [5]. The results in these papers, as well as [6, 4] are enumerative, whereas in this paper we explore tatami coverings from a computational perspective. There is no comprehensive structure theorem for tatami coverings of rectilinear grids, but evidently much of the structure is still there, as is illustrated in Fig. 1. In contrast with other tatami-related results, however, we make no attempt to characterise this structure. Instead, our reduction relies on the interactions between coverings of a few specific regions that are discovered using a `SAT`-solver.

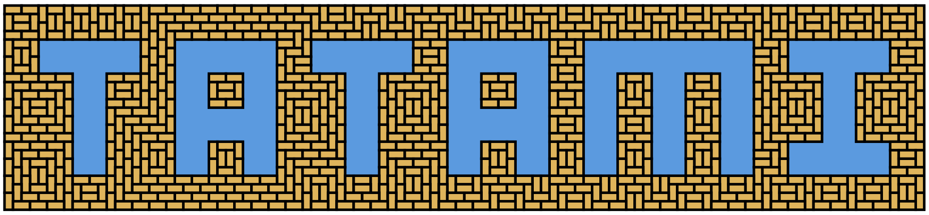


Fig. 1. A domino tatami covering of a rectilinear region, produced by a `SAT`-solver

There are some previous complexity results about tilings and domino coverings. Historically, perhaps the first concerned colour-constrained coverings, such as those of Wang tiles. It is well known, for example, that covering the $k \times k$ grid with Wang tiles is NP-complete ([8]). On the other hand tatami does not appear to be a special case of these, nor of similar colour restrictions on dominoes (e.g. [1, 13]).

A more closely related mathematical context is found, instead, among the graph matching problems discussed by Churchley, Huang, and Zhu, in [2]. In their paper, an *H-transverse matching* of a graph G, is a matching M, such that $G - M$ has no subgraph H. In a tatami covering of the rectilinear grid, G is a finite induced subgraph of the infinite grid-graph, H is a 4-cycle, and we require a perfect matching of the edges. In fact, if the matching is not required to be perfect, the problem is polynomial.

`SAT`-solvers have been applied to a broad range of industrial and mathematical problems in the last decade. Our reduction from planar `3SAT` uses Minisat ([3]) to help automate gadget generation, as was also done by Ruepp and Holzer ([11]). It is easy to see that instances of other locally restricted covering problems can be expressed as satisfiability formulae, which suggests that `SAT`-solvers may provide a methodological applicability in hardness reductions involving those problems.

2 Preliminaries

Let ϕ be a CNF formula, with variables U, and clauses C. The formula is *planar* if there exists a planar graph $G(\phi)$ with vertex set $U \cup C$ and edges

$\{u, c\} \in E$, where one of the literals u or $\bar{u}$ is in the clause c. When the clauses contain at most three literals, ϕ is an instance of planar `3SAT` (`P3SAT`), which is NP-complete ([9]).

We construct an instance of `DTC` which emulates a given instance, ϕ, of `P3SAT`, by replacing the vertices and edges of $G(\phi)$ with a rectilinear region, $R(\phi)$, that can be tatami-covered with dominoes if and only if ϕ is satisfiable. Let $n = |U \cup C|$. In Sec. 4 we show that $R(\phi)$ can be created in $\mathcal{O}(n)$ time, and that it fits in an $\mathcal{O}(n) \times \mathcal{O}(n)$ grid, by using Rosenstiehl and Tarjan's algorithm ([10]).

3 Gadgets

In this section we describe wire, `NOT` gates, and `AND` gates, which form the required gadgets. The functionality of our gadgets depends on the coverings of a certain 8×8 subgrid.

Lemma 1. *Let R be a rectilinear grid, with an 8×8 subgrid, S. If a domino crosses the boundary of S in a domino tatami covering of R, then at least one corner of S is also covered by a domino that crosses its boundary.*

Proof. Suppose R is covered by dominoes, and consider those dominoes which cover S. Such a cover may not be exact, in the sense that a domino may cross the boundary of S. If we consider all such dominoes to be monominoes within S, we obtain a monomino-domino covering of S. This covering inherits the tatami restriction from the covering of R, so it is one of the 8×8 monomino-domino coverings enumerated in [6] (and/or [5]).

The proof of Lemma 4 in [6] (third paragraph) states that there is a monomino in at least one corner of S if $0 < m < n$; Corollary 2 of [5] states that there is a monomino in at least one corner of S if $m = n$ (see examples in Fig. 2). This monomino corresponds to a domino which crosses the boundary in a corner of S, as required. □

The rectilinear region $R(\phi)$ incorporates a network of 8×8 squares, whose centres reside on a $16\mathbb{Z} \times 16\mathbb{Z}$ grid, and whose corners form part of the boundary of $R(\phi)$. Lemma 1 implies that no domino may cross their boundaries, and thus each one must be covered in one of the two ways shown in Fig. 3(a). (For proofs see [12] and Exercise 215, Sec. 7.1.4 in [7]).

The coverings of these squares are related to each other by connecting regions. The part of an 8×8 square which borders on a connector may be covered either by two tiles, denoted by `F` to signify "false", or three tiles, denoted by `T` to signify "true" (see Fig. 3(a)). Note that the covering of a square is not `T` or `F` by itself, because connectors below and beside it would meet the square at differing interfaces.

A connector, which imposes a relationship between the coverings of a set of 8×8 squares, is verified by showing that it can be covered if and only if the

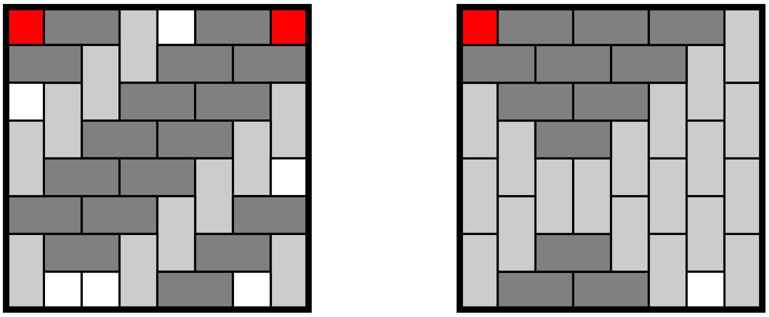
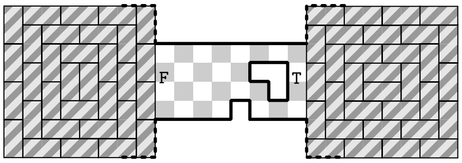


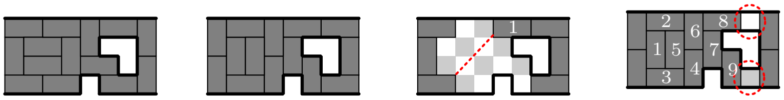
Fig. 2. All monomino-domino tatami coverings of the square have at least one monomino in their corners (see [5, 6]). The squares in $R(\phi)$ have isolate corners, so these must be covered in exactly one of the two ways given by Exercise 7.1.4.215 in [7], shown in the left and right-hand cross-hatched squares in Fig. 3(a).

relationship is satisfied. The connectors we describe were generated with `SAT`-solvers, but they are simple enough that we can verify them by hand, as is done below.

`NOT` gate. The `NOT` gate interfaces with two 8×8 squares (see Fig. 3(a)), and can be covered if and only if these squares are covered with differing configurations.



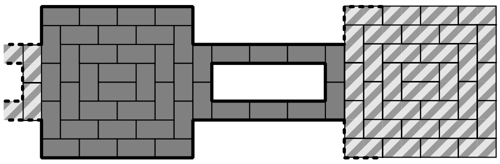
(a) `NOT` gate with `F` and `T` interfaces. This also shows the two possible domino coverings of the 8×8 square.

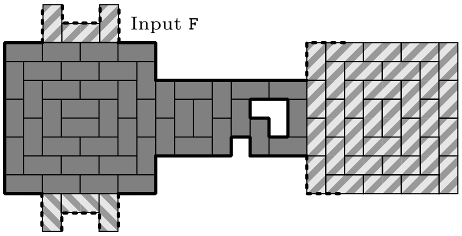


(b) `F`$\longrightarrow$`T`. (c) `T`$\longrightarrow$`F`. (d) `F`$\longrightarrow$`F`. (e) `T`$\longrightarrow$`T`.

Fig. 3. `NOT` gate can be covered if and only if the input differs from the output. Numbered tiles indicate the (non-unique) ordering in which their placement is forced. Red dotted lines indicate how domino coverings are impeded in *(d)* and *(e)*.

Wire gadget. Wire transmits `T` or `F` through a sequence of squares (see Fig. 4(a)). A turn may incorporate a `NOT` gate in order to maintain the same configuration (see Fig. 4(b)).


(a) Unit of wire, carrying `T`.



(b) Wire branch and turn, carrying `F`.

Fig. 4. Wire gadget

AND gate. The `AND` gate interfaces with two 8×8 input squares, and one output square (see Fig. 5). It can be covered with dominoes if and only if the output value is the `AND` of the inputs (see Figs. 6 and 7).

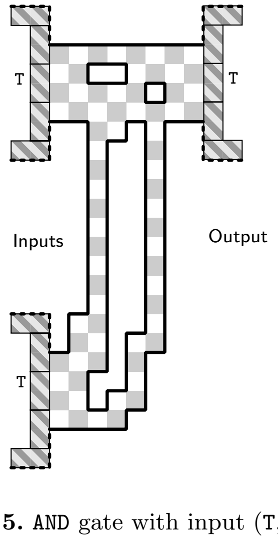


Fig. 5. `AND` gate with input (`T`,`T`)

Variable gadget. We use a vertical segment of wire. The variable gadget is set to `T` or `F` by choosing the appropriate covering of one of its 8×8 squares. Its value (or its negation) is propagated to clause gadgets via horizontal wire gadgets, representing edges.

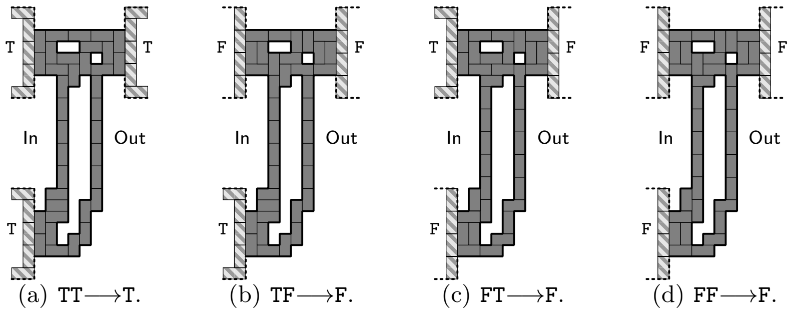


Fig. 6. `AND` gate coverings

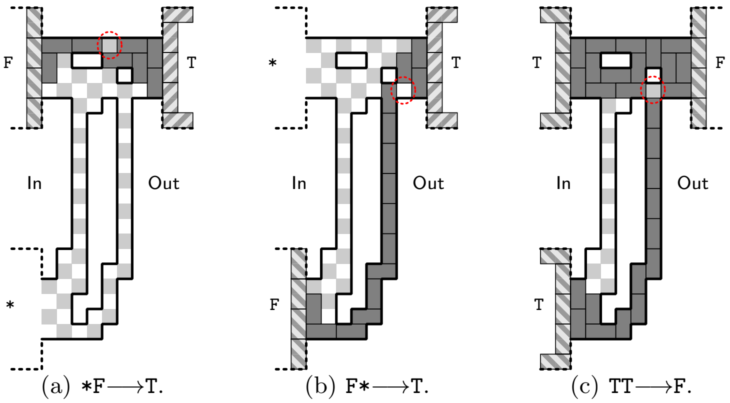


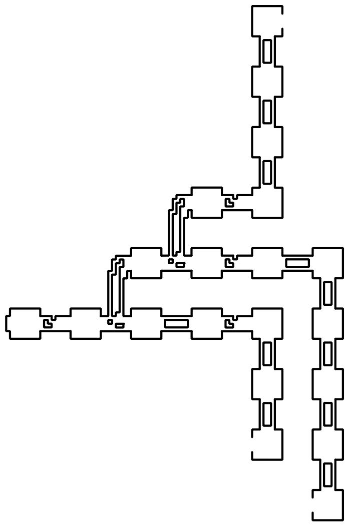
Fig. 7. Impossible `AND` gate coverings, where `*` denotes `F` or `T`

Clause gadget. The clause gadget is a circuit for $\neg(\bar{a} \wedge (\bar{b} \wedge \bar{c}))$, or the equivalent with fewer inputs, ending in a configuration that can be covered if and only if the output signal of the circuit is `T`. To satisfy the layout requirements, the inputs to the clause are vertically translated by wire (see Fig. 8).

4 Layout

Let $G(\phi)$ be a planar embedding of the Boolean 3CNF formula ϕ, using Rosenstiehl and Tarjan's ([10]) algorithm, so that each vertex is represented by a vertical line segment, and each edge is represented by a horizontal line segment. All parts lie on integer grid lines, inside of a $\mathcal{O}(n) \times \mathcal{O}(n)$ grid, where $n = |U \cup C|$, and the embedding is found in $\mathcal{O}(n)$ time.

There exists a constant K, which is the same for any planar 3CNF formula, such that $G(\phi)$ can be scaled to fit on the $nK \times nK$ grid, and its parts replaced by the gadgets described above. Each gadget has a constant number of grid squares, which ensures that $R(\phi)$ has $\mathcal{O}(n^2)$ grid squares altogether.



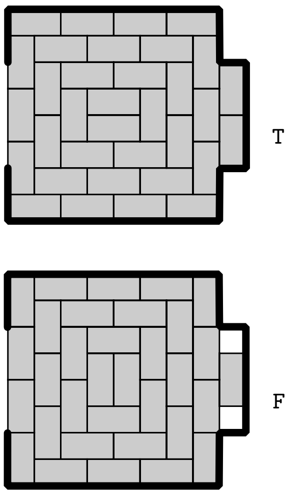


Fig. 8. A three input clause gadget from the circuit $\neg(\bar{a}\wedge(\bar{b}\wedge\bar{c}))$. Vertical wire translates horizontal inputs without changing the signal. The end of the clause is coverable if and only if its signal is T.

The variable gadget is connected to edges by branches. The layout of $G(\phi)$ prevents conflicts between edges meeting the variable gadget on the same side, while two edges can meet the left and right sides of the variable gadget without interfering with each other. The inputs of the clause gadget are symmetric, so there are no conflicts when connecting these to horizontal edges (see Fig. 8(a)).

Example. The planar Boolean formula from Fig. 1 in [9] gives the DTC instance in Fig. 9.

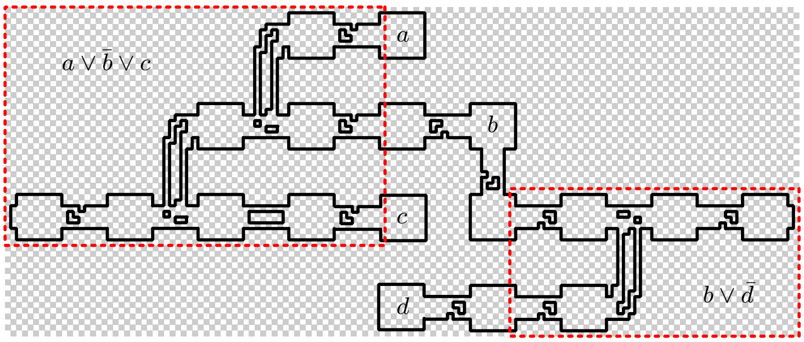


Fig. 9. An instance of DTC for the formula $(a \vee \bar{b} \vee c) \wedge (b \vee \bar{d})$

5 SAT-Solver

The search for logical gates required fast testing of small DTC instances. We reduced DTC to SAT in order to use the SAT-solver, Minisat ([3]), and efficiently test candidate regions connecting 8×8 squares while satisfying the conditions of the desired gate. The DTC solver was also allowed to make certain decisions about the region, rather than simply testing regions generated by another program.

Our search algorithm requires the following inputs:

– an $r \times c$ rectangle of grid squares, partitioned into pairwise disjoint sets K, X, A, C; and,
– a set of partial (good) coverings, G, and partial (bad) coverings, B, of C.

The output, R, is the region $A' \cup K$, where $A' \subseteq A$, which satisfies the following constraints.

(g) There exist coverings of R which form partial tatami domino coverings with each element of G.
(b) There exists no covering of R which forms a partial tatami domino covering with an element of B.

The outer loop of the search algorithm calls the SAT-solver to find a region that satisfies all elements of G, and avoids a list of forbidden regions, which is initially empty. Upon finding such a region, the inner loop checks whether the region satisfies any element of B. The search succeeds when *(g)* and *(b)* are both satisfied, and fails if the outer loop's SAT instance has no satisfying assignment.

The search space grows very quickly for several reasons, not least of which is the fact that 2^{160} regions are possible within the 20×8 rectangle occupied by our AND gate (if corners are allowed to meet one another). In addition, the list of forbidden regions, L, becomes too large for the SAT-solver to handle efficiently.

We used two heuristics on the inputs to obtain a feasible search. The first was searching for a smaller AND gate, which we modified to fit the placement of the 8×8 squares. The second was choosing forbidden squares, X, and required squares, K, to reduce the number of trivially useless regions that are tested.

5.1 DTC as a Boolean Formula

The SAT instances used above are modifications of a formula which is satisfiable if and only if a given region has a domino tatami covering.

Let R be the region we want to cover, and consider the graph whose vertices are the grid squares of R, and whose edges connect vertices of adjacent grid squares. Let H be the set of horizontal edges and let V be the set of vertical edges. The variables of the SAT instance are $H \cup V$, and those variables set to true in a satisfying assignment are the dominoes in the covering. The clauses are as follows, where $h, h' \in H$ and $v, v' \in V$.

1. Ensure a matching: For each pair of incident horizontal edges (h, h'), require the clause $\bar{h} \vee \bar{h}'$, and similarly for (v, v'), (h, v).

2. Ensure the matching is perfect: For each set of edges $\{h, h', v, v'\}$, which are incident to a vertex, require the clause $h \vee h' \vee v \vee v'$.
3. Enforce the tatami restriction: For each 4-cycle, $hvh'v'$, require the clause $h \vee h' \vee v \vee v'$.

6 Variations and Future Work

There are other locally constrained covering problems that are easily represented as Boolean formulae. Some of these are obviously polynomial, such as monomino-domino tatami covering, but others may be NP-complete. SAT-solvers can sometimes be used in such problems to create elaborate gadgets, which may help find a hardness reduction.

An example problem, whose computational complexity is open, is Lozenge-only Tatami Covering. This problem is the decision about whether or not a finite sub-grid of the triangular lattice can be covered with lozenges, such that no 5 lozenges meet at any point. A structure similar to that of tatami coverings occurs for this constraint (see Fig. 10).

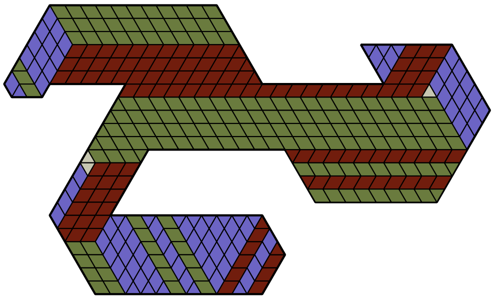


Fig. 10. A triangle-lozenge tatami covering

Our main question about DTC is the complexity of the case where the region is simply connected (no holes). It seems likely that the problem is still NP-complete, but a completely new approach will be required.

Secondarily, we are interested in H-transverse perfect matchings for H and G other than C_4 and grid-graphs. Are there other H-transverse perfect matchings of interest which induce a tatami-like global structure in the containing graph?

Another variant, mildly advocated by Don Knuth (personal communication), concerns inner corners of the coverings, such as occurs at the upper left in the letter T in Figure 1. If corners such as these, where a + occurs, are forbidden but corners such as the upper right one in the I are allowed (a $\perp$ shape or one of its rotations), then the nature of tatami coverings changes. The complexity of such coverings is unknown.

Acknowledgement. We thank Bruce Kapron for useful conversations about this problem, and Don Knuth for his comments on an earlier draft. Part of this research was conducted at the 9th McGill–INRIA Workshop on Computational Geometry.

References

[1] Biedl, T.: The complexity of domino tiling. In: Proceedings of the 17th Canadian Conference on Computational Geometry (CCCG), pp. 187–190 (2005)

[2] Churchley, R., Huang, J., Zhu, X.: Complexity of cycle transverse matching problems. In: Iliopoulos, C.S., Smyth, W.F. (eds.) IWOCA 2011. LNCS, vol. 7056, pp. 135–143. Springer, Heidelberg (2011)

[3] Eén, N., Sörensson, N.: An extensible sat-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)

[4] Erickson, A., Ruskey, F.: Enumerating maximal tatami mat coverings of square grids with v vertical dominoes. Submitted to a Journal (2013), `http://arxiv.org/abs/1304.0070`

[5] Erickson, A., Ruskey, F., Schurch, M., Woodcock, J.: Monomer-dimer tatami tilings of rectangular regions. The Electronic Journal of Combinatorics 18(1) #P109, 24 pages (2011)

[6] Erickson, A., Schurch, M.: Monomer-dimer tatami tilings of square regions. Journal of Discrete Algorithms 16, 258–269 (2012)

[7] Knuth, D.E.: The Art of Computer Programming: Combinatorial Algorithms, Part 1, 1st edn., vol. 4A. Addison-Wesley Professional (January 2011)

[8] Lewis, H.R.: Complexity of solvable cases of the decision problem for the predicate calculus. In: 19th Annual Symposium on Foundations of Computer Science, pp. 35–47 (October 1978)

[9] Lichtenstein, D.: Planar formulae and their uses. SIAM Journal on Computing 11(2), 329–343 (1982)

[10] Rosenstiehl, P., Tarjan, R.E.: Rectilinear planar layouts and bipolar orientations of planar graphs. Discrete & Computational Geometry 1(1), 343–353 (1986)

[11] Ruepp, O., Holzer, M.: The computational complexity of the Kakuro puzzle, revisited. In: Boldi, P. (ed.) FUN 2010. LNCS, vol. 6099, pp. 319–330. Springer, Heidelberg (2010)

[12] Ruskey, F., Woodcock, J.: Counting fixed-height tatami tilings. The Electronic Journal of Combinatorics 16(1) #R126, 20 pages (2009)

[13] Worman, C., Watson, M.D.: Tiling layouts with dominoes. In: Proceedings of the 16th Canadian Conference on Computational Geometry (CCCG), pp. 86–90 (2004)

The Complexity of the Identifying Code Problem in Restricted Graph Classes[*]

Florent Foucaud

Universitat Politècnica de Catalunya, Building C3, C/ Jordi Girona 1-3, 08034 Barcelona, Spain
florent.foucaud@gmail.com

Abstract. An identifying code is a subset of vertices of a graph such that each vertex is uniquely determined by its nonempty neighbourhood within the identifying code. We study the associated computational problem Minimum Identifying Code, which is known to be NP-hard, even when the input graph belongs to a number of specific graph classes such as planar bipartite graphs. Though the problem is approximable within a logarithmic factor, it is known to be hard to approximate within any sub-logarithmic factor. We extend the latter result to the case where the input graph is bipartite, split or co-bipartite. Among other results, we also show that for bipartite graphs of bounded maximum degree (at least 3), it is hard to approximate within some constant factor. We summarize known results in the area, and we compare them to the ones for the related problem Minimum Dominating Set. In particular, our work exhibits important graph classes for which Minimum Dominating Set is efficiently solvable, but Minimum Identifying Code is hard (whereas in all previously studied classes, their complexity is the same). We also introduce a graph class for which the converse holds.

1 Introduction

We study the computational complexity of the *identifying code problem*, where we want to find a set of vertices in a graph that uniquely identifies each vertex using its neighbourhood within the set. In particular, we study this complexity according to the graph class of the input. Identifying codes, introduced in 1998 [25], are a special case of the notion of a *test cover* in hypergraphs, first mentioned in Garey and Johnson's book [20]. Test covers have found applications in the areas of testing individuals (such as patients or computers) for diseases or faults, see [7,12]. In particular, as graphs model computer networks or buildings, identifying codes have been applied to the location of threats in facilities [34] and error detection in computer networks [25].

To avoid confusion, we will usually call hypergraphs and their vertex and edge sets $H = (I, A)$ and graphs $G = (V, E)$. Given a hypergraph H, a *set cover* of

[*] An extended version of this paper, containing the full proofs and further results, is available on the author's website [16].

© Springer-Verlag Berlin Heidelberg 2013

H is a subset $\mathcal{S}$ of its edges such that each vertex v belongs to at least one set S of $\mathcal{S}$. We say that S *dominates* v. A *test cover* of H is a subset $\mathcal{T}$ of edges such that for each pair u, v of distinct vertices of H, there is at least one set T of $\mathcal{T}$ that contains exactly one of u and v [20]. We say that T (and also $\mathcal{T}$) *separates* u from v. A set of edges that is both a set cover and a test cover is called a *discriminating code* of H [7]. It has to be mentioned that some hypergraphs may not admit any set cover (if some vertex is not part of any edge) or test cover (if two vertices belong to the same set of edges).

Given a graph G and a vertex v of G, we denote by $N[v]$ the closed neighbourhood of v. An *identifying code* of G is a subset $\mathcal{C} \subseteq V(G)$ such that $\mathcal{C}$ is a *dominating set*, i.e. for each $v \in V(G)$, $N[v] \cap \mathcal{C} \neq \emptyset$ and $\mathcal{C}$ is a *separating code*, i.e. for each pair $u, v \in V(G)$, if $u \neq v$, then $N[u] \cap \mathcal{C} \neq N[v] \cap \mathcal{C}$. The minimum size of an identifying code of a given graph G will be denoted $\gamma^{\text{ID}}(G)$. Identifying codes were introduced in [25]. Note that a graph may not admit a separating code if it contains *twin* vertices, i.e. vertices having the same closed neighbourhood. In a graph containing no twins, the whole vertex set is an identifying code; we call such graphs *twin-free*.

Identifying codes and further related notions have been studied extensively in the literature. We refer to Lobstein's on-line bibliography [27] on these topics. In particular, see [1,2,5,9,15,17,18,21,26,29,31,32] for studies of the computational complexity of these problems.

For definitions of computational complexity, we refer to the books of Ausiello et al. and of Garey and Johnson [3, 20]. Let us formally define the minimization problem associated with identifying codes (other problems used herein are defined analogously; we skip their definitions).

MIN ID CODE
INSTANCE: A graph G.
TASK: Find a minimum-size identifying code of G.

We will study MIN ID CODE from an approximation point of view, but also from a decision point of view; in that case MIN ID CODE is said to be NP-hard if the associated decision problem (consisting in deciding whether a given graph has an identifying code of a given size) is NP-hard.

We recall that the class APX is the class of all optimization problems that are c-approximable for some constant c. We also refer to the class log-APX as the class of all optimization problems that are $f(n)$-approximable, where n is the size of the instance and f is a poly-logarithmic function.

In this paper, we will study specific graph classes, of which many are standard, such as bipartite graphs, planar graphs or graphs of given maximum degree. Bipartite graphs which do not have any induced cycle of length more than 4 are called *chordal bipartite* (note that they are in general not chordal). Complements of bipartite graphs are called *co-bipartite* graphs. *Split graphs* are those whose vertex set can be partitioned into a clique and an independent set.

1.1 Related Work

It is well-known that Min Dominating Set is log-APX-hard (whereas a logarithmic factor approximation exists) [11,22]. The same properties hold for Min Test Cover [12] (a result that is easily seen to be transferrable to Min Discriminating Code[1]) and Min Id Code (see [5,26,32], for different proofs).

Regarding restrictions of the instances to specific graph classes, much is known for Min Dominating Set: NP-hardness of Min Dominating Set holds for many classes such as (chordal) bipartite graphs, split graphs, line graphs or planar graphs, but not for strongly chordal graphs, directed path graphs (which include the well-known interval graphs), or graphs having a dominating shortest path (see e.g. [13] for an on-line database, and [22] for a survey). The log-APX-completeness of Min Dominating Set is known to hold even for bipartite graphs and split graphs [11], however it does not hold for planar graphs or unit disk graphs (in which Min Dominating Set admits PTAS algorithms [4,23]) or in (bipartite) graphs of bounded maximum degree (at least 3), where it is APX-complete [11].

In comparison, much less is known about Min Id Code; extending this knowledge is the main goal of this paper. It was known that, in general, Min Id Code is NP-hard, even for bipartite graphs [9], planar graphs of maximum degree 3 [1,2], planar bipartite unit disk graphs [29], line graphs [17], split graphs [15,18], and, interestingly, interval graphs [15,18]. Regarding the approximation hardness, log-APX-completeness of Min Id Code is known only for general graphs [5,26,32], and APX-completeness, for graphs of bounded maximum degree at least 8 [21].

1.2 Our Contribution and Structure of the Paper

We extend the knowledge about the computational complexity of Min Id Code when restricted to specific classes of graphs. We compare these results to the corresponding ones for Min Dominating Set; see Table 1 for a summary of many known complexity results for these problems for selected graph classes.

We show in Section 2 that Min Id Code is log-APX-complete for bipartite, split and co-bipartite graphs. Prior, three different papers [5,26,32] showed that Min Id Code is log-APX-complete, but only in general graphs; intuitively speaking, the proximity between Min Discriminating Code and Min Id Code is used to design simpler reductions. Note that on co-bipartite graphs, Min Dominating Set is trivially solvable in polynomial time; in contrast, our result shows that Min Id Code is computationally very hard on this class.

In Section 3, we show that Min Id Code is APX-complete for bipartite graphs of maximum degree 3, improving on a result from Gravier et al. [21]. We also show that it is NP-hard for the same class with the additional restriction of planarity, as well as for chordal bipartite graphs.

[1] We can reduce Min Test Cover to Min Discriminating Code: given a hypergraph H, construct H' by adding to H a single vertex x and the set $V(H) \cup \{x\}$. Now H has a test cover of size k if and only if H' has a discriminating code of size $k+1$.

Finally, in Section 4, we exhibit a class of graphs, which we call SC-graphs, where Min Dominating Set is NP-hard, but Min Id Code is solvable in polynomial time. Until now, all known results for given graph classes were showing that Min Id Code was at least as hard as Min Dominating Set.

Table 1. Comparison of complexity lower bounds, "LB", and upper bounds, "UB", on approximation ratios (as functions of the order n of the input graph) of Min Id Code and Min Dominating Set for selected graph classes. Underlined entries are new results proved in this paper. Graph classes for which the precise complexity class of Min Id Code is not fully determined are marked with (∗). SC-graphs will be defined in Section 4. Definitions for classes that are not defined here can be found in [13].

	Min Id Code		Min Dominating Set	
graph class	LB	UB	LB	UB
in general	log-APX-h [5,26,32]	$O(\ln n)$ [12]	log-APX-h [11]	$O(\ln n)$ [24]
bipartite	$\underline{\text{log-APX-h}}$ (Co. 4)	$O(\ln n)$ [12]	log-APX-h [11]	$O(\ln n)$ [24]
chordal bipartite (∗)	$\underline{\text{NP-h}}$ (Th. 15)	$O(\ln n)$ [12]	NP-h [28]	$O(\ln n)$ [24]
split, chordal	$\underline{\text{log-APX-h}}$ (Th. 7)	$O(\ln n)$ [12]	log-APX-h [11]	$O(\ln n)$ [24]
planar (+ bipartite max. degree 3) (∗)	$\underline{\text{NP-h}}$ (Th. 11)	7 [31]	NP-h [33]	PTAS [4]
line (∗)	APX-h [15]	4 [17]	APX-h [10]	2 [10]
$K_{1,\ell}$-free ($\ell \geq 3$)	$\underline{\text{log-APX-h}}$ (Th. 7)	$O(\ln n)$ [12]	APX-h [10]	$\ell - 1$ [11]
max. degree Δ	APX-h $\Delta \geq 8$: [21]	$O(\ln \Delta)$ [12]	APX-h $\Delta \geq 3$: [30]	$O(\ln \Delta)$ [24]
max. degree $\Delta \geq 3$ and bipartite	$\underline{\text{APX-h}}$ (Th. 11)	$O(\ln \Delta)$ [12]	APX-h [11]	$O(\ln \Delta)$ [24]
unit disk (∗)	NP-h [29]	$O(\ln n)$ [12]	NP-h [8]	PTAS [23]
co-bipartite	$\underline{\text{log-APX-h}}$ (Th. 7)	$O(\ln n)$ [12]	P (trivial)	
interval, (∗)	NP-h [18]	$O(\ln n)$ [12]	P [6]	
permutation (∗)	OPEN	$O(\ln n)$ [12]	P [14]	
(planar) SC-graphs (∗)	$\underline{\text{P}}$ (Th. 18)		$\underline{\text{NP-h}}$ (Th. 19)	$O(\ln n)$ [24]

2 Bipartite, Co-bipartite and Split Graphs

In this section, we provide three reductions from Min Discriminating Code to Min Id Code for bipartite, split and co-bipartite graphs. We begin with preliminary considerations.

2.1 Useful Bounds and Constructions

Theorem 1 ([7]). *Let $H = (I, A)$ be a hypergraph admitting a discriminating code $\mathcal{C}$. Then $|\mathcal{C}| \geq \log_2(|I| + 1)$. If $\mathcal{C}$ is inclusion-wise minimal, then $|\mathcal{C}| \leq |I|$.*

We now describe two constructions that ensure that the vertices of some vertex set A are correctly identified using the vertices of another set L.

Construction 2 (bipartite logarithmic identification of A over (A, L)). Given two sets of vertices A and L with $|A| \leq 2^{|L|} - 1$, the bipartite logarithmic identification of A over (A, L), denoted $\mathcal{LOG}(A, L)$, is the graph of vertex set $A \cup L$ and where each vertex of A has a distinct nonempty subset of L as its neighbourhood.

The next construction is similar, but makes sure that each vertex of A has at least two neighbours in L.

Construction 3 (non-singleton bipartite logarithmic identification of A over (A, L)). Given two sets of vertices A and L with $|A| \leq 2^{|L|} - |L| - 1$,[2] the non-single bipartite logarithmic identification of A over (A, L), denoted $\mathcal{LOG}^*(A, L)$, is the graph of vertex set $A \cup L$ and where each vertex of A has a distinct subset of L of size at least 2 as its neighbourhood.

2.2 Bipartite Graphs

Theorem 4. Min Id Code *is log-APX-complete, even for bipartite graphs.*

Theorem 4 is proved using the following reduction.

Reduction 5. Given a hypergraph (I, A), we construct in polynomial time the bipartite graph $G(I, A)$ on $|I| + |A| + 9\lceil \log_2(|A| + 1)\rceil + 3$ vertices, with vertex set $V(G(I, A)) = I \cup A \cup \{x, y, z\} \cup \{a_j, b_j, c_j, d_j, e_j, f_j, g_j, h_j, i_j \mid 1 \leq j \leq \lceil \log_2(|A| + 1)\rceil\}$, and edge set:

$$\begin{aligned} E\left(G(I,A)\right) = \{x,y\} \cup \{y,z\} \cup \{\{z,i\} \mid i \in I\} \cup\ & E\left(\mathcal{B}(I,A)\right) \\ \cup\ & E\left(\mathcal{LOG}(A, \{a_j \mid 1 \leq j \leq \lceil \log_2(|A|+1)\rceil\})\right) \\ \cup\ & \{\{a_j,b_j\},\{b_j,c_j\},\{a_j,d_j\},\{d_j,g_j\} \mid 1 \leq j \leq \lceil \log_2(|A|+1)\rceil\} \\ \cup\ & \{\{d_j,e_j\},\{e_j,f_j\},\{g_j,h_j\},\{h_j,i_j\} \mid 1 \leq j \leq \lceil \log_2(|A|+1)\rceil\}. \end{aligned}$$

where $\mathcal{B}(I, A)$ denotes the bipartite incidence graph of (I, A) and $E\left(\mathcal{LOG}(A, L)\right)$ denotes the bipartite logarithmic identification of A over (A, L) (see Construction 2). The construction is illustrated in Figure 1.

Theorem 6. *A hypergraph (I, A) has a discriminating code of size at most k if and only if graph $G(I, A)$ has an identifying code of size at most $k + 6\lceil \log_2(|A| + 1)\rceil + 2$, and one can construct one using the other in polynomial time.*

[2] There are exactly $2^{|L|} - |L| - 1$ distinct subsets of L with size at least 2.

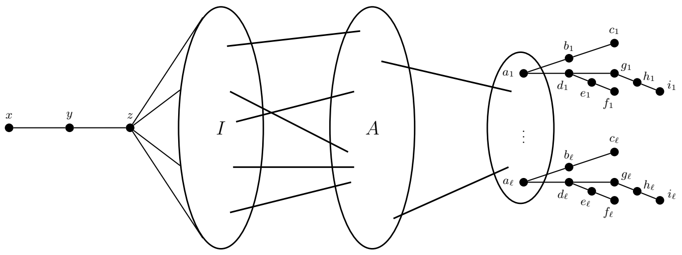


Fig. 1. Reduction from Min Discriminating Code to Min Id Code

Proof. Let $\mathcal{D} \subseteq A$ be a discriminating code of (I, A), $|\mathcal{D}| = k$. We define $\mathcal{C}(\mathcal{D})$ as follows: $\mathcal{C}(\mathcal{D}) = \mathcal{D} \cup \{x, z\} \cup \{a_j, c_j, d_j, f_j, g_j, i_j \mid 1 \le j \le \lceil \log_2(|A| + 1)\rceil\}$. One can easily check that $\mathcal{C}(\mathcal{D})$ has size $k + 6\lceil \log_2(|A| + 1)\rceil + 2$. Code $\mathcal{C}(\mathcal{D})$ has size $k + 6\lceil \log_2(|A| + 1)\rceil + 2$ is clearly a dominating set. To see that it is an identifying code of $G(I, A)$, observe that vertex z separates all vertices of I from all vertices which are not in $I \cup \{z\}$. Vertex z itself is the only vertex dominated only by z (each vertex of $\mathcal{I}$ being dominating by some vertex of $\mathcal{D}$); y is dominated by both x, y and x, only by itself. Since $\mathcal{D}$ a discriminating code of (I, A), all vertices of I are dominated by a distinct subset of $\mathcal{D}$. Furthermore, due to the bipartite logarithmic identification of A over $(A, \{a_j \mid 1 \le j \le \lceil \log_2(|A|+1)\rceil\})$ (and since each vertex a_j belongs to the code), all vertices of A are dominated by a unique subset of $\{a_j \mid 1 \le j \le \lceil \log_2(|A| + 1)\rceil\}$. Finally, it is easy to check that all vertices of type $a_j, b_j, c_j, d_j, e_j, f_j, g_j, h_h, i_j$ are correctly separated.

For the other direction, Let $\mathcal{C}$ be an identifying code of $G(I, A)$, $|\mathcal{C}| = k + 6\lceil \log_2(|A| + 1)\rceil + 2$. We first "normalize" $\mathcal{C}$ by constructing an identifying code $\mathcal{C}^*$ of $G(I, A)$, $|\mathcal{C}^*| \le |\mathcal{C}|$, such that the two following properties hold:

$$|\mathcal{C}^* \cap \{V(G(I, A)) \setminus \{I \cup A\}\}| = 6\lceil \log_2(|A| + 1)\rceil + 2 \quad (1)$$

$$|\mathcal{C}^* \cap I| = \emptyset. \quad (2)$$

To get Condition (1), we replace $|\mathcal{C} \cap \{V(G(I, A)) \setminus \{I \cup A\}\}|$ by $\{x, z\} \cup \{a_j, c_j, d_j,\ f_j, g_j, i_j \mid 1 \le j \le \lceil \log_2(|A| + 1)\rceil\}$ to get code $\mathcal{C}'$ (whose structure is similar to the one of the code constructed in the first part of the proof). We claim that $|\mathcal{C}'| \le |\mathcal{C}|$. First of all, observe that we had $|\mathcal{C} \cap \{V(G(I, A)) \setminus \{I \cup A\}\}| \ge 6\lceil \log_2(|A| + 1)\rceil + 2$. To see this, note that vertex z is the only one separating $\{x, y\}$, and $|\mathcal{C} \cap \{x, y\}| \ge 1$ since $\mathcal{C}$ must dominate x. Similarly, for any $j \in \{1, \ldots, \log_2(|A|+1)\}$, vertices a_j, d_j, g_j are the only ones separating $\{b_j, c_j\}$, $\{e_j, f_j\}$ and $\{h_j, i_j\}$, respectively, and $|\mathcal{C} \cap \{b_j, c_j\}| \ge 1$, $|\mathcal{C} \cap \{e_j, f_j\}| \ge 1$ and $|\mathcal{C} \cap \{h_j, i_j\}| \ge 1$, since $\mathcal{C}$ must dominate c_j, f_j and i_j, respectively.

To fulfill Condition (2), we replace each vertex $i \in I \cap \mathcal{C}'$ by some vertex in A. If $\mathcal{C}' \setminus \{i\}$ is an identifying code, we may just remove i from the code. Otherwise, note that i is not needed for domination since all vertices of I are dominated by z and all vertices of A are dominated by some vertex in $\{a_j \mid 1 \le j \le \lceil \log_2(|A| + 1)\rceil\}$. Hence, i separates i itself from some other vertex i' in I

(indeed, one can check that all other types of pairs which could be separated by i are actually already separated by some vertex of $\mathcal{C}' \cap (V(G(I,A)) \setminus I)$. But then, the pair $\{i, i'\}$ is unique (suppose i separates i itself from two distinct vertices i' and i'' of I, then i' and i'' would not be separated by $\mathcal{C}'$, a contradiction). Since (I, A) admits a discriminating code, there must be some vertex a of A separating i from some i'. Hence we replace i by a. Doing this for every $i \in \mathcal{C}' \cap I$, we get code $\mathcal{C}^*$, and $|\mathcal{C}^*| \leq |\mathcal{C}'| \leq |\mathcal{C}|$.

Using these observations and similar arguments as in the first part of the proof, one can check that the obtained code $\mathcal{C}^*$ is still an identifying code.

To complete the proof, we claim that $\mathcal{C}^* \cap A$ is a discriminating code of (I, A): indeed, all pairs $\{I, I'\}$ of I are separated by $\mathcal{C}^*$. By Condition (1), they must be separated by some vertex of A (note that z is adjacent to all vertices of I), and we are done. □

Theorem 6 proves that Min Id Code for bipartite graphs is NP-hard, and can be used to prove Theorem 4:

Proof (Proof of Theorem 4). We use Theorem 6 to show that any c-approximation algorithm $\mathscr{A}$ for Min Id Code for bipartite graphs can be turned into a $7c$-approximation algorithm for Min Discriminating Code. Min Discriminating Code being log-APX-complete [12] and Min Id Code being in log-APX, we get the claim.

Let (I, A) be a hypergraph with optimal value OPT, and let $G(I, A)$ be the bipartite graph constructed using Reduction 5. By Theorem 6, we have:

$$\gamma^{\text{ID}}(G(I,A)) \leq OPT + 6\lceil \log_2(|A|+1) \rceil + 2. \tag{3}$$

Let $\mathcal{C}$ be an identifying code of $G(I, A)$ computed by $\mathscr{A}$. We have:

$$|\mathcal{C}| \leq c\gamma^{\text{ID}}(G(I,A)). \tag{4}$$

By Theorem 6, we can compute in polynomial time a discriminating code $\mathcal{D}$ of (I, A). Using Inequalities 3 and 4 together with the fact that $\lceil \log_2(|A|) \rceil \leq OPT \leq |\mathcal{D}|$ (Theorem 1), we get:[3]

$$\begin{aligned}
|\mathcal{D}| &\leq |\mathcal{C}| - 6\lceil \log_2(|A|+1) \rceil - 2 \leq c\gamma^{\text{ID}}(G(I,A)) - 6\lceil \log_2(|A|+1) \rceil - 2 \\
&\leq c(OPT + 6\lceil \log_2(|A|+1) \rceil + 2) - 6\lceil \log_2(|A|+1) \rceil - 2 \\
&\leq cOPT + (c-1)(6\lceil \log_2(|A|) \rceil + 8) \leq cOPT + (c-1)(6OPT + 8) \\
&\leq 7cOPT.
\end{aligned}$$
□

2.3 Split Graphs and Co-bipartite Graphs

Theorem 7. Min Id Code *is log-APX-complete for split graphs and for co-bipartite graphs.*

[3] For the last line inequality, we assume here that $OPT \geq 2$.

Theorem 7 is proved using the two following reductions from Min Discriminating Code to Min Id Code.

Reduction 8. Given a hypergraph (I,A), we construct in polynomial time the following split graph $Sp(I,A)$ on $|I|+|A|+6\lceil\log_2(|A|+1)\rceil+1$ vertices, with vertex set $V(Sp(I,A)) = K \cup S$ (K is a clique and S, an independent set). More specifically, $K = I \cup \{u\} \cup \{k_j \mid 1 \leq j \leq 2\lceil\log_2(|A|+1)\rceil\}$ and $S = A \cup \{v\} \cup \{s_j, t_j \mid 1 \leq j \leq 2\lceil\log_2(|A|+1)\rceil\}$.

$Sp(I,A)$ has edge set:

$$\begin{aligned} E(Sp(I,A)) = \{u,v\} \cup\ & E(\mathcal{B}(I,A)) \\ \cup\ & E(\mathcal{LOG}^*(A, \{k_j \mid 1 \leq j \leq 2\lceil\log_2(|A|+1)\rceil\})) \\ \cup\ & \{\{k_j,s_j\},\{k_j,t_j\} \mid 1 \leq j \leq \lceil\log_2(|A|+1)\rceil\} \\ \cup\ & \{a,b \mid a,b \in K, a \neq b\}, \end{aligned}$$

where $\mathcal{B}(I,A)$ denotes the bipartite incidence graph of (I,A) and $E(\mathcal{LOG}^*(A,L))$ denotes the non-singleton bipartite logarithmic identification of A over (A,L) (see Construction 3). The construction is illustrated in Figure 2(a).

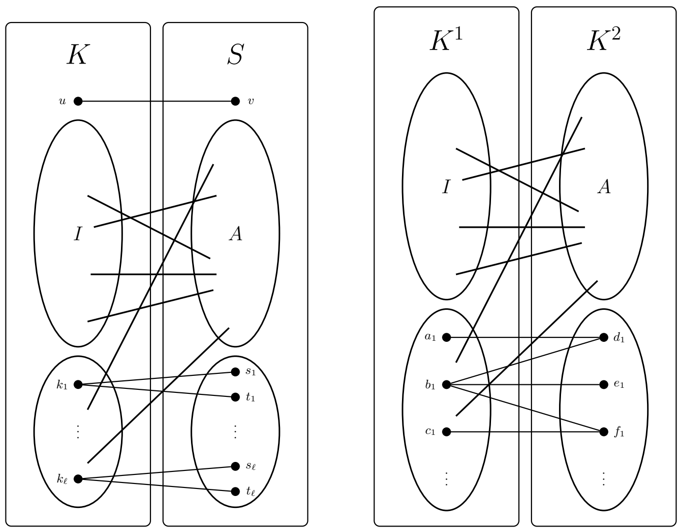
Reduction 9. Given a hypergraph (I,A), we construct in polynomial time the following co-bipartite graph $G(I,A)$ on $|I|+|A|+6\lceil\log_2(|A|+1)\rceil$ vertices, with vertex set $V(G(I,A)) = K^1 \cup K^2$, where K^1 and K^2 are two cliques over the two sets of vertices $K^1 = I \cup \{a_j, b_j, c_j \mid 1 \leq j \leq \lceil\log_2(|A|+1)\rceil\}$ and $K^2 = A \cup \{d_j, e_j, f_j \mid 1 \leq j \leq \lceil\log_2(|A|+1)\rceil\}$.

$G(I,A)$ has edge set:

$$\begin{aligned} E(G(I,A)) = E(\mathcal{B}(I,A)) \cup\ & E(\mathcal{LOG}(A, \{a_j \mid 1 \leq j \leq \lceil\log_2(|A|+1)\rceil\})) \\ \cup\ & \{\{a_j,d_j\},\{b_j,d_j\},\{b_j,e_j\},\{b_j,f_j\},\{c_j,f_j\} \mid 1 \leq j \leq \lceil\log_2(|A|+1)\rceil\} \\ \cup\ & \{x,y \mid x,y \in K^1\} \cup \{x,y \mid x,y \in K^2\}. \end{aligned}$$

where $\mathcal{B}(I,A)$ denotes the bipartite incidence graph of (I,A) and $E(\mathcal{LOG}(A,L))$ denotes the bipartite logarithmic identification of A over (A,L) (see Construction 2). The construction is illustrated in Figure 2(b).

Proof (Sketch of proof of Theorem 7). Reductions 8 and 9 can be used to show that, given a hypergraph (I,A), (I,A) has a discriminating code of size at most k if and only if $Sp(I,A)$ has an identifying code of size at most $k+4\lceil\log_2(|A|+1)\rceil+1$ and $G(I,A)$ has an identifying code of size at most $k+5\lceil\log_2(|A|+1)\rceil-2$, respectively. Moreover these constructions can be performed in polynomial time. Using similar arguments as for bipartite graphs, we can show that any c-approximation algorithm for Min Id Code for split graphs or co-bipartite graphs can be turned into a $5c$- or $6c$-approximation algorithm for Min Discriminating Code, respectively.



(a) Reduction for split graphs (b) Reduction for co-bipartite graphs

Fig. 2. Two reductions from Min Discriminating Code to Min Id Code

3 Reductions for (Planar) Bipartite Graphs of Bounded Maximum Degree and Chordal Bipartite Graphs

In this section, we improve results from the literature by showing that Min Id Code is NP-hard for planar bipartite graphs of maximum degree 3. We also improve and extend the APX-hardness results for Min Id Code for non-bipartite graphs of maximum degree at least 8 from [21] by showing that they are APX-hard even for bipartite graphs of maximum degree 3. Finally, we show that Min Id Code is NP-hard for chordal bipartite graphs.

We will use the standard concept of *L-reductions*, that is widely used to prove APX-hardness of optimization problems.

Definition 10 ([30]). *Let P and Q be two optimization problems. An* L-reduction *from P to Q is a four-tuple (f, g, α, β) where f and g are polynomial time computable functions and α, β are positive constants with the following properties:*

1. *Function f maps instances of P to instances of Q and for every instance I_P of P, $OPT_Q(f(I_P)) \leq \alpha \cdot OPT_P(I_P)$.*
2. *For every instance I_P of P and every solution $SOL_{f(I_P)}$ of $f(I_P)$, g maps the pair $(f(I_P), SOL_{f(I_P)})$ to a solution SOL_{I_P} of I_P such that $|OPT_P(I_P) - |SOL_{I_P}|| \leq \beta \cdot |OPT_Q(f(I_P)) - |SOL_{f(I_P)}||$.*

As discovered in [30], if there exists an L-reduction between two optimization problems P and Q with parameters α and β and it is NP-hard to approximate

P within ratio $r_P = 1 + \delta$, then it is NP-hard to approximate Q within ratio $r_Q = 1 + \frac{\delta}{\alpha\beta}$.

3.1 (Planar) Bipartite Graphs of Maximum Degree 3

Theorem 11. *Reduction 12 applied to graphs of maximum degree 3 is an L-reduction with parameters $\alpha = 4$ and $\beta = 1$. Therefore,* Min Id Code *is APX-complete, even for bipartite graphs of maximum degree 3. Moreover,* Min Id Code *is NP-hard, even for planar bipartite graphs of maximum degree 3.*

We prove Theorem 11 using the following reduction.

Reduction 12. Given a graph G, we construct the graph G' on vertex set

$$V(G') = V(G) \cup \{p_e, q_e \mid e \in E(G)\},$$

and edge set

$$E(G') = \{\{x, p_e\}, \{y, p_e\}, \{p_e, q_e\} \mid e = \{x, y\} \in E(G)\}.$$

The construction is illustrated in Figure 3 (where vertices of G are circled).

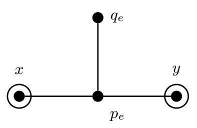


Fig. 3. Reduction 12 from Min Vertex Cover to Min Id Code

For the following claims, let G be a graph and G', the graph obtained from G using Reduction 12.

Claim 13. *Let $\mathcal{N}$ be a vertex cover of G. Using $\mathcal{N}$, one can build an identifying code of G' of size at most $|\mathcal{N}| + |E(G)|$.*

Proof. First of all, we may assume that G is connected. Furthermore, it has no vertex of degree less than 2. Indeed, assuming we have a vertex cover containing a degree 2-vertex x, we can always replace it by its neigbour in the solution. Removing x and its neighbour from the graph, one gets a computationally equivalent instance.

Let $\mathcal{C} = \mathcal{N} \cup \{p_e \mid e \in E(G)\}$. Set $\mathcal{C}$ is an identifying code of G': any original vertex x of G is dominated by the unique set of vertices $\{p_e \mid x \in e, e \in E(G)\}$ (this set having at least two elements). For each edge $\{x, y\} = e \in E(G)$, vertex p_e is dominated by itself and at least one of x, y; q_e is dominated by p_e only. □

Claim 14. *Let $\mathcal{C}$ be an identifying code of G'. One can use $\mathcal{C}$ to build a vertex cover of G of size at most $|\mathcal{C}| - |E(G)|$.*

Proof. For each edge $e = \{x, y\}$ of G, one of p_e, q_e belongs to $\mathcal{C}$, since $\mathcal{C}$ has to dominate q_e. Moreover, one of x, y belongs to $\mathcal{C}$ since p_e, q_e need to be separated. Hence, $\mathcal{C} \cap V(G)$ is a vertex cover of G with size at most $|\mathcal{C}| - |E(G)|$. □

We are now ready to prove Theorem 11. In what follows, let $\tau(G)$ denote the minimum size of a vertex cover of G.

Proof (Proof of Theorem 11). Let G be a graph of maximum degree 3 and G' the graph constructed from G using Reduction 12. We have to prove Properties 1 and 2 from Definition 10.

By Claim 13, given an optimal vertex cover $\mathcal{N}^*$ of G, we can construct an identifying code $\mathcal{C}$ with $\gamma^{\text{ID}}(G') \leq |\mathcal{C}| \leq |\mathcal{N}^*| + |E(G)| = \tau(G) + |E(G)|$. By Claim 14, given an optimal identifying code $\mathcal{C}^*$ of G', we can construct a vertex cover $\mathcal{N}$ of G such that $\tau(G) \leq |\mathcal{N}| \leq |\mathcal{C}^*| - |E(G)| = \gamma^{\text{ID}}(G) - |E(G)|$. Hence:

$$\gamma^{\text{ID}}(G') = \tau(G) + |E(G)|. \tag{5}$$

Proof of Property 1. Since G has maximum degree 3, each vertex can cover at most three edges, hence we have $\tau(G) \geq \frac{|E(G)|}{3}$, so $|E(G)| \leq 3\tau(G)$. Using Equality (5), we get that $\gamma^{\text{ID}}(G') = \tau(G) + |E(G)| \leq 4\tau(G)$.

Proof of Property 2. Let $\mathcal{C}$ be an identifying code of G'. Using Claim 14 applied to $\mathcal{C}$, we obtain a vertex cover $\mathcal{N}$ with $|\mathcal{N}| \leq |\mathcal{C}| - |E(G)|$. By Equality (5), we have $-\tau(G) = |E(G)| - \gamma^{\text{ID}}(G')$. So we obtain:

$$\begin{aligned} |\mathcal{N}| - \tau(G) \leq& |\mathcal{C}| - |E(G)| + |E(G)| - \gamma^{\text{ID}}(G') \\ |\tau(G) - |\mathcal{N}|| \leq& |\gamma^{\text{ID}}(G') - |\mathcal{C}||. \end{aligned}$$

For the second part of the statement, Min Vertex Cover is known to be APX-complete for graphs of maximum degree 3 [11]. It is easy to check that the constructed graphs have maximum degree 3 and are bipartite. For the final part of the statement, we apply Reduction 12 to Min Vertex Cover for *planar* graphs of maximum degree 3, which is known to be NP-hard [19]. Claims 13 and 14 applied to an optimal vertex cover and an optimal identifying code show that $\gamma^{\text{ID}}(G') = \tau(G) + |E(G)|$. □

3.2 Chordal Bipartite Graphs

Theorem 15. Min Id Code *is NP-hard, even for chordal bipartite graphs.*

Reduction 16. Given a graph G, we construct the graph G' on vertex set

$$V(G') = V(G) \cup \{a_x, b_x, c_x, d_x, e_x \mid x \in V(G)\},$$

and edge set

$$\begin{aligned} E(G') =& E(G) \cup \{\{x, a_x\}, \{x, e_x\}, \{a_x, b_x\}, \{a_x, c_x\}, \{a_x, d_x\}, \{e_x, b_x\}, \\ & \{e_x, c_x\}, \{e_x, d_x\} \mid x \in V(G)\}. \end{aligned}$$

The construction is illustrated in Figure 4.

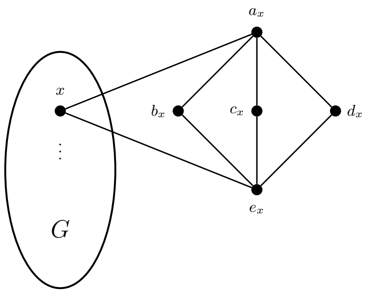


Fig. 4. Reduction from Min Dominating Set to Min Id Code

To prove Theorem 15, we show that G has a dominating set of size at most k if and only if G' has an identifying code of size at most $k + 3|V(G)|$. The proof is omitted due to lack of space.

4 Further Classes of Graphs for Which the Complexities of Min Dominating Set, and Min Id Code Differ

We saw that for co-bipartite graphs, Min Id Code is hard (whereas Min Dominating Set is trivially solvable in polynomial time). In this section, we define a class for which the converse holds: Min Dominating Set is NP-hard, but Min Id Code is solvable in polynomial time. We call these graphs *SC-graphs.*

Definition 17. *A graph G is an* SC-graph *if it can be built from a bipartite graph with parts S and T and an additional set S' with $|S'| = 2|S|$ such that:*

- *for each vertex x of S, there is a path x, u_x, v_x of length 2 starting at x with $u_x, v_x \in S'$, $deg_G(u_x) = 2$ and $deg_G(v_x) = 1$, and*
- *each vertex of T has a distinct neighbourhood within S, and this neighbourhood has at least two elements.*

An example of an SC-graph is pictured in Figure 5. We have the following theorems (proofs are omitted due to lack of space).

Theorem 18. *Let G be an SC-graph built from a bipartite graph with parts S and T, with S_1, the set of all degree 1-vertices of the pendant paths attached to the vertices of S. We have $\gamma^{ID}(G) = 2|S|$ and $S \cup S_1$ is an identifying code of G. Hence,* Min Id Code *can be solved in polynomial time in the class of SC-graphs.*

Theorem 19. Min Dominating Set *is NP-hard in planar (bipartite) SC-graphs of maximum degree 4.*

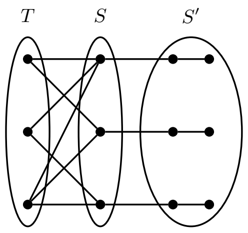


Fig. 5. Example of an SC-graph

5 Open Problems

The complexity for Min Id Code is open for several important input graph classes, as shown in Table 1. Regarding interval graphs, the approximation complexity of Min Id Code is still an open question. It is also of interest to determine the complexity of Min Id Code for permutation graphs (for which Min Dominating Set is polynomial-time solvable [14]). Finally, we remark that Min Dominating Set admits PTAS algorithms for planar graphs [4] and for unit disk graphs [23]. Does the same hold for Min Id Code?

References

1. Auger, D.: Minimal identifying codes in trees and planar graphs with large girth. Eur. J. Combin. 31(5), 1372–1384 (2010)
2. Auger, D., Charon, I., Hudry, O., Lobstein, A.: Complexity results for identifying codes in planar graphs. Int. T. Oper. Res. 17(6), 691–710 (2010)
3. Ausiello, G., Crescenzi, P., Gambosi, G., Kann, V., Marchetti-Spaccamela, A., Protasi, M.: Complexity and approximation. Springer (1999)
4. Baker, B.S.: Approximation algorithms for NP-complete problems on planar graphs. J. ACM 41(1), 153–180 (1994)
5. Berger-Wolf, T.Y., Laifenfeld, M., Trachtenberg, A.: Identifying codes and the set cover problem. In: Proc. 44th Allerton Conf. on Comm. Contr. and Comput. (2006)
6. Booth, K.S., Johnson, H.J.: Dominating sets in chordal graphs. SIAM J. Comput. 11(1), 191–199 (1982)
7. Charbit, E., Charon, I., Cohen, G., Hudry, O., Lobstein, A.: Discriminating codes in bipartite graphs: bounds, extremal cardinalities, complexity. Adv. Math. Commun. 4(2), 403–420 (2008)
8. Clark, B.N., Colbourn, C.J., Johnson, D.S.: Unit disk graphs. Discrete Math. 86(1-3), 165–177 (1990)
9. Charon, I., Hudry, O., Lobstein, A.: Minimizing the size of an identifying or locating-dominating code in a graph is NP-hard. Theor. Comput. Sci. 290(3), 2109–2120 (2003)
10. Chlebík, M., Chlebíková, J.: Approximation hardness of edge dominating set problems. J. Comb. Optim. 11, 279–290 (2006)
11. Chlebík, M., Chlebíková, J.: Approximation hardness of dominating set problems in bounded degree graphs. Inform. Comput. 206, 1264–1275 (2008)
12. De Bontridder, K.M.J., Halldórsson, B.V., Halldórsson, M.M., Hurkens, C.A.J., Lenstra, J.K., Ravi, R., Stougie, L.: Approximation algorithms for the test cover problem. Math. Programm. Ser. B 98, 477–491 (2003)

13. De Ridder, H.N., et al.: Information System on Graph Classes and their Inclusions (ISGCI), `http://www.graphclasses.org`
14. Farber, M., Keil, J.M.: Domination in permutation graphs. J. Algorithm. 6, 309–321 (1985)
15. Foucaud, F.: Combinatorial and algorithmic aspects of identifying codes in graphs. PhD thesis, Université Bordeaux 1, France (2012), `http://tel.archives-ouvertes.fr/tel-00766138`
16. Foucaud, F.: On the decision and approximation complexities for identifying codes and locating-dominating sets in restricted graph classes (2013) (manuscript), `http://www-ma4.upc.edu/~florent.foucaud/Research`
17. Foucaud, F., Gravier, S., Naserasr, R., Parreau, A., Valicov, P.: Identifying codes in line graphs. J. Graph Theor. 73(4), 425–448 (2013), doi:10.1002/jgt.21686
18. Foucaud, F., Mertzios, G., Naserasr, R., Parreau, A., Valicov, P.: Identifying codes in subclasses of perfect graphs. Manuscript (2012)
19. Garey, M.R., Johnson, D.S.: The rectilinear Steiner tree problem is NP-complete. SIAM J. Appl. Math. 32(4), 826–834 (1977)
20. Garey, M.R., Johnson, D.S.: Computers and intractability: a guide to the theory of NP-completeness. W. H. Freeman (1979)
21. Gravier, S., Klasing, R., Moncel, J.: Hardness results and approximation algorithms for identifying codes and locating-dominating codes in graphs. Alg. Oper. Res. 3(1), 43–50 (2008)
22. Haynes, T.W., Hedetniemi, S.T., Slater, P.J. (eds.): Domination in graphs: advanced topics. Marcel Dekker (1998)
23. Hunt III, H.B., Marathe, M.V., Radhakrishnan, V., Ravi, S.S., Rosenkrantz, D.J., Stearns, R.E.: NC-approximation schemes for NP-and PSPACE-hard problems for geometric graphs. J. Algor. 26, 238–274 (1998)
24. Johnson, D.S.: Approximation algorithms for combinatorial problems. J. Comput. Sys. Sci. 9, 256–278 (1974)
25. Karpovsky, M.G., Chakrabarty, K., Levitin, L.B.: On a new class of codes for identifying vertices in graphs. IEEE T. Inform. Theory 44, 599–611 (1998)
26. Laifenfeld, M., Trachtenberg, A.: Identifying codes and covering problems. IEEE T. Inform. Theory 54(9), 3929–3950 (2008)
27. Lobstein, A.: Watching systems, identifying, locating-dominating and discriminating codes in graphs: a bibliography, `http://www.infres.enst.fr/~lobstein/debutBIBidetlocdom.pdf`
28. Müller, H., Brandtädt, A.: The NP-completeness of Steiner Tree and Dominating Set for chordal bipartite graphs. Theor. Comput. Sci. 53, 257–265 (1987)
29. Müller, T., Sereni, J.-S.: Identifying and locating-dominating codes in (random) geometric networks. Comb. Probab. Comput. 18(6), 925–952 (2009)
30. Papadimitriou, C.H., Yannakakis, M.: Optimization, approximation, and complexity classes. J. Comput. Sys. Sci. 43(3), 425–440 (1991)
31. Slater, P.J., Rall, D.F.: On location-domination numbers for certain classes of graphs. Congr. Numer. 45, 97–106 (1984)
32. Suomela, J.: Approximability of identifying codes and locating-dominating codes. Inform. Process. Lett. 103(1), 28–33 (2007)
33. Suomela, J.: Answer to the question "Is the dominating set problem restricted to planar bipartite graphs of maximum degree 3 NP-complete?", `http://cstheory.stackexchange.com/a/2508/1930`
34. Ungrangsi, R., Trachtenberg, A., Starobinski, D.: An implementation of indoor location detection systems based on identifying codes. In: Aagesen, F.A., Anutariya, C., Wuwongse, V. (eds.) INTELLCOMM 2004. LNCS, vol. 3283, pp. 175–189. Springer, Heidelberg (2004)

Expanding the Expressive Power of Monadic Second-Order Logic on Restricted Graph Classes

Robert Ganian[1] and Jan Obdržálek[2]

[1] Vienna University of Technology, Austria*
rganian@gmail.com
[2] Masaryk University, Brno, Czech Republic**
obdrzalek@fi.muni.cz

Abstract. We combine integer linear programming and recent advances in Monadic Second-Order model checking to obtain two new algorithmic meta-theorems for graphs of bounded vertex-cover. The first one shows that the model checking problem for cardMSO$_1$, an extension of the well-known Monadic Second-Order logic by the addition of cardinality constraints, can be solved in FPT time parameterized by vertex cover. The second meta-theorem shows that the MSO partitioning problems introduced by Rao can also be solved in FPT time with the same parameter.

The significance of our contribution stems from the fact that these formalisms can describe problems which are W[1]-hard and even NP-hard on graphs of bounded tree-width. Additionally, our algorithms have only elementary dependence on the parameter and formula. We also show that both results are easily extended from vertex cover to neighborhood diversity.

1 Introduction

It is a well-known result of Courcelle, Makowski and Rotics that MSO$_1$ (and LinEMSO$_1$) model checking is in FPT on graphs of bounded clique-width [4]. However, this leads to algorithms which are far from practical – the time complexity includes a tower of exponents, the height of which depends on the MSO$_1$ formula. Recently it has been shown that much faster model checking algorithms are possible if we consider more powerful parameters such as vertex cover [15] – with only an elementary dependence of the runtime on both the MSO$_1$ formula and parameter.

Vertex cover has been generally used to solve individual problems for which traditional width parameters fail to help (see e.g. [1,6,9,10]). Of course, none of these problems can be described by the standard MSO$_1$ or LinEMSO$_1$ formalism. This raises the following, crucial question: would it be possible to naturally extend the language of MSO$_1$ to include additional well-studied problems without sacrificing the positive algorithmic results on graphs of bounded vertex-cover?

* Robert Ganian acknowledges support by ERC (COMPLEX REASON, 239962).
** Jan Obdržálek is supported by the Czech Science Foundation, project P202/11/0196.

© Springer-Verlag Berlin Heidelberg 2013

We answer this question by introducing cardMSO$_1$ (Definition 2.3) as the extension of MSO$_1$ by linear cardinality constraints – linear inequalities on vertex set cardinalities and input-specified variables. The addition of linear inequalities significantly increases the descriptive power of the logic, and allows to capture interesting problems which are known to be hard on graphs of bounded tree-width. We refer to Section 4 for a discussion of the expressive power and applications of cardMSO$_1$, including a new result for the c-balanced partitioning problem (Theorem 4.1).

The first contribution of the article lies in providing an FPT-time model checking algorithm for cardMSO$_1$ on graphs of bounded vertex cover. This extends the results on MSO$_1$ model checking obtained by Lampis in [15], which introduce an elementary-time FPT MSO$_1$ model checking algorithm parameterized by vertex cover. However, the approach used there cannot be straightforwardly applied to formulas with linear inequalities (cf. Section 3 for further discussion).

Theorem 1.1. *There exists an algorithm which, given a graph G with vertex cover of size k and a* cardMSO$_1$ *formula φ with q variables, decides if $G \models \varphi$ in time $2^{2^{O(k+q)}+|\varphi|} + 2^k|V(G)|$.*

The core of our algorithm rests on a combination of recent advances in MSO$_1$ model checking and the use of Integer Linear Programming (ILP). While using ILP to solve individual difficult graph problems is not new [9], the goal here was to obtain new graph-algorithmic meta-theorems for frameworks containing a wide range of difficult problems. The result also generalizes to the neighborhood diversity parameter introduced in [15] and to MSO$_2$ (as discussed in Section 6).

In the second part of the article, we turn our attention to a different, already studied extension of MSO$_1$: the MSO partitioning framework of Rao [19]. MSO partitioning asks whether a graph may be partitioned into an arbitrary number of sets so that each set satisfies a fixed MSO$_1$ formula, and has been shown to be solvable in XP time on graphs of bounded clique-width. Although MSO partitioning is fundamentally different from cardMSO$_1$ and both formalisms expand the power of MSO$_1$ in different directions, we show that a combination of MSO$_1$ model checking and ILP may also be used to provide an efficient FPT model-checking algorithm for MSO$_1$ partitioning parameterized by vertex-cover or neighborhood diversity.

Theorem 1.2. *There exists an algorithm which, given a graph G with vertex cover of size k and a MSO partitioning instance (φ, r) with q variables, decides if $G \models (\varphi, r)$ in time $2^{2^{O(q2^k)}} \cdot |(\varphi, r)| + 2^k|V(G)|$.*

2 Preliminaries and Definitions

2.1 Vertex Cover and Types

In the following text all graphs are simple and without loops. For a graph G we use $V(G)$ and $E(G)$ to denote the sets of its vertices and edges, and use $N(v)$ to denote the set of neighbors of a vertex $v \in V(G)$.

The graph parameter we are primarily interested in is vertex cover. A key notion related to graphs of bounded vertex cover is the notion of a vertex type.

Definition 2.1 ([15]). *Let G be a graph. Two vertices $u, v \in V$ are of the same* type T *if $N(u) \setminus \{v\} = N(v) \setminus \{u\}$. We use $\mathcal{T}_G$ to denote the set of all types of G (or just $\mathcal{T}$ if G is clear from the context).*

Since each type is associated with its vertices, we also use T to denote the set of vertices of type T. Note that then $\mathcal{T}_G$ forms a partition of the set $V(G)$.

For the sake of simplicity, we adopt the convention that, on a graph with a fixed vertex cover X, we additionally separate each cover vertex into its own type. Then it is easy to see that each type is an independent set, and a graph with vertex cover of size k has at most $2^k + k$ types.

It is often useful to divide vertices of the same type further into subtypes. The subtypes are usually identified by a system of sets, and all subtypes of a given type form a partition of that type:

Definition 2.2. *Let G be a graph and $\mathcal{U} \subseteq 2^{V(G)}$ a set of subsets of $V(G)$. Then two vertices $u, v \in V(G)$* are of the same subtype *(w.r.t. $\mathcal{U}$) if $u, v \in T$ for some $T \in \mathcal{T}_G$ and $\forall U \in \mathcal{U}. u \in U \iff v \in U$. We denote by $\mathcal{S}_T^{\mathcal{U}}$ the set of all subtypes of a type $T \in \mathcal{T}_G$, and also define the set of all subtypes of (w.r.t. $\mathcal{U}$) as $\mathcal{S}_G^{\mathcal{U}}$. (If G and $\mathcal{U}$ are clear form the context, we may write $\mathcal{S}$ instead of $\mathcal{S}_G^{\mathcal{U}}$)*

Finally, notice that $|\mathcal{S}_G^{\mathcal{U}}| \leq 2^{|\mathcal{U}|} |\mathcal{T}_G|$.

2.2 MSO$_1$ and Its Cardinality Extensions

Monadic Second Order logic (MSO$_1$) is a well established logic of graphs. It is the extension of first order logic with quantification over vertices and sets of vertices. MSO$_1$ in its basic form can only be used to describe decision problems. To solve optimization problems we may use LinEMSO$_1$ [4], which is capable of finding maximum- and minimum-cardinality sets satisfying a certain MSO$_1$ formula. This is useful for providing simple descriptions of well-known optimization problems such as Minimum Dominating Set (*adj* is the adjacency relation):

$$\mathrm{Min}(X)\colon \forall a \exists b \in X\colon (adj(a, b) \lor a = b)$$

The crucial point is that LinEMSO$_1$ only allows the optimization of set cardinalities over all assignments satisfying a MSO$_1$ formula. It is not possible to use LinEMSO$_1$ to place restrictions on cardinalities of sets considered in the formula. In fact, such restrictions may be used to describe problems which are W[1]-hard on graphs of bounded tree-width, whereas all LinEMSO$_1$-definable problems may be solved in FPT time even on graphs of bounded clique-width [4].

In this paper we define cardMSO$_1$, an extension of MSO$_1$ which allows restrictions on set cardinalities.

Definition 2.3 (cardMSO$_1$). *The language of* cardMSO$_1$ *logic consists of expressions built from the following elements:*

- *variables $x, y\dots$ for vertices, and $X, Y\dots$ for sets of vertices*
- *the predicates $x \in X$ and $\mathrm{adj}(x, y)$ with the standard meaning*
- *equality for variables, quantifiers $\forall, \exists$ and the standard Boolean connectives*
- *tt and ff as the standard valuation constants representing true and false*
- *the expressions $[\rho_1 \le \rho_2]$, for which the syntax of the ρ expressions is defined as $\rho ::= n \mid |X| \mid \rho + \rho$, where $n \in \mathbb{Z}$ ranges over integer constants and X over (vertex) set variables.*

We call expressions of the form $[\rho_1 \le \rho_2]$ linear (cardinality) constraints*, and write $[\rho_1 = \rho_2]$ as a shorthand for $[\rho_1 \le \rho_2] \wedge [\rho_2 \le \rho_1]$, and $[\rho_1 < \rho_2]$ for $[\rho_1 \le \rho_2] \wedge \neg[\rho_2 \le \rho_1]$. A* formula *$\varphi$ of* cardMSO$_1$ *is an expression of the form $\varphi = \exists Z_1 \dots \exists Z_m.\overline{\varphi}$ such that $\overline{\varphi}$ is a* MSO$_1$ *formula with linear constraints and $Z_1, \dots, Z_m$ are the only variables which appear in the linear constraints.*

To give the semantics *of* cardMSO$_1$ *it is enough to define the semantics of cardinality constraints, the rest follows the standard* MSO$_1$ *semantics. Let $\mathcal{V} : \mathcal{X} \to \mathbb{Z}$ be a valuation of set variables. Then the truth value of $[\rho_1 \le \rho_2]$ is obtained be replacing each occurrence of $|X|$ with the cardinality of $\mathcal{V}(X)$ and evaluating the expression as standard integer inequality.*

To give an example, the following cardMSO$_1$ formula is true if, and only if, a graph is bipartite and both parts have the same cardinality:

$$\exists X_1 \exists X_2.(\forall v \in V.(v \in X_1 \iff \neg v \in X_2)) \wedge [|X_1| = |X_2|] \wedge$$
$$(\forall u \in V.(adj(u, v) \implies ((u \in X_1 \wedge v \in X_2) \vee (u \in X_2 \wedge v \in X_1)))$$

For a cardMSO$_1$ formula $\varphi = \exists Z_1 \dots \exists Z_m.\overline{\varphi}$ we call $\exists Z_1 \dots \exists Z_m$ the *prefix* of φ, and the variables Z_i *prefix variables.* We also put $\mathcal{Z}(\varphi) = \{Z_1, \dots, Z_m\}$, and often write just $\mathcal{Z}$ if φ is clear from the context. Note that, since all prefix variables are existentially quantified set variables, checking whether $G \models \varphi$ (for some graph G) is equivalent to finding a variable assignment $\chi : \mathcal{Z} \to 2^{V(G)}$ such that $G \models_\chi \overline{\varphi}$. We call such χ the *prefix assignment* (for G and φ). Note that the sets $\chi(Z_i)$ can be used to determine subtypes, and therefore we often write $\mathcal{S}_G^\chi$ with the obvious meaning.

2.3 ILP Programming

Integer Linear Programming (ILP) is a well-known framework for formulating problems, and will be used extensively in our approach. We provide only a brief overview of the framework:

Definition 2.4 (p-Variable ILP Feasibility (p-ILP)). *Given matrices $A \in \mathbb{Z}^{m \times p}$ and $b \in \mathbb{Z}^{m \times 1}$, the* p-Variable ILP Feasibility (p-ILP) problem *is whether there exists a vector $x \in \mathbb{Z}^{p \times 1}$ such that $A \cdot x \le b$. The number of variables p is the parameter.*

Lenstra [16] showed that p-ILP, together with its optimization variant p-OPT-ILP, can be solved in FPT time. His running time was subsequently improved by Kannan [14] and Frank and Tardos [11].

Theorem 2.5 ([16,14,11,9]). *p-ILP and p-OPT-ILP can be solved using $O(p^{2.5p+o(p)} \cdot L)$ arithmetic operations in space polynomial in L, L being the number of bits in the input.*

3 cardMSO$_1$ Model Checking

The main purpose of this section is to give a proof of Theorem 1.1. The proof builds upon the following result of Lampis:

Lemma 3.1 ([15]). *Let φ be an* MSO$_1$ *formula with q_S set variables and q_v vertex variables. Let G_1 be a graph, $v \in V(G_1)$ a vertex of type T such that $|T| > 2^{q_S} \cdot q_v$, and G_2 a graph obtained from G_1 by deleting v. Then $G_1 \models \varphi$ iff $G_2 \models \varphi$.*

In other words, a formula φ of MSO$_1$ cannot distinguish between two graphs G_1 and G_2 which differ only in the cardinalities of some types, as long as the cardinalities in both graphs are at least $2^{q_S} \cdot q_v$.

This gives us an efficient algorithm for model checking MSO$_1$ on graphs of bounded vertex cover: We first "shrink" the sizes of types to $2^{q_S} \cdot q_v$ and then recursively evaluate the formula, at each quantifier trying all possible choices for each set and vertex variable[1].

Theorem 3.2 ([15]). *There exists an algorithm which, for a* MSO$_1$ *sentence φ with q variables and a graph G with n vertices and vertex cover of size at most k, decides $G \models \varphi$ in time $2^{2^{O(k+q)}} + O(2^k n)$.*

However, a straightforward adaptation of the approach sketched above does not work with linear constraints. To see this, simply consider e.g. the formula $\exists Z_1 \exists Z_2.[|Z_1| = |Z_2| + 1]$. Changing the cardinality of Z_1 by even a single vertex can alter whether the linear constraint is evaluated as true or false, even if $|Z_1 \cap T|$ is large for some type T. On the other hand, observe that the truth value of a linear inequality $[\rho_1 \leq \rho_2]$ depends only on the prefix variables, not on the rest of the formula. With this in mind, we continue by sketching the general strategy for proving Theorem 1.1:

Given a graph G and a formula φ we begin by creating the graph G_φ from G by reducing the size of each type to $2^{q_S} \cdot q_v$. Since this construction can impact the possible values of linear constraints in φ, we replace each linear constraint with either *tt* or *ff*, effectively claiming which linear constraints we expect to be satisfied in G (for some assignment to prefix variables). We try all 2^l possible truth valuations of linear constraints.

For each MSO$_1$ formula ψ obtained from φ by fixing some truth valuation of linear constraints we now check whether $G_\varphi \models \overline{\psi}$, generating all prefix assignments χ for which $G_\varphi \models_\chi \overline{\psi}$. The remaining step is to check whether some prefix assignment (in G_φ) can be extended to a prefix assignment in G in such a way

[1] Note that both Lemma 3.1 and Theorem 3.2 implicitly utilize the symmetry between vertices of the same type.

that ψ would still hold in G and all linear cardinality constraints would evaluate to their guessed values. This check is performed by the construction of an p-ILP formulation which is feasible if, and only if, there is such an extension.

We now formalize the proof we have just sketched. First, we need a few definitions. We start by formalizing the process of "shrinking" (some types of) a graph.

Definition 3.3. *Given a graph G and a* cardMSO$_1$ *formula $\varphi = \exists Z_1 \dots \exists Z_m . \overline{\varphi}$ with q_v vertex and $m + q_S$ set variables, we define the* reduced graph G_φ *to be the graph obtained from G by the following prescription:*

1. *For each type $T \in \mathcal{T}_G$ s.t. $|T| > 2^{q_S+m} q_v$ we delete the "extra" vertices of type T so that exactly $2^{q_S+m} q_v$ vertices of this type remain, and*
2. *we take the subgraph induced by the remaining vertices.*

Note that vertices of a type with cardinality at most $2^{q_S+m} q_v$ are never deleted in the process of "shrinking" G, and $|V(G_\varphi)| \leq |\mathcal{T}_{G_\varphi}| \cdot 2^{q_S+m} q_v$. Next we formalize the process of fixing the truth values of linear cardinality constraints.

Definition 3.4. *Let $l(\varphi) = \{l_1, \dots, l_k\}$ be the list of all linear cardinality constraints in the formula φ. Let $\alpha : l(\varphi) \to \{tt, ff\}$, called the* pre-evaluation function*, be an assignment of truth values to all linear constraints. Then by $\alpha(\varphi)$ we denote the formula obtained from φ by replacing each linear constraint l_i by $\alpha(l_i)$, and call $\alpha(\varphi)$ the* pre-evaluation of φ. *Note that $\alpha(\varphi)$ is a* MSO$_1$ *formula.*

As we mentioned earlier, the truth value for each linear cardinality constraint depends only on the values of prefix variables. Therefore all linear constraints can be evaluated once we have fixed a prefix assignment. We say that a prefix assignment χ, of a cardMSO$_1$ formula φ, *complies with* a pre-evaluation α if each linear constraint $l \in l(\varphi)$ evaluates to true (under χ) if, and only if, $\alpha(l) = tt$.

We also need a notion of extending a prefix assignment for G_φ to G. In the following definition we use the implicit matching between the subtypes S of G and the subtypes S_φ of its subgraph G_φ.

Definition 3.5. *Given a graph G and a* cardMSO$_1$ *formula $\varphi = \exists Z_1 \dots \exists Z_m . \overline{\varphi}$ with q_v vertex and q_S set variables in $\overline{\varphi}$, we say that a prefix assignment χ for G extends a prefix assignments χ_φ for G_φ if for all $S \in \mathcal{S}_G^\chi$:*

1. *$S = S_\varphi$ if $|S_\varphi| \leq 2^{q_S} q_v$*
2. *$S \supseteq S_\varphi$ if $|S_\varphi| > 2^{q_S} q_v$*

Finally we will need the following statement, which directly follows from the proof of Lemma 3.1 [15]:

Lemma 3.6. *Let $\varphi = \exists Z_1 \dots \exists Z_m . \overline{\varphi}$ be an* MSO$_1$ *formula, with q_S set variables in $\overline{\varphi}$ and q_v vertex variables, and let $\chi_1 : \mathcal{Z} \to 2^{V(G_1)}$ be a prefix assignment in G_1. Let $v \in V(G_1)$ be a vertex of subtype $S \in \mathcal{S}_{G_1}^\chi$ such that $|S| > 2^{q_s} q_v$, and G_2 a graph obtained from G_1 by deleting v. Then $G_1 \models_{\chi_1} \varphi$ iff $G_2 \models_{\chi_2} \varphi$, where χ_2 is the prefix assignment induced by χ_1 on G_2.*

For the remainder of this section let us fix a cardMSO$_1$ formula $\varphi = \exists Z_1 \dots \exists Z_m . \overline{\varphi}$ with q_v vertex variables, q_S set variables in $\overline{\varphi}$ and with linear cardinality constraints $l(\varphi) = \{l_1, \dots, l_k\}$. We are now ready to state the main lemma:

Lemma 3.7. *Let G be a graph, φ be a* cardMSO$_1$ *formula, χ_φ be a prefix assignment for G_φ, and α a pre-evaluation such that $G_\varphi \models_{\chi_\varphi} \alpha(\overline{\varphi})$. Then we can, in time $O(|\mathcal{T}_G| \cdot 2^m |l(\varphi)|)$, construct a p-ILP formulation which is feasible iff χ_φ can be extended to a prefix assignment χ for G such that (a) χ complies with α, and (b) $G \models_\chi \overline{\varphi}$. Moreover, the formulation has $|\mathcal{T}_G| \cdot 2^m$ variables.*

Proof. We start by showing the construction of the p-ILP formulation. The set of variables is created as follows: For each subtype $S \in \mathcal{S}_{G_\varphi}^{\chi_\varphi}$ we introduce a variable x_S which will represent the cardinality of S in G. There are three groups of constraints:

1. We need to make sure that, for each type $T \in \mathcal{T}_G$, the cardinalities of all subtypes of T sum up to the cardinality of a type T. This is easily achieved by including a constraint $\sum_{S \subseteq T} x_S = |T|$ for each type T (note that here $|T|$ is a constant).

2. We need to guarantee that χ extends χ_φ. Therefore we include $x_S = |S_\varphi|$ for each subtype with $|S_\varphi| \leq 2^{q_S} q_v$, and $x_S > |S_\varphi|$ if $|S_\varphi| > 2^{q_S} q_v$.

3. We need to check that χ complies with α, i.e. that each linear constraint l is either true or false based on the value of $\alpha(l)$. For each constraint l we first replace each occurrence of $|Z_i|$ with the sum of cardinalities of all subtypes which are contained in Z_i, i.e. by $\sum_{S_\varphi \subseteq Z_i} x_S$. Then if $\alpha(l) = tt$, we simply insert the modified constraint into the formulation. Otherwise we first reverse the inequality (e.g. $>$ instead of $\leq$), and then also insert it.

To prove the forward implication, let us assume that the p-ILP formulation is feasible. To define χ we start with $\chi = \chi_\varphi$. Then for each subtype $S \in \mathcal{S}_G$ if $x_S > |S_\varphi|$ we add $x_S - |S_\varphi|$ unassigned vertices of type T, where T is the supertype of S. This is always possible thanks to constraints in 1. and 2. The constraints in 3. guarantee that χ complies with α. Finally $G \models_\chi \overline{\varphi}$ by Lemma 3.6.

For the reverse implication let $S \in \mathcal{S}_G$ be the subtype identified by the set $\mathcal{Y} \subset \mathcal{Z}$. Then we put $x_S = |\{v \in V(G) | \forall Z \in \mathcal{Z}. v \in \chi(Z) \iff Z \in \mathcal{Y}\}$, and the p-ILP formulation is satisfiable by our construction. Finally, it is easy to verify that the size of this p-ILP formulation is at most $O(|\mathcal{T}_G| \cdot 2^{q_S} |l(\varphi)|)$. ∎

Proof of Theorem 1.1. We start by constructing G_φ from G, which may be done by finding a vertex cover in time $O(2^k \cdot n)$, dividing vertices into at most $2^k + k$ types (in linear time once we have a vertex cover) and keeping at most $2^{q_S + m} q_v$ vertices in each type.

Now for each pre-evaluation $\alpha : l(\varphi) \to \{tt, ff\}$ we do the following: We run the trivial recursive MSO$_1$ model checking algorithm on G_φ, by trying all possible assignments of vertices of G_φ to set and vertex variables. Each time we find a satisfying assignment, we remember the values of the prefix variables $\mathcal{Z}$, and proceed to finding the next satisfying assignment. Since the prefix variables of φ (and $\alpha(\varphi)$) are existentially quantified, their value is fixed before $\alpha(\overline{\varphi})$ starts being evaluated and therefore is the same at any point of evaluating $\alpha(\overline{\varphi})$. At the end of this stage we end up with at most $(2^{|V(G_\varphi)|})^m$ different satisfying prefix assignments of $Z_1, \ldots, Z_m$ for each pre-evaluation α.

We now need to check whether some combination of a pre-evaluation α and its satisfying prefix assignment χ_φ from the previous step can be extended to a satisfying assignment for $\overline{\varphi}$ and G. This can be done by Lemma 3.7.

To prove correctness, assume that there exists a satisfying assignment χ for G. We create G'_φ by, for each $T \in \mathcal{T}_G$ such that $|T| > 2^{q_S+m}q_v$, inductively deleting vertices from subtypes $S \subseteq T$ such that $|S| > 2^{q_s}q_v$, until $|T| = 2^{q_S+m}q_v$ for every T. Observe that G'_φ is isomorphic to G_φ and that there is a satisfying assignment χ' induced by χ on G'_φ. Then applying the isomorphism to χ' creates a satisfying assignment χ_2 on G_φ, and Lemma 3.7 ensures that our p-ILP formulation is feasible for χ_2.

To compute the time complexity of this algorithm, note that we first need time $O(2^k \cdot n)$ to compute G_φ. Then for each of the $2^{|l|}$ pre-evaluations we compute all the satisfying prefix assignments in time $2^{2^{O(k+q_S+m)}q_v}$ by Theorem 3.2. For each of the at most $(2^{|V(G_\varphi)|})^m = (2^{(2^k+k)\cdot 2^{q_S+m}q_v})^m$ satisfying prefix assignments for G_φ, we check whether it can be extended to an assignment for G, which can be done in time at most $2^{2^{O(k+q_S+m)}}$ by applying Theorem 2.5 on the p-ILP formulation constructed by Lemma 3.7. We therefore need time $O(2^k \cdot n) + 2^m \cdot (2^{2^{O(k+q_S+m)}q_v+|l|} + (2^{(2^k+k)\cdot 2^{q_S+m}q_v})^m \cdot 2^{2^{O(k+q_S+m)}})$, and the bound follows. ∎

Remark: The space complexity of the algorithm presented above may be improved by successively applying Lemma 3.7 to each iteratively computed satisfying prefix assignment (for each pre-evaluation).

Before moving on to the next section, we show how these results can be extended towards well-structured dense graphs. It is easy to verify that the only reference to an actual vertex cover of our graph is in Theorem 3.2 – all other proofs rely purely on bounding the number of types. In [15] Lampis also considered a new parameter called *neighborhood diversity*, which is the number of different types of a graph. I.e. graph G has neighborhood diversity k iff $|\mathcal{T}_G| = k$. Since there exist classes of graphs with unbounded vertex cover but bounded neighborhood diversity (for instance the class of complete graphs), parameterizing by neighborhood diversity may in some cases lead to better results than using vertex cover.

Corollary 3.8. *There exists an algorithm which, given a graph G with neighborhood diversity k and a* cardMSO$_1$ *formula φ with q variables, decides if $G \models \varphi$ in time $2^{k2^{O(q)}+|\varphi|} + k \cdot poly(|V(G)|)$.*

Proof. The proof is nearly identical to the proof of Theorem 1.1. The only change is that we begin by computing the neighborhood diversity and the associated partition into types (which may be done in polynomial time, cf. Theorem 5 in [15]), and we of course use the fact that the number of types is now at most k instead of $2^k + k$. ∎

4 Applications

4.1 Equitable Problems

Perhaps the most natural class of problems which may be captured by cardMSO$_1$ but not by MSO$_1$ (or even MSO$_2$) are equitable problems. Equitable problems generally ask for a partitioning of the graph into a (usually fixed) number of specific sets of equal (± 1) cardinality.

Equitable c-coloring [18] is probably the most extensively studied example of an equitable problem. It asks for a partitioning of a graph into c equitable independent sets and has applications in scheduling, garbage collection, load balancing and other fields (see e.g. [5,3]). While even equitable 3-coloring is W[1]-hard on graphs of bounded tree-width [8], equitable c-coloring may easily be expressed in cardMSO$_1$:

$$\exists A,B,C : part(A,B,C) \wedge \forall x,y : ((x,y \in A \vee x,y \in B \vee x,y \in C) \implies \neg adj(x,y))$$
$$\wedge equi(A,B) \wedge equi(A,C) \wedge equi(B,C), \text{ where}$$

- $part(A,B,C) = \big(\forall x : (x \in A \wedge \neg x \in B \wedge \neg x \in C) \vee (\neg x \in A \wedge x \in B \wedge \neg x \in C) \vee (\neg x \in A \wedge \neg x \in B \wedge x \in C)\big)$.
- $equi(T,U) = (\big[|T| = |U|+1\big] \vee \big[|T|+1 = |U|\big] \vee \big[|T| = |U|\big])$.

Equitable connected c-partition [6] is another studied equitable problem which is known to be W[1]-hard even on graphs of bounded path-width but which admits a simple description in cardMSO$_1$:

$$\exists A,B,C : part(A,B,C) \wedge conn(A) \wedge conn(B) \wedge conn(C)$$
$$\wedge equi(A,B) \wedge equi(A,C) \wedge equi(B,C), \text{ where}$$

- $conn(U) = \big(\forall T : (\forall x : x \in T \implies x \in U) \implies (T = U \vee (\neg \exists a : a \in T) \vee \exists a,b : a \in U \wedge \neg a \in T \wedge b \in T \wedge adj(a,b)\big)$.

4.2 Solution Size as Input

cardMSO$_1$ allows us to restrict the set cardinalities by constants given as part of the input. For instance, the formula below expresses the existence of an Independent Dominating Set of cardinality k:

$$\exists X : (\forall a,b \in X . \neg adj(a,b)) \wedge$$
$$\wedge (\forall b \in V . b \in X \vee (\exists a \in X .\, adj(a,b))) \wedge [|X| = k]$$

Notice that there is an equivalent MSO$_1$ formula for any fixed k. However, the number of variables in the MSO$_1$ formula would depend on k, which would negatively impact on the runtime of model checking. On the other hand, using an input-specified variable only requires us to change a constant in the p-ILP formulation, with no impact on runtime.

4.3 c-Balanced Partitioning

Finally, we show an example of how our approach can be used to obtain new results even for optimization problems, which are (by definition) not expressible by cardMSO$_1$. While the presented algorithm does not rely directly on Theorem 1.1, it is based on the same fundamental ideas.

The problem we focus on is c-balanced partitioning, which asks for a partition of the graph into c equitable sets such that the number of edges between different sets is minimized. The problem was first introduced in [17], has applications in parallel computing, electronic circuit design and sparse linear solvers and has been studied extensively (see e.g. [7,2]). The problem is notoriously hard to approximate, and while an exact XP algorithm exists for the c-balanced partitioning of trees parameterized by c [7,17], no parameterized algorithm is known for graphs of bounded tree-width.

Theorem 4.1. *There exists an algorithm which, given a graph G with vertex cover of size k and a constant c, solves c-balanced partitioning in time $2^{2^{O(k+c)}} + 2^k|V(G)|$.*

Proof. We begin by applying the machinery of Theorem 1.1 to the cardMSO$_1$ formula φ for equitable c-partitioning φ:

$$\exists A, B, C : part(A, B, C) \wedge equi(A, B) \wedge equi(A, C) \wedge equi(B, C)$$

Recall that this means trying all possible assignments of the c set variables in G_φ and testing whether each assignment can be extended to G in a manner satisfying φ. Unlike in Theorem 1.1 though, we need to tweak the p-ILP formulations to not only check the existence of an extension χ for our pre-evaluation α, but also to find the χ which minimizes the size of the cut between vertex sets.

To do so, we add one variable β into the formulation and use a p-OPT-ILP formulation minimizing β. We also add a single equality into the formulation to make β equal to the size of the cut between the c vertex sets. While it is not possible to count the edges directly, the fact that we always have a fixed satisfying prefix assignment in G_φ allows us to calculate β as:

$$\beta = const_0 + \textstyle\sum_{S\in U} const_S\, x_S, \text{ where}$$

- $const_0$ is the number of edges between all pairs of cover vertices with different types (this is obtained from the prefix assignment in G_φ),
- U is the set of subtypes which do not contain cover vertices (recall that each cover vertex has its own subtype),
- x_S is the ILP variable for the cardinality of subtype S (cf. Lemma 3.7),
- For each subtype S, $const_S$ is the number of adjacent vertices in the cover assigned to a different vertex set than S. The values of $const_S$ depend only on the subtype S and the chosen prefix assignment χ_φ in G_φ.

For each satisfying prefix assignment χ_φ in G_φ, the p-OPT-ILP formulation will not only check that this may be extended to an assignment χ in G, but also find the assignment in G which minimizes β. All that is left is to store the best computed β for each satisfying prefix assignment and find the satisfying prefix assignment with minimum β after the algorithm from Theorem 1.1 finishes.

For correctness, assume that there exists a solution which is smaller than the minimal β found by the algorithm. Such a solution would correspond to an assignment of φ in G, which may be reduced to a prefix assignment χ of a pre-evaluation $\alpha(\varphi)$ in G_φ. If we construct the p-ILP formulation for χ and $\alpha(\varphi)$, then the obtained β would equal the size of the cut. However, our algorithm computes the β for all pre-evaluations and satisfying prefix assignments in G_φ, so this gives a contradiction. ∎

5 MSO Partitioning

The MSO (or MSO$_1$) partitioning framework was introduced by Rao in [19] and allows the description of many problems which cannot be formulated in MSO, such as Chromatic number, Domatic number, Partitioning into Cliques etc. While a few of these problems (e.g. Chromatic number) may be solved on graphs of bounded tree-width in FPT time by using additional structural properties of tree-width, MSO partitioning problems in general are W[1]-hard on such graphs.

Definition 5.1 (MSO partitioning). *Given a MSO formula φ, a graph G and an integer r, can $V(G)$ be partitioned into sets $X_1, X_2, \dots, X_r$ such that $\forall i \in \{1, 2, \dots, r\} : X_i \models \varphi$?*

Similarly to Section 3, we will show that a combination of ILP and MSO model checking allows us to design efficient FPT algorithms for MSO partitioning problems on graphs of bounded vertex cover. However, here the total number of sets is specified on the input and so the number of subtypes is not fixed, which prevents us from capturing the cardinality of subtypes by ILP variables. Instead we use the notion of *shape*:

Definition 5.2. *Given a graph G and a MSO$_1$ formula φ with q_S, q_v set and vertex variables respectively, two sets $A, B \subseteq V(G)$ have the same* shape *iff for each type T it holds that either $|A \cap T| = |B \cap T|$ or both $|A \cap T|, |B \cap T| > 2^{q_S} q_v$.*

Let A be any set of shape s. We define $|s \cap T|$, for any type T, as:

$$|s \cap T| = \begin{cases} |A \cap T| & \text{if } |A \cap T| \le 2^{q_S} q_v \\ \top & \text{otherwise} \end{cases}$$

Since φ is a MSO$_1$ formula, from Lemma 3.1 we immediately get:

(5.3) For any two sets $A, B \subseteq V(G)$ of the same shape, it holds that $A \models \varphi$ iff $B \models \varphi$, and

(5.4) Given a MSO formula with q variables, a graph G with vertex cover of size k has at most $(2^{q_S}q_v)^{2^k+k}$ distinct shapes.

With these in hand, we may proceed to:

Proof of Theorem 1.2. First, we consider all at most $(2^{q_S}q_v)^{2^k+k}$ shapes of a set X. For each such shape s, we decide whether a set X_s of shape s satisfies φ by Theorem 3.2. We then create an ILP formulation with one variable x_s for each shape s satisfying φ. The purpose of x_s is to capture the number of sets X_s of shape s in the partitioning of G.

Two conditions need to hold for the number of sets of various shapes. First, the total number of sets needs to be r. This is trivial to model in our formulation by simply adding the constraint that the sum of all x_s equals r.

Second, it must be possible to map each vertex in G to one and only one set X (to ensure that the sets form a partition). Notice that if a partition were to only contain shapes with at most $2^{q_S}q_v$ vertices in each T, then the cardinality of $s \cap T$ would be fixed and so the following set of constraints for each $T \in \mathcal{T}$ would suffice:

$\sum_{\forall s} x_s \cdot |s \cap T| = |T|$

However, in general the partition will also contain shapes with more than $2^{q_S}q_v$ vertices in T, and in this case we do not have access to the exact cardinality of their intersection with T. To this end, for each $T \in \mathcal{T}$ we add the following two sets of constraints:

a) $\sum_{\forall s:|s\cap T| \le 2^{q_S}q_v} x_s \cdot |s \cap T| + \sum_{\forall s:|s\cap T| = \top} x_s \cdot (2^{q_S}q_v) \le |T|$

b) $\sum_{\forall s:|s\cap T| \le 2^{q_S}q_v} x_s \cdot |s \cap T| + \sum_{\forall s:|s\cap T| = \top} x_s \cdot |T| \ge |T|$

Here a) ensures that a partitioning of G into $\sum_{\forall s} x_s$ sets of shape s can "fit" into each T and b) ensures that there are no vertices which cannot be mapped to any set. Notice that if the partition contains any shape s which intersects with T in over $2^{q_S}q_v$ vertices then b) is automatically satisfied, since all unmapped vertices in T can always be added to s without changing $X_s \models \varphi$.

If the p-ILP formulation specified above has a feasible solution, then we can construct a solution to (φ, r) on G by partitioning G as follows: For each shape s we create sets $X_{s,1} \ldots X_{s,x_s}$. Then in each type T in G, we map $|T \cap s|$ yet-unmapped vertices to each set $X_{s,i}$. Constraints a) make sure this is possible. If there are any vertices left unmapped in T, then due to constraint b) there must exist some set X' such that $|X' \cap T| > 2^{q_S}q_v$. We map the remaining unmapped vertices in T to any such set X', resulting in a partition of G. Finally, the fact that each of our sets satisfies φ follows from our selection of shapes.

On the other hand, if a solution to (φ, r) on G exists, then surely each set in the partition has some shape and so it would be found by the p-ILP formulation. The total runtime is the sum of finding the vertex cover, the time of model-checking all the shapes and the runtime of p-ILP, i.e. $O(2^k|V(G)) + 2^{2^{O(k+q)}} \cdot q^{(2^k+k)} + q^{(2^k+k)\cdot q^{O(2^k+k)}}$. ∎

Theorem 1.2 straightforwardly extends to neighborhood diversity as well. Directly bounding the number of types by k results in a bound of $(2^{q_S} q_v)^k$ on the number of distinct shapes in Claim 5.4, and so we get:

Corollary 5.5. *There exists an algorithm which, given a graph G with neighborhood diversity at most k and a MSO partitioning instance (φ, r) with q variables, decides if $G \models (\varphi, r)$ in time $2^{2^{O(qk)}} \cdot |(\varphi, r)| + k|V(G)|$.*

6 Concluding Notes

The article provides two new meta-theorems for graphs of bounded vertex cover. Both considered formalisms can describe problems which are W[1]-hard on graphs of bounded clique-width and even tree-width. On the other hand, we provide FPT algorithms for both cardMSO$_1$ and MSO partitioning which have an elementary dependence on both the formula and parameter (as opposed to the results of Courcelle et al. for tree-width).

The obtained time complexities are actually fairly close to the lower bounds provided in [15] for MSO$_1$ model checking (already $2^{2^{o(k+q)}} \cdot poly(n)$ would violate ETH); this is surprising since the considered formalisms are significantly more powerful than MSO$_1$. Our methods may also be of independent interest, as they show how to use p-ILP as a powerful tool for solving general model checking problems.

Let us conclude with future work and possible extensions of our results. As correctly observed by Lampis in [15], any MSO$_2$ formula can be expressed by MSO$_1$ on graphs of bounded vertex cover. This means that an (appropriately defined) cardMSO$_2$ or MSO$_2$ partitioning formula could be translated to an equivalent cardMSO$_1$ or MSO partitioning formula on graphs of bounded vertex cover. However, the details of these formalisms would need to be laid out in future work.

Another direction would be to extend the results of Theorems 1.1 and 1.2 to more general parameters, such as twin-cover [12] or shrub-depth [13]. Finally, it would be interesting to extend cardMSO$_1$ to capture more hard problems. Theorem 4.1 provides a good indication that the formalism could be adapted to also describe a number of optimization problems on graphs.

References

1. Adiga, A., Chitnis, R., Saurabh, S.: Parameterized algorithms for boxicity. In: Cheong, O., Chwa, K.-Y., Park, K. (eds.) ISAAC 2010, Part I. LNCS, vol. 6506, pp. 366–377. Springer, Heidelberg (2010)
2. Andreev, K., Räcke, H.: Balanced graph partitioning. Theory Comput. Syst. 39(6), 929–939 (2006)
3. Bazewicz, J., Ecker, K., Pesch, E., Schmidt, G., Weglarz, J.: Scheduling Computer and Manufacturing Processes, 2nd edn. Springer-Verlag New York, Inc., Secaucus (2001)

4. Courcelle, B., Makowsky, J.A., Rotics, U.: Linear time solvable optimization problems on graphs of bounded clique-width. Theory Comput. Syst. 33(2), 125–150 (2000)
5. Das, S., Finocchi, I., Petreschi, R.: Conflict-free star-access in parallel memory systems. J. Parallel Distrib. Comput. 66(11), 1431–1441 (2006)
6. Enciso, R., Fellows, M.R., Guo, J., Kanj, I., Rosamond, F., Suchý, O.: What makes equitable connected partition easy. In: Chen, J., Fomin, F.V. (eds.) IWPEC 2009. LNCS, vol. 5917, pp. 122–133. Springer, Heidelberg (2009)
7. Feldmann, A., Foschini, L.: Balanced Partitions of Trees and Applications. In: STACS 2012. Leibniz International Proceedings in Informatics (LIPIcs), vol. 14, pp. 100–111. Schloss Dagstuhl, Dagstuhl (2012)
8. Fellows, M., Fomin, F., Lokshtanov, D., Rosamond, F., Saurabh, S., Szeider, S., Thomassen, C.: On the complexity of some colorful problems parameterized by treewidth. Inf. Comput. 209, 143–153 (2011)
9. Fellows, M.R., Lokshtanov, D., Misra, N., Rosamond, F.A., Saurabh, S.: Graph layout problems parameterized by vertex cover. In: Hong, S.-H., Nagamochi, H., Fukunaga, T. (eds.) ISAAC 2008. LNCS, vol. 5369, pp. 294–305. Springer, Heidelberg (2008)
10. Fiala, J., Golovach, P., Kratochvíl, J.: Parameterized complexity of coloring problems: Treewidth versus vertex cover. Theoret. Comput. Sci. 412(23), 2513–2523 (2011)
11. Frank, A., Tardos, É.: An application of simultaneous diophantine approximation in combinatorial optimization. Combinatorica 7(1), 49–65 (1987)
12. Ganian, R.: Twin-cover: Beyond vertex cover in parameterized algorithmics. In: Marx, D., Rossmanith, P. (eds.) IPEC 2011. LNCS, vol. 7112, pp. 259–271. Springer, Heidelberg (2012)
13. Ganian, R., Hliněný, P., Nešetřil, J., Obdržálek, J., Ossona de Mendez, P., Ramadurai, R.: When trees grow low: Shrubs and fast MSO_1. In: Rovan, B., Sassone, V., Widmayer, P. (eds.) MFCS 2012. LNCS, vol. 7464, pp. 419–430. Springer, Heidelberg (2012)
14. Kannan, R.: Minkowski's convex body theorem and integer programming. Math. Oper. Res. 12, 415–440 (1987)
15. Lampis, M.: Algorithmic meta-theorems for restrictions of treewidth. Algorithmica 64(1), 19–37 (2012)
16. Lenstra, H.: Integer programming with a fixed number of variables. Math. Oper. Res. 8, 538–548 (1983)
17. MacGregor, R.: On partitioning a graph: a theoretical and empirical study. PhD thesis, University of California, Berkeley (1978)
18. Meyer, W.: Equitable coloring. American Mathematical Monthly 80, 920–922 (1973)
19. Rao, M.: MSOL partitioning problems on graphs of bounded treewidth and clique-width. Theoret. Comput. Sci. 377, 260–267 (2007)

Dynamising Interval Scheduling: The Monotonic Case

Alexander Gavruskin[1], Bakhadyr Khoussainov[1], Mikhail Kokho[1], and Jiamou Liu[2]

[1] Department of Computer Science, University of Auckland, New Zealand
[2] School of Computing and Mathematical Sciences, Auckland University of Technology, New Zealand
{a.gavruskin,m.kokho}@auckland.ac.nz, bmk@cs.auckland.ac.nz, jiamou.liu@aut.ac.nz

Abstract. We investigate dynamic algorithms for the interval scheduling problem. We focus on the case when the set of intervals is monotonic. This is when no interval properly contains another interval. We provide two data structures for representing the intervals that allow efficient insertion, removal and various query operations. The first dynamic algorithm, based on the data structure called compatibility forest, runs in amortised time $O(\log^2 n)$ for insertion and removal and $O(\log n)$ for query. The second dynamic algorithm, based on the data structure called linearised tree, runs in time $O(\log n)$ for insertion, removal and query. We discuss differences and similarities of these two data structures through theoretical and experimental results.

1 Introduction

Background. Imagine a number of processes all need to use a particular resource for a period of time. Each process i specifies a starting time $s(i)$ and a finishing time $f(i)$ between which it needs to continuously occupy the resource. The resource cannot be shared by two processes at any instance. One is required to design a scheduler which chooses a subset of these processes so that 1) there is no time conflict between processes in using the resource; and 2) there are as many processes as possible that get chosen.

The above is a typical set-up for the interval scheduling problem, one of the basic problems in the study of algorithms. Formally, given a collection of intervals on the real line all specified by starting and finishing times, the problem asks for a subset of maximal size consisting of pairwise non-overlapping intervals. The interval scheduling problem and its variants appear in a wide range of areas in computer science and applications such as in logistics, telecommunication, and manufacturing. They form an important class of scheduling problems and have been studied under various names and with application-specific constraints [9].

The interval scheduling problem, as stated above, can be solved by a *greedy* scheduler as follows [8]. The scheduler sorts intervals based on their finishing time, and then iteratively selects the interval with the least finishing time that

© Springer-Verlag Berlin Heidelberg 2013

is compatible with the intervals that have already been scheduled. The set of intervals chosen in this manner is guaranteed to have maximal size. This algorithm works in a *static* context in the sense that the set of intervals is given a priori and it is not subject to change.

In a *dynamic* context the instance of the interval scheduling problem is usually changed by a real-time events, and a previously optimal schedule may become not optimal. Examples of such real-time events include job cancelation, arrival of an urgent job, change in job processing time. To avoid the repetitive work of rerunning the static algorithm every time when the problem instance has changed, there is a demand for efficient *dynamic algorithms* for solving the scheduling problem on the changed instances. In this dynamic context, the set of intervals change through a number of *update operations* such as insertion or removal. Our goal is to design data structures that allow us to solve the interval scheduling problem in a dynamic setting.

In our effort to dynamise the interval scheduling problem, we focus on a special class of interval sets which we call *monotonic interval sets.* In monotonic interval sets no interval is properly contained by another interval. Considering monotonic intervals is a natural setting for the problem. For example, if all processes require the same amount of time to be completed, then the set of intervals is monotonic. Moreover, monotonic interval sets are closely related to proper interval graphs. An *interval graph* is an undirected graph whose nodes are intervals and two nodes are adjacent if the two corresponding intervals overlap. A *proper interval graph* is an interval graph for a monotonic set of intervals. There exist linear time algorithms for representing a proper interval graph by a monotonic set of intervals [1,6,2]. Furthermore, solving the interval scheduling problem for monotonic intervals corresponds to finding a maximal independent set in a proper interval graph.

Related work. On a somewhat related work, S. Fung, C. Poon and F. Zheng [3] investigated an online version of interval scheduling problem for weighted intervals with equal length (hence, the intervals are monotonic), and designed randomised algorithms. We also mention that R. Lipton and A. Tompkins [5] initiated the study of online version of the interval scheduling problem. In this version a set of intervals are presented to a scheduler in order of start time. Upon seeing each interval the algorithm must decide whether to include the interval into the schedule.

A related problem on a set of intervals I asks to find a minimal set of points S such that every interval from I intersects with at least one point from S. Such a set S is called a *piercing set* of I. A dynamic algorithm for maintaining a minimal piercing set S is studied in [4]. The dynamic algorithm runs in time $O(|S| \log |I|)$. We remark here that if one has a maximal set J of disjoint intervals in I, one can use J to find a minimal piercing set of I, where each point in the piercing set corresponds to the finishing time of an interval in J in time $O(|J|)$. Therefore our dynamic algorithm can be adapted to one that maintains a minimal piercing set. Our algorithm improves the results in [4] when the interval set I is monotonic.

Kaplan et al. in [7] studied a problem of maintaining a set of nested intervals with priorities. The problem asks for an algorithm that given a point p finds the interval with maximal priority containing p. Similarly to our dynamic algorithm, the solution in [7] also uses dynamic trees to represent a set of intervals.

Our results. We provide two dynamic algorithms for solving the interval scheduling problem on monotonic set of intervals. Both algorithms allow efficient insertion, removal and query operation. Formal explanation are in the next sections.

The first algorithm maintains the *compatibility forest* data structure denoted by CF. We say the *right compatible interval* of an interval i is the interval j such that $f(i) < s(j)$ and there does not exist an interval ℓ such that $f(i) < s(\ell)$ and $f(\ell) < f(j)$. The CF data structure maintains the right compatible interval relation. The implementation of the data structure utilises, nontrivially, the dynamic tree data structure of Sleator and Tarjan [10]. As a result, in **Theorem 4** of Section 3 we prove that the insert and remove operations take amortised time $O(\log^2 n)$ and the query operation takes amortised time $O(\log n)$.

The second dynamic algorithm maintains the *linearised tree* data structure denoted by LT. We say that intervals are *equivalent* if their right compatible intervals coincide. The LT data structure maintains both the right compatibility relation and the equivalence relation. Then, in **Theorem 9** of Section 4 we prove that the insertion, removal and query operations take time amortised $O(\log n)$. However, this comes with a cost. As opposed to the CF data structure that keeps a representation of an optimal set after each update operation, the linearised tree data structure does not explicitly represent the optimal solution.

To test the performance of our algorithms, we carried out experiments on random sequences of update and query operations. The experiments show that the two data structures CF and LT perform similarly. The reason for this is that the first dynamic algorithm based on CF reaches the bound of $\log^2 n$ only on specific sequences of operations, while on uniformly random sequences the algorithm may run much faster. Both algorithms outperform the modified naive algorithm (described in Sec. 2).

Organisation of the paper. Section 2 introduces the problem, monotonic interval sets and the modified naive dynamic algorithm. Section 3 and 4 describe the CF and LT data structures and present our dynamic algorithms. Section 5 extends the data structures by adding the report operation that outputs the full greedy solution. Section 6 discusses the experiments.

2 Preliminaries

Interval scheduling basics. An *interval* is a pair $(s(i), f(i)) \in \mathbb{R}^2$ with $s(i) < f(i)$, where $s(i)$ is the *starting time* and $f(i)$ is the *finishing time* of the interval. We abuse notation and write i for the interval $(s(i), f(i))$. Two intervals i and j are *compatible* if $f(i) < s(j)$ or $f(j) < s(i)$. Otherwise, these two intervals *overlap*. Given a collection of intervals $I = \{i_1, i_2, \ldots, i_k\}$, a *compatible set* of I is a subset $J \subseteq I$ such that the intervals in J are pairwise compatible. An *optimal set* of I

is a compatible set of maximal size. The *interval scheduling problem* consists of designing an algorithm that finds an optimal set.

We recall the greedy algorithm that solves the problem [8]. The algorithm sorts intervals by their finishing time, and then iteratively chooses the interval with the least finishing time compatible with the last selected interval. The set of thus selected intervals is optimal. The algorithm is in $O(n \log n)$ where n is the size of I. If the sorting is already given then the algorithm runs in linear time. Below, we formally define the greedy optimal set found by this greedy algorithm.

Let $\preceq$ be the ordering of the intervals by their finishing time. Throughout, by the *least interval*, the *greatest interval*, the *next interval*, the *previous interval*, we mean the least, greatest, next and previous interval with respect to $\preceq$. Without loss of generality we may assume that the intervals in I have pairwise distinct finishing times. Given the collection I, we inductively define the set $J = \{i_1, i_2, \ldots\}$, the *greedy optimal set* of I, as follows. The interval i_1 is the least interval in I. The interval i_{k+1} is the least interval compatible with i_k such that $i_k \prec i_{k+1}$. The set J obtained this way is an optimal set [8].

Dynamic setting. In this setting the collection I of intervals changes over time. Thus, the input to the problem is an arbitrary sequence $o_1, \ldots, o_m$ of update and query operations described as follows:

- *Update operations*: $\mathsf{insert}(i)$ inserts an interval i and $\mathsf{remove}(i)$ removes an interval i.
- *Query operation*: The operation $\mathsf{query}(i)$ returns true if i belongs to the greedy optimal set and false otherwise.

Our goal is to design algorithms for performing these operations that minimise the total running time. We will use the following data structures.

- *Interval tree.* We maintain the ordered set of intervals I in a balanced binary search tree. We call this tree the *interval tree* and denote it by $T(I)$. The interval tree supports all operations of a binary search tree and performs them in $O(\log n)$ worst-case time.
- *Splay tree.* A *splay tree* is a self-balancing binary search tree for storing linearly ordered objects. In addition to the standard binary search tree operations, the splay tree data structure supports the following operations. Operation $\mathsf{splay}(u)$ reorganises a splay tree so that u becomes the root. Operation $\mathsf{join}(A, B)$ merges two splay trees A and B, such that any element in A is less than any element in B, into one tree. Finally, operation $\mathsf{split}(A, u)$ divides a splay tree into two splay trees $R(u)$ and $L(u)$, where $R(u) = \{x \in A \mid u \leq x\}$ and $L(u) = \{x \in A \mid x < u\}$. All the operations for splay trees take $O(\log n)$ amortised time [11].
- *Dynamic trees.* This data structure maintains a forest. Basic update operations are $\mathsf{link}(v, w)$, which creates an edge from a root v to a vertex w (thus v becomes a child of w) and $\mathsf{cut}(v)$, which deletes the edge from v to its parent. Query operations for dynamic tree depend on specific application. Usually, a query operation searches for a node or an edge on a path from a given node.

For example, operation $\mathsf{findmin}(u)$ returns an edge with a minimal value on a path from u to a root. These operations have $O(\log n)$ amortised time complexity [10].

Monotonic Interval Sets. The set I of intervals is called *monotonic* if no interval in I contains another interval. Since I changes over time through update operations, to preserve monotonicity we assume that the $\mathsf{insert}(i)$ operation never adds an interval i which contains or is contained in an existing interval. Recall that the *right compatible interval* of i, denoted by $\mathsf{rc}(i)$, is the least interval j compatible with i such that $i \prec j$. Similarly, the *left compatible interval* of i, written $\mathsf{lc}(i)$, is the greatest interval j compatible with i such that $j \prec i$.

Monotonicity of I implies an important property of the interval tree $T(I)$: if an interval $i \in T(I)$ is not compatible with an interval j, then the left subtree of i does not contain $\mathsf{rc}(j)$ and the right subtree of i does not contain $\mathsf{lc}(j)$. This allows us to define two efficient operations: $\mathsf{right\_compatible}(j)$, which is defined below, and $\mathsf{left\_compatible}(j)$, which is similar except we replace "$\preceq$" with "$\succeq$" and swap "left" and "right".

Algorithm 1. $\mathsf{right\_compatible}(i)$

```
1: $r \leftarrow \mathsf{nil}$
2: $j \leftarrow$ the root in the interval tree $T(I)$.
3: while $j \neq \mathsf{nil}$ do
4:     if $j \preceq i$ or $j$ overlaps $i$ then
5:         $j \leftarrow$ the right child of $j$
6:     else
7:         $r \leftarrow j$
8:         $j \leftarrow$ the left child of $j$
9: return $r$
```

Lemma 1. *On monotonic set I of intervals the operations* $\mathsf{right\_compatible}(i)$ *and* $\mathsf{left\_compatible}(i)$ *run in time $\Theta(\log n)$ and return* $\mathsf{rc}(i)$ *and* $\mathsf{lc}(i)$ *respectively.*

To prove the lemma we observe that for a monotonic set I of intervals and $i, j \in I$, if i overlaps j, then each of the intervals between i and j overlaps both i and j.

Proof. We only prove the lemma for $\mathsf{right\_compatible}$. The operation takes time $\Theta(\log n)$ as the length of paths from a leaf to the root in $T(I)$ is $\lfloor \log n \rfloor + 1$.

For the correctness of $\mathsf{right\_compatible}$, we use the following loop invariant: *If I contains* $\mathsf{rc}(i)$*, then the subtree rooted at j contains* $\mathsf{rc}(i)$ *or r equals* $\mathsf{rc}(i)$.

Initially, j is the root of $T(I)$, so the invariant holds. Each iteration of the **while** loop executes either line 5 or lines 7-8 of Alg. 1. If line 5 is executed, then we have $j \preceq i$ or j overlaps i. If $j \preceq i$ then all intervals in the left subtree of j are less than i. If $j \succeq i$ but j overlaps i, then by the observation above, all intervals between i and j overlap i. In both cases, none of the intervals in the left subtree of j is $\mathsf{rc}(i)$. Therefore setting j to be the right child of j preserves the invariant.

If lines 7-8 are executed, then we have $j \succeq i$ and j is compatible with i. If there exists an interval that is less than j and compatible with i, then such an interval is in the left subtree of j. If such an interval does not exist, j is the smallest interval which is compatible with i. Therefore setting r to be j and j to be the right child of j preserves the invariant.

Thus, the algorithm outputs $\mathsf{rc}(i)$ if it exists and outputs nil otherwise. Indeed, the loop terminates when $j = \mathsf{nil}$. Hence if the set of intervals I contains $\mathsf{rc}(i)$ then $r = \mathsf{rc}(i)$. If I does not contain $\mathsf{rc}(i)$ then line 5 is executed at every iteration, so $r = \mathsf{nil}$. $\square$

Modified naive dynamic algorithm. A naive dynamic algorithm for the interval scheduling problem is to keep intervals sorted and construct the greedy optimal set from scratch at each query operation. Another modified yet still naive dynamic algorithm is this. Store the greedy optimal set in a self-balancing binary search tree T. After each $\mathsf{insert}(i)$ or $\mathsf{delete}(i)$ operation search for the greatest interval $j_0 \in T$ such that $f(j_0) < s(i)$. Then insert a sequence $j_1 = \mathsf{rc}(j_0), \ldots, j_k = \mathsf{rc}(j_{k-1})$ of intervals into T. The sequence ends with the interval j_k such that $\mathsf{rc}(j_k)$ does not exist or is already in T. While inserting, we remove all intervals between j_0 and j_{k+1} from T. The $\mathsf{query}(i)$ operation of this algorithm takes $O(\log n)$ worst-case time. The $\mathsf{insert}(i)$ and $\mathsf{remove}(i)$ operations take $O(k \log n)$ worst-case time, where k is the number of intervals inserted into T. In Section 6 we compare this modified algorithm with the algorithms provided by the CF and LT data structures.

3 Compatibility Forest Data Structure (CF)

Building the data structure. Let I be a set of intervals. We define the *compatibility forest* as a graph $\mathcal{F}(I) = (V, E)$ where $V = I$ and $(i, j) \in E$ if $j = \mathsf{rc}(i)$. By a forest we mean a directed graph where the edge set contains links from nodes to their parents. We use $p(v)$ to denote the parent of node v. The *roots* and *leaves* are standard notions that we do not define. Figure 1 shows an example of a monotonic set of intervals with its compatibility forest. We note that for every forest one can construct in a linear time a monotonic set of intervals whose compatibility forest coincides (up to isomorphism) with the forest.

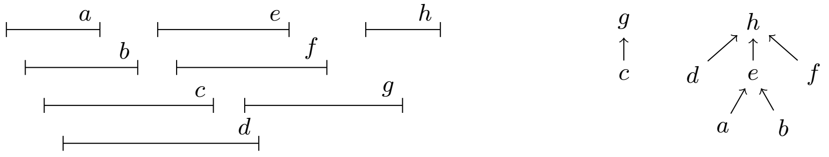


Fig. 1. Example of a monotonic set of intervals and its compatibility forest

A *path* in the compatibility forest $\mathcal{F}(I)$ is a sequence of nodes $i_1, i_2, \ldots, i_k$ where $(i_t, i_{t+1}) \in E$ for any $t = 1, \ldots, k-1$. It is clear that any path in the forest $\mathcal{F}(I)$ consists of compatible intervals. Essentially, the forest $\mathcal{F}(I)$ connects nodes by the greedy rule: for any node i in the forest $\mathcal{F}(I)$, if the greedy rule is applied to i, then the rule selects the parent j of i in the forest. Hence, the longest paths in the compatibility forest correspond to optimal sets of I. In particular, the path starting from the least interval is the greedy optimal set. Our first dynamic algorithm amounts to maintaining this path in the forest $\mathcal{F}(I)$.

We explain how we maintain paths in the compatibility forest $\mathcal{F}(I)$. The representation of the forest is developed from the dynamic tree data structure as in [10]. The idea is to partition the compatibility forest into a set of node-disjoint paths. Paths are defined by two types of edges, *solid edges* and *dashed edges*. Each node in the compatibility forest is required to have at most one incoming solid edge. A sequence of edges $(u_0, u_1), (u_1, u_2), \ldots, (u_{k-1}, u_k)$ where each (u_i, u_{i+1}) is a solid edge is called a *solid path*. A solid path is *maximal* if it is not properly contained in any other solid path. Therefore, the solid edges in $\mathcal{F}(I)$ form several maximal solid paths in the forest. Furthermore, the data structure ensures that each node belongs to some maximal solid path. There is an important subroutine in the dynamic tree data structure called the *expose* operation [10]. The operation starts from a node v and traverses the path from v to the root: while traversing, if the edge $(x, p(x))$ is dashed, we declare $(x, p(x))$ solid and declare the incoming solid edge (if it exists) incident to $p(x)$ dashed. Thus, after exposing node v, all the edges on the path from v to the root become solid. Note that in CF data structure the $p(x)$ and $\mathsf{rc}(x)$ are the same.

To represent CF we use two data structures. The first is the interval tree $T(I)$. The operation right_compatible computes the outgoing dashed edges of the compatibility forest. The second is a set of splay trees. Each splay tree stores the nodes of a maximal solid path in the compatibility forest with the underlying order $\preceq$. We denote by ST_u the splay tree containing the node u.

Dynamic Algorithm 1. We now describe algorithms for maintaining compatibility forest data structure. We call the algorithms queryCF, insertCF and removeCF for the query, insertion, and removal operations, respectively.

The operation queryCF*:* To perform this operation on an interval i, we first find in the interval tree $T(I)$ the minimum element m. We then check if i belongs to the splay tree ST_m. We return true if $i \in \mathrm{ST}_m$; otherwise we return false.

The operation expose*:* To expose an interval i, we find the maximum element j in the splay tree ST_i. Then find the right compatible interval $i' = \mathsf{rc}(j)$. If i' does not exist (that is, j is a root in the compatibility forest), we stop the process. Otherwise, (j, i') is a dashed edge. We split the splay tree at i' into trees $L(i')$ and $R(i')$ and join ST_i with $R(j')$. We then repeat the process taking i' as i.

The operation insertCF*:* To insert an interval i, we add i into the tree $T(I)$. Then we locate the next interval r of i in the ordering $\preceq$. If such r exists, we access r in the splay tree ST_r and find the interval j such that (j, r) is a solid edge. If such a j exists and j is compatible with i, we delete the edge (j, r) and create a new edge (j, i) and declare it solid. We restore the longest path of the compatibility forest by exposing the least interval in $T(I)$.

The operation removeCF*:* To delete an interval i, we delete the incoming and outgoing solid edges of i if such edges exist. We then delete i from the tree $T(I)$. We restore the longest path of the CF by exposing the least interval in $T(I)$.

Correctness of the operations. For correctness, we use the following invariants.

(A1) Every splay tree represents a maximal path formed from solid edges.
(A2) Let m be the least interval in I. The splay tree ST_m contains all intervals on the path from m to the root.

Note that (A2) guarantees that the query operation correctly determines if a given interval i is in the greedy optimal set. The next lemma shows that (A1) and (A2) are invariants indeed and that the operations correctly solve the dynamic monotonic interval scheduling problem.

Lemma 2. *(A1) and (A2) are invariants of* insertCF*,* removeCF*, and* queryCF*.*

Proof. For (A1), first consider the operation of joining two splay trees A and B via the operation expose(i). Let j be the maximal element in A and j' be the minimum element in B. In this case, j' is obtained by the operation right_compatible(j). It is clear that (j, j') is an edge in the forest $\mathcal{F}(I)$. Next, consider the case when we apply insertCF(i) into the splay tree A. In this case, A is $L(r)$ where r is the next interval of i in I. Let j be the previous interval of r in the tree ST_r. By (A1), before inserting i, (j, r) is an edge in $\mathcal{F}(I)$ and thus $r = \mathsf{rc}(j)$. Note we only insert i to $L(r)$ when j is compatible with i. Since $i < r$, after inserting i, i becomes the new right compatible interval of j. So, joining $L(r)$ with i preserves (A1). Operations removeCF(i) and queryCF(i) do not create new edges in splay trees. Thus, (A1) is preserved under all operations.

For (A2), the expose(i) operation terminates when it reaches a root of the compatibility forest. As a result, ST_i contains all nodes on the path from i to the root. Since expose(minimum($T(I)$)) is called at the end of both insertCF(i) and removeCF(i) operations, (A2) is preserved under every operation. □

Complexity. Let n be the number of intervals in I. As discussed in Section 2, all operations for the interval tree have $O(\log n)$ worst case complexity, and all operations for splay trees have $O(\log n)$ amortised complexity. The query operation, involves finding the minimum interval in $T(I)$ and searching i in a splay tree. Hence, the query operation runs in amortised time $O(\log n)$. For each insert and remove operation, we perform a constant number of operations on $T(I)$ and the splay trees plus one expose operation.

To analyse expose operation, define the size size(i) of an interval i to be the number of nodes in the subtree rooted at i in $\mathcal{F}(I)$. Call an edge (i, j) in $\mathcal{F}(I)$ *heavy* if $2 \cdot \mathsf{size}(i) > \mathsf{size}(j)$, and *light* otherwise. It is not hard to see that this partition of edges has the following properties:

($\star$) Every node has at most one incoming heavy edge.
($\star\star$) Every path in the compatibility forest consists of at most $\log n$ light edges.

Lemma 3. *In a sequence of k update operations, the total number of dashed edges, traversed by* expose *operation, is $O(k \log n)$.*

Proof. The number of iterations in expose operation is the number of dashed edges in a path from the least interval to the root. A dashed edge is either heavy or light. From ($\star\star$), there are at most $\log n$ light dashed edges in the path. To count the number of heavy edges, consider the previous update operations. After deletion of i, all children of i become children of the next interval of i. After inserting i, the children of the next interval of i that are compatible with i become children of i. Figure 2 illustrates these structural changes. Thus, an update operation transforms at most $\log n$ light dashed edges to heavy dashed edges in each path, starting at the next interval or the right compatible interval of i. Execution of expose in an update operation creates at most $\log n$ heavy dashed edges from heavy solid edges. Hence, the total number of heavy dashed edges created after k update operations is $O(k \log n)$. □

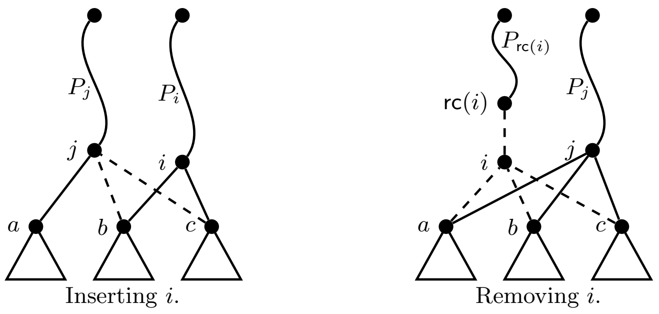


Fig. 2. Redirections of edges in CF, where j is the next interval of i

Lemma 2 and Lemma 3 give us the following theorem:

Theorem 4. *The algorithms* queryCF, insertCF *and* removeCF *solve the dynamic monotonic interval scheduling problem. The algorithms perform insert interval and remove interval operations in $O(\log^2 n)$ amortised time and query operation in $O(\log n)$ amortised time, where n is the size of the set I of intervals.*

Remark. Tarjan and Sleator's dynamic tree data structure has amortised time $O(\log n)$ for update and query operations. To achieve this, the algorithm maintains dashed edges explicitly. Their technique cannot be adapted directly to CF because insertion or removal of intervals may result in redirections of a linear number of edges. Therefore, more care should be taken; for instance, one needs to maintain dashed edges implicitly in $T(I)$ and compute them calling right_compatible operation.

Proposition 5 (Sharpness of the $\log^2 n$ bound). *In* CF *data structure there exists a sequence of k update operations with $\Theta(k \log^2 n)$ total running time.*

Proof. Consider a sequence which creates a set of $n < k$ intervals. We assume that $n = 2^{h+1} - 1$ for an $h \in \omega$. The first n operations of the sequence are insertCF such that the resulted compatibility forest is a perfect binary tree T_n, that is, each internal node of T_n has exactly two children and the height of each leaf in T_n is h. The next $k - n$ operations starting form T_n are pairs of insertCF followed by removeCF. At stage $s = n+2m+1$, insertCF inserts an interval i_s into T_s producing the tree T_{s+1}. The interval i_s is such that in T_{s+1} the path from i_s to the root is of length $h + 1$ and the path consists of dashed edges only. Then, at stage $s + 1$ we delete i_s. This produces a tree T_{s+2} which is a perfect binary tree of height h. We repeat this $k - n$ times. We can select i_s as desired since each perfect binary tree T_s always has a path of length h consisting of dashed edges only. Therefore a sequence of k such operations takes time $\Theta(k \log^2 n)$. □

4 Linearised Tree Data Structure (LT)

Building the data structure. We describe a second dynamic algorithm for solving the monotonic interval scheduling problem. Our goal is to improve the running time for the update operations by introducing the *linearised tree* data structure.

We say that intervals i and j are *equivalent*, written as $i \sim j$, iff $\mathsf{rc}(i) = \mathsf{rc}(j)$. Denote the equivalence class of i by $[i]$. Thus, two intervals are in the same equivalence class if they are siblings in the compatibility forest. In the linearised tree we arrange all intervals in an equivalence class in a path using the $\preceq$-order. The linearised tree consists of all such "linearised" equivalence classes joined by edges. Hence, there are two types of edges in the linearised tree. The first type connects intervals in the same equivalence class. The second type joins the greatest interval in an equivalence class with its right compatible interval. Formally, the *linearised tree* $\mathcal{L}(I)$ is a triple $(I; E_\sim, E_c)$, where $E_\sim$ and E_c are disjoint set of edges such that:

- $(i, j) \in E_\sim$ if and only if $i \sim j$ and i is the previous interval of j. Call i the *equivalent child* of j.
- $(i, j) \in E_c$ if and only if i is the greatest interval in $[i]$ and $j = \mathsf{rc}(i)$. Call i the *compatible child* of j.

Figure 3 shows an example of a linearised tree. We stress three crucial differences between the CF and LT data structures. The first is that a path in a linearised tree may not be a compatible set of intervals. The second is that linearised trees are binary. The third is when we insert or remove an interval we need to redirect at most two existing edges in the linearised tree. We explain the last fact in more details below when we introduce the dynamic algorithm.

$$\begin{tikzcd}
g & & h & \\
c \arrow[u] & d \arrow[ur] & e \arrow[u] & f \arrow[ul] \\
& a \arrow[ur] & & b \arrow[ul]
\end{tikzcd}
\qquad
\begin{tikzcd}
& g \arrow[rrrr] & & & & h \\
c \arrow[ur] & & d \arrow[r] & e \arrow[r] & f \arrow[ur] & \\
& & a \arrow[r] & b \arrow[u] & &
\end{tikzcd}$$

Fig. 3. Example of a compatibility forest (left) and linearised tree (right)

We use the dynamic tree data structure to represent the linearised tree. We also maintain the interval tree $T(I)$ as an auxiliary data structure. The interval tree is used to compute previous and next intervals as well as left compatible and right compatible intervals of a given interval.

Dynamic Algorithm 2. We now describe algorithms for maintaining linearised tree data structure. We call the algorithms queryLT, insertLT and removeLT for the query, insertion, and removal operations, respectively.

The operation queryLT*:* To detect if an interval i is in the greedy optimal set, consider the path P from the least node m to the root in the linearised tree $\mathcal{L}(I)$. If $i \notin P$, return false. Otherwise, consider the direct predecessor j of i in the path P. If j does not exist or $(j, i) \in E_c$, return true. Otherwise, we return false.

Algorithm 2. queryLT(i)

```
 1: $m \leftarrow$ minimum($T(I)$)
 2: if $i = m$ then                                   ▷ $i$ is the least interval
 3:     return true
 4: expose($m$)                         ▷ Make the path from $m$ to the root solid
 5: if $i \neq$ find($\mathrm{ST}_m, i$) then     ▷ $i$ is not on the path from $m$ to the root
 6:     return false
 7: $j \leftarrow$ predecessor($\mathrm{ST}_m, i$)               ▷ $(j, i)$ is an edge in LT
 8: if $i$ is compatible with $j$ then
 9:     return true
10: else
11:     return false
```

Lemma 6. *The operation* queryLT(i) *returns* true *if and only if a given interval i belongs to the greedy optimal set of I.*

The operation insertLT*:* Given i, we insert i into $T(I)$. If i is the greatest interval in $[i]$, then we add the edge $(i, \mathsf{rc}(i))$ into E_c. Otherwise, we add the edge (i, j) to $E_\sim$, where j is the next interval equivalent to i. If i has an equivalent child k then we add the edge (k, i) to $E_\sim$ and delete the old outgoing edge from k in case such edge exists. If i has a compatible child ℓ then we add the edge (ℓ, i) to E_c and delete the old outgoing edge in case such edge exists.

Lemma 7. *The operation* $\mathsf{insertLT}(i)$ *preserves linearised tree data structure.*

The operation $\mathsf{removeLT}$*:* Given i, we delete i from $T(I)$. We delete an edge from i to the parent of i and redirect the edge from the equivalent child j of i to the parent of i. Then we redirect an edge from the compatible child ℓ of i. Removing i may add new intervals to the equivalence class of ℓ. Therefore if ℓ is still the greatest interval in the updated equivalence class, we add an edge $(\ell, \mathsf{rc}(\ell)$ to E_c. Otherwise, we add the edge (i, j) to $E_\sim$, where j is the next interval of ℓ.

Lemma 8. *The operation* $\mathsf{removeLT}(i)$ *preserves linearised tree data structure.*

Lemmas 6-8 lead us to the following theorem:

Theorem 9. *The* $\mathsf{queryLT}$*,* $\mathsf{insertLT}$ *and* $\mathsf{removeLT}$ *operations solve the dynamic monotonic interval scheduling problem in* $O(\log n)$ *amortised time, where* n *is the size of the set* I *of intervals.*

Note. The time complexity of the operations above depends on the type of dynamic trees, representing paths of LT. We can achieve the worst-case bound instead of amortized if we use globally biased trees instead of splay trees [10]. However, after each operation we must ensure that for every pair of edges (v, u) and (w, u) of the linearised tree, nodes v and u are in the same dynamic tree if and only if the numbers of nodes in the subtree rooter at v is greater or equal to the number of nodes in the subtree rooted at u.

5 Extending Functionality of CF and LT Data Structures

The operations $\mathsf{queryCF}$ and $\mathsf{queryLT}$ detect if a given interval i belongs to the current greedy optimal set. Alternatively, another intuitive meaning of the query operation is to report the full greedy optimal set. The report operation, given a set I of monotonic intervals, outputs all the intervals (with their starting and finishing times) in the greedy optimal set. It turns out, our data structures allow an efficient implementation of $\mathsf{reportCF}$ and $\mathsf{reportLT}$ operations.

In the CF data structure, the greedy schedule is the set of intervals on the path from the least node m to the root. This path is represented by the splay tree ST_m and is maintained after every update operation. Therefore the $\mathsf{reportCF}$ amounts to in-order traversal of ST_m. The only thing we need to remember is the root of ST_m after every update operation.

Theorem 10. *The amortised complexity of the* $\mathsf{reportCF}$ *operation is* $O(|\mathrm{ST}_m|)$*, where* ST_m *is the greedy optimal set.*

The theorem above also shows a subtle difference between the two data structures CF and LT. In the LT data structure, in order to perform the $\mathsf{reportLT}$ operation, one needs to examine the path P starting form the minimal element in the tree $\mathcal{L}(I)$. But the path might contain nodes that are not necessarily in the greedy optimal solution. Namely, we need to filter out those nodes v in P

for which there exists a $u \in P$ such that $(u, v) \in E_{\sim}$. Hence, reportLT runs in linear time on the size of P, where in the worst case $P = I$.

In the modified naive algorithm, reporting of the greedy optimal set J takes $O(|J|)$ time. However, to maintain the set J, update operations of the algorithm take $O(k \log n)$ time as described in Section 2, where k is the number of changes in J. In the worst case, $k = \Theta(n)$.

6 Experiments

Here we experimentally compare the naive (N), modified naive (MN), CF and LT data structure algorithms. We implemented these algorithms in Java and run experiments on a laptop with *4GB of RAM* memory and *Intel Core 2 Duo 2130 Mhz, 3MB of L2 cache memory* processor.

In our tests, we measure the average running time in a randomly chosen sequence of $m = n + rn + qn$ operations on initially empty interval set, where n, rn and qm are the number of insert, remove and query operations respectively. Here both q and r are parameters. For insert(i) operation, the starting time $s(i)$ is a random number in $[0, 1]$ and the finishing time is $s(i) + 1/n$. The operations remove and query are always randomly applied to the current set I of intervals. The summary of our experiments are the following:

– CF performs similarly to LT in spite of the fact that CF takes $O(\log^2 n)$ in average as opposed to $O(\log n)$ of LT data structure. Figure 4 shows the results of an experiment with $q = n$ and $r = 0.5n$. Here our sequences of operations do not contain report operation.
– CF data structure performs better than LT if we replace q query operations with q report operations. This confirms our remarks at the end of Section 5.

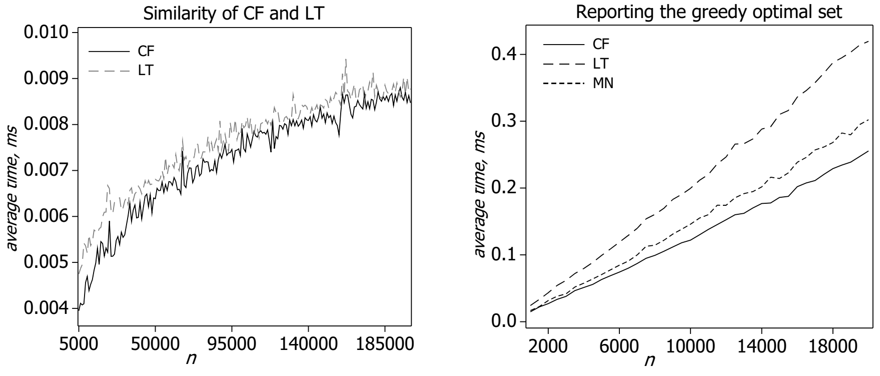


Fig. 4. The parameters of the experiment on the left plot are $q = n$ and $r = 0.5$. The parameters of the experiment on the right plot are $q = n$ and $r = 0$.

- More surprisingly, even MN performs better than LT if we replace q query operations with q report operations.
- CF outperforms MN if we replace q query operations with q report operations.
- N is outperformed by all algorithms in most of the settings.

7 Conclusions and Open Problems

Several directions for further research remain open. One of them is to remove the monotonic restriction and allow intervals to be contained in other intervals. To treat this general case a result in line with Lemma 1 would perhaps play a crucial role. Another direction is to allow an arbitrary, but fixed number of available resources. Data structures, solving these more general interval scheduling problems, would be valuable in practical applications.

References

1. Corneil, D.: A simple 3-sweep LBFS algorithm for the recognition of unit interval graphs. Discrete Applied Mathematics 138(3), 371–379 (2004)
2. Deng, X., Hell, P., Huang, J.: Linear-time representation algorithms for proper circular-arc graphs and proper interval graphs. SIAM Journal on Computing 25(2), 390–403 (1996)
3. Fung, S.P.Y., Poon, C.K., Zheng, F.: Online interval scheduling: Randomized and Multiprocessor cases. In: Lin, G. (ed.) COCOON 2007. LNCS, vol. 4598, pp. 176–186. Springer, Heidelberg (2007)
4. Katz, M.J., Nielsen, F., Segal, M.: Maintenance of a piercing set for intervals with applications. Algorithmica 36(1), 59–73 (2003)
5. Lipton, R., Tompkins, A.: Online interval scheduling. In: Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 302–311 (1994)
6. Heggernes, P., Meister, D., Papadopoulos, C.: A new representation of proper interval graphs with an application to clique-width. Electronic Notes in Discrete Mathematics 32, 27–34 (2009)
7. Kaplan, H., Molad, E., Tarjan, R.: Dynamic rectangular intersection with priorities. In: Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing, pp. 639–648 (June 2003)
8. Kleinberg, J., Tardos, E.: Algorithm Design (2006)
9. Kolen, A., Lenstra, J.K., Papadimitriou, C.H., Spieksma, F.C.: Interval scheduling: A survey. Naval Research Logistics 54(5), 530–543 (2007)
10. Sleator, D., Tarjan, R.: A Data Structure for Dynamic Trees. Journal of Computer and System Sciences 26(3), 362–391 (1983)
11. Sleator, D., Tarjan, R.: Self-adjusting binary search trees. Journal of the ACM 32(3), 652–686 (1985)

Graph Editing to a Fixed Target[⋆]

Petr A. Golovach[1], Daniël Paulusma[2], and Iain Stewart[2]

[1] Department of Informatics, Bergen University,
PB 7803, 5020 Bergen, Norway
petr.golovach@ii.uib.no

[2] School of Engineering and Computing Sciences, Durham University,
Science Laboratories, South Road, Durham DH1 3LE, United Kingdom
{daniel.paulusma,i.a.stewart}@durham.ac.uk

Abstract. For a fixed graph H, the H-Minor Edit problem takes as input a graph G and an integer k and asks whether G can be modified into H by a total of at most k edge contractions, edge deletions and vertex deletions. Replacing edge contractions by vertex dissolutions yields the H-Topological Minor Edit problem. For each problem we show polynomial-time solvable and NP-complete cases depending on the choice of H. Moreover, when G is AT-free, chordal or planar, we show that H-Minor Edit is polynomial-time solvable for all graphs H.

1 Introduction

Graph editing problems are well studied both within algorithmic and structural graph theory and beyond (e.g. [1, 4, 22, 23]), as they capture numerous graph-theoretic problems with a variety of applications. A graph editing problem takes as input a graph G and an integer k, and the question is whether G can be modified into a graph that belongs to some prescribed graph class $\mathcal{H}$ by using at most k operations of one or more specified types. So far, the most common graph operations that have been considered are vertex deletions, edge deletions and edge additions. Well-known problems obtained in this way are Feedback Vertex Set, Odd Cycle Transversal, Minimum Fill-In, and Cluster Editing. Recently, several papers [9, 10, 15–17] appeared that consider the setting in which the (only) permitted type of operation is that of an *edge contraction*. This operation removes the vertices u and v of the edge uv from the graph and replaces them by a new vertex that is made adjacent to precisely those remaining vertices to which u or v was previously adjacent. So far, the situation in which we allow edge contractions *together with* one or more additional types of graph operations has not been studied. This is the main setting that we consider in our paper.

A natural starting approach is to consider families of graphs $\mathcal{H}$ of cardinality 1. For such families, straightforward polynomial-time algorithms exist if the set of permitted operations may only include edge additions, edge deletions and vertex

[⋆] Research supported by EPSRC (EP/G043434/1) and ERC (267959).

© Springer-Verlag Berlin Heidelberg 2013

deletions. However, we show that this is no longer necessarily true in our case, in which we allow both edge contractions and vertex deletions to be applied.

It so happens that setting $\mathcal{H} = \{H\}$ for some graph H, called the *target* graph from now on, yields graph editing problems that are closely related to problems that ask whether a given graph H appears as a "pattern" within another given graph G so that G can be transformed to H via a sequence of operations *without* setting a bound k on the number of operations allowed. These 'unbounded' problems are ubiquitous in computer science, and below we shortly survey a number of known results on them; those results that we will use in our proofs are stated as lemmas.

We start with some additional terminology. A *vertex dissolution* is the removal of a vertex v with exactly two neighbors u and w, which may not be adjacent to each other, followed by the inclusion of the edge uw. If we can obtain a graph H from a graph G by a sequence that on top of vertex deletions and edge deletions may contain operations of one additional type, namely edge contractions or vertex dissolutions, then G contains H as a *minor* or *topological minor*, respectively. For a *fixed* graph H, that is, H is not part of the input, this leads to the decision problems H-Minor and H-Topological Minor, respectively. Grohe, Kawarabayashi, Marx, and Wollan [13] showed that H-Topological Minor can be solved in cubic time for all graphs H, whereas Robertson and Seymour [25] proved the following seminal result.

Lemma 1 ([25]). *H-Minor can be solved in cubic time for all graphs H.*

We say that a containment relation is *induced* if edge deletions are excluded from the permitted graph operations. In the case of minors and topological minors, this leads to the corresponding notions of being an *induced minor* and *induced topological minor*, respectively, with corresponding decision problems H-Induced Minor and H-Induced Topological Minor. In contrast to their non-induced counterparts, the complexity classifications of these two problems have not yet been settled. In fact, the complexity status of H-Induced Minor when H is restricted to be a tree has been open since it was posed at the AMS-IMS-SIAM Joint Summer Research Conference on Graph Minors in 1991. Up until now, only forests on at most seven vertices have been classified [8] (with one forest still outstanding), and no NP-complete cases of forests H are known. The smallest known NP-complete case is the graph H^* on 68 vertices displayed in Figure 1; this result is due to Fellows, Kratochvíl, Middendorf and Pfeiffer [7].

Lemma 2 ([7]). *H^*-Induced Minor is NP-complete.*

Lévêque, Lin, Maffray, and Trotignon [20] gave both polynomial-time solvable and NP-complete cases for H-Induced Topological Minor. In particular they showed the following result, where we denote the complete graph on n vertices by K_n.

Lemma 3 ([20]). *K_5-Induced Topological Minor is NP-complete.*

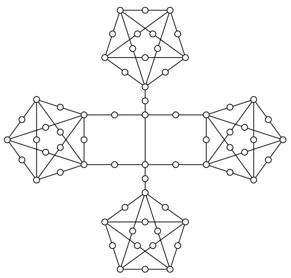


Fig. 1. The smallest graph H^* for which H-Induced Minor is NP-complete [7]

The complexity of H-Induced Topological Minor is still open when H is a complete graph on 4 vertices. Lévêque, Maffray, and Trotignon [21] gave a polynomial-time algorithm for recognizing graphs that neither contain K_4 as an induced topological minor nor a wheel as an induced subgraph. However, they explain that a stronger decomposition theorem (avoiding specific cutsets) is required to resolve the complexity status of K_4-Induced Topological Minor affirmatively.

Before we present our results, we first introduce some extra terminology. Let G be a graph and H a minor of G. Then a sequence of minor operations that modifies G into H is called an *H-minor sequence* or just a minor sequence of G if no confusion is possible. The *length* of an H-minor sequence is the number of its operations. An H-minor sequence is *minimum* if it has minimum length over all H-minor sequences of G. For a fixed graph H, the H-Minor Edit problem is that of testing whether a given graph G has an H-minor sequence of length at most k for some given integer k. Also, for the other containment relations we define such a sequence and corresponding decision problem.

Because any vertex deletion, vertex dissolution and edge contraction reduces a graph by exactly one vertex, any H-induced minor sequence and any H-topological induced minor sequence of a graph G has the same length for any graph H, namely $|V_G|-|V_H|$. Hence, H-Induced Minor Edit and H-Induced Topological Minor Edit are polynomially equivalent to H-Induced Minor and H-Induced Topological Minor, respectively. We therefore do not consider H-Induced Minor Edit and H-Induced Topological Minor Edit, but will focus on the H-Minor Edit and H-Topological Minor Edit problems from now on. For these two problems edge deletions are permitted, and this complicates the situation. For example, let $G = K_n$ and $H = K_1$. Then a minimum H-minor sequence of G consists of $n-1$ vertex deletions, whereas the sequence that consists of $n(n-1)/2$ edge deletions followed by $n-1$ vertex deletions is an H-minor sequence of G that has length $n(n-1)/2+n-1$.

Our Results. In Section 2 we pinpoint a close relationship between H-Minor Edit and H-Induced Minor, and also between H-Topological Minor Edit and H-Induced Topological Minor. We use this observation in Section 3.1, where we show both polynomial-time solvable and NP-complete cases for

H-Minor Edit and H-Topological Minor Edit; note that the hardness results are in contrast with the aforementioned tractable results for H-Minor [25] and H-Topological Minor [13]. There is currently not much hope in settling the complexity of H-Minor Edit and H-Topological Minor Edit for all graphs H, due to their strong connection to H-Induced Minor and H-Induced Topological Minor, the complexity classification of each of which still must be completed. However, in Section 3.2, we are able to show that H-Minor Edit is polynomial-time solvable on AT-free graphs, chordal graphs and planar graphs. In Section 3.3 we discuss parameterized complexity aspects, whereas Section 4 contains our conclusions and directions for further research.

Proofs of theorems and lemmas that are marked with the symbol ★ have been omitted either completely or partially due to space restrictions.

2 Preliminaries

In this section we state some results from the literature and make some basic observations; we will need these results and observations later on. We only consider undirected finite graphs with no loops and with no multiple edges. We denote the vertex set and edge set of a graph G by V_G and E_G, respectively. If no confusion is possible, we may omit subscripts. We refer the reader to the textbook of Diestel [5] for any undefined graph terminology.

The *disjoint union* of two graphs G and H with $V_G \cap V_H = \emptyset$ is the graph $G + H$ that has vertex set $V_G \cup V_H$ and edge set $E_G \cup E_H$. We let P_n and C_n denote the path and cycle on n vertices, respectively, whereas $K_{1,n}$ is the star on $n+1$ vertices; note that $K_{1,1} = P_2$ and $K_{1,2} = P_3$. The subgraph of a graph $G = (V, E)$ induced by a subset $S \subseteq V$ is denoted by $G[S]$. A subgraph G' of a graph G is *spanning* if $V_{G'} = V_G$. Let G be a graph that contains a cycle C as a subgraph. If $|V_C| = |V_G|$ then C is a *hamilton cycle*, and G is called *hamiltonian.* An edge $uv \in E_G \setminus E_C$, with C some cycle and with $u, v \in V_C$, is a *chord* of C.

We will frequently make use of the following observation.

Lemma 4. *If (G, k) is a yes-instance of H-*Minor Edit *or H-*Topological Minor Edit*, for some graph H, then $|V_H| \le |V_G| \le |V_H| + k$.*

Proof. Let (G, k) be a yes-instance of H-Minor Edit or H-Topological Minor Edit for some graph H. An edge contraction, vertex deletion or vertex dissolution reduces a graph by exactly one vertex, whereas an edge deletion does not change the number of vertices. This has the following two implications. First, no graph operation involved increases the number of vertices of a graph. Hence, $|V_H| \le |V_G|$. Second, any H-minor sequence of G has length at least $|V_G| - |V_H|$. Hence, $|V_G| - |V_H| \le k$, or equivalently, $|V_G| \le |V_H| + k$. □

We write $\Pi_1 \le \Pi_2$, for two decision problems Π_1 and Π_2, to denote that Π_2 generalizes Π_1. The following observation shows a close relationship between our two editing problems and the corresponding induced containment problems.

Lemma 5. *Let H be a graph. Then the following two statements hold:*

(i) H-Induced Minor $\leq$ H-Minor Edit.
(ii) H-Induced Topological Minor $\leq$ H-Topological Minor Edit.

Proof. We start with the proof of (i). Let H be a graph, and let G be an instance of H-Induced Minor. We define $k = |V_G| - |V_H|$. We will show that (G, k) is an equivalent instance of H-Minor Edit. If $k < 0$, then G is a no-instance of H-Induced Minor, and by Lemma 4, (G, k) is a no-instance of H-Minor Edit. Suppose that $k \geq 0$. We claim that G contains H as an induced minor if and only if (G, k) has an H-minor sequence of length at most k.

First suppose G contains H as an induced minor. Because one edge contraction or one vertex deletion reduces a graph by exactly one vertex, any H-induced minor sequence of G is an H-minor sequence of G that has length $|V_G|-|V_H| = k$.

Now suppose that G has an H-minor sequence S of length at most k. Because $|V_G| - |V_H| = k$, we find that S contains at least k operations that are vertex deletions or edge contractions. Because S has length at most k, this means that S contains no edge deletions. Hence, S is an H-induced minor sequence. We conclude that G contains H as an induced minor.

The proof of (ii) uses the same arguments as the proof of (i); in particular any vertex dissolution in a graph reduces the graph by exactly one vertex. $\square$

Two disjoint vertex subsets U and W of a graph G are *adjacent* if there exists some vertex in U that is adjacent to some vertex in W. The following alternative definition of being a minor is useful. Let G and H be two graphs. An *H-witness structure* $\mathcal{W}$ is a vertex partition of a subgraph G' of G into $|V_H|$ nonempty sets $W(x)$ called (*H-witness*) *bags*, such that

(i) each $W(x)$ induces a connected subgraph of G, and
(ii) for all $x, y \in V_H$ with $x \neq y$, bags $W(x)$ and $W(y)$ are adjacent in G if x and y are adjacent in H.

An *H-witness structure* $\mathcal{W}$ corresponds to at least one H-minor sequence of G. In order to see this, we can first delete all vertices of $V_G \setminus V_{G'}$, then modify all bags $W(x)$ into singletons via edge contractions and finally delete all edges uv with $u \in W(x)$ and $v \in W(y)$ whenever $uv \notin E_H$. The remaining graph is isomorphic to H. We note that G may have more than one H-witness structure.

An *edge subdivision* is the operation that removes an edge uv of a graph and adds a new vertex w adjacent (only) to u and v. This leads to an alternative definition of being a topological minor, namely that a graph G contains a graph H as a topological minor if and only if G contains a subgraph H' that is a *subdivision* of H; that is, H' can be obtained from H by a sequence of edge subdivisions. A *subdivided star* is a graph obtained from a star after p edge subdivisions for some $p \geq 0$.

Let G be a graph that contains a graph H as a minor. An H-minor sequence S of G is called *nice* if S starts with all its vertex deletions, followed by all its edge contractions and finally by all its edge deletions. It is called *semi-nice* if S starts with all its vertex deletions, followed by all its edge deletions and finally by all its edge contractions. By replacing edge contractions with vertex dissolutions, we obtain the notions of a *nice* and a *semi-nice* topological minor sequence.

Lemma 6 (★). *Let H be a graph and k an integer. If a graph G has an H-minor sequence of length k, then G has a nice H-minor sequence of length at most k.*

We note that a lemma for topological minors similar to Lemma 6 does not hold. For example, build G as follows: take two disjoint copies of K_n, where $n \geq 5$; subdivide an edge in each copy, to introduce new vertices u and v; and join the two new vertices u and v. Build H as two disjoint copies of K_n. If we delete the edge (u, v) of G and next perform two vertex dissolutions (of u and v) then we obtain an H-topological minor sequence of length 3 for G. It is not difficult to see that there is no nice H-topological minor sequence for G of length at most 3. However, for topological minor sequences the following holds.

Lemma 7 (★). *Let H be a graph and k an integer. If a graph G has an H-topological minor sequence of length k, then G has a semi-nice H-topological minor sequence S of length at most k, such that the vertices not deleted by the vertex deletions of S induce a subgraph that contains a subdivision of H as a spanning subgraph.*

We note that a lemma for minors similar to Lemma 7 does not hold. The following example, which we will use later on as well, illustrates this.

Example 1. Let $H = C_6$. We take a cycle C_r for some integer $r \geq 7$. Let u be one of its vertices. Add an edge between u and every (non-adjacent) vertex of the cycle except the two vertices at distance two from u. This yields the graph G. Then $(G, r-5)$ is a yes-instance of H-Minor Edit (via a sequence of $r-6$ edge contractions followed by an edge deletion). It is not difficult to see that every H-minor sequence of length $r-5$ of G must start with $r-6$ edge contractions followed by one edge deletion.

Let $K_{k,\ell}$ be the complete bipartite graph with partition classes of size k and ℓ. Fellows et al. [7] showed that for all graphs H, H-Induced Minor is polynomial-time solvable on planar graphs, that is, graphs that contain neither $K_{3,3}$ nor K_5 as a minor. This result has been extended by van 't Hof et al. [18] to any minor-closed graph class that is *nontrivial*, i.e., that does not contain all graphs.

Lemma 8 ([18]). *Let $\mathcal{G}$ be any nontrivial minor-closed graph class. Then, for all graphs H, the H-*Induced Minor *problem is polynomial-time solvable on $\mathcal{G}$.*

An *asteroidal triple* in a graph is a set of three mutually non-adjacent vertices such that each two of them are joined by a path that avoids the neighborhood of the third, and *AT-free* graphs are exactly those graphs that contain no such triple. A graph is *chordal* if it contains no induced cycle on four or more vertices. We will also need the following two results.

Lemma 9 ([11]). *For all graphs H, the H-*Induced Minor *problem can be solved in polynomial time on AT-free graphs.*

Lemma 10 ([2]). *For all graphs H, the H-*Induced Minor *problem can be solved in polynomial time on chordal graphs.*

3 Complexity Results

In Section 3.1 we consider general input graphs G, whereas in Section 3.2 we consider special classes of input graphs. In Section 3.3 we discuss parameterized complexity aspects.

3.1 General Input Graphs

We first show that the computational complexities of H-Minor Edit and H-Topological Minor Edit may differ from those of H-Minor and H-Topological Minor, respectively.

Theorem 1. *The following two statements hold:*

*(i) There is a graph H for which H-*Minor Edit *is* NP*-complete.*
*(ii) There is a graph H for which H-*Topological Minor Edit *is* NP*-complete.*

Proof. For (i) we take the graph H^* displayed in Figure 1. Then the claim follows from Lemma 2 combined with Lemma 5-(i). For (ii) we take $H = K_5$. Then the claim follows from Lemma 3 combined with Lemma 5-(ii). □

The remainder of this section is devoted to results for some special classes of target graphs H. We start by considering the case when H is a complete graph; note that Theorem 2-(ii) generalizes Theorem 1-(ii).

Theorem 2. *The following two statements hold:*

*(i) K_r-*Minor Edit *can be solved in cubic time for all $r \geq 1$.*
*(ii) K_r-*Topological Minor Edit *can be solved in polynomial time, if $r \leq 3$, and is* NP*-complete, if $r \geq 5$.*

Proof. We first prove (i). Let (G, k) be an instance of K_r-Minor Edit. If $|V_G| - r < 0$ or $|V_G| - r > k$, then (G, k) is a no-instance of K_r-Topological Minor Edit due to Lemma 4. Suppose that $0 \leq |V_G| - r \leq k$. Because we may remove without loss of generality any edge deletions from a K_r-minor sequence of a graph, we find that (G, k) is a yes-instance of K_r-Minor Edit if and only if G contains K_r as a minor. Hence, the result follows after applying Lemma 1.

We now prove (ii). The cases $H = K_1$ and $H = K_2$ are trivial. The case $H = K_3 = C_3$ follows from Theorem 5, which we will prove later. The case $H = K_5$ follows from the proof of Theorem 1-(ii).

Let $r \geq 6$ and assume that K_{r-1}-Topological Minor Edit is NP-complete. Let (G, k) be an instance of K_{r-1}-Topological Minor Edit. From G we construct a graph G' by adding a new vertex v that we make adjacent to all vertices of G. Trivially, if (G, k) is a yes-instance of K_{r-1}-Topological Minor Edit then (G', k) is a yes-instance of K_r-Topological Minor Edit. Conversely, suppose that (G', k) is a yes-instance of K_r-Topological Minor Edit. Then G' has a K_r-topological minor sequence S of length at most k. We modify S as follows. We remove all edge deletions involving v. We replace the dissolution of any vertex $w \neq v$ that involves v with the deletion of w. If v is not deleted by S, then this yields a K_r-topological minor sequence of G' with length at most k that is a K_{r-1}-topological minor sequence of G with length at most k. If v is deleted by S, then we modify S further by removing this vertex deletion. Because v is adjacent to all vertices of G' other than itself, the resulting sequence is a K_{r+1}-topological minor sequence of G' with length at most $k-1$. By extending this sequence with the deletion of one of the remaining vertices not equal to v, we obtain a K_r-topological minor sequence of G' with length at most k that is a K_{r-1}-topological minor sequence of G with length at most k. □

We now consider H-Minor Edit for the case in which H is a path or a star.

Theorem 3 (★). P_r-Minor Edit *and* $K_{1,r}$-Minor Edit *can both be solved in polynomial time for all* $r \geq 1$.

For topological minors we can show a stronger result than Theorem 3.

Theorem 4. *Let H be a subdivided star. Then H-*Topological Minor Edit *is polynomial-time solvable.*

Proof. Let H be a subdivided star. Let (G, k) be an-instance of H-Topological Minor Edit with $|V_G| - |V_H| = k'$. We run the algorithm described below.

If $k' < 0$ or $k' > k$, then we return `no`. Otherwise, that is, if $0 \leq k' \leq k$, we proceed as follows. We consider each subset U of $|V_H|$ vertices of G. We check if $G[U]$ contains an H-topological minor sequence of length at most $k - k'$ (note that such a sequence only involves edge deletions). If so, then we return `yes`. Otherwise, after considering all such subsets U, we return `no`.

We now analyze the running time. Let $|V_G| = n$. Then there are at most $n^{|V_H|}$ possible choices of subgraphs G' on $|V_H|$ vertices. This is a polynomial number, because of our assumption that H is fixed. By the same assumption, every such subgraph G' has constant size, and hence can be processed in polynomial time. We conclude that the total running time is polynomial.

We now prove the correctness of our algorithm. If $k' < 0$ or $k' > k$, then (G, k) is a no-instance of H-Topological Minor Edit due to Lemma 4. Suppose that $0 \leq k' \leq k$. We claim that our algorithm returns `yes` if and only if G has an H-topological minor sequence of length at most k.

First suppose that our algorithm returns `yes`. Then G contains H as a topological minor and our algorithm has obtained H from a graph $G[U]$ on $|V_H|$ vertices by at most $k - k'$ operations (which are all edge deletions), whereas the total number of vertex deletions is $|V_G| - |V_H| = k'$. Hence, G has an H-topological minor sequence of length at most $k - k' + k' = k$.

Now suppose that G has an H-topological minor sequence S of length at most k. By Lemma 7, we may assume without loss of generality that S is a semi-nice H-topological minor sequence S of length at most k, such that the vertices not deleted by the vertex deletions of S induce a subgraph G' that contains a subdivision H' of H as a spanning subgraph. We also assume without loss of generality that the number of vertex deletions in S is maximum over all such H-topological minor sequences of G. Then, because H is a subdivided star, H' must be isomorphic to H (to see this, note that after all vertex deletions of S, we must have a subdivided star, together with possibly some additional edges, and that any subsequent vertex dissolutions can be replaced with vertex deletions so contradicting the maximality of the number of vertex deletions in S). Because $H' \simeq H$ is a spanning subgraph of G', our algorithm will consider G' at some point. Because the remaining operations in S are at most $k - k'$ edge deletions, they form an H-topological minor sequence of G' that has length at most $k - k'$. This will be detected by our algorithm, which will then return `yes`. This completes the proof of Theorem 4. □

Note that Theorem 3 can be generalized to be valid for target graphs H that are linear forests (disjoint unions of paths) and Theorem 4 to be valid for target graphs H that are forests, all connected components of which have at most one vertex of degree at least 3, respectively.

We now consider the case when the target graph H is a cycle and show the following result, which holds for topological minors only.

Theorem 5. C_r-Topological Minor Edit *can be solved in polynomial time for all* $r \geq 3$.

Proof. Let $r \geq 3$. Let (G, k) be an instance of C_r-Topological Minor Edit. We run the following algorithm. Let $k' = |V_G| - r$. If $k' < 0$ or $k' > k$, then we return `no`. Otherwise, we do as follows. We check if G contains C_r as an induced topological minor. If so, then we return `yes`. If not, then we do as follows for each subgraph G' of G with $r \leq |V_{G'}| \leq 2r$. We check if G' contains a C_r-topological minor sequence of length at most $k - (|V_G| - |V_{G'}|)$. If so, then we return `yes`. Otherwise, after having considered all subgraphs G' of G on at most $2r$ vertices, we return `no`.

We first analyze the running time of this algorithm. Checking whether G contains C_r as an induced topological minor is equivalent to checking whether G contains an induced cycle of length at least r; the latter can be done in polynomial time. Suppose G does not contain C_r as an induced topological minor. Then the algorithm considers at most $|V_G|^{2r}$ subgraphs of G, which is a polynomial number because r is fixed. For the same reason, our algorithm can process every such subgraph G' in constant time.

We are left to prove correctness. If $k' < 0$ or $k' > k$, then (G, k) is a no-instance of C_r-Topological Minor Edit due to Lemma 4. Suppose that $0 \leq k' \leq k$. We claim that our algorithm returns `yes` if and only if G has a C_r-topological minor sequence of length at most k.

First suppose that our algorithm returns `yes`. If G contains C_r as an induced topological minor, then G has a C_r-topological minor sequence of length at most $k' \leq k$. Otherwise, G contains a subgraph G' of at most $2r$ vertices that has a C_r-topological minor sequence of length at most $k - (|V_G| - |V_{G'}|)$. Adding the

$|V_G| - |V_{G'}|$ vertex deletions that yielded G' to this sequence gives us a C_r-topological minor sequence of G that has length at most k.

Now suppose that G has a C_r-topological minor sequence of length at most k. By Lemma 7, we may assume without loss of generality that S is a semi-nice H-topological minor sequence S of length at most k, such that the vertices not deleted by the vertex deletions of S induce a subgraph G' that contains a subdivision H' of C_r as a spanning subgraph; note that $H' = C_s$ for some $s \geq r$. We also assume without loss of generality that the number of vertex deletions in S is maximum over all such H-topological minor sequences of G, and moreover, that C_r is obtained from G' by first deleting all chords and then by dissolving $s - r$ vertices.

If G' has at most $2r$ vertices, then the algorithm would consider G' at some point. Because S is a C_r-topological minor sequence of length at most k, we find that G' has a C_r-topological minor sequence of length at most $k - (|V_G| - |V_{G'}|)$. Hence, our algorithm would detect this and return `yes`.

Now suppose that G' has at least $2r + 1$ vertices. Suppose that C_s has at least one chord e. Because $s = |V_{G'}| \geq 2r + 1$, this means that G' contains a smaller cycle C_t on $t \geq r$ vertices that has exactly $t - 1$ edges in common with C_s. We modify S as follows. We first remove all vertices of G' not on C_t. Then we remove all chords of C_t. Finally, we perform $t - r$ vertex dissolutions on C_t in order to obtain a graph isomorphic to C_r. Hence, the sequence S' obtained in this way is a C_r-topological minor sequence of G as well. By our construction, S' is semi-nice. Moreover, because C_s and C_t have $t - 1$ edges in common, any edge deletion in S' is an edge deletion in S as well. Hence S' has length at most k. However, S' contains at least one more vertex deletion than S, because there exists at least one vertex in G' that is not on C_t. This contradicts the maximality of the number of vertex deletions in S. Hence, C_s has no chords. Then C_s is an induced cycle on at least $s \geq r$ vertices in G', and consequently, in G. This means that G contains C_r as an induced topological minor. The algorithm checks this and thus returns `yes`. □

3.2 Input Graphs Restricted to Some Nontrivial Graph Class

Instead of restricting the target graph H to belong to some special graph class, as is done in Section 3.1, we can also restrict the input graph G to some special graph class. In this section we do this for the H-Minor Edit problem.

For the H-Minor Edit problem, we may use the following lemma that strengthens the relationship between H-Minor Edit and H-Induced Minor.

Lemma 11. *Let $\mathcal{G}$ be a graph class and H a graph. If H'-*Induced Minor *is polynomial-time solvable on $\mathcal{G}$ for each spanning supergraph H' of H, then H-*Minor Edit *is polynomial-time solvable on $\mathcal{G}$.*

Proof. Let $\mathcal{G}$ be a graph class and H a graph. Suppose that H'-Induced Minor is polynomial-time solvable on $\mathcal{G}$ for each spanning supergraph H' of H. Let $G \in \mathcal{G}$ and $k \in \mathbb{Z}$ form an instance of H-Minor Edit. Let $k^* = |V_G| - |V_H|$.

If $k^* < 0$ or $k^* > k$, then we return `no`. Suppose $0 \leq k^* \leq k$. Then, for every spanning supergraph H' of H with at most $k - k^*$ additional edges, we check if G contains H' as an induced minor. As soon as we find that this is the case for some H' we return `yes`. Otherwise, after having considered all spanning supergraphs of H, we return `no`.

The running time of the above algorithm is polynomial for the following two reasons. First, because H is fixed, the number of spanning supergraphs H' of H is a constant. Second, by our assumption, we can solve H'-Induced Minor in polynomial time on $\mathcal{G}$ for each spanning supergraph H' of H.

We now prove that our algorithm is correct. If $k^* < 0$ or $k^* > k$ then (G, k) is a no-instance of H-Minor Edit due to Lemma 4. From now on we assume that $0 \leq k^* \leq k$. We claim that our algorithm returns `yes` if and only if G has an H-minor sequence of length at most k.

First suppose that our algorithm returns `yes`. Then there exists a spanning supergraph H' of H with at most $k - k^*$ additional edges, such that H' is an induced minor of G. Let S' be an H'-induced minor sequence of G. Then S' has length exactly $|V_G| - |V_{H'}| = |V_G| - |V_H| = k^*$. We extend S' by deleting the edges in $E_{H'} \setminus E_H$. This yields an H-minor sequence S of G. As $|E_{H'} \setminus E_H| \leq k - k^*$, we find that S has length at most $k^* + k - k^* = k$. Hence, (G, k) is a yes-instance of H-Minor Edit.

Now suppose that G has an H-minor sequence S of length at most k. By Lemma 6, we may assume without loss of generality that S is nice, thus all edge deletions in S take place after all its edge contractions and vertex deletions. Let S' be the prefix of S obtained by omitting the edge deletions of S. Then S' is an H'-induced minor sequence of G for some spanning supergraph H' of H. Because every operation in S' is a vertex deletion or edge contraction, S' has length $|V_G| - |V_{H'}| = |V_G| - |V_H| = k^*$. Because S has length at most k, this means that H' can have at most $k - k^*$ more edges than H. Hence, as our algorithm considers all spanning supergraphs of H with at most $k - k^*$ additional edges, it will return `yes`, as desired. This completes the proof of Lemma 11. □

Combining Lemmas 8–10 with Lemma 11 yields the following result.

Theorem 6. *For all graphs H, the H-*Minor Edit *problem is polynomial-time solvable on*

(i) the class of AT-free graphs
(ii) the class of chordal graphs
(iii) any nontrivial minor-closed class of graphs.

3.3 Parameterized Complexity

A parameterized problem is *fixed-parameter tractable* if an instance (I, p) (where p is the parameter) can be solved in time $f(p) \cdot |V_G|^{O(1)}$ for some function f that only depends on p. Here, the natural parameter is the number of permitted operations k. The following proposition follows immediately from Lemma 4.

Proposition 1. *For all graphs H, the H-*Minor Edit *and H-*Topological Minor Edit *problems are fixed-parameter tractable when parameterized by k.*

We now take $|V_H|$ as the parameter. Theorem 2 shows that in that case H-Minor Edit and H-Topological Minor Edit are fixed-parameter tractable and para-NP-complete, respectively, when H is a complete graph. The running times of the algorithms given by Theorems 3–5 are bounded by $O(n^{|V_H|})$, where H is a path, subdivided star or cycle, respectively. A natural question would be if we can show fixed-parameter tractability with parameter $|V_H|$ for these cases. However, the following result shows that this is unlikely (the class W[1] is regarded as the parameterized analog to NP).

Proposition 2 (★). *For $H \in \{C_r, P_r, K_{1,r}\}$, the problems H-*Minor Edit *and H-*Topological Minor Edit *are* W[1]*-hard when parameterized by r.*

Proof. Due to space restrictions, we only consider the cases $H = P_r$ and $H = C_r$.

Let $H = P_r$. Papadimitriou and Yannakakis [24] proved that the problem of testing whether a graph G contains P_r as an induced subgraph is W[1]-hard when parameterized by r (their proof was not done in terms of parameterized complexity theory and was later rediscovered by Haas and Hoffmann [14]). We observe that a graph G contains P_r as an induced subgraph if and only if G contains a P_r-(topological) minor sequence of length at most $|V_G| - r$. Hence, P_r-Minor Edit and P_r-Topological Minor Edit are W[1]-hard when parameterized by r.

Let $H = C_r$. It is known that the problem of testing whether a graph G contains C_r as an induced subgraph is W[1]-hard when parameterized by r [14, 24]. The corresponding hardness proofs in [14, 24] immediately imply that the problem of testing whether a graph G contains a cycle C_s with $s \geq r$ as an induced subgraph is W[1]-hard as well, when parameterized by r. We observe that a graph G contains a cycle C_s with $s \geq r$ as an induced subgraph if and only if G contains a C_r-(topological) minor sequence of length at most $|V_G| - r$. Hence, C_r-Minor Edit and C_r-Topological Minor Edit are W[1]-hard when parameterized by r. □

4 Conclusions

The ultimate goal is to complete our partial complexity classifications of H-Minor Edit and H-Topological Minor Edit. For this purpose, the following three research questions must be addressed.

1. Is H-Minor Edit polynomial-time solvable for all subdivided stars H?
2. Is C_r-Minor Edit polynomial-time solvable for all $r \geq 3$?
3. Is K_4-Topological Minor Edit polynomial-time solvable?

Answering Question 1 in the affirmative would generalize Theorem 3, in which target graphs H that are paths or stars are considered. Note that generalizing

Theorem 3 to all trees H may be very challenging, because a positive result would solve the aforementioned open problem on H-Induced Minor restricted to trees H, due to Lemma 5-i. The same holds for Question 3: a positive answer to Question 3 would imply membership in P for K_4-Induced Topological Minor, the complexity status of which is a notorious open case (see e.g. [21]). As regards Question 2, Example 1 shows that we cannot guess a bounded set of vertices and consider the subgraph that these vertices induce instead of the whole input graph, as was done for C_r-Topological Minor Edit. Hence, new techniques are needed. So far, we only know that the statement is true if $r \leq 4$.

Proposition 3. C_r-Minor Edit *is polynomial-time solvable if* $r \leq 4$.

Proof. If $r = 3$ then $H = C_3 = K_3$ and we apply Theorem 2-(i). Let $r = 4$, and let (G, k) be an instance of C_4-Minor Edit. We run the following algorithm. Let $k' = |V_G| - r$. If $k' < 0$ or $k' > k$, then we return `no` due to Lemma 4. Otherwise, we do as follows. We check if G contains C_4 as an induced minor. If so then we return `yes`. Note that this is equivalent to checking if G contains an induced cycle on at least four vertices, which can be done in polynomial time. If not then G is chordal, and we apply Theorem 6-(ii). □

Another question is whether we can prove an analog of Theorem 6 for H-Topological Minor Edit. It is known that for all graphs H, the H-Induced Topological Minor problem is polynomial-time solvable for AT-free graphs [12], chordal graphs [2] and planar graphs [19]. However, as noted in Section 2, we cannot always guarantee the existence of a nice topological minor sequence of sufficiently small length. Hence, our proof technique used to prove Theorem 6 can no longer be applied.

Finally, we can consider other graph containment relations as well. An *edge lift* removes two edges uv and vw that share a common vertex v and adds an edge between the other two vertices u and w involved (should this edge not exist already). A graph G contains a graph H as an *immersion* if G can be modified into H by a sequence of operations consisting of vertex deletions, edge deletions and edge lifts. If edge deletions are not allowed then G contains H as an *induced immersion*. It is known that the corresponding decision problems H-Immersion [13] and H-Induced Immersion [3] polynomial-time solvable for all fixed graphs H (the first problem can even be solved in cubic time [13]). What is the computational complexity of H-Immersion Edit and H-Induced Immersion Edit? The main difficulty is that for both problems, we can swap neither edge lifts with vertex deletions nor edge deletions with vertex deletions.

References

1. Alon, N., Shapira, A., Sudakov, B.: Additive approximation for edge-deletion problems. In: Proc. FOCS 2005, pp. 419–428. IEEE Computer Society (2005)
2. Belmonte, R., Golovach, P.A., Heggernes, P., van, P.: Detecting fixed patterns in chordal graphs in polynomial time. Algorithmica (to appear)

3. Belmonte, R., van 't Hof, P., Kamiński, M.: Induced Immersions. In: Chao, K.-M., Hsu, T.-S., Lee, D.-T. (eds.) ISAAC 2012. LNCS, vol. 7676, pp. 299–308. Springer, Heidelberg (2012)
4. Burzyn, P., Bonomo, F., Duran, G.: NP-completeness results for edge modification problems. Discrete Applied Mathematics 154, 1824–1844 (2006)
5. Diestel, R.: Graph Theory, Electronic Edition. Springer (2005)
6. Downey, R.G., Fellows, M.R.: Parameterized complexity. Monographs in Computer Science. Springer, New York (1999)
7. Fellows, M.R., Kratochvíl, J., Middendorf, M., Pfeiffer, F.: The complexity of induced minors and related problems. Algorithmica 13, 266–282 (1995)
8. Fiala, J., Kamiński, M., Paulusma, D.: Detecting induced star-like minors in polynomial time. Journal of Discrete Algorithms 17, 74–85 (2012)
9. Golovach, P.A., van 't Hof, P., Paulusma, D.: Obtaining planarity by contracting few edges. Theoretical Computer Science 476, 38–46 (2013)
10. Golovach, P.A., Kamiński, M., Paulusma, D., Thilikos, D.M.: Increasing the minimum degree of a graph by contractions. Theoretical Computer Science (to appear)
11. Golovach, P.A., Kratsch, D., Paulusma, D.: Detecting induced minors in AT-free graphs. Theoretical Computer Science (to appear)
12. Golovach, P.A., Paulusma, D., van Leeuwen, E.J.: Induced disjoint paths in AT-free graphs. In: Fomin, F.V., Kaski, P. (eds.) SWAT 2012. LNCS, vol. 7357, pp. 153–164. Springer, Heidelberg (2012)
13. Grohe, M., Kawarabayashi, K., Marx, D., Wollan, P.: Finding topological subgraphs is fixed-parameter tractable. In: Proc. STOC 2011, pp. 479–488 (2011)
14. Haas, R., Hoffmann, M.: Chordless paths through three vertices. Theoretical Computer Science 351, 360–371 (2006)
15. Heggernes, P., van 't Hof, P., Lévêque, B., Paul, C.: Contracting chordal graphs and bipartite graphs to paths and trees. Discrete Applied Mathematics (to appear)
16. Heggernes, P., van 't Hof, P., Lokshtanov, D., Paul, C.: Obtaining a bipartite graph by contracting few edges. In: Proc. FSTTCS 2011, pp. 217–228 (2011)
17. Heggernes, P., van 't Hof, P., Lévêque, B., Lokshtanov, D., Paul, C.: Contracting graphs to paths and trees. Algorithmica (to appear)
18. van 't Hof, P., Kamiński, M., Paulusma, D., Szeider, S., Thilikos, D.M.: On graph contractions and induced minors. Discrete Applied Mathematics 160, 799–809 (2012)
19. Kobayashi, Y., Kawarabayashi, K.: A linear time algorithm for the induced disjoint paths problem in planar graphs. Journal of Computer and System Sciences 78, 670–680 (2012)
20. Lévêque, B., Lin, D.Y., Maffray, F., Trotignon, N.: Detecting induced subgraphs. Discrete Applied Mathematics 157, 3540–3551 (2009)
21. Lévêque, B., Maffray, F., Trotignon, N.: On graphs with no induced subdivision of K_4. Journal of Combinatorial Theory, Series B 102, 924–947 (2012)
22. Lewis, J.M., Yannakakis, M.: The node-deletion problem for hereditary properties is NP-complete. Journal of Computer and System Sciences 20, 219–230 (1980)
23. Natanzon, A., Shamir, R., Sharan, R.: Complexity classification of some edge modification problems. Discrete Applied Mathematics 113, 109–128 (2001)
24. Papadimitriou, C.H., Yannakakis, M.: On limited nondeterminism and the complexity of the VC-dimension. Journal of Computer and System Sciences 43, 425–440 (1991)
25. Robertson, N., Seymour, P.D.: Graph minors. XIII. The disjoint paths problem. Journal of Combinatorial Theory, Series B 63, 65–110 (1995)

Tight Bound on the Diameter of the Knödel Graph

Hayk Grigoryan and Hovhannes A. Harutyunyan

Department of Computer Science and Software Engineering, Concordia University, Montreal, Quebec, Canada H3G 1M8
h_grig@encs.concordia.ca, haruty@cse.concordia.ca

Abstract. The Knödel graph $W_{\Delta,n}$ is a regular graph of even order and degree Δ where $2 \leq \Delta \leq \lfloor \log_2 n \rfloor$. Despite being a highly symmetric and widely studied graph, the diameter of $W_{\Delta,n}$ is known only for $n = 2^{\Delta}$. In this paper we present a tight upper bound on the diameter of the Knödel graph for general case. We show that the presented bound differs from the diameter by at most 2 when $\Delta < \alpha \lfloor \log_2 n \rfloor$ for some $0 < \alpha < 1$ where $\alpha \to 1$ when $n \to \infty$. The proof is constructive and provides a near optimal diametral path for the Knödel graph $W_{\Delta,n}$.

Keywords: Knödel graph, diametral path, broadcasting, minimum broadcast graph.

1 Introduction

The Knödel graph $W_{\Delta,n}$ is a regular graph of even order and degree Δ where $2 \leq \Delta \leq \lfloor \log n \rfloor$ (all logarithms in this paper are base 2, unless otherwise specified). It was introduced by Knödel for $\Delta = \lfloor \log n \rfloor$ and was used in an optimal gossiping algorithm [17]. For smaller Δ, the Knödel graph is defined in [8].

The Knödel graph was widely studied as an interconnection network topology and proven to be having good properties in terms of broadcasting and gossiping. The Knödel graph $W_{\Delta,2^{\Delta}}$ is one of the three non-isomorphic infinite graph families known to be minimum broadcast and gossip graphs (graphs that have the smallest possible broadcast and gossip times and the minimum possible number of edges). The other two families are the well known hypercube [5] and the recursive circulant graph [18].

The Knödel graph $W_{\Delta-1,2^{\Delta}-2}$ is a minimum broadcast and gossip graph also for $n = 2^{\Delta} - 2 (\Delta \geq 2)$ [16],[3]. One of the advantages of the Knödel graph, as a network topology, is that it achieves the smallest diameter among known minimum broadcast and gossip graphs for $n = 2^{\Delta} (\Delta \geq 1)$. All the minimum broadcast graph families — k-dimensional hypercube, $C(4, 2^k)$-recursive circulant graph and $W_{k,2^k}$ Knödel graph — have the same degree k, but have diameters equal to k, $\lceil \frac{3k-1}{4} \rceil$ and $\lceil \frac{k+2}{2} \rceil$ respectively. A detailed description of some graph theoretic and communication properties of these three graph families and their comparison can be found in [6].

© Springer-Verlag Berlin Heidelberg 2013

As shown in [1], the edges of the Knödel graph can be grouped into dimensions which are similar to hypercube dimensions. This allows to use these dimensions in a similar manner as in hypercube for broadcasting and gossiping. Unlike the hypercube, which is defined only for $n = 2^k$, the Knödel graph is defined for any even number of vertices. Properties such as small diameter, vertex transitivity as a Cayley graph [15], high vertex and edge connectivity, dimensionality, embedding properties [6] make the Knödel graph a good candidate as a network topology and good architecture for parallel computing. $W_{\lfloor \log n \rfloor, n}$ guarantees the minimum time for broadcasting and gossiping. So, it is a broadcast and gossip graph [1],[7],[8]. Moreover, $W_{\lfloor \log n \rfloor, n}$ is used to construct sparse broadcast graphs of a bigger size by interconnecting several smaller copies or by adding and deleting vertices [13],[10],[9],[2],[4],[11],[16],[12].

Multiple definitions are known for the Knödel graph. We use the following definition from [8], which explicitly presents the Knödel graph as a bipartite graph.

Definition 1. *The Knödel graph on an even number of vertices n and of degree Δ were $2 \leq \Delta \leq \lfloor \log n \rfloor$ is defined as $W_{\Delta,n} = (V, E)$ where*

$$V = \{(i,j) \mid i = 1,2 \; j = 0, ..., n/2 - 1\},$$

$$E = \{((1,j),(2,(j + 2^k - 1) \bmod (n/2))) \mid \\ j = 1, ..., n/2 \; k = 0,1,...,\Delta - 1\}.$$

We say that an edge $((1,j'),(2,j'')) \in E$ is r-dimensional if $j' = (j'' + 2^r - 1) \bmod (n/2)$ where $r = 0,1,...,\Delta - 1$. In this case, $(1,j')$ and $(2,j'')$ are called r-dimensional neighbors.

Fig. 1 illustrates $W_{3,14}$ and its 0, 1 and 2-dimensional edges. We can simplify the illustration of the Knödel graph by minimizing the number of intersecting edges. For this, we repeat few vertices and present the Knödel graph from Fig. 1 as illustrated in Fig. 2.

Despite being a highly symmetric and widely studied graph, the diameter of the Knödel graph $D(W_{\Delta,n})$ is known only for $n = 2^{\Delta}$. In [7], it was proved that $D(W_{\Delta,2^{\Delta}}) = \lceil \frac{\Delta+2}{2} \rceil$. The nontrivial proof of this result is algebraic and the actual diametral path is not presented. The problem of finding the shortest path between any pair of vertices in the Knödel graph $W_{\Delta,2^{\Delta}}$ is studied in [14], where an 2-approximation algorithm with the logarithmic time complexity is presented.

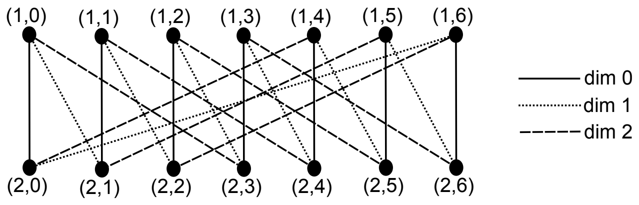


Fig. 1. The $W_{3,14}$ graph and its 0, 1 and 2-dimensional edges

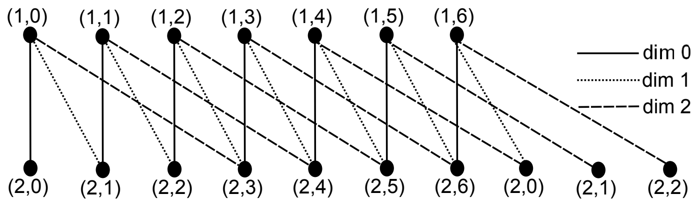


Fig. 2. The $W_{3,14}$ graph

Most properties of the Knödel graph are known only for $W_{\Delta,2^\Delta}$ and $W_{\Delta-1,2^\Delta-2}$. In this paper we present a tight upper bound on the diameter of the Knödel graph $D(W_{\Delta,n})$ for all even n and $2 \leq \Delta \leq \lfloor \log n \rfloor$. We show that the presented bound may differ from the actual diameter by at most 2 for almost all Δ. Our proof is constructive and provides a near optimal diametral path in $W_{\Delta,n}$.

Usually the partition in which a vertex occurs is not relevant, so we just use x to refer to either vertex $(1,x)$ or vertex $(2,x)$. The distance between vertices u and v is denoted by $dist(u,v)$. Using these notations and the vertex transitivity of the Knödel graph, we can state that $D(W_{\Delta,n}) = max\{dist(0,x) | 0 \leq x < n/2\}$. In this paper, we actually give a tight upper bound on $dist(0,x)$ for all $0 \leq x < n/2$.

2 Paths in the Knödel Graph

In this section we construct three different paths between two vertices in the Knödel graph $W_{\Delta,n}$. These paths have certain properties and are used in the next section to prove the upper bound on the diameter of $W_{\Delta,n}$.

Before presenting our formal statements, let us get better understanding of the Knödel graph and the set of vertices which can be reached from vertex 0 using only 0 and $(\Delta-1)$-dimensional edges. Note that we can "move" in two different directions from vertex $0 = (1,0)$ or $0 = (2,0)$ of $W_{\Delta,n}$. Fig. 3 illustrates the discussed paths. We can choose the path $(1,0) \rightarrow (2, 2^{\Delta-1}-1) \rightarrow (1, 2^{\Delta-1}-1) \rightarrow (2, 2(2^{\Delta-1}-1)) \rightarrow ...$ or we can move in the opposite direction following the path $(1,0) \rightarrow (2,0) \rightarrow (1, n/2 - (2^{\Delta-1}-1)) \rightarrow (2, n/2 - (2^{\Delta-1}-1)) \rightarrow ...$. Every second edge in these paths is 0-dimensional. The $(\Delta-1)$-dimensional edges are used to move "forward" by $2^{\Delta-1}-1$ vertices, while the 0-dimensional edges are only to change the partition. These two paths will eventually intersect or overlap somewhere near vertex $\lceil n/4 \rceil$. Excluding vertex 0, we have only $n/2-1$ vertices in each partition. The $(\Delta-1)$-dimensional edges will split $W_{\Delta,n}$ into $\left\lceil \frac{n/2-1}{2^{\Delta-1}-1} \right\rceil$ segments, each having length $2^{\Delta-1}-1$, except the one containing vertex $\lceil n/4 \rceil$. We can perform only $\left\lfloor \frac{1}{2} \left\lceil \frac{n/2-1}{2^{\Delta-1}-1} \right\rceil \right\rfloor = \left\lfloor \frac{1}{2} \left\lceil \frac{n-2}{2^{\Delta}-2} \right\rceil \right\rfloor$ $(\Delta-1)$-dimensional passes in each of these two paths before they intersect. Therefore, we will never use more than $\left\lfloor \frac{1}{2} \left\lceil \frac{n-2}{2^{\Delta}-2} \right\rceil \right\rfloor$ $(\Delta-1)$-dimensional passes to reach a vertex in $W_{\Delta,n}$.

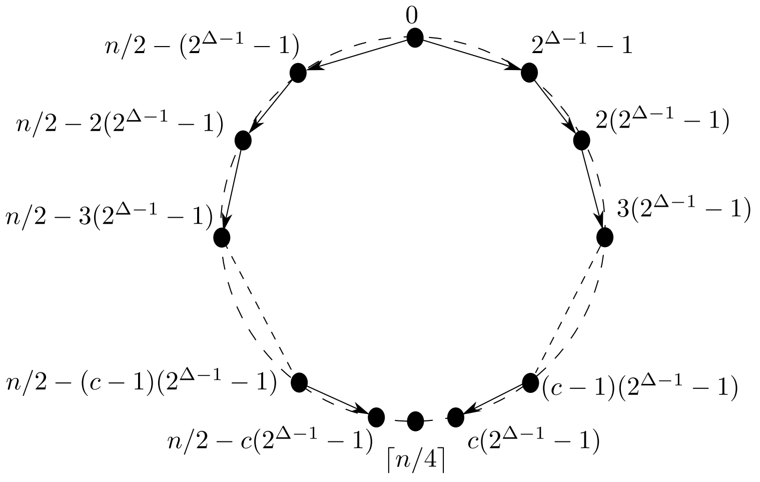


Fig. 3. Schematic illustration of the paths. $c = \left\lfloor \frac{1}{2} \left\lceil \frac{n-2}{2^{\Delta}-2} \right\rceil \right\rfloor$

Our first lemma constructs a path between vertex 0 and some vertex y which is relatively close to our destination vertex x. Vertex y will have a special form making such construction straightforward. Recall that x refers to $(1, x)$ or $(2, x)$, and y refers to $(1, y)$ or $(2, y)$.

Lemma 2. *For any vertex x of $W_{\Delta,n}$, by using at most $2c + 1$ edges where $c = \left\lfloor \frac{1}{2} \left\lceil \frac{n-2}{2^{\Delta}-2} \right\rceil \right\rfloor$, we can construct a path from vertex 0 to reach some vertex y such that $|x - y| \leq 2^{\Delta-1} - 1$.*

Proof. Our goal is to reach some vertex y of form $y = c(2^{\Delta-1} - 1)$ or $y = n/2 - c(2^{\Delta-1} - 1)$ such that $|x - y| \leq 2^{\Delta-1} - 1$. We use only 0 and $(\Delta - 1)$-dimensional edges and one of two paths described above and illustrated in Fig. 3. We consider two cases. In the first case we cover the values of x that can be reached by moving in "clockwise" direction from vertex 0. For the remaining values of x, we use the path from Fig. 3 moving to the opposite direction.

Case 1: $x < (c + 1)(2^{\Delta-1} - 1)$. By alternating between 0 and $(\Delta - 1)$-dimensional edges, we can reach a vertex y of form $y = c'(2^{\Delta-1} - 1)$ and closest to x from vertex $0 = (2, 0)$. We will need at most $2c' + 1$ edges for that. The path to reach $y = (1, y)$ will be $(2, 0) \to (1, 0) \to (2, 2^{\Delta-1} - 1) \to (1, 2^{\Delta-1} - 1) \to (2, 2(2^{\Delta-1} - 1)) \to ... \to (2, c'(2^{\Delta-1} - 1)) \to (1, c'(2^{\Delta-1} - 1)) = y$. It is clear that $c' \leq c = \left\lfloor \frac{1}{2} \left\lceil \frac{n-2}{2^{\Delta}-2} \right\rceil \right\rfloor$, hence the bound on the length of constructed path follows. From the form of y follows that $|x - y| \leq 2^{\Delta-1} - 1$. Fig. 4 shows the described path from $(2, 0)$ to $y = 6 = (1, 6)$.

Case 2: $x > n/2 - c(2^{\Delta-1} - 1)$. This case is similar to case 1 except in order to construct shorter path to y of form $y = n/2 - c'(2^{\Delta-1} - 1)$, we are moving from vertex $0 = (1, 0)$ in anticlockwise direction. The path for $y = (2, y)$ will

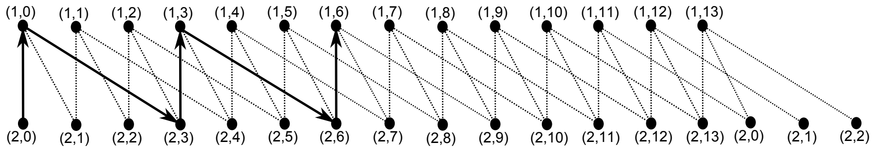


Fig. 4. A path between $(2,0)$ and $(1,6)$ vertices in $W_{3,28}$ graph

be $(1,0) \rightarrow (2,0) \rightarrow (1, n/2 - (2^{\Delta-1}-1) \rightarrow (2, n/2 - (2^{\Delta-1}-1)) \rightarrow ... \rightarrow (1, n/2 - (c'-1)(2^{\Delta-1}-1)) \rightarrow (2, n/2 - (c'-1)(2^{\Delta-1}-1)) = y$ and will have length at most $2c+1$. Obviously we will have $|x-y| \leq 2^{\Delta-1}-1$ as well. □

The following lemma constructs a path between two vertices of $W_{\Delta,n}$ that are relatively close to each other. More precisely, when the difference of their labels is upper bounded by $2^{\Delta-1}-1$. We construct a path between two vertices x_1 and x_2 which is not necessarily a shortest path between them. To reach the given vertex with label $x_2 > x_1$ from vertex labeled x_1, we first use a large dimensional edge to "jump over" vertex x_2 and reach some vertex $y \geq x_2$, such that $y - x_2$ is the smallest. After that, we start moving from y in backward direction till we reach x_2 from right. This backward steps are performed in a greedy way. At each step, we are using the largest dimensional edge to reach some new vertex y' such that $y' - x_2$ is minimal and y' is on the right side of x_2 i.e. $y' \geq x_2$.

Lemma 3 (Existence of a special path). *For any two vertices of $W_{\Delta,n}$ labeled x_1 and x_2, if $|x_2 - x_1| \leq 2^{\Delta-1}-1$, then there exists a special path between x_1 and x_2 of length at most $2\Delta-3$. This path contains one "direct" d-dimensional edge where $d \leq \Delta-1$, some 0-dimensional edges and some edges having dimensions between 1 and $d-1$ pointing in "backward" direction. The number of these backward edges is at most $\Delta-2$.*

Proof. Without loss of generality, we assume that $x_1 = 0$ and $x_2 > x_1$. In order to construct the described path, we use an edge to get from vertex 0 to some vertex y closest to x_2 such that $y > x_2$ and y is directly connected to 0. This will be our "direct" d-dimensional edge. After reaching vertex y, we start to move in "backward" direction towards x_2. Once started moving in backward direction, the distance from y to x_2 which is upper bounded by $2^{\Delta-2}$, will be cut at least by half with each backward edge. Therefore we need at most $\Delta-2$ backward edges. Combined with the 0-dimensional edges between these backward edges, this will give a path of length $2(\Delta-2)$. By adding the initial edge, we get the $2\Delta-3$ upper bound on the length of the constructed path.

Fig. 5 shows the described path between vertices $x_1 = (1,0)$ and $x_2 = (2,5)$. In the illustrated example $y = 7$, $d = 4$, the "direct" edge is $((1,0),(2,7))$ and the "backward" edges are $((2,7),(1,6))$ and $((2,6),(1,5))$.

The reason we chose this particular path between x_1 to x_2 is that the backward passes can be performed in the path constructed by Lemma 2. This will be crucial in the proof of the main theorem. □

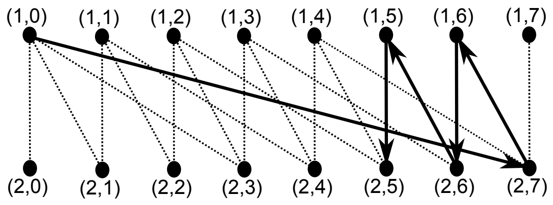


Fig. 5. A path between $(1, 0)$ and $(2, 5)$ vertices in a section of the Knödel graph of degree 5

Our last lemma deals with the problem of finding the shortest path in a particular section of the Knödel graph.

Lemma 4 (Shortest path approximation). *For any two vertices of $W_{\Delta,n}$ labeled x_1 and x_2, if $|x_2 - x_1| \leq 2^d - 1$ for some $d \leq \Delta - 1$, then there exist a path between x_1 and x_2 of length at most $3\lceil d/4 \rceil + 4$.*

Proof. Without loss of generality, we assume that $x_1 = 0$ and $x_2 > x_1$. Our goal is to construct a short path from vertex 0 to vertex $x_2 = x \leq 2^d - 1$. The proof is based on a recursive construction of a path between vertices 0 and x having length at most $3\lceil d/4 \rceil + 4$. The recursion will be on d.

The base case is when $d \leq 3$. This case is illustrated in Fig. 6, from which we observe that we can reach any vertex x where $0 \leq x \leq 2^d - 1 = 7$ with a path of length at most 4.

For $d > 3$, using at most three edges, we can cut the distance between 0 and x by a factor of 16. Fig. 7 presents a schematic illustration of this. We divide the initial interval of length $2^d - 1$ into eight smaller intervals $A_1, A_2, ..., A_8$, each having length at most $\lceil (2^d - 1)/8 \rceil$, where $A_i = [(i-1)m, im)$, $i = 1, ..., 8$ and $m = 2^{d-3}$.

It is not difficult to see that all these intervals, except A_6, have both their end vertices reachable from 0 by using at most three edges. For A_6, using at most 3 edges we can reach its middle vertex $11m/2 - 1$ and the end vertex $6m$. The paths, which use at most 3 edges, are illustrated in Fig. 7. This means that when $x \in A_i$ for all $1 \leq i \leq 8$, using at most three edges, we will be within distance $m/2$ from x. After relabeling the vertices, we will get the same problem of finding a path between vertices 0 and x, but the new x will be at least 16 times smaller.

It will take at most $\lceil \log_{16}(2^d - 1) \rceil$ recursive steps to reach the base case, and we will use at most three edges in each step. By combining this with at most 4 edges used for the base case, we will get that $dist(0, x) \leq 3\lceil \log_{16}(2^d - 1) \rceil + 4 \leq 3\lceil d/4 \rceil + 4$. □

We note that each recursive step in Lemma 4 involves only constant number of operations. Therefore the described path can be constructed by an algorithm of complexity $O(\log n)$.

Lemma 4 can be used to construct a short path between any two vertices of $W_{\Delta,n}$ for the case when $\Delta = \lfloor \log n \rfloor$. The length of the constructed path will

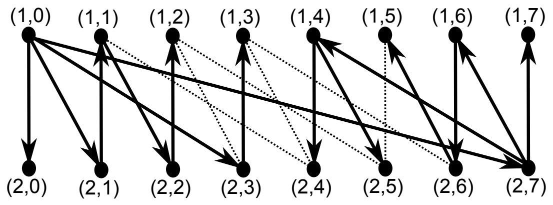


Fig. 6. Paths from vertex 0 to all other vertices $x \leq 7$ in a section of the Knödel graph

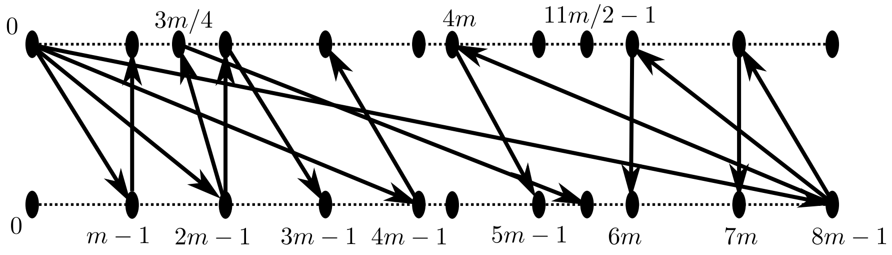


Fig. 7. Illustration of the recursive step. $m = 2^{d-3}$

be at most $3\lceil(\Delta - 1)/4\rceil + 4$. It follows that $D(W_{\Delta,n}) \leq 3\lceil(\Delta - 1)/4\rceil + 4$ for $\Delta = \lfloor \log n \rfloor$.

3 Upper Bound on Diameter

In this section, using the lemmas from Section 2, we construct a path between vertices 0 and x for any vertex x in $W_{\Delta,n}$. The maximum length of such a path will be an upper bound on the diameter of $W_{\Delta,n}$.

Our first upper bound on $D(W_{\Delta,n})$ will trivially follow from Lemma 2 and Lemma 4.

Theorem 5 (Trivial). $D(W_{\Delta,n}) \leq 2\left\lfloor \frac{1}{2}\left\lceil \frac{n-2}{2^{\Delta}-2}\right\rceil\right\rfloor + 3\lceil(\Delta - 1)/4\rceil + 5.$

Proof. According to Lemma 2, for any vertex x in $W_{\Delta,n}$, we need at most $2\left\lfloor \frac{1}{2}\left\lceil \frac{n-2}{2^{\Delta}-2}\right\rceil\right\rfloor + 1$ edges to reach from vertex 0 to a vertex y of form $y = c(2^{\Delta-1} - 1)$ or $y = n/2 - c(2^{\Delta-1} - 1)$ within distance $2^{\Delta-1} - 1$ from x i.e. $|x - y| \leq 2^{\Delta-1} - 1$. Now we can apply Lemma 4 and claim that $dist(x, y) \leq 3\lceil d/4\rceil + 4$ where $d \leq \Delta - 1$. Thus, we have that $dist(0, x) \leq dist(0, y) + dist(y, x) \leq 2\left\lfloor \frac{1}{2}\left\lceil \frac{n-2}{2^{\Delta}-2}\right\rceil\right\rfloor + 3\lceil(\Delta - 1)/4\rceil + 5$. □

Theorem 5 combines the paths described in Lemmas 2 and 4 in the most trivial way. With the slight modification of the path described in Lemma 2 and combining it with paths from Lemmas 3 and 4 we can significantly improve the presented upper bound on $D(W_{\Delta,n})$.

Theorem 6 (Main). *Let* $a = \left\lfloor \frac{1}{2} \left\lceil \frac{n-2}{2^{\Delta}-2} \right\rceil \right\rfloor$ *and* $b = \Delta - 2$ *(*$\Delta \geq 3$*). If* $a \geq b$ *then* $D(W_{\Delta,n}) \leq 2a + 3 = 2 \left\lfloor \frac{1}{2} \left\lceil \frac{n-2}{2^{\Delta}-2} \right\rceil \right\rfloor + 3$*, otherwise* $D(W_{\Delta,n}) \leq 2a + 3 \lceil (\Delta - 2 - a)/4 \rceil + 7 \leq \frac{3}{4}\Delta + \frac{5}{4}a + \frac{17}{2}$.

Proof. Case 1: $a \geq b$. From Lemma 2, we recall that a is the maximum number of $(\Delta-1)$-dimensional edges necessary to reach a vertex of form $y = c(2^{\Delta-1} - 1)$ or $y = n/2 - c(2^{\Delta-1} - 1)$ closest to our destination vertex x. Recall that $b = \Delta - 2$ is the maximum number of "backward" edges used in the path from Lemma 3. We observe that when $a \geq b$, then all the "backward" passes can be performed by modifying the path described in Lemma 2 used to reach vertex y. We just need to replace some of the 0-dimensional edges from Lemma 2 used only for switching the graph partition with the corresponding "backward" passes from Lemma 3. As a result of this modification, instead of reaching y, with $2a + 1$ edges we will reach some vertex y' precisely at distance $2^{\Delta} - 1$ from x. By using one more $(\Delta-1)$-dimensional and one more 0-dimensional edge, we can perform the final pass and reach x with a path of length at most $2a + 3$.

Case 2: $a < b$. In this case we will be able to perform only some of the reverse passes from Lemma 3 by modifying the path from Lemma 2. More precisely, out of $b = \Delta - 2$ reverse passes, we will be able to perform only a of them in the modified path. We note that each reverse pass in Lemma 3 cuts the distance to x by half. This means that performing $b - a$ reverse passes in the path constructed by Lemma 2 of length $2a+3$, we will be within distance $2^{\Delta-2-a}$ from x compared to $2^{\Delta-2}$ without performing these reverse passes. Now we can use Lemma 4 with $d = \Delta - 2 - a$ and claim that we will be able to reach x by using at most $3 \lceil (\Delta - 2 - a)/4 \rceil + 4$ additional edges. Thus, $D(W_{\Delta,n}) \leq (2a + 3) + (3 \lceil (\Delta - 2 - a)/4 \rceil + 4) \leq \frac{3}{4}\Delta + \frac{5}{4}a + \frac{17}{2}$. □

4 Tightness of the Upper Bound

In this section we analyze the tightens of the upper bound on the diameter of the Knödel graph from Theorem 6. To do that we will first present a lower bound on the diameter of the Knödel graph.

Theorem 7 (Lower bound). $D(W_{\Delta,n}) \geq 2 \left\lfloor \frac{1}{2} \left\lceil \frac{n-2}{2^{\Delta}-2} \right\rceil \right\rfloor + 1$.

Proof. First, note that in order to reach vertex $x = (1, c(2^{\Delta-1} - 1))$ where $c = \left\lfloor \frac{1}{2} \left\lceil \frac{n-2}{2^{\Delta}-2} \right\rceil \right\rfloor$ from vertex $(2, 0)$, we cannot construct a path shorter than the one described in Lemma 2 and illustrated in Fig. 3. This path contains exactly $c + 1$ 0-dimensional edges used for changing the graph partition and c $(\Delta - 1)$-dimensional edges used for moving towards x in the fastest possible way. Thus, the lower bound $D(W_{\Delta,n}) \geq 2c + 1 = 2 \left\lfloor \frac{1}{2} \left\lceil \frac{n-2}{2^{\Delta}-2} \right\rceil \right\rfloor + 1$ follows. □

The following theorem shows that the presented upper bound is tight, in particular it is within additive factor 2, for almost all possible values of Δ.

Theorem 8 (Tightness). *For any* $0 < \alpha < 1$*, there exists some* $N(\alpha)$ *such that for all* $n \geq N(\alpha)$ *and* $\Delta < \alpha \lfloor \log n \rfloor$*, the* $D(W_{\Delta,n}) \leq 2a + 3$ *upper bound from Theorem 6 (*$a = \left\lfloor \frac{1}{2} \left\lceil \frac{n-2}{2^{\Delta}-2} \right\rceil \right\rfloor$*) differs from actual diameter by at most 2, i.e.* $2\left\lfloor \frac{1}{2} \left\lceil \frac{n-2}{2^{\Delta}-2} \right\rceil \right\rfloor + 1 \leq D(W_{\Delta,n}) \leq 2\left\lfloor \frac{1}{2} \left\lceil \frac{n-2}{2^{\Delta}-2} \right\rceil \right\rfloor + 3.$

Proof. From Theorem 7 it follows that the upper bound from Theorem 6 for the case when $a \geq b$ may differ from actual diameter by at most 2. Now, we find a sufficient condition for $a \geq b$ to be true. By observing that $a = \left\lfloor \frac{1}{2} \left\lceil \frac{n-2}{2^{\Delta}-2} \right\rceil \right\rfloor \geq \frac{n/2}{2^{\Delta}} - 1$ and $b = \Delta - 2 \leq \Delta - 1$ we get that if $\frac{n/2}{2^{\Delta}} - 1 \geq \Delta - 1$ then $a \geq b$ equality is true. After further simplification, we get the $2\Delta 2^{\Delta} \leq n$ sufficient condition for $a \geq b$ to be true.

It follows that for given n and Δ, where $2 \leq \Delta \leq \lfloor \log n \rfloor$ such that $\Delta 2^{\Delta+1} \leq n$, we have $2a + 1 \leq D(W_{\Delta,n}) \leq 2a + 3$. Finally, we observe that for any $0 < \alpha < 1$ and $\Delta < \alpha \lfloor \log n \rfloor$ the $\Delta 2^{\Delta+1} \leq n$ inequality is always true for sufficiently large n. □

Note that Theorem 6, in almost all cases, actually gives an approximation algorithm to find the diameter of $W_{\Delta,n}$ with an additive factor 2.

5 Summary

In this paper we obtained tight lower and upper bounds on the diameter of the Knödel graph $W_{\Delta,n}$ for all even n and $2 \leq \Delta \leq \lfloor \log n \rfloor$. We showed that the presented bound differs from actual diameter by at most 2 for almost all Δ. Our proofs are constructive and provide a near optimal diametral path in $W_{\Delta,n}$.

Recall that the only known results, regarding the diameter of the Knödel graph, were the exact value $D(W_{\Delta,2^{\Delta}}) = \left\lceil \frac{\Delta+2}{2} \right\rceil$ [7] and an 2-approximation algorithm with logarithmic time complexity for finding shortest path between any pair of vertices in $W_{\Delta,2^{\Delta}}$ [14]. Lemma 4 provides $D(W_{\Delta,2^{\Delta}}) \leq 3\lceil(\Delta-1)/4\rceil + 4$. Comparing this with the exact expression above, we see that Lemma 4 provides an 3/2-approximation algorithm for the problem of finding a diametral path. This is much better than the 2-approximation algorithm presented in [14]. However, we note that [14] addresses more general problem of finding a shortest path in the Knödel graph and the 2-approximation ratio is for the shortest path between any two vertices, while our result is only for the diametral path.

Our future research will be focused on routing and broadcasting problems in the Knödel graph $W_{\Delta,n}$ for all $2 \leq \Delta \leq \lfloor \log n \rfloor$.

References

1. Bermond, J.C., Harutyunyan, H.A., Liestman, A.L., Perennes, S.: A note on the dimensionality of modified knödel graphs. Int. J. Found. Comput. Sci. 8(2), 109–116 (1997)

2. Bermond, J.C., Fraigniaud, P., Peters, J.G.: Antepenultimate broadcasting. Networks 26(3), 125–137 (1995)
3. Dinneen, M., Fellows, M., Faber, V.: Algebraic constructions of efficient broadcast networks. Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, 152–158 (1991)
4. Dinneen, M.J., Ventura, J.A., Wilson, M.C., Zakeri, G.: Construction of time-relaxed minimal broadcast networks. Parallel Processing Letters 9(1), 53–68 (1999)
5. Farley, A.M., Hedetniemi, S., Mitchell, S., Proskurowski, A.: Minimum broadcast graphs. Discrete Mathematics 25, 189–193 (1979)
6. Fertin, G., Raspaud, A.: A survey on knödel graphs. Discrete Applied Mathematics 137(2), 173–195 (2004)
7. Fertin, G., Raspaud, A., Schröder, H., Sýkora, O., Vrťo, I.: Diameter of the knödel graph. In: Brandes, U., Wagner, D. (eds.) WG 2000. LNCS, vol. 1928, pp. 149–160. Springer, Heidelberg (2000)
8. Fraigniaud, P., Peters, J.G.: Minimum linear gossip graphs and maximal linear (δ, k)-gossip graphs. Networks 38(3), 150–162 (2001)
9. Harutyunyan, H.A.: Minimum multiple message broadcast graphs. Networks 47(4), 218–224 (2006)
10. Harutyunyan, H.A.: An efficient vertex addition method for broadcast networks. Internet Mathematics 5(3), 211–225 (2008)
11. Harutyunyan, H.A., Liestman, A.L.: More broadcast graphs. Discrete Applied Mathematics 98(1-2), 81–102 (1999)
12. Harutyunyan, H.A., Liestman, A.L.: On the monotonicity of the broadcast function. Discrete Mathematics 262(1-3), 149–157 (2003)
13. Harutyunyan, H.A., Liestman, A.L.: Upper bounds on the broadcast function using minimum dominating sets. Discrete Mathematics 312(20), 2992–2996 (2012)
14. Harutyunyan, H.A., Morosan, C.D.: On the minimum path problem in knödel graphs. Networks 50(1), 86–91 (2007)
15. Heydemann, M.C., Marlin, N., Pérennes, S.: Complete rotations in cayley graphs. European Journal of Combinatorics 22(2), 179–196 (2001)
16. Khachatrian, L.H., Harutounian, O.S.: Construction of new classes of minimal broadcast networks. In: Conference on Coding Theory, Dilijan, Armenia, pp. 69–77 (1990)
17. Knödel, W.: New gossips and telephones. Discrete Mathematics 13(1), 95 (1975)
18. Park, J.H., Chwa, K.Y.: Recursive circulant: a new topology for multicomputer networks (extended abstract). In: International Symposium on Parallel Architectures, Algorithms and Networks, ISPAN 1994, pp. 73–80 (1994)

Structural Properties of Subdivided-Line Graphs[⋆]

Toru Hasunuma

Institute of Socio-Arts and Sciences,
The University of Tokushima,
1–1 Minamijosanjima, Tokushima 770–8502 Japan
hasunuma@ias.tokushima-u.ac.jp

Abstract. Motivated by self-similar structures of Sierpiński graphs, we newly introduce the subdivided-line graph operation Γ and define the n-iterated subdivided-line graph $\Gamma^n(G)$ of a graph G. We then study structural properties of subdivided-line graphs such as edge-disjoint Hamilton cycles, hub sets, connected dominating sets, and completely independent spanning trees which can be applied to problems on interconnection networks. From our results, the maximum number of edge-disjoint Hamilton cycles, the minimum cardinality of a hub set, the minimum cardinality of a connected dominating set, and the maximum number of completely independent spanning trees in Sierpiński graphs are obtained as corollaries. In particular, our results for edge-disjoint Hamilton cycles and hub sets on iterated subdivided-line graphs are generalizations of the previously known results on Sierpiński graphs, while our proofs are simpler than those for Sierpiński graphs.

1 Introduction

Throughout the paper, a graph may have self-loops but not multiple edges, unless otherwise stated. Let G be a graph. The vertex set and the edge set of G are denoted by $V(G)$ and $E(G)$, respectively. The notion of Sierpiński graphs was introduced by Klavžar and Milutinović [17]. The *Sierpiński graph* $S(n, k)$ is the graph with the vertex set consisting of all n-tuples of k numbers $1, 2, \ldots, k$, i.e, $V(S(n, k)) = \{(v_1, v_2, \ldots, v_n) \mid 1 \leq v_i \leq k, 1 \leq i \leq n\}$ and in which two vertices $(u_1, u_2, \ldots, u_n)$ and $(v_1, v_2, \ldots, v_n)$ are adjacent if and only if there exists an integer j, where $1 \leq j \leq n$, such that $u_i = v_i$ for $1 \leq i < j$, $u_j \neq v_j$, and $u_i = v_j$, $v_i = u_j$ for $j < i \leq n$. Fig. 1 shows the Sierpiński graphs $S(2, 3)$ and $S(3, 3)$. A vertex of the form $(i, i, \ldots, i)$ of $S(n, k)$ is called an *extreme vertex*. Every vertex except for the k extreme vertices in $S(n, k)$ has degree k, while the degree of each extreme vertex is $k - 1$. The Sierpiński graph $S(n, k)$ has a recursive structure, i.e., $S(n, k)$ can be constructed from k copies of $S(n-1, k)$ by joining extreme vertices in a fashion of the complete graph with k vertices. Since $S(n, k)$ is not regular, two regular variations called the *extended Sierpiński graphs* $S^+(n, k)$ and $S^{++}(n, k)$ were also introduced by Klavžar and Mohar [18]. An extended Sierpiński graph $S^+(n, k)$ is obtained from $S(n, k)$ by adding a new vertex and joining it to all the extreme vertices of $S(n, k)$. Another extended Sierpiński graph $S^{++}(n, k)$ is obtained from $k+1$ copies of $S(n-1, k)$ by joining the extreme vertices in a fashion of the complete graph with $k + 1$ vertices

[⋆] This work was supported by JSPS KAKENHI 25330015.

© Springer-Verlag Berlin Heidelberg 2013

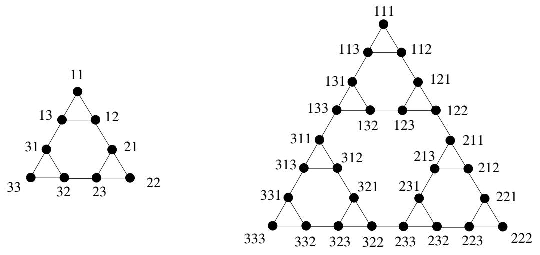


Fig. 1. The Sierpiński graphs $S(2,3)$ and $S(3,3)$

(see Fig. 2). Because of their interesting self-similar structures with relations to the problem of the Tower of Hanoi, various properties on the Sierpiński-like graphs have been investigated: hamiltonicity [25], hub numbers [19], colorings [13,15], covering codes [2,9], average eccentricity [14], and crossing number [18]. On the other hand, the *WK-recursive networks* have independently been proposed by Vecchia and Sanges [23] as interconnection networks, and their topological properties have been investigated in [4,7]. The WK-recursive network and the Sierpiński graph have very similar structures, and the difference between them is the existence of "open edges" incident to each extreme vertex in the WK-recursive network. By deleting such open edges, they become isomorphic. From the point of interconnection networks, WK-recursive networks have a remarkable nice property for extendability.

Motivated by self-similar structures of the Sierpiński graphs (WK-recursive networks), we newly introduce the subdivided-line graph operation Γ and define the n-iterated subdivided-line graphs $\Gamma^n(G)$ of a graph G (the definitions of these notions will be described in Section 3). We then study structural properties of subdivided-line graphs such as edge-disjoint Hamilton cycles, hub sets, connected dominating sets, and completely independent spanning trees (the definition of each term will be also described in Section 4). These structural properties can be applied to problems on interconnection networks. Especially, edge-disjoint Hamilton cycles and completely independent spanning trees have applications to fault-tolerant communication problems. The Sierpiński graph $S^\circ(n,k)$ with a self-loop at each extreme vertex and the extended Sierpiński graph $S^{++}(n,k)$ can be defined as the n-iterated subdivided-line graphs of the graph with one vertex and k self-loops and the complete graph with $k+1$ vertices, respectively. Thus, the class of iterated subdivided-line graphs generalizes the class of these Sierpiński-like graphs. By applying our results on subdivided-line graphs with the results on decompositions of complete graphs to the Sierpiński-like graphs, the maximum number of edge-disjoint Hamilton cycles [25], the minimum cardinality of a hub set [19], the minimum cardinality of a connected dominating set, and the maximum number of completely independent spanning trees in the Sierpiński-like graphs are obtained as corollaries. In particular, our results for edge-disjoint Hamilton cycles and hub sets on iterated subdivided-line graphs are generalizations of the previously known results on Sierpiński-like graphs [25,19], while our proofs are simpler than those for Sierpiński-like graphs.

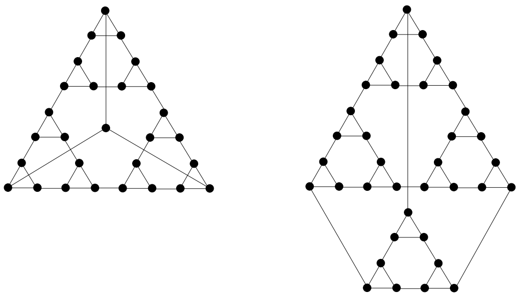


Fig. 2. The extended Sierpiński graphs $S^+(3,3)$ and $S^{++}(3,3)$

This paper is organized as follows. Section 2 presents terminology and notations used in the paper. The subdivided-line graph operation and the n-iterated subdivided-line graph of a graph are introduced in Section 3. Section 3 also presents fundamental properties of iterated subdivided-line graphs such as diameter and connectivity. Structural properties of subdivided-line graphs and their applications to the Sierpiński-like graphs are given in Section 4. Section 5 concludes the paper with several remarks.

2 Preliminaries

Let $G = (V, E)$ be a graph. The number of self-loops in G is denoted by $\ell(G)$. For $v \in V(G)$, let $N_G(v)$ and $E_G(v)$ denote the set of vertices adjacent to v and the set of edges incident to v in G, respectively. The degree of v in G is denoted by $\deg_G(v)$, i.e., $\deg_G(v) = |E_G(v)|$. Note that in this paper, if v has a self-loop, then we count it one in $\deg_G(v)$ instead of two. Let $S \subseteq V(G)$ and $T \subseteq E(G)$. Then, $G - S$ (resp., $G - T$) is the graph obtained from G by deleting every element of S (resp., T). The subgraph of G induced by S is denoted by $\langle S \rangle_G$. Let $\delta(G)$ and $\Delta(G)$ denote the minimum degree and the maximum degree of a vertex of G, respectively. The complete graph with k vertices is denoted by K_k. For $u, v \in V(G)$, the *distance* $\mathrm{dist}_G(u, v)$ of u and v in G is the length of a shortest path between u and v in G. If there is no path between u and v in G, $\mathrm{dist}_G(u, v)$ is defined to be ∞. The *diameter* $\mathrm{diam}(G)$ of G is the maximum distance of any pair of vertices in G, i.e., $\mathrm{diam}(G) = \max_{u,v \in V(G)} \mathrm{dist}_G(u, v)$. The *vertex-connectivity* $\kappa(G)$ is the minimum cardinality of a proper subset S of $V(G)$ such that $G - S$ is disconnected or $G - S \cong K_1$. Also, the *edge-connectivity* $\bar{\kappa}(G)$ of G is the minimum cardinality of a subset T of $E(G)$ such that $G - T$ is disconnected.

The *line graph* $L(G)$ of G is the graph whose vertex set is $E(G)$ and in which two distinct vertices $\{u, v\}$ and $\{x, y\}$ are adjacent if and only if they are adjacent in G, i.e., $\{u, v\} \cap \{x, y\} \neq \emptyset$. Besides, a vertex $\{w, w\}$ corresponding to a self-loop in G also has a self-loop in $L(G)$. Let $e = \{x, y\} \in E(G)$. Then, let G_e be the graph with $V(G_e) = V(G) \cup \{v_e\}$, where $v_e \notin V(G)$, and $E(G_e) = (E(G) - \{\{x, y\}\}) \cup \{\{x, v_e\}, \{v_e, y\}\}$. We say that G_e is obtained from G by *elementary subdividing* the edge e. The *barycentric subdivision* $B(G)$ of G is the graph obtained from G by elementary subdividing every edge of G except for self-loops.

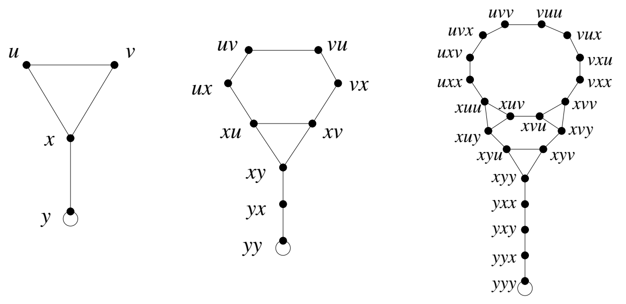


Fig. 3. G, $\Gamma(G)$, and $\Gamma^2(G)$, where the vertex-labeling of $\Gamma^2(G)$ follows those for iterated subdivided-line graphs

3 The Subdivided-Line Graph Operation and Iterated Subdivided-Line Graphs

Combining the notions of the line graph and the barycentric subdivision of a graph, we define the subdivided-line graph operation.

Definition 1. *Let G be a graph. The subdivided-line graph $\Gamma(G)$ of G is defined to be the line graph of the barycentric subdivision of G. i.e., $\Gamma(G) = L(B(G))$. We call Γ the subdivided-line graph operation.*

Each vertex in $\Gamma(G)$ can be denoted by the ordered pair of vertices. Namely, for each non-loop edge $\{u, v\}$ of G, there exist two corresponding vertices uv, vu in $\Gamma(G)$. For a self-loop $\{w, w\}$ of G, the only corresponding vertex in $\Gamma(G)$ is ww. Note that a vertex of the form ww also has a self-loop in $\Gamma(G)$. Two distinct vertices uv, xy in $\Gamma(G)$ are adjacent if and only if either $u = x$, or $u = y$ and $v = x$ (see the middle graph in Fig. 3). The edge set of $\Gamma(G)$ is naturally divided into two categories. An edge joining vertices of the forms uv and vu is corresponding to the original edge $\{u, v\}$ in G, while an edge joining vertices uv and ux, where $v \neq x$, is newly generated in $\Gamma(G)$. Then, we call edges of the former type *original edges* and edges of the latter type *generated edges*. Note that a self-loop at a vertex ww is an original edge. The subgraph of $\Gamma(G)$ induced by the set of generated edges is the disjoint union of complete graphs, i.e., $\cup_{v \in V(G)} K_{\deg_G(v)}$. For each $v \in V(G)$, we denote by $K(v)$ the complete graph induced by a set $\{\{vw, vw'\} \mid w, w' \in N_G(v), w \neq w'\}$ of generated edges. Original edges of the form $\{vw, wv\}$, where $w \in N_G(v)$, are injectively incident to vertices of $K(v)$. Thus, the degree of every vertex of $K(v)$ in $\Gamma(G)$ is equal to $\deg_G(v)$.

Using the subdivided-line graph operation, we can naturally define the n-iterated subdivided-line graph of a graph.

Definition 2. *The n-iterated subdivided-line graph $\Gamma^n(G)$ of G is the graph obtained from G by iteratively applying the subdivided-line graph operation n times*

For $n \geq 1$, each vertex of $\Gamma^n(G)$ can be expressed by a sequence $v_0v_1 \cdots v_n$ of vertices of G, where $v_1 \cdots v_n$ is any sequence on $N_G(v_0)$. Intuitively, we can say that each vertex

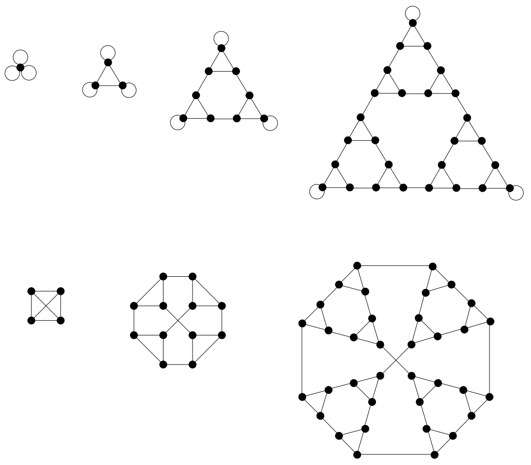


Fig. 4. $\Gamma^i(K_1^3)$ for $i \le 3$ and $\Gamma^i(K_4)$ for $i \le 2$

v in G is expanded to $\deg_G(v)^n$ vertices in $\Gamma^n(G)$. Thus, $\Gamma^n(G)$ has $\sum_{v\in V(G)} \deg_G(v)^n$ vertices. Two vertices $u_0u_1\cdots u_n$ and $v_0v_1\cdots v_n$ are adjacent if and only if there exists an integer $0 \le h \le n$ such that $u_0\cdots u_{h-1} = v_0\cdots v_{h-1}$, $u_h \ne v_h$, and $u_j = v_h, v_j = u_h$ for $h < j \le n$ (see the right graph in Fig. 3). Such an h for adjacent two vertices $u_0u_1\cdots u_n$ and $v_0v_1\cdots v_n$ is the *level* of the edge joining the vertices. Note that any edge with level n is a generated edge, while any edge with level $h < n$ is an original edge corresponding to a generated edge in $\Gamma^h(G)$. For any vertex $v_0v_1\cdots v_n$ in $\Gamma^n(G)$, the degree is equal to $\deg_G(v_0)$. Hence, $\Gamma^n(G)$ has $\frac{1}{2}(|\ell(G)| + \sum_{v\in V(G)} \deg_G(v)^{n+1})$ edges. Note that $\ell(G) = \ell(\Gamma^n(G))$. Besides, a vertex $v_0v_1\cdots v_n$ is incident to $\deg_G(v_0) - 1$ edges with level n and one edge with level $h < n$.

Let $S^\circ(n,k)$ be the graph obtained from the Sierpiński graph $S(n,k)$ by adding a self-loop to each extreme vertex. Then, we have the following lemma, where K_1^k is the graph with one vertex and k self-loops, and K_k° is the complete graph with k vertices and a self-loop at each vertex. Fig. 4 shows examples for $S^\circ(n,3)$ and $S^{++}(n,3)$.

Lemma 1. *Let* $n \ge 1$.

- $\Gamma^n(K_1^k) = \Gamma^{n-1}(K_k^\circ) \cong S^\circ(n,k)$.
- $\Gamma^{n-1}(K_{k+1}) \cong S^{++}(n,k)$.

By definition, we immediately have the following. Namely, the minimum degree and the maximum degree are invariant with respect to the subdivided-line graph operation.

Proposition 1. $\delta(G) = \delta(\Gamma^n(G)) \le \Delta(\Gamma^n(G)) = \Delta(G)$ *for all* $n \ge 0$.

Let $P = (u_0, u_1, \ldots, u_t)$ be a path of length t between vertices u_0 and u_t in G. Consider a vertex $u_0v_i \in V(K(u_0))$ and a vertex $u_tv_j \in V(K(u_t))$ in $\Gamma(G)$. Then, there is a path between u_0v_i and u_tv_j of length at most $2t + 1$. From this observation, we can see that $\mathrm{diam}(\Gamma(G)) \le 2(\mathrm{diam}(G)) + 1$. Therefore, the next proposition holds.

Proposition 2. $\mathrm{diam}(\Gamma^n(G)) \leq 2^n(\mathrm{diam}(G) + 1) - 1$ *for all* $n \geq 0$.

On vertex-connectivity and edge-connectivity, the following proposition holds.

Proposition 3. $\kappa(\Gamma^n(G)) = \bar{\kappa}(\Gamma^n(G)) = \bar{\kappa}(G)$ *for all* $n \geq 1$.

Proof: It clearly holds that $\kappa(\Gamma(G)) \leq \bar{\kappa}(\Gamma(G))$. If F is a cut of G, i.e., $F \subseteq E(G)$ such that $G - F$ is disconnected, then the set of original edges of $\Gamma(G)$ corresponding to F is also a cut of $\Gamma(G)$. Thus, it holds that $\bar{\kappa}(\Gamma(G)) \leq \bar{\kappa}(G)$. Therefore, it is sufficient to show that $\kappa(\Gamma(G)) \geq \bar{\kappa}(G)$.

If $\bar{\kappa}(G) = 1$, then G is connected and thus $\Gamma(G)$ is also connected, i.e., $\kappa(\Gamma(G)) \geq 1$. Let $\bar{\kappa}(G) \geq 2$ and $ux, vy \in V(\Gamma(G))$. If $u \neq v$, then there are $\bar{\kappa}(G)$ edge-disjoint paths from u to v in G, thus there are $\bar{\kappa}(G)$ vertex-disjoint paths from ux to vy in $\Gamma(G)$. Suppose that $u = v$ and $x \neq y$. In $K(u)$, there are $\deg_G(u) - 1$ vertex-disjoint paths from ux to uy. In G, if there is a path from x to y without containing u, then there are totally $\deg_G(u)$ vertex-disjoint paths from ux to uy. Suppose that in G every path from x to y contains u. Then, the maximum number of edge-disjoint paths from x to y is at most $\lfloor \frac{\deg_G(u)}{2} \rfloor \leq \deg_G(u) - 1$ and $\bar{\kappa}(G) \leq \lfloor \frac{\deg_G(u)}{2} \rfloor$. Thus, there are at least $\bar{\kappa}(G)$ vertex-disjoint paths from ux to uy in $\Gamma(G)$. Hence, $\kappa(\Gamma(G)) \geq \bar{\kappa}(G)$. □

4 Structural Properties of Subdivided-Line Graphs and Sierpiński-Like Graphs

4.1 Edge-Disjoint Hamilton Cycles

Edge-disjoint Hamilton cycles are fundamental subjects and have been studied from not only a theoretical point of view [5] but also a practical point of view, since the notion has applications in interconnection networks [1,6,20,22].

We first present a necessary and sufficient condition for the subdivided-line graph $\Gamma(G)$ to be Hamiltonian. A graph is called *supereulerian* if the graph has a spanning Eulerian subgraph.

Theorem 1. *$\Gamma(G)$ is Hamiltonian if and only if G is supereulerian.*

Proof: Suppose that $\Gamma(G)$ is Hamiltonian. Let C be a Hamilton cycle in $\Gamma(G)$. Contracting the vertices in each complete graph $K(v)$ into a single vertex v, a spanning connected subgraph F_C of G is obtained from C. Since C is a Hamilton cycle, the degree of each vertex of F_C must be even. Thus, F_C is a spanning Eulerian subgraph of G.

Conversely, suppose that G is supereulerian. Let F be a spanning Eulerian subgraph of G with an Euler tour. Based on F, we can construct a Hamilton cycle in $\Gamma(G)$ as follows. Let $v \in V(G)$ and $T(v) = \{\{\{u_1, v\}, \{v, w_1\}\}, \{\{u_2, v\}, \{v, w_2\}\}, \ldots, \{\{u_p, v\}, \{v, w_p\}\}\}$ be the set of pairs of edges which are incident to v and consecutively appear in the Euler tour in F. For every pair $\{\{u_i, v\}, \{v, w_i\}\}$ in $T(v)$ except for one pair $\{\{u_p, v\}, \{v, w_p\}\}$, we replace it with the consecutive three edges $\{u_iv, vu_i\}$, $\{vu_i, vw_i\}$ and $\{vw_i, w_iv\}$ in $\Gamma(G)$. Note that $\{vu_i, vw_i\} \in E(K(v))$. For the pair of edges $\{\{u_p, v\}, \{v, w_p\}\}$ in F, we replace it with the path starting from the original edge $\{u_pv, vu_p\}$ and ending at the original edge $\{vw_p, w_pv\}$ which contains all the vertices in $V(K(v)) - \{vu_1, vw_1, \ldots, vu_{p-1}, vw_{p-1}\}$. Repeating the similar process for all the other vertices in G and combining them according to the Euler tour, we have a Hamilton cycle in $\Gamma(G)$. □

A Hamilton cycle in a graph is clearly a spanning Euler subgraph of the graph. Thus, we have the following corollary.

Corollary 1. *$\Gamma^n(G)$ is Hamiltonian if G is supereulerian for all $n \geq 1$.*

Next, we consider a sufficient condition for the existence of edge-disjoint Hamilton cycles in $\Gamma(G)$. For complete graphs, the existence and constructions of edge-disjoint Hamilton cycles (paths) are already well-known. Namely, K_k can be decomposed into Hamilton cycles (respectively, Hamilton paths) if k is odd (respectively, even) (e.g., see Theorem 9.21 and Corollary 9.22 in [3]). This means that for any $\lfloor \frac{k}{2} \rfloor$ disjoint pairs $(u_1, v_1), (u_2, v_2), \ldots, (u_{\lfloor \frac{k}{2} \rfloor}, v_{\lfloor \frac{k}{2} \rfloor})$ of vertices of K_k, there are $\lfloor \frac{k}{2} \rfloor$ edge-disjoint Hamilton paths between u_i and v_i $(i = 1, 2, \ldots, \lfloor \frac{k}{2} \rfloor)$. Using the results, we can show the following result.

Theorem 2. *If there are ℓ edge-disjoint Hamilton cycles in G, then there are ℓ edge-disjoint Hamilton cycles in $\Gamma(G)$.*

Proof: Suppose that there are ℓ edge-disjoint Hamilton cycles $C_1, C_2, \ldots, C_\ell$ in G. For every vertex v of G, it holds that $deg_{C_i}(v) = 2$ for $i = 1, 2, \ldots, \ell$ and $\delta(G) \geq 2\ell$. Then, let $N_{C_i}(v) = \{u_i, w_i\}$ for $i = 1, 2, \ldots, \ell$. In $\Gamma(G)$, each vertex v of G is replaced with the complete graph $K(v)$ such that original edges of $\Gamma(G)$ are injectively incident to the vertices of $K(v)$. Since $K(v)$ has $\lfloor \frac{\deg_G(v)}{2} \rfloor$ $(\geq \lfloor \frac{\delta(G)}{2} \rfloor \geq \ell)$ edge-disjoint Hamilton paths between any $\lfloor \frac{\mathrm{dge}_G(v)}{2} \rfloor$ disjoint pairs of vertices for each $v \in V(G)$, by selecting ℓ edge-disjoint Hamilton paths $P_i(v)$ between vu_i and vw_i $(1 \leq i \leq \ell)$ in $K(v)$, and concatenating $P_i(v)$ and the original edges corresponding to the edges in C_i, ℓ edge-disjoint Hamilton cycles in $\Gamma(G)$ are obtained. □

By the similar construction, we can also obtain ℓ edge-disjoint Hamilton paths in $\Gamma(G)$ if G has ℓ edge-disjoint Hamilton paths, where $\ell \leq \lfloor \frac{\delta(G)}{2} \rfloor$. Applying Theorem 2 (with this remark) iteratively, we have the following corollary.

Corollary 2. *Let $\ell \leq \lfloor \frac{\delta(G)}{2} \rfloor$. If there are ℓ edge-disjoint Hamilton cycles (resp., paths) in G, then there are ℓ edge-disjoint Hamilton cycles (resp., paths) in $\Gamma^n(G)$ for all $n \geq 0$.*

Setting $G = K_k^\circ$ and $G = K_{k+1}$ in the above corollary and using the fact that there are $\lfloor \frac{k-1}{2} \rfloor$ edge-disjoint Hamilton cycles in K_k, the next results which were recently shown by Xue, Zuo, and Li [25], are immediately obtained. Note that the numbers of edge-disjoint Hamilton cycles in Theorem 3 are optimal. Figure 5 shows an example of two edge-disjoint Hamilton cycles in $S(2, 5)$.

Theorem 3. [25]

- *There are $\lfloor \frac{k-1}{2} \rfloor$ edge-disjoint Hamilton cycles in $S(n, k)$.*
- *There are $\lfloor \frac{k}{2} \rfloor$ edge-disjoint Hamilton cycles in $S^{++}(n, k)$.*

Besides, we can easily check that there are $\lfloor \frac{k}{2} \rfloor$ edge-disjoint Hamilton paths between $\lfloor \frac{k}{2} \rfloor$ disjoint pairs of extreme vertices in $S(n, k)$. Thus, there are $\lfloor \frac{k}{2} \rfloor$ edge-disjoint Hamilton cycles in $S^+(n, k)$, which was also shown in [25].

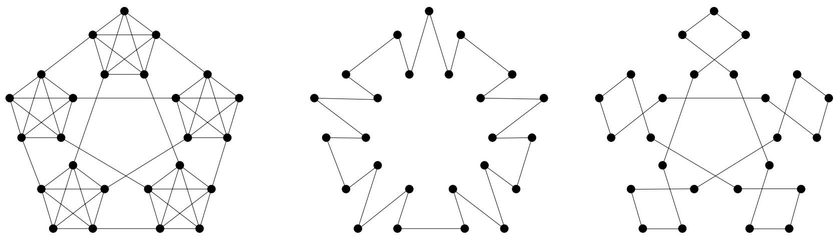


Fig. 5. The Sierpiński graph $S(2,5)$ and its two edge-disjoint Hamilton cycles

4.2 Hub Sets and Connected Dominating Sets

The notion of a hub set was introduced by Walsh [24]. A *hub set* S_h of G is a subset of $V(G)$ such that for every pair of distinct vertices $u, v \in V(G) - S_h$, there exists a path between u and v in which all the internal vertices are in S_h (in what follows, we call such a path a *correct path* between u and v). Note that there is no internal vertex for a path of length one. Thus, the condition of a hub set vacuously holds for any pair of adjacent vertices. The minimum cardinality of a hub set of G is the *hub number* $h(G)$ of G. A *connected hub set* S'_h of G is a hub set of G such that $\langle S'_h \rangle_G$ is connected and the minimum cardinality of a connected hub set of G is denoted by $h_c(G)$. There is a notion closely related to a connected hub set. A *connected dominating set* S_c of G is a subset of $V(G)$ such that for every vertex $v \in V(G) - S_c$, there exists a vertex w in S_c such that $\{v, w\} \in E(G)$ and $\langle S_c \rangle_G$ is connected. A connected dominating set has an application to location problems of resources in communication networks. The minimum cardinality of a connected dominating set of G is the *connected domination number* $\gamma_c(G)$ of G. A connected dominating set is a connected hub set. Thus, it holds that $h(G) \leq h_c(G) \leq \gamma_c(G)$. Grauman et. al [8] proved that $\gamma_c(G) \leq h(G) + 1$ and presented a polynomial-time algorithm for checking $h_c(G) = \gamma_c(G)$. Johnson, Slater, and Wlash [16] also characterized the graph G for which $\gamma_c(G) = h_c(G) + 1$.

We determine the hub number, the connected hub number, and the connected domination number of $\Gamma(G)$.

Theorem 4. *Let G be a connected graph with $\Delta(G) \geq 3$. Then,*

$$h(\Gamma(G)) = h_c(\Gamma(G)) = \gamma_c(\Gamma(G)) = 2(|V(G)| - 1).$$

Proof: First, we show that $\gamma_c(\Gamma(G)) \leq 2(|V(G)| - 1)$. Let T be a spanning tree in G. Let S_T be the set of vertices of $\Gamma(G)$ incident to original edges corresponding to edges in T, i.e., $S_T = \{vw \mid \{v, w\} \in E(T)\}$. Then, $\langle S_T \rangle_{\Gamma(G)}$ is clearly connected. Besides, for any vertex $uv \in V(\Gamma(G)) - S_T$, uv is incident to a vertex in S_T, since for any $u \in V(G)$, $K(u)$ contains at least one vertex in S_T. Therefore, S_T is a connected dominating set of $\Gamma(G)$.

Next, we will show that $h(\Gamma(G)) \geq 2(|V(G)| - 1)$. Let $S \subseteq V(\Gamma(G))$. For each connected component C of $\langle S \rangle_{\Gamma(G)}$, define the extended component $Ex(C)$ with respect to S as $C \cup (\cup_{v \in V(G), K(v) \cap V(C) \neq \emptyset} K(v))$. Besides, for each $K(u)$ containing no vertex in S, we call it an extended component for convenience. Now assume that $h(\Gamma(G)) < 2(|V(G)| - 1)$ and let S_h be a minimum hub set of $\Gamma(G)$ such that the number $N_c(S_h)$ of extended components with respect to S_h is minimum over all minimum hub sets of $\Gamma(G)$. Since $|S_h| < 2(|V(G)| - 1)$, $N_c(S_h) \geq 2$.

Since $\Gamma(G)$ is connected, there exist two extended components M and M' such that there is an original edge joining a vertex in M and a vertex in M'. Let $vw \in V(M)$ and $wv \in V(M')$. By definition, it does not hold that $vw, wv \in S_h$. Assume that $vw, wv \notin S_h$. If there is a vertex $xy \notin S_h$, where $x \neq y$, such that xy is in an extended component M'' different from M and M', then the vertex yx must be in $V(M) \cap S_h$ for a correct path between xy and vw and also must be in $V(M') \cap S_h$ for a correct path between xy and wv, which is a contradiction. Thus, for any vertex $xy \notin S_h \cup \{vw, wv\}$, xy is in $V(M) \cup V(M')$ such that if $xy \in V(M)$ (resp., $xy \in V(M')$), then $yx \in V(M') \cap S_h$ (resp., $yx \in V(M) \cap S_h$). Since $|V(\Gamma(G))| \geq 2|V(G)| + 1$, $|V(\Gamma(G)) - S_h| \geq 4$. Let $x_1y_1, x_2y_2 \notin S_h \cup \{vw, wv\}$. Then, letting $S'_h = (S_h - \{y_1x_1, y_2x_2\}) \cup \{vw, wv\}$, S'_h is also a minimum hub set, however, $N_c(S'_h) < N_c(S_h)$, which contradicts the definition of S_h. Therefore, one of vw and wv is in S_h and the other is not in S_h. Without loss of generality, we may assume that $vw \notin S_h$ and $wv \in S_h$. If there is another vertex $ab \in V(M) - \{vw\}$ such that $ba \in V(M')$, then one of ab and ba is in S_h, the other is not in S_h by the previous discussion, and we can also decrease the number of extended components while preserving the cardinality of a minimum hub set. Thus, there is only one original edge between M and M'. By the similar discussion, for any two extended components, there is at most one original edge between them. Suppose that there is a vertex $xy \in V(M) - S_h$ and $ab \in V(M') - S_h$. Then, for a correct path between xy and ab, there exists an extended component M'' different from M and M' such that $yx, ba \in V(M'') \cap S_h$. However, in such a case, we can check that there exists a vertex $cd \notin S_h \cup \{vw, xy, ab\}$ and a vertex ef in $\{vw, xy, ab\}$ such that there is no correct path between cd and ef. Hence $|V(M) - S_h| = 1$ or $|V(M') - S_h| = 0$.

Suppose that there is a vertex $xy \in V(M) - (S_h \cup \{vw\})$, i.e., $|V(M) - S_h| \geq 2$. Assume that there is a vertex $cd \in V(M'') - S_h$, where M'' is an extended component different from M and M' such that $dc \in V(M')$. Note that $dc \in S_h$. In this case, for the existence of a correct path between xy and cd, yx must be in $V(M'') \cap S_h$. It can be checked that there is no extended component containing a vertex not in S_h except for M, M', and M''. Besides, since $|V(\Gamma(G)) - S_h| \geq 4$, there is a vertex in $(V(M) \cup V(M') \cup V(M'')) - (S_h \cup \{vw, xy, cd\})$. However, it follows that there exist two original edges between M and M'' or between M' and M'', which is a contradiction by the previous discussion. Hence, if there is a vertex $cd \in V(M'') - S_h$, then $dc \in V(M) \cap S_h$. Since $V(M') \subseteq S_h$ and there is no vertex not in S_h adjacent to a vertex in $V(M')$, we can select an appropriate vertex ef in M' such that by letting $S'_h = (S_h - \{ef\}) \cup \{vw\}$, S'_h is a minimum hub set and $N_c(S'_h) < N_c(S_h)$, which is a contradiction. Therefore, $|V(M) - S_h| = 1$. Similarly, we can check that $|V(M') - S_h| = 0$.

If an extended component M^*, where $M^* \neq M, M'$, contains two vertices $x_1y_1, x_2y_2 \notin S_h$, then $y_1x_1, y_2x_2 \in (V(M) \cup V(M')) \cap S_h$, which implies the case that there are three extended components with three original edges among them, which is a contradiction by the previous discussions. Therefore, for any extended component M^* different from M and M', M^* contains at most one vertex not in S_h. Besides, if there are two extended

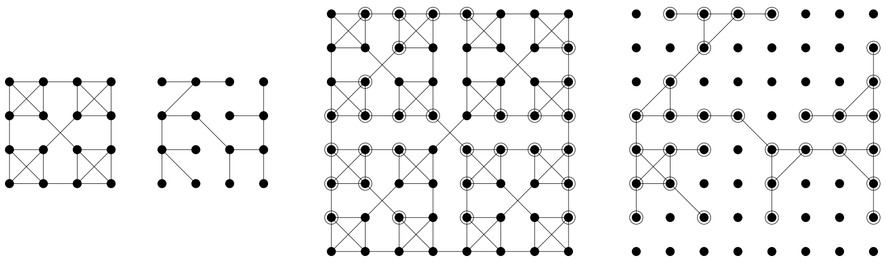


Fig. 6. The Sierpiński graph $S(2,4)$, a spanning tree T in $S(2,4)$, the minimum connected dominating set S_c in $S(3,4)$ based on T, and $\langle S_c \rangle_{S(3,4)}$ with vertices not in S_c, where each vertex in S_c is denoted by an additional circle

components M'' and M''' such that $ab \in V(M'')-S_h$, $cd \in V(M''')-S_h$, $ba \in V(M)\cap S_h$, and $dc \in V(M')\cap S_h$, then there is no correct path between ab and cd. Therefore, we can select an appropriate vertex ef in M or M' such that by letting $S'_h = (S_h - \{ef\}) \cup \{vw\}$, S'_h is a minimum hub set and $N_c(S'_h) < N_c(S_h)$, which is a contradiction. Consequently, we have $h(\Gamma(G)) \geq 2(|V(G)| - 1)$. □

Corollary 3. *Let G be a connected k-regular graph such that $k \geq 3$. Then,*

$$h(\Gamma^n(G)) = h_c(\Gamma^n(G)) = \gamma_c(\Gamma^n(G)) = 2(k^{n-1}|V(G)| - 1) \textit{ for all } n \geq 1.$$

From Corollary 3, the following results for $S(n,k)$ and $S^{++}(n,k)$, which were shown by Lin et. al [19] except for $\gamma_c(S(n,k))$ and $\gamma_c(S^{++}(n,k))$, are immediately obtained. By the proof of Theorem 4, based on a spanning tree in $S(n-1,k)$ (resp., $S^{++}(n-1,k)$), we can construct a minimum connected dominating set with $2(|V(S(n-1,k))| - 1)$ (resp., $2(|V(S^{++}(n-1,k))| - 1)$) vertices for $S(n,k)$ (resp., $S^{++}(n,k)$). Figure 6 illustrates a minimum connected dominating set in $S(3,4)$ based on a spanning tree in $S(2,4)$.

Theorem 5. [19]

- $h(S(n,k)) = h_c(S(n,k)) = \gamma_c(S(n,k)) = 2(k^{n-1} - 1)$ *for all* $n > 1$.
- $h(S^{++}(n,k)) = h_c(S^{++}(n,k)) = \gamma_c(S^{++}(n,k)) = 2(k^{n-1} + k^{n-2} - 1)$ *for all* $n > 1$.

By slightly modifying the proof of Theorem 4, we can show that $h(S^{+}(n,k)) = h_c(S^{+}(n,k)) = \gamma_c(S^{+}(n,k)) = 2k^{n-1} - k + 1$, which were also shown in [19] except for $\gamma_c(S^{+}(n,k))$.

4.3 Completely Independent Spanning Trees

Motivated by fault-tolerant communication problems in interconnection networks, the notion of completely independent spanning trees was introduced in [10]. *Completely independent spanning trees* $T_1, T_2, \ldots, T_\ell$ in G are spanning trees in G such that for every pair of distinct vertices u and v, the ℓ paths between u and v in $T_1, T_2, \ldots, T_\ell$ are pairwise internally vertex-disjoint. It was also shown in [10] that $T_1, T_2, \ldots, T_\ell$ in G are completely independent spanning trees if and only if $T_1, T_2, \ldots, T_\ell$ are edge-disjoint spanning trees such that for every vertex v in G, the degrees of v in $T_1, T_2, \ldots, T_\ell$ are one except for at most one spanning tree T_i. Based on this characterization, we can construct ℓ completely independent spanning trees in G by coloring the vertices of G using ℓ colors $c_1, \ldots, c_\ell$ so that there are ℓ edge-disjoint spanning trees $T_1, \ldots, T_\ell$ such that all the internal (non-leaf) vertices of T_i are colored by c_i. The notion of completely independent spanning trees is related to connected dominating sets. Namely, if there are ℓ completely independent spanning trees in G, then there are ℓ vertex-disjoint connected dominating sets. Until now, the existence of completely independent spanning trees have been studied for several graph classes [10,11,12]. Besides, it was shown in [21] that there is no direct relationship between the vertex-connectivity of G and the number of completely independent spanning trees in G.

Lemma 2. *For any partition of $V(K_k)$ with ℓ parts $P_1, P_2, \ldots, P_\ell$, if $|P_i| \geq 2$ for each i, then K_k has ℓ completely independent spanning trees $T_1, T_2, \ldots, T_\ell$ such that the set of internal vertices of T_i is contained in P_i for each i.*

Proof: For the complete graph K_k with even k, we can directly construct $\frac{k}{2}$ completely independent spanning trees as mentioned in [11]. Let $V(K_k) = \{v_0, v_1, \ldots, v_{k-1}\}$. Define completely independent spanning trees T_i in K_k as follows:

$$T_i = \left\langle \left\{ \{v_i, v_{i+j}\} \;\middle|\; 1 \leq j \leq \frac{k}{2} \right\} \cup \left\{ \{v_{i+\frac{k}{2} \bmod k}, v_{i+\frac{k}{2}+j \bmod k}\} \;\middle|\; 1 \leq j < \frac{k}{2} \right\} \right\rangle_{K_k}$$

for $i = 0, 1, \ldots, \frac{k}{2} - 1$. Namely, T_i is obtained from T_0 by clockwise-shifting i times provided that the vertices of K_k are placed in circular positions.

Select two elements a_i, b_i from each P_i and let $S = \{a_1, b_1, a_2, b_2, \ldots, a_\ell, b_\ell\}$. Then, we construct ℓ completely independent spanning trees $T_1, T_2, \ldots, T_\ell$ in $\langle S \rangle_{K_k}$ so that $\{a_i, b_i\}$ is the set of internal vertices in T_i for each i by the above construction. For each vertex $x \in V(K_k) - S$, adding x to T_i with the edge $\{x, a_i\}$, ℓ completely independent spanning trees in K_k are obtained. □

Theorem 6. *If there are ℓ edge-disjoint spanning trees $T_1, T_2, \ldots, T_\ell$ in a graph G such that for any vertex v of G,*

$$\deg_G(v) - \sum_{1 \leq i \leq \ell} \deg_{T_i}(v) \geq |\{i \mid \deg_{T_i}(v) = 1\}|,$$

then there are ℓ completely independent spanning trees in $\Gamma(G)$.

Proof: Let $T_1, T_2, \ldots, T_\ell$ be ℓ edge-disjoint spanning trees in G. Let H_i be the subgraph of $\Gamma(G)$ induced by the set of the original edges corresponding to the edges in $E(T_i)$, i.e., $H_i = \langle \{\{vw, wv\} \mid \{v, w\} \in E(T_i)\} \rangle_{\Gamma(G)}$. Then, H_i is a non-spanning tree of $\Gamma(G)$ for each $i \in \{1, 2, \ldots, \ell\}$ such that $H_1, H_2, \ldots, H_\ell$ are edge-disjoint.

For each $K(v)$ of $\Gamma(G)$, let $P_i(v) = V(K(v)) \cap V(H_i)$ for $i = 1, 2, \ldots, \ell$. From the condition that $\deg_G(v) - \sum_{1 \leq i \leq \ell} \deg_{T_i}(v) \geq |\{i \mid \deg_{T_i}(v) = 1\}|$, we can select $S_i \subseteq V(K(v)) - \cup_{1 \leq i \leq \ell} V(H_i)$, where $S_i \cap S_j = \emptyset$ for $i \neq j$, so that $V(K_k)$ is partitioned to ℓ parts $P_1(v) \cup S_1, P_2(v) \cup S_2, \ldots, P_\ell(v) \cup S_\ell$ and $|P_i(v) \cup S_i| \geq 2$ for all $i \in \{1, 2, \ldots, \ell\}$.

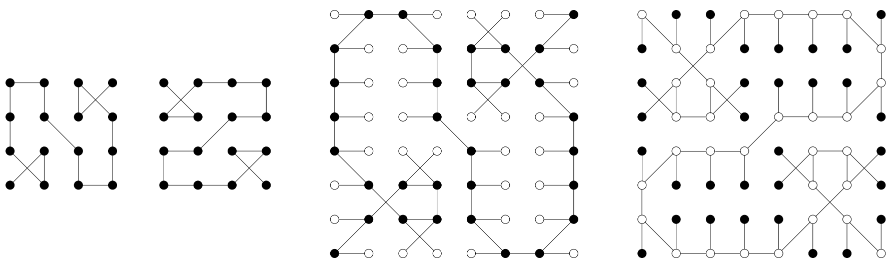


Fig. 7. Two edge-disjoint Hamilton paths in $S(2,4)$ and the corresponding two isomorphic completely independent spanning trees T_1 and T_2 in $S(3,4)$, where a vertex of $S(3,4)$ is colored white if it is an internal vertex of T_2

From Lemma 2, there are ℓ completely independent spanning trees $T_1^*, T_2^*, \ldots, T_\ell^*$ in each $K(v)$ such that for each T_i^*, the set of internal vertices is contained in $P_i(v) \cup S_i$. Hence, for $i = 1, 2, \ldots, \ell$, combining T_i^* with H_i and deleting appropriate edges if $|P_i(v)| > 2$, we have ℓ completely independent spanning trees in $\Gamma(G)$. □

If $\ell \leq \lfloor \frac{\delta(G)}{2} \rfloor$ and there are ℓ edge-disjoint Hamilton paths $T_1, T_2, \ldots, T_\ell$ in G, then the degree condition in Theorem 6 is satisfied since $\deg_{T_i}(v) \leq 2$ for every T_i and every $v \in V(G)$. Therefore, the following corollary holds.

Corollary 4. *Let $\ell \leq \lfloor \frac{\delta(G)}{2} \rfloor$. If there are ℓ edge-disjoint Hamilton paths in a graph G, then there are ℓ completely independent spanning trees in $\Gamma(G)$.*

From Corollaries 2 and 4, we have the next result on iterated subdivided-line graphs.

Corollary 5. *Let $\ell \leq \lfloor \frac{\delta(G)}{2} \rfloor$. If there are ℓ edge-disjoint Hamilton paths in G, then there are ℓ completely independent spanning trees in $\Gamma^n(G)$ for all $n \geq 1$.*

Applying Corollary 5 to Sierpiński-like graphs, the following theorem is obtained. We can easily check that the number of completely independent spanning trees in this theorem is optimal since completely independent spanning trees are edge-disjoint each other. In particular, when k is even, the $\frac{k}{2}$ completely independent spanning trees in $S(n,k)$ ($S^{++}(n,k)$) are isomorphic each other. Figure 7 shows two isomorphic completely independent spanning trees in $S(3,4)$ based on two edge-disjoint Hamilton paths in $S(2,4)$.

Theorem 7.

- *There are $\lfloor \frac{k}{2} \rfloor$ completely independent spanning trees in $S(n,k)$.*
- *There are $\lfloor \frac{k}{2} \rfloor$ completely independent spanning trees in $S^{++}(n,k)$.*

As mentioned in Subsection 4.1, there are $\lfloor \frac{k}{2} \rfloor$ edge-disjoint Hamilton paths between $\lfloor \frac{k}{2} \rfloor$ disjoint pairs of extreme vertices in $S(n,k)$. Thus, we can also construct $\lfloor \frac{k}{2} \rfloor$ completely independent spanning trees in $S^{+}(n,k)$.

5 Concluding Remarks

In this paper, we have introduced a notion of the subdivided-line graph operation and iterated subdivided-line graphs. The class of iterated subdivided-line graphs generalizes the class of Sierpiński-like graphs. By investigating structural properties on iterated subdivided-line graphs, we have obtained generalized results with simpler proofs for the previously known results on edge-disjoint Hamilton cycles and minimum hub sets of the Sierpiński-like graphs. Besides, we have obtained new results on completely independent spanning trees in iterated subdivided-line graphs which can be applied to fault-tolerant communication problems in WK-recursive networks.

For digraphs, the line digraph operation L is well-known as a useful graph operation to construct a class of digraphs with bounded degree and small diameter. In fact, the classes of de Brujin digraphs and Kautz digraphs which are known as interconnection networks can be constructed by using the line digraph operation. By the subdivided-line graph operation Γ, we can generate a class of graphs $\Gamma^n(G)$ with bounded degree and good extendability. Unfortunately, the diameter of a graph in the class is not small in general, i.e., the diameter is not a logarithm of the number of vertices. However, for small n, there are cases for which graphs generated by the subdivided-line graph operation have advantages on the number of vertices compared with graphs with logarithmic diameter. For example, Table 1 shows the comparison of the hypercube and the (extended) Sierpiński graphs for the order (the number of vertices), degree, and diameter for small n. The k-dimensional hypercube Q_k is one of the most popular interconnection networks with logarithmic diameter. In each case of Table 1, $S(n,k)$ and/or $S^{++}(n,k)$ have more vertices than the hypercube with the same degree and the same (or larger) diameter. The hypercube has a logarithmic diameter, while it does not have a property of bounded degree. Then the hybrid class $\Gamma^n(Q_k)$ might be another candidate for interconnection networks. In particular, as a spanning subgraph, $\Gamma(Q_k)$ contains the cube-connected-cycles CCC_k which is also known as an interconnection network, and when $n = 3$ it holds that $\Gamma(Q_3) \cong CCC_3$.

Table 1. The cases for which $S(n,k)$ and/or $S^{++}(n,k)$ have advantages on the order compared with the hypercube Q_k when $n \leq 4$

		Hypercube Q_k		Sierpiński Graph $S(n,k)$		Extended Sierpiński $S^{++}(n,k)$	
n	degree k	diameter k	order 2^k	diameter 2^n-1	order k^n	diameter 2^n-1	order k^n+k^{n-1}
2	3	3	8	3	9	3	12
2	4	4	16	3	16	3	20
3	7	7	128	7	343	7	392
3	8	8	256	7	512	7	576
3	9	9	512	7	729	7	810
3	10	10	1024	7	1000	7	1100
4	15	15	32768	15	50625	15	54000
4	16	16	65536	15	65536	15	69632

References

1. Bae, M.M., Bose, B.: Edge disjoint Hamiltonian cycles in k-ary n-cubes and hypercubes. IEEE Trans. Comput. 52, 1271–1284 (2003)
2. Beaudou, L., Gravier, S., Klavžar, S., Kovše, M., Mollard, M.: Covering codes in Sierpiński graphs. Discret. Math. Theor. Comput. Sci. 12, 63–74 (2010)
3. Chartrand, G., Lesniak, L.: Graphs & Digraphs, 4th edn. Chapman & Hall/CRC (2005)
4. Chen, G.-H., Duh, D.-R.: Topological properties, communication, and computation on WK-recursive networks. Networks 24, 303–317 (1994)
5. Christofides, D., Kühn, D., Osthus, D.: Edge-disjoint Hamilton cycles in graphs. J. Combin. Theory Ser. B 102, 1035–1060 (2012)
6. Čada, R., Kaiser, T., Rosenfeld, M., Ryjáček, Z.: Disjoint Hamilton cycles in the star graph. Inform. Process. Lett. 110, 30–35 (2009)
7. Duh, D.-R., Chen, G.-H.: Topological properties of WK-recursive networks. J. Parallel and Distrib. Comput. 23, 468–474 (1994)
8. Grauman, T., Hartke, S.G., Jobson, A., Kinnersley, B., West, D.B., Wiglesworth, L., Worah, P., Wu, H.: The hub number of a graph. Inf. Process. Lett. 108, 226–228 (2008)
9. Gravier, S., Kovše, M., Mollard, M., Moncel, J., Parreau, A.: New results on variants of covering codes in Sierpiński graphs. Des. Codes Cryptogr. (to appear)
10. Hasunuma, T.: Completely independent spanning trees in the underlying graph of a line digraph. Discrete Math. 234, 149–157 (2001)
11. Hasunuma, T.: Completely independent spanning trees in maximal planar graphs. In: Kučera, L. (ed.) WG 2002. LNCS, vol. 2573, pp. 235–245. Springer, Heidelberg (2002)
12. Hasunuma, T., Morisaka, C.: Completely independent spanning trees in torus networks. Networks 60, 59–69 (2012)
13. Hinz, A.M., Parisse, P.: Coloring Hanoi and Sierpiński graphs. Discrete Math. 312, 1521–1535 (2012)
14. Hinz, A.M., Parisse, P.: The average eccentricity of Sierpiński graphs. Graphs Combin. 28, 671–686 (2012)
15. Jakovac, M., Klavžar, S.: Vertex-, edge-, and total-coloring of Sierpiński-like graphs. Discrete Math. 309, 1548–1556 (2009)
16. Johnson, P., Slater, P., Walsh, M.: The connected hub number and the connected domination number. Networks 58, 232–237 (2011)
17. Klavžar, S., Milutinović, U.: Graphs $S(n,k)$ and a variant of the Tower of Hanoi problem. Czechoslovak Math. J. 47(122), 95–104 (1997)
18. Klavžar, S., Mohar, B.: Crossing numbers of Sierpiński-like graphs. J. Graph Theory 50, 186–198 (2005)
19. Lin, C.-H., Liu, J.-J., Wang, Y.-L., Yen, W.C.-K.: The hub number of Sierpiński-like graphs. Theory Comput. Syst. 49, 588–600 (2011)
20. Micheneau, C.: Disjoint Hamiltonian cycles in recursive circulant graphs. Inform. Process. Lett. 61, 259–264 (1997)
21. Péterfalvi, F.: Two counterexamples on completely independent spanning trees. Discrete Math. 312, 808–810 (2012)
22. Rowley, R., Bose, B.: Edge-disjoint Hamiltonian cycles in de Bruijn networks. In: Proceedings of the Distributed Memory Computing Conference, vol. 6, pp. 707–709 (1991)
23. Vecchia, G.D., Sanges, C.: A recursively scalable network VLSI implementation. Future Generat. Comput. Syst. 4, 235–243 (1988)
24. Walsh, M.: The hub number of a graph. Int J. Math. Comput. Sci. 1, 117–124 (2006)
25. Xue, B., Zuo, L., Li, G.: The hamiltonicity and path t-coloring of Sierpiński-like grraphs. Discrete Applied Math. 160, 1822–1836 (2012)

Induced Subtrees in Interval Graphs*

Pinar Heggernes[1], Pim van 't Hof[1], and Martin Milanič[2]

[1] Department of Informatics, University of Bergen, Norway
{pinar.heggernes,pim.vanthof}@ii.uib.no
[2] UP IAM and UP FAMNIT, University of Primorska, Slovenia
martin.milanic@upr.si

Abstract. The Induced Subtree Isomorphism problem takes as input a graph G and a tree T, and the task is to decide whether G has an induced subgraph that is isomorphic to T. This problem is known to be NP-complete on bipartite graphs, but it can be solved in polynomial time when G is a forest. We show that Induced Subtree Isomorphism can be solved in polynomial time when G is an interval graph. In contrast to this positive result, we show that the closely related Subtree Isomorphism problem is NP-complete even when G is restricted to the class of proper interval graphs, a well-known subclass of interval graphs.

1 Introduction and Background

The problems Subgraph Isomorphism and Induced Subgraph Isomorphism both take as input two graphs G and H, and the task is to determine whether G has a subgraph or an induced subgraph, respectively, that is isomorphic to H. Subgraph Isomorphism and Induced Subgraph Isomorphism are two well-studied and notoriously hard problems in the area of graph algorithms, generalizing classical NP-complete problems such as Clique, Independent Set and Hamiltonian Path.

Both Subgraph Isomorphism and Induced Subgraph Isomorphism are known to be NP-complete already when each of G and H is a disjoint union of paths [7, 9], and thus both problems are NP-complete on any hereditary graph class that contains arbitrarily long induced paths. In particular, both problems are NP-complete on proper interval graphs and on bipartite permutation graphs. Interestingly, Induced Subgraph Isomorphism can be solved in polynomial time on connected proper interval graphs and connected bipartite permutation graphs [13], whereas Subgraph Isomorphism remains NP-complete on these connected graph classes [16]. Both problems can be solved in polynomial time when G is a forest and H is a tree [23], but remain NP-complete when G is a tree and H is a forest [9].

The problems Subgraph Isomorphism and Induced Subgraph Isomorphism remain hard also on several classes of graphs that do not contain long induced paths. For example, both problems are NP-complete on connected

* This work is supported by the Research Council of Norway, by the Slovenian Research Agency, and by the European Science Foundation.

© Springer-Verlag Berlin Heidelberg 2013

cographs [1, 5, 7], and remain NP-complete even on connected trivially perfect graphs [2, 16], a subclass of cographs. Both Subgraph Isomorphism and Induced Subgraph Isomorphism are also NP-complete when both input graphs are split graphs [7, 11]. Kijima et al. [16] showed that Subgraph Isomorphism is NP-complete when G is restricted to the class of chain graphs, cochain graphs, or threshold graphs, but the problem becomes polynomial-time solvable when, in addition, H is also restricted to the same class as G.

Given the large amount of hardness results on the two problems, even under severe restrictions on G, it makes sense to consider different restrictions on H in an attempt to obtain tractability. A natural candidate for such a restriction is demanding H to be a tree. This brings us to the two problems that we focus on in this paper:

(Induced) Subtree Isomorphism
Input: A graph G and a tree T.
Question: Does G have an (induced) subgraph isomorphic to T?

Since Subtree Isomorphism is a generalization of Hamiltonian Path, this problem is NP-complete on all graph classes on which Hamiltonian Path is NP-complete, like planar graphs, chordal bipartite graphs, and strongly chordal split graphs [9, 26]. Similarly, the fact that finding a longest induced path in a bipartite graph is NP-hard [9] implies that Induced Subtree Isomorphism is NP-complete on bipartite graphs. Both Subtree Isomorphism and Induced Subtree Isomorphism are also NP-complete on graphs of treewidth at most 2 [22], but can be solved in polynomial time when G is a forest [23].

The main result of this paper is a polynomial-time algorithm for Induced Subtree Isomorphism on interval graphs, which gives a nice contrast to the NP-completeness of Induced Subgraph Isomorphism on this graph class. On the negative side, we show that Subtree Isomorphism is NP-complete already on proper interval graphs. Note that the problem of finding a longest path in an interval graph can be solved in polynomial time [14] (and even on cocomparability graphs [24]). Hence, our negative result on proper interval graphs shows that finding a given tree as a subgraph in a proper interval graph is much harder than finding a path of given length in such a graph, despite the linear structure that proper interval graphs possess.

2 Definitions and Notation

All graphs considered in this paper are finite, undirected and simple. We refer to the monograph by Diestel [8] for basic graph terminology not defined below. Detailed information on all the graph classes mentioned in this paper can be found in the books by Golumbic [12] and Brandstädt, Le, and Spinrad [3]. Figure 1 below shows the inclusion relations between most of the graph classes mentioned this paper.

A graph is an *interval graph* if there is a bijection between its vertices and a family of closed intervals of the real line such that two vertices are adjacent if

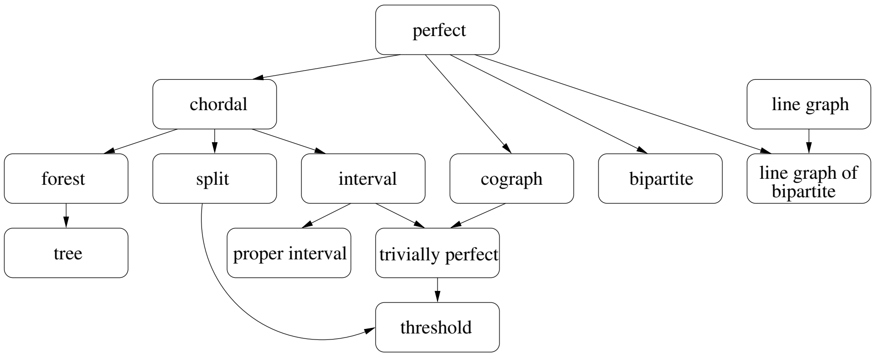


Fig. 1. An overview of most of the graph classes mentioned in this paper. An arrow from a class $\mathcal{G}$ to a class $\mathcal{H}$ indicates that $\mathcal{H}$ is a subset of $\mathcal{G}$.

and only if the two corresponding intervals overlap. Such a bijection is called an *interval representation* of the graph. A graph is a *proper interval graph* if it has an interval representation where no interval properly contains another interval. Many different characterizations of interval graphs are known in the literature. In order to state the one we will use in our algorithm, we need the following definition.

Definition 1. *Let $G = (V, E)$ be a graph. An ordering $(u_1, \ldots, u_n)$ of V is called an* interval order *of G if, for every triple (i, j, k) with $1 \leq i < j < k \leq n$, it holds that $u_i u_k \in E$ implies $u_j u_k \in E$.*

Olariu [27] showed that a graph G is an interval graph if and only if G has an interval order.

A *tree* is a connected graph without cycles. Let G be a graph and let T be a tree. If G has a induced subgraph that is isomorphic to T, then we say that T is an *induced subtree* of G; a *subtree* of G is defined analogously. A tree T is a *caterpillar* if it has a path that contains every vertex of degree at least 2 in T; such a path is called a *backbone* of T. A well-known characterization of interval graphs by Lekkerkerker and Boland [20] immediately implies that a tree is an interval graph if and only if it is a caterpillar.

Let G and H be two graphs. A mapping $\varphi : V(H) \to V(G)$ is said to be an *induced subgraph isomorphism*, or ISI *mapping* for short, of H into G, if φ is injective and $uv \in E(H)$ if and only if $\varphi(u)\varphi(v) \in E(G)$ for all $u, v \in V(H)$. Consequently, H is an induced subgraph of G if and only if there exists an ISI mapping of H into G.

3 Induced Subtree Isomorphism on Interval Graphs

In this section, we show that Induced Subtree Isomorphism can be solved in polynomial time on interval graphs. Before presenting our algorithm in the proof of Theorem 1 below, we first prove a sequence of five lemmas, as well as a corollary of these lemmas that forms the main ingredient of our algorithm.

Throughout this section, up to the statement of Theorem 1, let $G = (V, E)$ be a connected interval graph and let T be a caterpillar on at least three vertices. We fix an interval order $\sigma = (u_1, \dots, u_n)$ of G. For any two vertices $x, y \in V$, we write $x \prec_\sigma y$ if x appears before y in the interval order σ, i.e., if $x = u_i$ and $y = u_j$ for some $i < j$.

Suppose there exists an ISI mapping φ of T into G. Then, for any ordered path $P = (t_1, \dots, t_p)$ of T, we say that P is *φ-increasing* if $\varphi(t_1) \prec_\sigma \varphi(t_2) \prec_\sigma \cdots \prec_\sigma \varphi(t_p)$, and P is *φ-decreasing* if $\varphi(t_p) \prec_\sigma \varphi(t_{p-1}) \prec_\sigma \cdots \prec_\sigma \varphi(t_1)$.

Lemma 1. *Let $P = (t_1, \dots, t_p)$ be an ordered path in T whose vertices all have degree at least 2 in T. Then, for any* ISI *mapping φ of T into G, the path P is either φ-increasing or φ-decreasing.*

Proof. The statement is trivially true if $p \leq 2$. Let $p \geq 3$. Suppose, for contradiction, that there exists an ISI mapping φ of T into G such that P is neither φ-increasing nor φ-decreasing. Then, in particular, there exist three consecutive vertices t_i, t_{i+1}, t_{i+2} of P such that the ordered path (t_i, t_{i+1}, t_{i+2}) is neither φ-increasing nor φ-decreasing. Let $u_{j_1} = \varphi(t_i)$, $u_{j_2} = \varphi(t_{i+1})$ and $u_{j_3} = \varphi(t_{i+2})$. Without loss of generality, we may assume that $j_1 < j_3$. Observe that Definition 1 implies that $j_1 < j_2$. Since we assumed the ordered path (t_i, t_{i+1}, t_{i+2}) to be neither φ-increasing nor φ-decreasing, it holds that $j_2 > j_3$.

Let w be a T-neighbor of t_{i+2} other than t_{i+1}; such a vertex w exists since we assume that all vertices of P have degree at least 2 in T. Let $u_{j_w} = \varphi(w)$. We consider three cases according to the value of j_w.

- Suppose that $j_w < j_1$. Since φ preserves adjacencies and $wt_{i+2} \in E(T)$, we have $u_{j_w}u_{j_3} \in E(G)$. On the other hand, since φ preserves non-adjacencies and $t_it_{i+2} \notin E(T)$, we have $u_{j_1}u_{j_3} \notin E(G)$. However, this contradicts Definition 1 applied to the triple $(i, j, k) = (j_w, j_1, j_3)$.
- Suppose that $j_1 < j_w < j_2$. Since φ preserves adjacencies and $t_it_{i+1} \in E(T)$, we have $u_{j_1}u_{j_2} \in E(G)$. On the other hand, since φ preserves non-adjacencies and $wt_{i+1} \notin E(T)$, we have $u_{j_w}u_{j_2} \notin E(G)$. Again, we have a contradiction to Definition 1, this time for the triple $(i, j, k) = (j_1, j_w, j_2)$.
- Suppose that $j_2 < j_w$. Since φ preserves adjacencies and $t_{i+2}w \in E(T)$, we have $u_{j_3}u_{j_w} \in E(G)$. On the other hand, since φ preserves non-adjacencies and $t_{i+1}w \notin E(T)$, we have $u_{j_2}u_{j_w} \notin E(G)$. We have a contradiction to Definition 1 for the triple $(i, j, k) = (j_3, j_2, j_w)$.

This completes the proof of the lemma. $\square$

We will show in Lemma 2 below that the problem of determining whether G has an induced subgraph isomorphic to T can be reduced to computing the values of a certain Boolean-valued function f_B for each $B \in \mathcal{B}$, where $\mathcal{B}$ is a set of three or four so-called *representative backbones* of T. In order to state the lemma, we first need to define the set $\mathcal{B}$ and the function f_B.

Recall that a backbone of T is any path B containing all vertices of degree at least 2 in T. The *order* of a backbone B is the number of vertices in B and is denoted by $|B|$. For any ordered backbone $B = (t_1, t_2, \ldots, t_p)$, the ordered backbone $B^{-1} = (t_p, t_{p-1}, \ldots, t_1)$ is called the *reverse* of B. Given two ordered backbones $B = (t_1, t_2, \ldots, t_p)$ and $B' = (t'_1, \ldots, t'_{p'})$ of T, we say that B and B' are *equivalent* if $p = p'$ and $t_i = t'_i$ for all $i \in \{1, 2, \ldots, p-1\}$.

We now define a set $\mathcal{B}$ consisting of three or four non-equivalent ordered backbones of T as follows. Let $B_{\min} = (t_1, t_2, \ldots, t_p)$ be an ordered backbone of T of minimum order, i.e., $B_{\min}$ is an ordered backbone of T whose vertex set consists of exactly those vertices that have degree at least 2 in T. Let $B_{\min}^{-1} = (t_p, t_{p-1}, \ldots, t_1)$ be the reverse of $B_{\min}$, where $B_{\min} = B_{\min}^{-1}$ if $p = 1$. Note that every ordered backbone of T other than $B_{\min}$ or $B_{\min}^{-1}$ contains either $|V(B_{\min})| + 1$ or $|V(B_{\min})| + 2$ vertices. Let us fix a neighbor t'_1 of t_1 and a neighbor t'_p of t_p such that $t'_1 \neq t_p$ and such that both t'_1 and t'_p have degree 1 in T. Such neighbors t'_1 and t'_p exist due to the fact that both t_1 and t_p have degree at least 2 in T and the assumption that T has at least three vertices. We now define two ordered backbones $B^+ = (t_1, t_2, \ldots, t_p, t'_p)$ and $B^- = (t_p, t_{p-1}, \ldots, t_1, t'_1)$. Finally, we define $\mathcal{B} = \{B_{\min}, B_{\min}^{-1}, B^+, B^-\}$. The backbones in $\mathcal{B}$ are called the *representative backbones* of T. Since we assume that T contains at least 3 vertices and $B_{\min} = B_{\min}^{-1}$ if and only if $p = 1$, we have that $|\mathcal{B}| = 3$ if $p = 1$ and $|\mathcal{B}| = 4$ if $p \geq 2$.

Let $B = (t_1, \ldots, t_p) \in \mathcal{B}$ be a representative backbone of T. For every $i \in \{1, \ldots, p\}$, let B_i denote the subgraph of T induced by the first i vertices of B together with their neighbors outside the backbone, that is,

$$B_i = T\left[\{t_1, \ldots, t_i\} \cup L_1 \cup \ldots \cup L_i\right] ,$$

where L_i denotes the set of neighbors of t_i outside B, i.e., $L_i = N_T(t_i) \setminus V(B)$. The following straightforward property of representative backbones will be useful in later proofs.

Observation 1. *For every representative backbone $B = (t_1, \ldots, t_p) \in \mathcal{B}$ and every $i \in \{1, \ldots, p\}$, vertex t_i has a neighbor in B_i.* □

For $1 \leq j < k \leq n$, let G_j^k denote the subgraph of G induced by $\{u_1, \ldots, u_j, u_k\}$, that is,

$$G_j^k = G\big[\{u_1, \ldots, u_j, u_k\}\big] .$$

With these definitions in mind, we define a Boolean-valued function $f_B : \mathcal{T}_B \to \{0, 1\}$, where

$$\mathcal{T}_B = \{(i, j, k) \mid 1 \leq i \leq p \quad \text{and} \quad 1 \leq j < k \leq n\},$$

as follows: for all $(i, j, k) \in \mathcal{T}_B$, we set $f_B(i, j, k) = 1$ if and only if there exists an ISI mapping φ of B_i into G_j^k such that $\varphi(t_i) = u_k$.

Lemma 2. *Graph G has an induced subgraph isomorphic to T if and only if there exists a backbone $B \in \mathcal{B}$ and an integer $k \in \{2, \ldots, n\}$ such that $f_B(|B|, k-1, k) = 1$.*

Proof. Suppose first that there exists a backbone $B \in \mathcal{B}$ and an integer k with $1 < k \leq n$ such that $f_B(|B|, k-1, k) = 1$. Then, denoting $B = (t_1, \ldots, t_p)$, there exists an ISI mapping φ of $B_p = T$ into $G^k_{k-1} = G[\{u_1, \ldots, u_k\}]$ such that $\varphi(t_p) = u_k$. Trivially, φ is an ISI mapping of T into G, and hence G has an induced subgraph isomorphic to T.

Conversely, suppose G has an induced subgraph that is isomorphic to T. Then there exists an ISI mapping φ of T into G. Since every vertex of $B_{\min}$ and $B^{-1}_{\min}$ has degree at least 2 in T, either $B_{\min}$ or $B^{-1}_{\min}$ is φ-increasing due to Lemma 1. Among all pairs (φ, B) where φ is an ISI mapping of T into G and B is a φ-increasing backbone in $\mathcal{B}$, choose a pair (φ, B) such that $B = (t_1, \ldots, t_p) \in \mathcal{B}$ is of maximum possible order. Let $\varphi(t_j) = u_{i_j}$ for all $j \in \{1, \ldots, p\}$. Let $k = i_p$.

We claim that, with B and k defined as above, it holds that $f_B(p, k-1, k) = 1$. By definition of f_B, this is equivalent to verifying the existence of an ISI mapping ψ of $B_p = T$ into $G^k_{k-1} = G[\{u_1, \ldots, u_k\}]$ such that $\psi(t_p) = u_k$. We claim that such a mapping is obtained by taking $\psi = \varphi$. Condition $\psi(t_p) = u_k$ follows from the definition of k, and the fact that ψ is an injective mapping that preserves adjacencies and non-adjacencies follows trivially from the corresponding properties of φ. It remains to verify that $\psi(T) = \varphi(T) \subseteq \{u_1, \ldots, u_k\}$.

Suppose for a contradiction that there exists a vertex $v \in V(T)$ such that $\varphi(v) = u_r$ for $r > k$. Since B is φ-increasing, we have $i_j \leq i_p = k$ for all $1 \leq j \leq p$, and hence v is not a vertex of B. Therefore, vertex v has a unique neighbor t_j in B. Suppose first that $1 \leq j < p$. Then, $p \geq 2$ and since φ preserves adjacencies and non-adjacencies, we conclude that $u_{i_j}u_r \in E(G)$ and $u_k u_r \notin E(G)$, which contradicts Definition 1 applied to the triple (i_j, k, r) with $i_j < k < r$. Therefore, we may assume that $j = p$, and v is adjacent to t_p. Consider the backbone $B' = (t_1, \ldots, t_p, v)$ of T, and let $B'' \in \mathcal{B}$ be a backbone in $\mathcal{B}$ equivalent to B'. Since B'' is equivalent to B', there exists a neighbor w of t_p such that $B'' = (t_1, \ldots, t_p, w)$. By the maximality of B, it follows that B'' is not φ-increasing, and consequently $\varphi(w) = u_s$ where $s < k$. Consider the mapping $\varphi' : V(T) \to V(G)$ obtained from φ by switching the images of v and w. Formally, for every $t \in V(T)$, set

$$\varphi'(t) = \begin{cases} \varphi(t) & \text{if } t \notin \{v, w\}\,; \\ \varphi(w) & \text{if } t = v\,; \\ \varphi(v) & \text{if } t = w\,. \end{cases}$$

Then φ' is an ISI mapping of T into G. However, since $\varphi'(w) = \varphi(v) = u_r$ and $r > k$, backbone B'' is a φ'-increasing backbone strictly longer than B, contradicting the definition of the pair (φ, B). This shows that if T is isomorphic to an induced subgraph of G, then there exists a backbone $B \in \mathcal{B}$ and an integer k with $1 < k \leq n$ such that $f_B(|B|, k-1, k) = 1$. □

In what follows, we show that for any backbone $B \in \mathcal{B}$, the values of the function f_B can be computed recursively. Definition 1 implies that for every $j \in \{1, \ldots, n\}$, there exists an index $\ell(j) \leq j$ such that $N_{G_j}[u_j] = \{u_{\ell(j)}, u_{\ell(j)+1}, \ldots, u_j\}$, where G_j denotes the subgraph of G induced by

$\{u_1,\ldots,u_j\}$. Also recall that for any given backbone $B=(t_1,\ldots,t_p)\in\mathcal{B}$ and any $i\in\{1,\ldots,p\}$, we write L_i to denote the set of neighbors of t_i outside B.

First, we consider the simplest case, namely computing $f_B(i,j,k)$ when $i=1$.

Lemma 3. *Let $B\in\mathcal{B}$ be a representative backbone of T, and let $(1,j,k)\in\mathcal{T}_B$. Then $f_B(1,j,k)=1$ if and only if $\alpha(G[\{u_{\ell(k)},\ldots,u_j\}])\geq|L_1|$.*

Proof. The definition of f_B implies that $f_B(1,j,k)=1$ if and only if there exists an ISI mapping φ of B_1 into G_j^k such that $\varphi(t_1)=u_k$. Notice that B_1 is isomorphic to a star with $|L_1|$ leaves. Hence, there exists an ISI mapping φ of B_1 into G_j^k such that $\varphi(t_1)=u_k$ if and only if u_k has at least $|L_1|$ pairwise non-adjacent neighbors in the graph G_j^k. But this last condition is equivalent to the condition that the independence number of the subgraph of G induced by $\{u_{\ell(k)},\ldots,u_j\}$ is at
least $|L_1|$. □

Now, let us consider the problem of computing the value of $f_B(i,j,k)$ for some $(i,j,k)\in\mathcal{T}_B$ with $i>1$, assuming that we have already computed the values of $f_B(i',j',k')$ for all $(i',j',k')\in\mathcal{T}_B$ with $i'<i$. Lemma 4 states a necessary condition for $f_B(i,j,k)=1$.

Lemma 4. *Let $B\in\mathcal{B}$ be a representative backbone of T, and let (i,j,k) be a triple in $\mathcal{T}_B$ with $i\geq 2$ such that $f_B(i,j,k)=1$. Then there exists an integer $k'\in\{\ell(k),\ldots,j\}$ such that*

$$f_B(i-1,\ell(k)-1,k')=1 \quad \text{and} \quad \alpha\Big(G\big[\{u_{k'+1},\ldots,u_j\}\setminus N_G(u_{k'})\big]\Big)\geq|L_i|\,.$$

Proof. Suppose that the conditions in the lemma are satisfied. By the definition of f_B, there exists an ISI mapping φ of B_i into G_j^k such that $\varphi(t_i)=u_k$. Let $k'\in\{1,\ldots,n\}$ be the index satisfying $\varphi(t_{i-1})=u_{k'}$. Let us verify that k' has all the desired properties. First of all, it holds that $k'\leq j$, since $k'\neq k$ by the injectivity of φ and since φ maps $V(B_i)$ to $V(G_j^k)=\{u_1,\ldots,u_j,u_k\}$. Second, since φ preserves adjacencies and $t_{i-1}t_i\in E(T)$, we have $u_{k'}u_k=\varphi(t_{i-1})\varphi(t_i)\in E(G)$. Consequently, $u_{k'}\in N_{G_k}[u_k]$ and hence $k'\geq\ell(k)$.

Now, let us show that $f_B(i-1,\ell(k)-1,k')=1$. This is equivalent to showing the existence of an ISI mapping φ' of B_{i-1} into $G_{\ell(k)-1}^{k'}$ such that $\varphi'(t_{i-1})=u_{k'}$. Let φ' be the restriction of φ to $V(B_{i-1})$. We will verify that φ' is an ISI mapping of B_{i-1} into $G_{\ell(k)-1}^{k'}$ such that $\varphi'(t_{i-1})=u_{k'}$. Since φ is an injective mapping that preserves adjacencies and non-adjacencies, so is φ'. The condition $\varphi'(t_{i-1})=u_{k'}$ is also satisfied, by the definition of k'. It remains to verify that for every $w\in V(B_{i-1})$, we have $\varphi'(w)\in V(G_{\ell(k)-1}^{k'})$, or, equivalently, that $\varphi(w)\in\{u_1,\ldots,u_{\ell(k)-1},u_{k'}\}$. For $w=t_{i-1}$, this is clear, and we only need to check that $\varphi(w)\in\{u_1,\ldots,u_{\ell(k)-1}\}$ for all $w\in V(B_{i-1})\setminus\{t_{i-1}\}$. Suppose for a contradiction that there exists a vertex $w\in V(B_{i-1})\setminus\{t_{i-1}\}$ with $\varphi(w)=u_r$ for some $r\in\{\ell(k),\ldots,j\}$. Then $u_ru_k\in E(G)$, and since φ preserves non-adjacencies, this implies $wt_i\in E(T)$. This contradiction to the fact that the only neighbor of t_i in B_{i-1} is t_{i-1} implies that $f_B(i-1, \ell(k)-1, k') = 1$, as claimed.

It remains to show that the independence number of the graph $G' = G\big[\{u_{k'+1}, \ldots, u_j\} \setminus N_G(u_{k'})\big]$ satisfies $\alpha(G') \geq |L_i|$. The vertices of L_i form an independent set of size $|L_i|$ in B_i, and since φ is an injective mapping preserving non-adjacencies, its image $\varphi(L_i)$ is an independent set of size $|L_i|$ in G^k_j. Since φ preserves non-adjacencies and $L_i \cap N_{B_i}(t_{i-1}) = \emptyset$, we have $\varphi(L_i) \cap N_G(u_{k'}) = \emptyset$. Hence, it is enough to show that $\varphi(L_i) \subseteq \{u_{k'+1}, \ldots, u_j\}$. Clearly, $\varphi(L_i) \subseteq \{u_1, \ldots, u_j\}$, so the only way the condition $\varphi(L_i) \subseteq \{u_{k'+1}, \ldots, u_j\}$ could fail is if there exists a vertex $w \in L_i$ such that $\varphi(w) = u_{i_w}$ for some integer $i_w \leq k'$. Since φ maps t_{i-1} to $u_{k'}$ and $w \neq t_{i-1}$, we have $i_w < k'$. Moreover, since φ preserves adjacencies and $wt_i \in E(B_i)$, we have $u_{i_w} u_k \in E(G)$. By Observation 1, vertex t_{i-1} has a neighbor, say z, in B_{i-1}. Clearly $z \neq t_i$; moreover, $u_{i_z} u_{k'} \in E(G)$, where $u_{i_z} = \varphi(z)$. Furthermore, Definition 1 implies that $i_z < k'$. Since φ preserves non-adjacencies and $wt_{i-1} \notin E(B_i)$, we have $u_{i_w} u_{k'} \notin E(G)$. Similarly, since $zt_i \notin E(B_i)$, we have $u_{i_z} u_k \notin E(G)$. If $i_z < i_w$ then $i_z < i_w < k'$, and we have a contradiction to Definition 1 for the triple $(i, j, k) = (i_z, i_w, k')$. Hence, $i_w < i_z$, and consequently $i_w < i_z < k$, and we have a contradiction to Definition 1 for the triple $(i, j, k) = (i_w, i_z, k)$. This shows that $\varphi(L_i) \subseteq \{u_{k'+1}, \ldots, u_j\}$. We conclude that $\varphi(L_i)$ is an independent set in the graph G', implying $\alpha(G') \geq |L_i|$, as claimed. □

We now show that the necessary condition in Lemma 4 is also a sufficient condition for $f_B(i, j, k) = 1$.

Lemma 5. *Let $B \in \mathcal{B}$ be a representative backbone of T, and let (i, j, k) be a triple in $\mathcal{T}_B$ with $i \geq 2$. If there exists an integer $k' \in \{\ell(k), \ldots, j\}$ such that*

$$f_B(i-1, \ell(k)-1, k') = 1 \quad \text{and} \quad \alpha\Big(G\big[\{u_{k'+1}, \ldots, u_j\} \setminus N_G(u_{k'})\big]\Big) \geq |L_i|\,,$$

then $f_B(i, j, k) = 1$.

Proof. Suppose that the conditions in the lemma are satisfied. By the definition of f_B, there exists an ISI mapping φ of B_{i-1} into $G^{k'}_{\ell(k)-1}$ such that $\varphi(t_{i-1}) = u_{k'}$. We need to show that there exists an ISI mapping φ' of B_i into G^k_j such that $\varphi'(t_i) = u_k$. Let I be an independent set of size $|L_i|$ in the graph $G' = G\big[\{u_{k'+1}, \ldots, u_j\} \setminus N_G(u_{k'})\big]$. We fix a bijection $\psi : L_i \to I$, and we define a mapping $\varphi' : V(B_i) \to V(G)$ as follows: for every $v \in V(B_i)$, we have

$$\varphi'(v) = \begin{cases} \varphi(v) & \text{if } v \in V(B_{i-1})\,; \\ u_k & \text{if } v = t_i\,; \\ \psi(v) & \text{if } v \in L_i\,. \end{cases}$$

Notice that since the vertex set of B_i is the disjoint union of sets $V(B_{i-1})$, $\{t_i\}$ and L_i, the mapping φ' is well-defined. In order to complete the proof, we will verify that φ' is an ISI mapping of B_i into G^k_j such that $\varphi'(t_i) = u_k$. In what follows, we will use the fact that $G^{k'}_{\ell(k)-1}$ is an induced subgraph of G^k_j.

(i) Since

$$\begin{aligned}\varphi'(V(B_i)) &= \varphi(V(B_{i-1})) \cup \{u_k\} \cup \psi(L_i)\\ &\subseteq V(G^{k'}_{\ell(k)-1}) \cup \{u_k\} \cup I\\ &\subseteq \{u_1,\dots,u_{k'}\} \cup \{u_k\} \cup \{u_{k'+1},\dots,u_j\}\\ &= V(G^k_j)\,,\end{aligned}$$

mapping φ' is indeed a mapping from $V(B_i)$ to $V(G^k_j)$.

(ii) Condition $\varphi'(t_i) = u_k$ is satisfied by definition.

(iii) The injectivity of φ' follows immediately from the injectivity of φ and the bijectivity of ψ.

(iv) *φ' preserves adjacencies:*
Let $uv \in E(B_i)$. If $u, v \in V(B_{i-1})$, then

$$\varphi'(u)\varphi'(v) = \varphi(u)\varphi(v) \in E(G^{k'}_{\ell(k)-1}) \subseteq E(G^k_j)\,,$$

where the fact that $\varphi(u)\varphi(v)$ is an edge of $G^{k'}_{\ell(k)-1}$ holds since φ preserves adjacencies. If $u = t_{i-1}$ and $v = t_i$ then $\varphi'(u)\varphi'(v) = u_{k'}u_k$, which is an edge of G^k_j since $\ell(k) \le k' \le j$. Finally, if $u = t_i$ and $v \in L_i$, then $\varphi'(u)\varphi'(v) = u_k\psi(v)$, which is an edge of G^k_j, since $\psi(v) = u_r$ for some $r \in \{k'+1,\dots,j\} \subseteq \{\ell(k)+1,\dots,j\}$, implying $u_r \in N_{G^k_j}(u_k)$.

(v) *φ' preserves non-adjacencies:*
Let u, v be a pair of distinct non-adjacent vertices of B_i.
If $u, v \in V(B_{i-1})$, then, since φ preserves non-adjacencies, φ' maps $\{u, v\}$ to a pair of non-adjacent vertices in $G^{k'}_{\ell(k)-1}$, and hence in G^k_j.

Suppose that $u \in V(B_{i-1})$ and $v = t_i$. Then $u \neq t_{i-1}$ and hence $\varphi'(u) \in \{u_1,\dots,u_{\ell(k)-1}\}$. Consequently, by the definition of $\ell(k)$, vertex $\varphi(u)$ is not adjacent to $u_k = \varphi(v)$ in G^k_j.

Suppose that $u = t_{i-1}$ and $v \in L_i$. Then $\varphi'(u) = u_{k'}$, and $\varphi'(v)$ is not adjacent to $\varphi'(u) = u_{k'}$ since $\varphi'(v) \in I \subseteq \{u_{k'+1},\dots,u_j\} \setminus N_G(u_{k'})$.

Finally, suppose that $u \in V(B_{i-1}) \setminus \{t_{i-1}\}$ and $v \in L_i$. Then $\varphi'(u) = u_r$ for some $r \in \{1,\dots,\ell(k)-1\}$, and $\varphi'(v) = u_s$ for some $s \in \{k'+1,\dots,j\}$. Suppose for a contradiction that u_r and u_s are adjacent in G^k_j. Since G^k_j is an induced subgraph of G, u_r and u_s are adjacent in G. On the other hand, the definition of I implies that $u_{k'}$ and u_s are non-adjacent in G. However, since $r < k' < s$, this contradicts Definition 1 applied to the triple $(i, j, k) = (r, k', s)$.

The above properties imply that φ' is an ISI mapping of B_i into G^k_j such that $\varphi'(t_i) = u_k$. Consequently $f_B(i, j, k) = 1$, completing the proof of the lemma. □

The results of Lemmas 3–5 can be summarized as follows.

Corollary 1. *For any representative backbone $B = (t_1,\dots,t_p) \in \mathcal{B}$ of T, the values of the function $f_B : \mathcal{T}_B \to \{0, 1\}$ can be computed recursively as follows:*

– *for $i = 1$ and all $1 \leq j < k \leq n$, we have $f_B(1, j, k) = 1$ if and only if*

$$\alpha(G[\{u_{\ell(k)}, \ldots, u_j\}]) \geq |L_1|\,;$$

– *for all $i \in \{2, \ldots, p\}$ and all $1 \leq j < k \leq n$, we have $f_B(i, j, k) = 1$ if and only if there exists an integer $k' \in \{\ell(k), \ldots, j\}$ such that*

$$f_B(i-1, \ell(k)-1, k') = 1 \quad \text{and} \quad \alpha\Big(G\big[\{u_{k'+1}, \ldots, u_j\} \setminus N_G(u_{k'})\big]\Big) \geq |L_i|\,.$$

We are now ready to prove the main result of this paper.

Theorem 1. Induced Subtree Isomorphism *can be solved in polynomial time on interval graphs.*

Proof. Let (G, T) be an instance of Induced Subtree Isomorphism, where $G = (V, E)$ is an interval graph and T is a tree. We assume that T has at least three vertices, as the problem can trivially be solved otherwise. We also assume that $|V(T)| \leq |V(G)|$, as otherwise we have a trivial no-instance. Finally, we assume that G is connected; if G is disconnected, then the polynomial-time algorithm described below can be applied to each of the connected components of G within the same overall time bound.

Let $t = |V(T)|$, $n = |V(G)|$ and $m = |E(G)|$. We start by checking whether T is a caterpillar, which can easily be done in time linear in the size of T. As mentioned in Section 2, every induced subtree of an interval graph is a caterpillar due to a characterization of interval graphs by Lekkerkerker and Boland [20]. Hence, if T is not a caterpillar, then we output "no". Suppose T is a caterpillar. Then we compute a set $\mathcal{B} = \{B_{\min}, B_{\min}^{-1}, B^+, B^-\}$ of at most four representative backbones of T in the way described just below Lemma 1. It is clear that such a set $\mathcal{B}$ can be computed in time $O(t)$. We also compute an interval order $\sigma = (u_1, u_2, \ldots, u_n)$ of G, which can be done in $O(n+m)$ time [27]. Using this interval order σ, we then compute, for all $i \in \{1, \ldots, n\}$, the indices $\ell(i)$ that were defined just above Lemma 3; this takes $O(n+m)$ time in total.

Lemma 2 and Corollary 1 imply that we can determine whether or not T is isomorphic to an induced subgraph of G by computing, for each backbone $B \in \mathcal{B}$, the value of $f_B(|B|, k-1, k)$ for every $k \in \{2, \ldots, n\}$. We will now describe how this can be done in polynomial time for a fixed backbone $B \in \mathcal{B}$. Since $\mathcal{B}$ contains at most four backbones, this suffices to complete the proof of Theorem 1.

Let $B = (t_1, \ldots, t_p) \in \mathcal{B}$ be a representative backbone of T. For every $i \in \{1, \ldots, p\}$, we compute the number $|L_i| = |N_T(t_i) \setminus V(B)|$. Then, for every pair (j, k) with $1 \leq j < k \leq n$, we compute the independence number of the graph $G\big[\{u_{\ell(k)}, \ldots, u_j\}\big]$. Since $G\big[\{u_{\ell(k)}, \ldots, u_j\}\big]$ is an induced subgraph of G, and thus an interval graph, its independence number can be computed in time $O(n+m)$ [10]. Hence, computing $\alpha\big(G\big[\{u_{\ell(k)}, \ldots, u_j\}\big]\big)$ for all pairs (j, k) takes $O(n^2(n+m))$ time in total. We also compute the independence number of the graph $G\big[\{u_{k'+1}, \ldots, u_j\} \setminus N_G(u_{k'})\big]$ for every pair (k', j) with $1 \leq k' \leq j \leq n$, which can be done in $O(n^2(n+m))$ time in total for similar reasons.

Having precomputed these independence numbers and values of $|L_i|$, we can now use the recursions from Corollary 1 to compute the value of $f_B(i,j,k)$ for every $(i,j,k) \in \mathcal{T}_B$, in increasing order of $i \in \{1,\ldots,|B|\}$, in time $O(tn^3)$ in total: each of the values $f_B(i,j,k)$ can be computed in constant time for $i=1$ and in time $O(n)$ for $i \geq 2$ from the already computed values. The overall time complexity of the algorithm is $O(n^2(n+m)) + O(tn^3) = O(n^2(tn+m))$.

The algorithm can be easily extended so that it also produces an ISI mapping of T into G, in case such a mapping exists. We just need to store, for each $B \in \mathcal{B}$ and each $(i,j,k) \in \mathcal{T}_B$ such that $f_B(i,j,k) = 1$, an ISI mapping φ of B_i into G_j^k such that $\varphi(t_i) = u_k$. The proofs of Lemmas 3 and 5 show that such mappings can efficiently be computed in a recursive way. □

4 Subtree Isomorphism on Interval Graphs

To complement our positive result on interval graphs in the previous section, we show in this section that SUBTREE ISOMORPHISM is NP-complete on interval graphs. In fact, we prove that SUBTREE ISOMORPHISM is NP-complete already on proper interval graphs, a well-known subclass of interval graphs.

We first need to introduce some additional terminology. Let $G = (V,E)$ be a graph. The *width* of an ordering $(u_1,\ldots,u_n)$ of V is $\max\{|i-j| : u_iu_j \in E\}$. The *bandwidth* of G is the minimum width of any ordering of the vertices of G. The BANDWIDTH problem takes as input a graph G and an integer k, and the task is to decide whether the bandwidth of G is at most k. An ordering $(u_1,\ldots,u_n)$ of V is a *proper interval order* of G if, for every triple (i,j,k) with $1 \leq i < j < k \leq n$, it holds that $u_iu_k \in E$ implies $u_iu_j \in E$ and $u_ju_k \in E$. A graph is a proper interval graph if and only if it has a proper interval order [21].

It follows from the definition of a proper interval order that for any proper interval graph G, the width of a proper interval order of G is exactly the bandwidth of G. Since a proper interval order of a proper interval graph can be computed in linear time [21], BANDWIDTH is solvable in linear time on proper interval graphs. However, BANDWIDTH is NP-complete on trees [25], and it is from this problem that we reduce in the proof of the following result.

Theorem 2. SUBTREE ISOMORHISM *is NP-complete on proper interval graphs.*

Proof. We give a reduction from BANDWIDTH on trees. Let T be a tree on n vertices which, together with an integer k, constitutes an instance of BANDWIDTH. We construct a proper interval graph G as follows: start from a simple path $(v_1,v_2,\ldots,v_n)$, and add edges so that v_i is adjacent to v_j if and only if $j-i \leq k$, for all $1 \leq i < j \leq n$. Such a graph is called a *k-path power* on n vertices, and is well-known to be a proper interval graph. We show that T is a subgraph of G if and only if T has bandwidth at most k.

If T is a subgraph of G, then clearly the bandwidth of T is at most k, since $|j-i| \leq k$ for every edge v_iv_j in G. If the bandwidth of T is at most k, then we can take an ordering of the vertices of T of width at most k, and add edges to make it a k-path power on n vertices. Since no original edge of T has endpoints that are more than k apart in the ordering, it is indeed possible to obtain a k-path power G in this way, which means that T is a subgraph of G. The proof is completed by observing that the problem is trivially in NP. □

Observe that we in fact proved a stronger result than the statement of Theorem 2. A tree is a *spanning subtree* of a graph G if it is a subtree of G and it has the same number of vertices as G. The above proof shows that Spanning Subtree Isomorphism is NP-complete on path powers, which form a subclass of proper interval graphs.

5 Concluding Remarks

As a consequence of our results in this paper and previously known results, the following boundaries are now established on subgraph problems on interval graphs. The Induced Subgraph Isomorphism problem is NP-complete even if both input graphs are connected interval graphs [2, 7], whereas it becomes polynomial-time solvable if G is an interval graph and H is a tree. The Subgraph Isomorphism problem is NP-complete even if G is a proper interval graph and H is a tree, but it becomes polynomial-time solvable if G is an interval graph and H is a path.

The problem of deciding, given a graph G and an integer k, whether there exists a (not necessarily induced) path of length k in G, is NP-complete on bipartite graphs [19]. An easy reduction from this problem, using the observation that a graph contains a path of length k if and only if its line graph contains an induced path of length $k-1$, shows that Induced Subtree Isomorphism is NP-complete on line graphs of bipartite graphs, a well-known subclass of perfect graphs. This contrasts our positive result on interval graphs in the following sense. Line graphs are claw-free, implying that every induced tree of a line graph is a path. The restricted nature of induced subtrees of interval graphs allowed us to obtain a polynomial-time algorithm for Induced Subtree Isomorphism on interval graphs. However, although line graphs have even more restricted induced subtrees than interval graphs, this does not imply tractability for Induced Subtree Isomorphism on line graphs.

We would also like to mention that Induced Subtree Isomorphism is trivially solvable in polynomial time on cographs and on split graphs, since every induced subtree of a cograph or a split graph is a caterpillar that has a backbone on at most two vertices. By similar arguments, the problem can also be solved in polynomial time on $3K_2$-free graphs, a superclass of split graphs.

We conclude with the following two questions regarding the complexity of Induced Subtree Isomorphism problem on two graph classes generalizing interval graphs:

– What is the computational complexity of Induced Subtree Isomorphism on chordal graphs, a superclass of both interval graphs and split graphs? Note that this problem is NP-complete on perfect graphs, a superclass of chordal graphs, due to the aforementioned NP-completeness results on bipartite graphs [9] and on line graphs of bipartite graphs. Also note that the easier problem of finding a longest induced path can be solved in polynomial time on chordal graphs, or more generally, in $O(n^k)$ time on k-chordal graphs, i.e., on graphs having no induced cycles on more than k vertices [15].

– What is the computational complexity of Induced Subtree Isomorphism on AT-free graphs? This is a superclass of interval graphs, which also generalizes the cocomparability graphs. AT-free graphs share many features with interval graphs that were used by our algorithm in Section 3: they have some kind of linear structure [6, 17], the only possible induced subtrees in an AT-free graph are caterpillars, and computing the independence number of an AT-free graph is a polynomially solvable task [4]. Also note that the problem of finding a longest induced path can be solved in polynomial time on AT-free graphs [15, 18].

References

1. Agarwal, R.K.: An investigation of the subgraph isomorphism problem. M.Sc. Thesis, Dept. of Computer Science, University of Toronto, TR 138180 (1980)
2. Belmonte, R., Heggernes, P., van 't Hof, P.: Edge contractions in subclasses of chordal graphs. Discrete Appl. Math. 160, 999–1010 (2012)
3. Brandstädt, A., Le, V.B., Spinrad, J.: Graph Classes: A Survey. SIAM, Philadelphia (1999)
4. Broersma, H., Kloks, T., Kratsch, D., Müller, H.: Independent sets in asteroidal triple-free graphs. SIAM J. Discrete Math. 12, 276–287 (1999)
5. Corneil, D.G., Lerchs, H., Stewart Burlingham, L.: Complement reducible graphs. Discrete App. Math. 3, 163–174 (1981)
6. Corneil, D.G., Olariu, S., Stewart, L.: Asteroidal triple-free graphs. SIAM J. Discrete Math. 10, 399–430 (1997)
7. Damaschke, P.: Induced subgraph isomorphism for cographs is NP-complete. In: Proceedings WG 1991. LNCS, vol. 484, pp. 72–78. Springer (1991)
8. Diestel, R.: Graph Theory. Electronic Edition (2005)
9. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-completeness. W.H. Freeman & Co. (1979)
10. 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, 180–187 (1972)
11. Golovach, P.A., Kamiński, M., Paulusma, D., Thilikos, D.M.: Containment relations in split graphs. Discrete Appl. Math. 160, 155–163 (2012)
12. Golumbic, M.C.: Algorithmic Graph Theory and Perfect Graphs, 2nd edn. Annals of Discrete Mathematics, vol. 57. North Holland (2004)
13. Heggernes, P.: van 't Hof, P., Meister, D., Villanger, Y.: Induced subgraph isomorphism on proper interval and bipartite permutation graphs (submitted)
14. Ioannidou, K., Mertzios, G.B., Nikolopoulos, S.D.: The longest path problem has a polynomial solution on interval graphs. Algorithmica 61, 320–341 (2011)
15. Ishizeki, T., Otachi, Y., Yamazaki, K.: An improved algorithm for the longest induced path problem on k-chordal graphs. Discrete Appl. Math. 156, 3057–3059 (2008)

16. Kijima, S., Otachi, Y., Saitoh, T., Uno, T.: Subgraph isomorphism in graph classes. Discrete Math. 312, 3164–3173 (2012)
17. Kratsch, D.: Domination and total domination on asteroidal triple-free graphs. Discrete Appl. Math. 99, 111–123 (2000)
18. Kratsch, D., Müller, H., Todinca, I.: Feedback vertex set and longest induced path on AT-free graphs. In: Bodlaender, H.L. (ed.) WG 2003. LNCS, vol. 2880, pp. 309–321. Springer, Heidelberg (2003)
19. Krishnamoorthy, M.S.: An NP-hard problem in bipartite graphs. ACM SIGACT News 7(1), 26 (1975)
20. Lekkerkerker, C.G., Boland, J.C.: Representation of a finite graph by a set of intervals on the real line. Fund. Math. 51, 45–64 (1962)
21. Looges, P.J., Olariu, S.: Optimal greedy algorithms for indifference graphs. Computers Math. Applic. 25, 15–25 (1993)
22. Matousek, J., Thomas, R.: On the complexity of finding iso- and other morphisms for partial k-trees. Discrete Math. 108, 343–364 (1992)
23. Matula, D.W.: An algorithm for subtree identification. SIAM Rev. 10, 273–274 (1968)
24. Mertzios, G.B., Corneil, D.G.: A simple polynomial algorithm for the longest path problem on cocomparability graphs. SIAM J. Discrete Math. 26, 940–963 (2012)
25. Monien, B.: The bandwidth minimization problem for caterpillars with hair length 3 is NP-complete. SIAM J. Alg. Discr. Meth. 7, 505–512 (1986)
26. Müller, H.: Hamiltonian circuits in chordal bipartite graphs. Discrete Math. 156, 291–298 (1996)
27. Olariu, S.: An optimal greedy heuristic to color interval graphs. Inform. Proc. Lett. 37, 21–25 (1991)

Protein Folding in 2D-Triangular Lattice Revisited

(Extended Abstract)

A.S.M. Shohidull Islam and M. Sohel Rahman

AℓEDA Group,
Department of CSE, BUET, Dhaka 1000, Bangladesh
sohansayed@gmail.com, msrahman@cse.buet.ac.bd

Abstract. In this paper, we present a novel approximation algorithm to solve the protein folding problem in the H-P model. Our algorithm is polynomial in terms of the length of the given H-P string. The expected approximation ratio of our algorithm is $1 - \dfrac{2\log n}{n-1}$ for $n \geq 6$, where n^2 is the total number of H in a given H-P string. The expected approximation ratio tends to 1 for large values of n. Hence our algorithm is expected to perform very well for larger H-P strings.

Keywords: Protein Folding, Approximation Ratio, Algorithms, H-P Model.

1 Introduction

A long standing problem in Molecular Biology and Biochemistry is to determine the three dimensional structure of a protein given only the sequence of amino acid residues that compose protein chains. This problem is known as the Holy Grail of Computational Molecular Biology, also termed as "cracking the second half of the genetic code". There exist a variety of models attempting to simplify the problem by abstracting only the "essential physical properties" of real proteins. In these models, the three-dimensional space is often represented by a lattice. Residues which are adjacent (i.e., covalently linked) in the primary sequence must be placed at adjacent points in the lattice.

In this paper, we consider the Hydrophobic-Polar Model, HP Model for short, introduced by Dill [2]. The HP model is based on the assumption that hydrophobicity is the dominant force in protein folding. This lattice model simplifies a protein's primary structure to a linear chain of beads. Each bead represents an amino acid, which can be one of two types: **H** (hydrophobic or nonpolar) or **P** (hydrophilic or polar). Conformations of proteins are embedded in either a two-dimensional or three-dimensional square/triangular/hexagonal lattice. A *conformation* of a protein is simply a self-avoiding walk along the lattice. The goal of the protein folding problem is to find a conformation of the protein sequence on the lattice such that the overall *energy* is minimized, for some reasonable definition of energy. Each amino acid in the chain is represented by

© Springer-Verlag Berlin Heidelberg 2013

occupying one lattice point, connected to its chain neighbour(s) on adjacent lattice points. An optimal embedding is one that maximizes the number of H-H contacts which are not adjacent in amino acid chain. So, in effect, an input to the problem is a finite string over the alphabet $(H, P)^+$. Often, in what follows, the input strings to our problem will be referred to as H-P strings. For a more biological background and motivations the readers are referred to [2,1]. A number of approximation algorithms have been developed for the HP model on the 2D square lattice, 3D cubic lattice, triangular lattice and the face-cantered-cubic (FCC) lattice [3,10,11]. Istrail and lam [7] have presented a survey which is composed of wide range of algorithms on different model of protein folding problem. The first approximation algorithm developed for this problem on the square lattice by Hart and Istrail has an approximation ratio of 1/4 [3]. The approximation ratio for this problem was improved to 1/3 by Newman [10]. The algorithm presented in [3] can be generalized to an approximation algorithm for the problem on the 3D cubic lattice. In [4] a general method for protein folding on the HP model called $masterapproximationalgorithm$ was presented by Hart and Istrail. This method can be applied to a large class of lattice models. Hart and Istrail [5] provided the first approximation algorithn for the problem of folding on the side-chain model which can be applied to 2D square, 3D cubic lattices, and FCC lattices. Their provided approximation ratio remains the best ratio for an approximation algorithm in any 3D HP-models to date. In [5] the authors also illustrate the transformation of approximation algorithm from lattice models to off-lattice models. Another approximation algorithm, based on different geometric ideas was presented in [11]. Heun [6] presented a linear-time approximation algorithm for protein folding in the HP side chain model on extended cubic lattice having approximate ration 0.84. In [8] the authors presented an approximation algorithm with approximation ratio 0.17 that folds an arbitrary protein sequence in the 2D hexagonal lattice HP-model.

2 Our Contribution

In this paper, we present an approximation algorithm for protein folding in 2D-triangular lattice. To the best of our knowledge the best approximation ratio for this problem was obtained by Agarwalla et al. [1], which is $\frac{6}{11}$. For our algorithm we do a probabilistic analysis and deduce that the expected approximation ratio of our algorithm is $1 - \frac{2 \log n}{n-1}$ for $n \geq 6$ where n^2 is the total number of H's in a given H-P string. Clearly our approximation ratio depends on n which in turn depends on the number of H's in the H-P string. For large values of n, this ratio tends to reach 1. So it can be expected that our algorithm would provide very close to optimal results for large values of n.

3 Roadmap

The rest of the paper is organized as follows. In Section 4, we define some notations and notions. Section 5 describes our approach to solve the problem.

In Section 6 we deduce the expected approximation ratio. We briefly conclude in Section 7.

4 Preliminaries

In this section, we present some notions and definitions (mostly in relation to the underlying lattice) that we need to explain our algorithm. In a triangular lattice, each lattice point has six neighbouring lattice points [1]. In the literature it is also called a hexagonal lattice. Note that, by definition, a lattice is infinite. However, in what follows, when we refer to a lattice we will refer to a finite part of it. This finite part of the lattice would essentially be a hexagon. We now define some notions related to a hexagon in the context of our approach. Note that a hexagon is said to be perfect (or regular) if it has six equal sides and six equal angles. Throughout the paper, when we refer to a hexagon we assume that the opposite sides of it are parallel having the same length. Also, when we consider a non-regular hexagon we assume that the sides of it can be grouped into two groups based on their length. In particular, two of its sides (that are parallel to each other) have a particular length, say, p and the other four sides have a different length, say m. Clearly, when $p = m$, we have a regular hexagon. Following the above discussion, it would be useful to define the former couple of sides of the (non-regular) hexagon (i.e., that having a length of p each) as $\mathcal{D}$-sides and the latter four sides (i.e., that having a length of m each) as $\mathcal{Q}$-sides.

The discussion that follows can be better understood with the help of Figure 1. As has been mentioned above the finite portion of the lattice of our interest can be seen as a hexagon, the *boundary* of which consists of those lattice points that have fewer than six neighbours within the hexagon. An *edge* is formed by

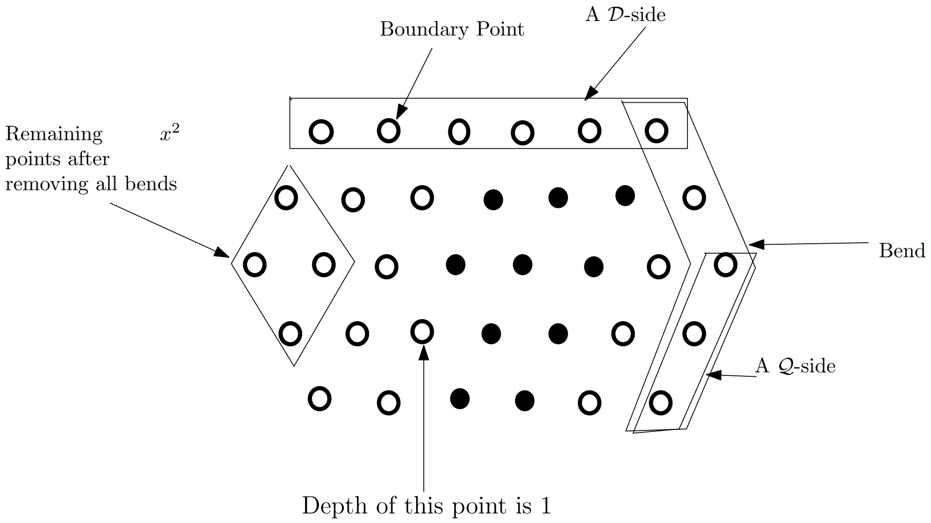


Fig. 1. Lattice

two neighbouring lattice points. If the lattice points are filled by H, the two neighbouring H's also form an edge. If two H's are non-adjacent in an H-P string and placed on neighbouring lattice points to form an edge, they form a *bond*. The points on the boundary are referred to as the boundary points. The *depth* of a point in a lattice is the minimum number of points it needs to traverse to reach any boundary point. Naturally, the depth of a boundary point is 0. The depth of a hexagon is the maximum depth of all points in the hexagon. In Figure 1, the depth of the hexagon is 2.

The *length* of the hexagon (or lattice) is the total number of points along the $\mathcal{D}$-sides. In Figure 1, the length of the hexagon is 6. A *region* in the hexagon is a group of the lattice points such that each point in it has at least two neighbours from within it. Similar to the boundary of a hexagon we also define the boundary of a region. The *boundary of a region* consists of those lattice points that have fewer than six neighbours within the region. A region must not contain any point such that deleting that point creates two separate regions. In graph-theoretic terms, the region cannot have a *cut vertex*. Also all the lattice points inside the boundary of a region are parts of the region. So, by definition, only the boundary itself cannot be considered as a region unless there are no points inside the boundary at all. In Figure 1, the black vertices comprise a region (which has only one point inside the boundary). The size of a region is the total number of lattice points inside it including the boundary points.

We also introduce a notion of a *bend* for a hexagon if its length is greater than its depth. A bend refers to the combined bent line along the 2 $\mathcal{Q}$-sides to the right. A bend could be defined identically considering the two $\mathcal{Q}$-sides to the left as well. However, for our purpose, we exclude that option from our definition. A bend is illustrated in Figure 1. Notice that if the depth of such a hexagon is x, then a bend contains $2x+1$ points. There are a total of ℓ bends in a hexagon, where ℓ is the length of the hexagon. Removing all bends from the hexagon leaves a total of x^2 lattice points (see Figure 1).

We use the usual notion of a *run* in an H-P string. In particular, a run in an H-P string denotes the consecutive H's or P's. For example, in the H-P string $HHHPPHHPHHHH$, we have a run of 3 H's, followed by a run of 2 P's and so on. Here the *run-length* of the first run of H (P) is 3 (2). We will sometimes use the term H-run (P-run) to indicate a run of H's (P's). The longest H-run (P-run) of a string denotes the run of H (P) which has the highest run-length among all the H-runs (P-runs) of the string. For the sake of conciseness, the H-P strings shall often be represented as H and P's with the corresponding run-lengths as their powers. So, the H-P string $\mathcal{S} = HHHPPHHPHHHH$ will often be conveniently represented by $\hat{\mathcal{S}} = H^3P^2H^2P^1H^4$. Further, we will use $S(i), 1 \leq i \leq |S|$ to denote the ith character of the H-P string $\mathcal{S}$. Similarly, $\hat{\mathcal{S}}(j)$ denotes the jth run of $\hat{\mathcal{S}}$. For example, $\hat{\mathcal{S}}(1)$ refers to H^3, $\hat{\mathcal{S}}(2)$ refers to P^2 and so on. We will use $SumH$ as the sum of the run-lengths of all the H-runs of a given string $\hat{\mathcal{S}}$. We end this section with a formal definition of the problem we handle in this paper.

Problem 1. Given an H-P string $\hat{\mathcal{S}}$, the problem is to place the H-P string on a triangular lattice such that the total number of bonds are maximized.

5 Our Approach

Our approach is a simple and intuitive one. Our idea is to identify the length and depth of a suitable hexagon and then try to put all the H's of a particular H-run inside the hexagon and put the P's of the following P-run (if any) outside that hexagon. The length and depth of the hexagon depend on SumH. The motivation here is to get the maximum number of bonds between H's. Note carefully that if we can fully fill a hexagon with n lattice points and get a total of k edges, the number of total bonds will be $k - n$. This is evident from Figure 2(h), where we have a hexagon of 37 points. Here after filling the hexagon fully we get a total of 90 edges. It is easy to verify that the total number of bonds are 53. We continue this process for every H-run and P-run of the string. We illustrate the approach with an example below. In the figures throughout this paper the bonds and edges are not shown explicitly. A connection between 2 lattice points indicate the presence of 2 H's that are adjacent in the input H-P string.

Example 1. Suppose we have an H-P string as follows:

$$\hat{\mathcal{S}} = H^6P^5H^2P^6H^4P^5H^6P^3H^2P^5H^4PH^7P^6H^2P^2H^4.$$

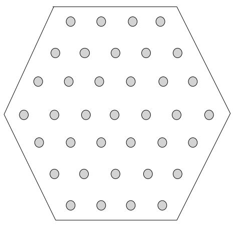
Figure 2(a) gives us a suitable regular hexagon for $\hat{\mathcal{S}}$ on the underlying lattice. Our approach starts with the longest H-run of $\hat{\mathcal{S}}$. In Figure 2(b) the longest H-run, i.e, $\hat{\mathcal{S}}(13) = H^7$, is first positioned within the hexagon. Then, in Figure 2(c), the subsequent P-run is positioned outside the hexagon. Similarly the approach continues through Figures 2(d) to 2(f) where we illustrate the positioning of H-runs and P-runs upto $\hat{\mathcal{S}}(17)$. Then we wrap around and starts with $\hat{\mathcal{S}}(1)$ in Figure 2(g). The final position of all the runs of $\hat{\mathcal{S}}$ is shown in Figure 2(h).

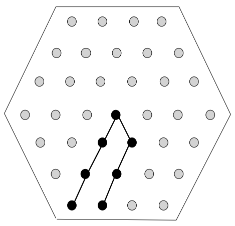
Notably, if two hexagons have the same number of lattice points and are filled up fully with H by a given H-P string, the hexagon with higher number of total edges have the higher number of total bonds as the difference between the total number of the edges and that of bonds is a constant (i.e., the total number of lattice points).

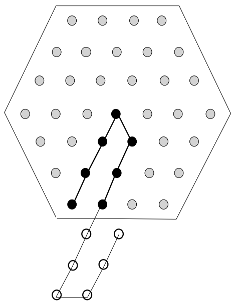
Now that we have discussed our main approach to fill up the hexagon, we can shift our focus to the question of whether we can accommodate all the H-runs of the input H-P string within the current hexagon. Recall that our goal is to increase the number of edges as much as possible. We have the following useful lemmas. The proofs of these lemmas will be given in the fuller version.

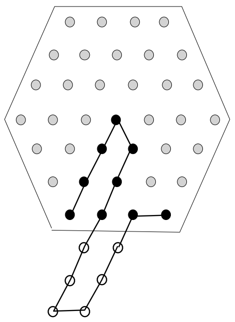
Lemma 1. *If two hexagons have the same number of lattice points then the hexagon with the greater depth will not have fewer edges.*

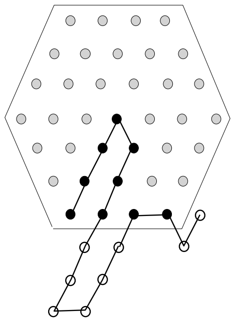
Lemma 2. *Suppose we have a regular hexagon H_1 containing N points. Now we reduce the length of this hexagon to get another hexagon H_2 such that H_2 contains N points as well. Then H_2 will have fewer edges inside it than that of H_1.*

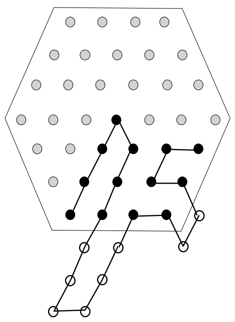

(a) Lattice points, i.e., Hexagon

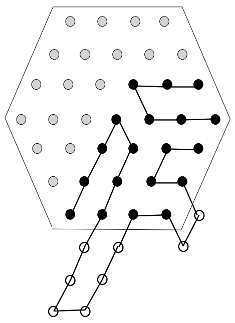

(b) Positioning $\hat{\mathcal{S}}(13) = H^7$


(c) Positioning $\hat{\mathcal{S}}(14) = P^6$


(d) Positioning $\hat{\mathcal{S}}(15) = H^2$


(e) Positioning $\hat{\mathcal{S}}(16) = P^2$


(f) Positioning $\hat{\mathcal{S}}(17) = H^4$


(g) Positioning $\hat{\mathcal{S}}(1) = H^6$

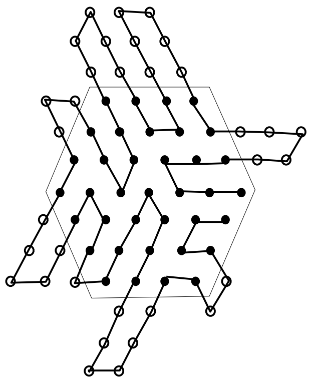

(h) Final state

Fig. 2. Construction

Lemma 3. *Assuming that we can fill up all the points of the hexagon, the number of edges (as well as the total number of bonds) will be maximum if, and only if, the hexagon is a regular hexagon.*

Proof. Lemma 3 follows readily from Lemmas 1 and 2. □

As has been mentioned before, our algorithm proceeds in an iterative fashion in order to achieve the highest possible number of edges by iteratively changing the length and depth of the hexagon. We start with an appropriate regular hexagon. Note carefully that, by Lemma 3, if we can fill the points of a regular hexagon, we get the optimum number of edges. If we fail to fill up all the points of a regular hexagon, we reduce the depth of the hexagon and increase its length with the hope that the number of edges will increase in the new hexagon. We continue the iteration (i.e., reducing the depth of the hexagon and filling it up) until we reach a case when the total number of edges decreases than that of the previous iteration. In that case, we terminate our algorithm and return the result of the previous iteration.

Notably, to fill up a regular hexagon with depth n at least one H-run having length $2n+1$ is needed. Besides, we need at least two H-runs of length $2n$, three of length $2n-2$, three of length $2n-4$... three H-runs of length 1 or 2 (depending on the size of n) in the input. The string in Example 1 presented before meets this criteria assuming $n=3$. To explain a bit more, note that, in the H-P string of Example 1 we have one H-run with run-length 7, two H-runs with run-length 6, three H-runs with run length 4 and the rest of the H-runs have run-length 2. Another example is given below where we cannot put all H's in a regular hexagon.

Example 2. Consider an H-P string

$$\hat{\mathcal{S}} = H^6P^3H^4P^4H^3P^6H^4P^3H^2P^4H^2P^3H^6H^5P^2H^2PH^3.$$

For this string, the length of a suitable regular hexagon is 4 (i.e., depth is 3) as SumH is 37. But in Figure 3 we can see that we cannot properly fill the hexagon. In such a case we have to increase the length of the hexagon to 6 as well as decreasing its depth to 2. The new hexagon is shown in Figure 4. As is evident from Figure 4, we can now fill the hexagon properly with $\hat{\mathcal{S}}$.

Now we formally present the steps of our algorithm below.

Step 1 Let SumH of the input H-P string $\hat{\mathcal{S}}$ is z and the longest H-run is $\hat{\mathcal{S}}(i)$ having run-length k. Compute $n = \frac{1+\sqrt{1+\frac{4\times(z-1)}{3}}}{2}$. Set $globalB = 0$.

Step 2 If $\frac{k-1}{2} \geq n$, then construct a regular hexagon with n lattice points at each side. Otherwise set, $n = \frac{k-1}{2}$ and $\ell = \frac{z-n^2}{2n+1}$ and construct a non-regular hexagon with length ℓ and depth n.

Step 3 For each of the H-runs and P-runs, starting from $\hat{\mathcal{S}}(i)$ and wrapping around the end (if applicable) execute the following two steps. For an H-run, execute Step a and Step b; for a P-run execute Step c.

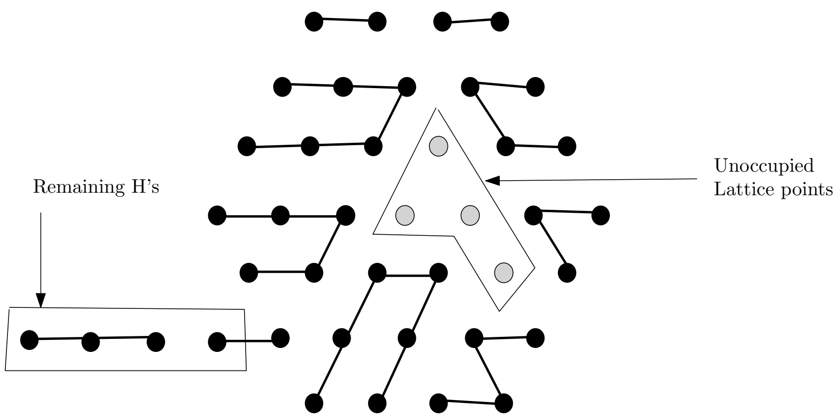


Fig. 3. $\hat{\mathcal{S}}$ of Example 2 cannot properly fill the hexagon

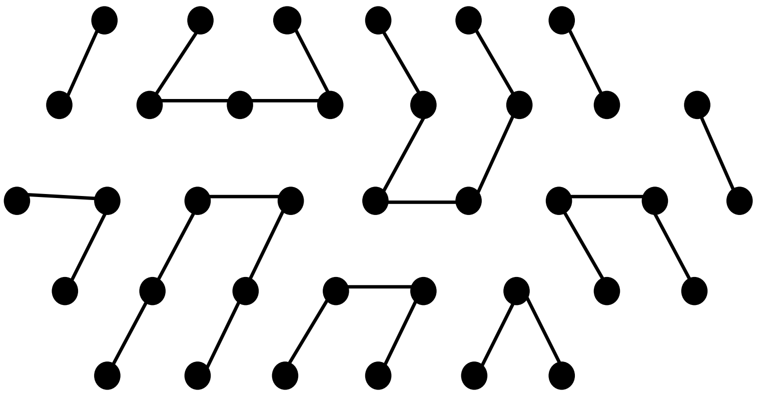


Fig. 4. $\hat{\mathcal{S}}$ of Example 2 can fill the adjusted hexagon

Step a [for H-runs] If the run length of the H-run is less than 3 then we take lattice points on the boundary of the hexagon. Otherwise, we try to find a region from the remaining unoccupied points. Here the total number of points in the region must be equal to the run-length of the current H-run and at least two of these points must be boundary points of the hexagon. We find the region executing the following steps (Steps i to iv). We ensure that the region property is maintained as we proceed by including the points one after another.

Step i Take two points on the boundary of the hexagon. These are the first two points of the region.

Step ii Identify the unoccupied points in the hexagon such that each of those has two neighbouring lattice points in the region. Find the point having the highest depth among these points and add this point to the region. Thus we increase the size of the region (by one).

```
        Step iii If no such point is found then go to Step 6.
        Step iv  If the region size is less than the run length of the current H-run, go
                 to Step ii.
Step b [for H-runs]Fill up the identified region's lattice points with the H-run.
Step c [for P-runs]Put the P-run outside the hexagon in two rows. The first P
       of the run will be a neighbour of the previous H-run's last H, while the
       last P of the run will be a neighbour of the next H-run's first H.
Step 4 Count the total number of bonds $B$.
Step 5 If $globalB > B$, return $globalB$.
Step 6 Otherwise set $globalB = B$ and $n = n - 1$
Step 7 If $n = 2$, return $B$; otherwise compute $\ell = \frac{z-n^2}{2n+1}$. Construct a hexagon
       with length $\ell$ and depth $n$ and go to Step 3.
```

Now we present and prove the following theorem which basically proves the correctness of our approach.

Theorem 1. *Given a region consisting of lattice points, a starting and an ending point such that those are boundary points of the hexagon, there always exists a path that starts at the starting point, ends at the ending point visiting each point in the region exactly once.*

Proof. We can traverse the points row wise from left to right within the region starting from, say, Row i and then right to left in Row $i + 1$ and so on. If the number of rows are even, then, in this manner we can traverse all the points 5). If it is odd then we traverse in a similar way except for the last two rows, where we simultaneously traverse those in a zigzag fashion 6). So filling up a region appropriately can be done in linear time with respect to the run-length of the corresponding H-run. □

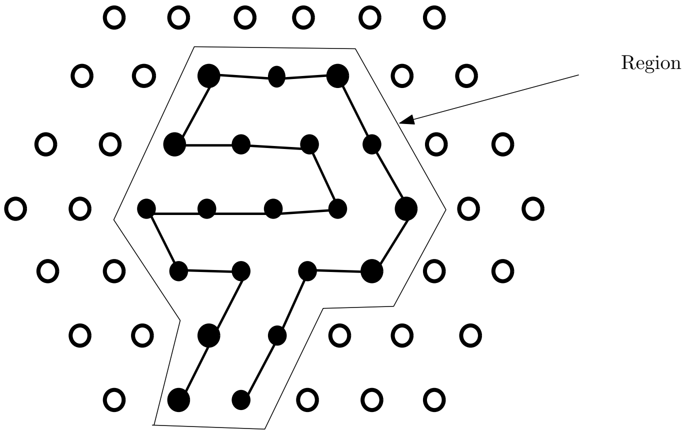


Fig. 5. For even number of rows

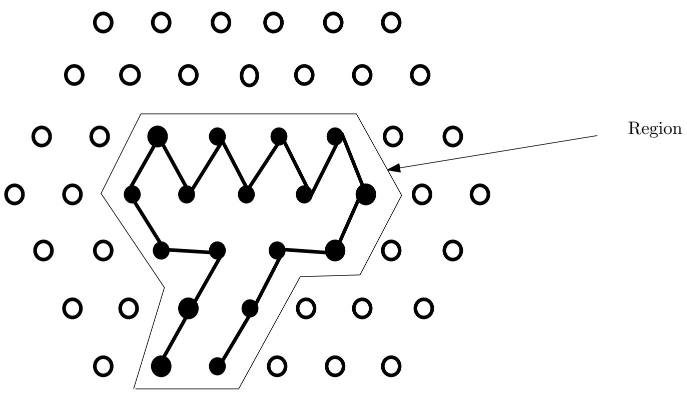


Fig. 6. For odd number of rows

Our algorithm runs in polynomial time as discussed below. Firstly, the algorithm iterates over at most n times. Now we have $n \leq \sqrt{z}$, because, $z = 3 \times n \times (n-1)$ which is proved in Lemma 6 in the following section. In each iteration we have to find a region for the current H-run. If a H-run has run-length ℓ, then Step 3(a) in the algorithm needs $O(\ell^2)$ time as Step 3(a(ii)) needs at most $O(\ell)$ time. So total time needed to perform this operation in each iteration, is at most $O(z^2)$. As each of the other steps need constant time, the complete runtime of the algorithm is $O(z^2 \times \sqrt{z})$.

6 Expected Approximation Ratio

In this section, we are going to deduce the expected approximation ratio of our algorithm. As the total number and run-lengths of H-runs may vary, in this analysis, we will find the expected number of H-runs and the expected run-length of each of the H-runs. These two values will depend only on SumH. Consider a regular hexagon with a side having n points. So, its depth is $n-1$. Assume that the total number of points in the hexagon is S. Then we have the following lemma.

Lemma 4. *Suppose we have a regular hexagon with depth* $n-1$ *and* S *points. The total number of bonds,* B*, is* $6 \times (n-1)^2 - 1$ *when all the points are filled with H.*

The proof will be given in the fuller version. Now the following lemma considers non-regular hexagons as well.

Lemma 5. *Consider a hexagon (either regular or non regular) having* n^2 *points. Then, the total number of bonds* B *is less than* $2 \times n(n-1)$.

The proof will be given in the fuller version.

We will now deduce the approximation ratio based on an expected value of the total number of bonds. We assume that all H-runs have equal length.

This assumption is valid in the context of our analysis and does not lose generality as follows. In what follows, we will be working with the expected number of H-runs and the expected length (say k_{Ex}) of an H-run. Hence in our analysis, each H-run will be assumed to have length k_{Ex}. We will now compute the expected values of the total number of edges (bonds), E_{Ex} (B_{Ex}) under this assumption.

From Figure 1, we can see that, the length of the hexagon is ℓ and depth is x. So each bend contains $2x + 1$ points and there are a total of ℓ bends. There are x^2 remaining lattice points outside the ℓ bends. So if the total number of points are z (see Figure 1) then,

$$z = (2x + 1) \times \ell + x^2 \tag{1}$$

So for a given z and x we can get,

$$\ell = \frac{(z - x^2)}{2x + 1} \tag{2}$$

To calculate the total number of edges, at first we have to identify how many edges can be formed by individual points. The arguments of Lemma 4 for calculating E and B also apply here. Note that, on the perimeter, aside from the corner points, total number of points are $2 \times (\ell - 2) + 4 \times (x - 1)$. So E_{Ex} can be computed as follows:

$$\begin{aligned}
&2E_{Ex} = 6 \times ((2x + 1) \times \ell + x^2) - 6 \times 3 - 2 \times (2 \times (\ell - 2) + 4 \times (x - 1))\\
&\Rightarrow 2E_{Ex} = 2 \times 3 \times ((2x + 1) \times \ell + x^2) - 9 - (2\ell - 4 + 4x - 4)\\
&\Rightarrow E_{Ex} = 3 \times ((2x + 1) \times \ell + x^2) - 1 - 2 \times (\ell + 2x)
\end{aligned}$$

And B_{Ex} can be computed as follows:
$B_{Ex} = E_{Ex} - z$
$B_{Ex} = 3 \times ((2x + 1) \times \ell + x^2) - 1 - 2 \times (\ell + 2x) - ((2x + 1) \times \ell + x^2)$
$\Rightarrow B_{Ex} = 2 \times ((2x + 1) \times \ell + x^2) - 2 \times (\ell + 2x) - 1$
Hence, we get the following equation.

$$B_{Ex} = 2z - 2 \times (\ell + 2x) - 1 \tag{3}$$

Note that according to our approach, the value of x is dependent on SumH. For this analysis, we now derive the expected run-length of H for a given H-P string where SumH is n^2. This problem can be mapped into the problem of *Integer Partitioning* as defined below.

Problem 2. Given an integer Y, the problem of Integer Partitioning aims to provide all possible ways of writing Y, as a sum of positive integers.

Note that the ways that differ only in the order of their summands are considered to be the same partition. A summand in a partition is called a part. Now, if we consider SumH as the input of Problem 2 (i.e., Y) then each run-length can be viewed as parts of the partition. So at first, we have to find the expected number of partitions, i.e., the expected number of runs of H. Kessler and Livingston [9] showed that to get an integer partition of an integer Y, the expected number of required parts is-

$$\sqrt{\frac{3Y}{2\pi}} \times (\log Y + 2\gamma - 2\log\sqrt{\frac{\pi}{6}}),$$

where γ is the famous Euler's constant.

For our problem $Y = SumH = n^2$. If we denote $E[P]$ as the expected number of H-runs then,

$$E[P] = \sqrt{\frac{6}{\pi}} \times n \times (\log n + \gamma - \log\sqrt{\frac{\pi}{6}}).$$

Now, as $(\log n + \gamma - \log\sqrt{\frac{\pi}{6}}) \leq (\sqrt{\frac{2\pi}{3}} \times \log n)$ for $n \geq 5$, we can say that

$$E[P] \leq 2n \times \log n.$$

Since SumH is n^2, expected value of each part, i.e., expected length of each run is greater than or equal to $\frac{n^2}{2n\times\log n} = \frac{n}{2\log n}$. Since all the H-runs are assumed to have the same length so each of them will construct a bend of $2x+1$ points in the lattice. So we must have $2x+1 \geq \frac{n}{2\log n}$. Hence we get the following equations:

$$x \geq \frac{n}{4\log n} - \frac{1}{2} \tag{4}$$

$$\ell \leq \frac{n^2 - (\frac{n^2}{16(\log n)^2} - \frac{n}{4\log n} + \frac{1}{4})}{\frac{n}{2\log n}} \tag{5}$$

Now, let us consider a hexagon $\mathcal{H}_1$ with length $\ell_{max} = \frac{n^2 - (\frac{n^2}{16(\log n)^2} - \frac{n}{4\log n} + \frac{1}{4})}{\frac{n}{2\log n}}$ and depth $x_{min} = \frac{n}{4\log n} - \frac{1}{2}$. Now, in $\mathcal{H}_1$ we also must have n^2 points. So, from Lemma 1 and Equations 4 and 5, clearly the number of bonds in $\mathcal{H}_1$ is less than or equal to than that in the hexagon having length ℓ and depth x. So from Equation 3 we have the following:

$$B_{Ex} \geq 2n^2 - 2 \times (2n\log n - \frac{n}{8\log n} - \frac{\log n}{2n} + \frac{1}{2} + \frac{n}{2\log n} - 1) - 1$$
$$\Rightarrow B_{Ex} \geq 2n^2 - 2 \times (2n\log n + \frac{3n}{8\log n} - \frac{\log n}{2n})$$

Now from Lemma 5, recall the upper bound for the total number of bonds, which is as follows: $B < 2 \times n(n-1)$. Hence we get the following expected approximation ratio :

$$\frac{B_{Ex}}{B} \geq \frac{2n^2 - 2 \times (2n\log n + \frac{3n}{8\log n} - \frac{\log n}{2n})}{2n \times (n-1)}$$

$$\Rightarrow \frac{B_{Ex}}{B} \geq \frac{n - 2\log n - \dfrac{3}{8\log n} + \dfrac{\log n}{2n^2}}{n-1}$$

As the term $\dfrac{\log n}{2n^2}$ is very small we can ignore it from the final result. Hence we have:

$$\frac{B_{Ex}}{B} \geq \frac{n - 2\log n - \dfrac{3}{8\log n}}{n-1}$$

As, $\dfrac{3}{8\log n} \leq 1$ for $n \geq 2$, so, $\dfrac{B_{Ex}}{B} \geq \dfrac{n - 2\log n - 1}{n-1}$ or

$$\frac{B_{Ex}}{B} \geq 1 - \frac{2\log n}{n-1} \quad for\ n \geq 6.$$

This is the final expected approximation ratio.

$\log n$	n	z	ratio
3	8	64	0.142
4	16	256	0.466
5	32	1024	0.677
6	64	4096	0.809

Fig. 7. Expected approximation ratio for different values of n

Note that the ratio increases significantly with the increase of the value of n as presented in Figure 7. So we can see that for large values of n, the expected approximation ratio tends to 1. So for large n it is expected that our algorithm will outperform the approximation algorithm presented in [1]. Recall that the approximation ratio of the algorithm of [1] is $\frac{6}{11}$, i.e., around 0.55.

7 Conclusion

In this paper, we have given a novel approximation algorithm to solve the protein folding problem in the H-P model introduced by Dill [2]. Our algorithm is polynomial and the expected approximation ratio is $1 - \dfrac{2\log n}{n-1}$ for $n \geq 6$ where n^2 is total number of H's in a given H-P string. For larger H-P strings it is expected that our algorithm will give a better result than the algorithm provided in [1], which currently gives the best approximation ratio for a 2D-triangular lattice. Additionally, our expected approximation ratio tends to reach one for large values of n. Hence our algorithm is expected to perform very well for larger H-P strings.

Acknowledgement. The authors would like to express their gratitude to the anonymous reviewers for their helpful comments and suggestions which helped a lot in improving the quality of the presentation of the paper. Also, the authors are grateful to Md. Mahbubul Hasan for insightful discussions and comments on an early draft.

References

1. Agarwala, R., Batzoglou, S., Dancík, V., Decatur, S.E., Hannenhalli, S., Farach, M., Muthukrishnan, S., Skiena, S.: Local rules for protein folding on a triangular lattice and generalized hydrophobicity in the hp model. Journal of Computational Biology 4(3), 275–296 (1997)
2. Dill, K.A.: Theory for the folding and stability of globular-proteins. Biochemistry 24(6), 1501–1509 (1985)
3. Hart, W.E., Istrail, S.: Fast protein folding in the hydrophobic-hydrophillic model within three-eights of optimal. Journal of Computational Biology 3(1), 53–96 (1996)
4. Hart, W.E., Istrail, S.: Invariant patterns in crystal lattices: Implications for protein folding algorithms (extended abstract). In: Hirschberg, D.S., Meyers, G. (eds.) CPM 1996. LNCS, vol. 1075, pp. 288–303. Springer, Heidelberg (1996)
5. Hart, W.E., Istrail, S.: Lattice and off-lattice side chain models of protein folding: Linear time structure prediction better than 86% of optimal. Journal of Computational Biology 4(3), 241–259 (1997)
6. Heun, V.: Approximate protein folding in the HP side chain model on extended cubic lattices (Extended abstract). In: Nešetřil, J. (ed.) ESA 1999. LNCS, vol. 1643, pp. 212–223. Springer, Heidelberg (1999)
7. Istrail, S., Lam, F.: Combinatorial algorithms for protein folding in lattice models: A survey of mathematical results. Communications in Information and Systems 9(4), 303–346 (2009)
8. Jiang, M., Zhu, B.: Protein folding on the hexagonal lattice in the hp model. J. Bioinformatics and Computational Biology 3(1), 19–34 (2005)
9. Kessler, I., Livingston, M.: The expected number of parts in a partition of n. Monatshefte für Mathematik 81(3), 203–212 (1976)
10. Newman, A.: A new algorithm for protein folding in the hp model. In: SODA, pp. 876–884 (2002)
11. Newman, A., Ruhl, M.: Combinatorial problems on strings with applications to protein folding. In: Farach-Colton, M. (ed.) LATIN 2004. LNCS, vol. 2976, pp. 369–378. Springer, Heidelberg (2004)

SAT and IP Based Algorithms for Magic Labeling with Applications

Gerold Jäger

Department of Mathematics and
Mathematical Statistics
University of Umeå
SE-90187 Umeå, Sweden
gerold.jaeger@math.umu.se

Abstract. A labeling of a graph with n vertices and m edges is a one-to-one mapping from the union of the set of vertices and edges onto the set $\{1, 2, \ldots, m + n\}$. Such a labeling is defined as magic, if one or both of the following two conditions is fulfilled: the sum of an edge label and the labels of its endpoint vertices is constant for all edges; the sum of a vertex label and the labels of its incident edges is constant for all vertices. We present effective IP and SAT based algorithms for the problem of finding a magic labeling for a given graph. We experimentally compare the resulted algorithms by applying it to random graphs. Finally, we demonstrate its performance by solving five open problems within the theory of magic graphs, posed in the book of Wallis.

Keywords: Boolean Satisfiability, Integer Programming, Magic Labeling.

1 Introduction

In this work we propose algorithms for *Magic Labeling* which has the following important application. Consider a communication network which consists of a set of devices and communication lines between some pairs of devices (see [1,4]). In order to prevent collisions, all communication lines have to be unique. Thus it is reasonable to assign each device and each communication line a different number. Because of security reasons, knowing the numbers of two devices should identify the communication line between them. This aim has been reached, if the numbers of two devices and its corresponding communication line sum to a constant value. Another application of Magic Labeling is radar impulses used to measure the distance of objects (see [1,5] for details). More exactly, both problems can be modeled as a so-called *EML* which will be formally defined in the following.

Let an undirected graph $G = (V, E)$ be given with vertex set V and edge set E, where $|V| = n$ and $|E| = m$. A *labeling* is a one-to-one mapping $\lambda : V \cup E \to \{1, 2, \ldots, m + n\}$. For a given labeling define the *weight* $\omega(e)$ of an edge $e \in E$ as the sum of the label of e and of the labels of its two endpoints.

© Springer-Verlag Berlin Heidelberg 2013

An *edge-magic labeling (EML)* is a labeling λ for which a constant $k \in \mathbb{N}$ exists such that $\omega(e) = k$ for each edge $e \in E$ (for examples see Figure 1). Similarly, define the *weight* $\omega(v)$ of a vertex $v \in V$ as the sum of the label of v and of the labels of all edges incident to v. A *vertex-magic labeling (VML)* is a labeling λ for which a constant $h \in \mathbb{N}$ exists such that $\omega(v) = h$ for each vertex $v \in V$ (for an example see Figure 1). Finally, a *totally magic labeling (TML)* is a labeling λ for which (not necessarily equal) constants $h, k \in \mathbb{N}$ exist such that λ is edge-magic with constant k and vertex-magic with constant h (for examples see Figure 1). h and k are called *magic constants*. A vertex $v \in V$ and an edge $e \in E$ are denoted *neighboring*, if e is incident to v. Note that different EMLs exist for the same graph (see Figure 1), and the same holds for VMLs. Surveys of results for magic labeling and related problems are given in [8] and [22]. Properties and algorithms for magic labeling on special graphs like cycles, wheels, paths and complete graphs can be found in [2,16,17]. We consider the problems, whether there exists an EML with given magic constant $k \in \mathbb{N}$, a VML with given magic constant $h \in \mathbb{N}$, and a TML with given magic constants $h, k \in \mathbb{N}$. Furthermore, we also consider the corresponding problems, where the constants are not given in the problem formulation.

Only few complexity results or general effective algorithm are known for these problems. Whereas in [13] the relationship between magic labeling and other combinatorial problems is discussed, which can be used to derive complexity results and algorithms, in [7] a sieve method for totally magic labelings is described, and in [3] a heuristic for magic (as well as antimagic) labelings is presented, where also an IP algorithm is utilized. Furthermore, an algorithm exists for a similar labeling problem, where only the edges are labeled and the labels do not necessarily differ [19]).

In the Sections 2.1 and 2.2, we present two different approaches for magic labeling, namely based on integer programming (IP) and on Boolean satisfiability (Sat). Whereas integer programming is a standard tool for solving combinatorial problems, see, e.g., [20], Sat encodings have only been used in recent times for this purpose. For instance, such encodings were given for the *Hamiltonian Cycle Problem* [11,12,18,21], *Haplotype Inference by Pure Parsimony* [14], the *Social Golfer Problem* [9], and the game *Sudoku* [15]. The reason is that in the last years much progress has been made in the optimization of practical Sat solvers (see [10] and the Sat competition [25]).

In Section 3 we compare the performance of these IP and Sat based algorithms for the problems EML and VML. Note that TML has been omitted in the experiments, as TMLs are rather rare. In particular, it has been proven that there are only six graphs with 10 and fewer vertices, for which a TML exists [22] (e.g., the TMLs of Figure 1).

In Section 4 we apply the introduced algorithms to the theory of magic labelings. We succeed in solving five unsolved research problems from this area, posed in the book of Wallis [22] about magic graphs.

We close this work with some suggestions for future work in Section 5

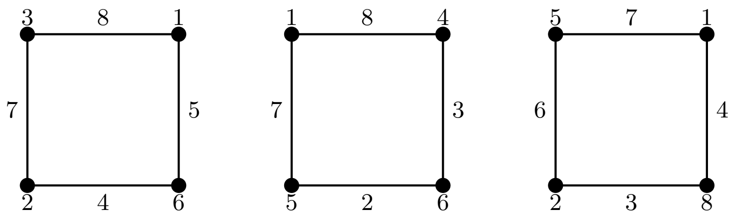


Fig. 1. Three EMLs of graphs with $n = 4$, $m = 4$, $k = 12$, $n = 4$, $m = 4$, $k = 13$, and $n = 4$, $m = 4$, $k = 13$

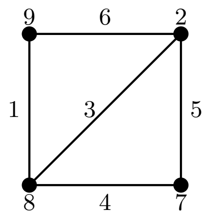


Fig. 2. VML of a graph with $n = 4$, $m = 5$ and $h = 16$

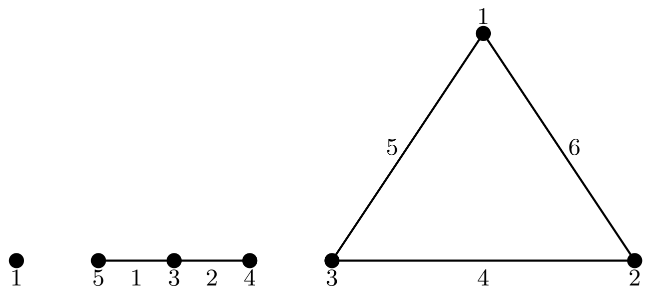


Fig. 3. Three TMLs of graphs with $n = 1$, $m = 0$, $h = 1$, $k = 0$, $n = 3$, $m = 2$, $h = 6$, $k = 9$, and $n = 3$, $m = 3$, $h = 12$, $k = 9$

2 Algorithms for Magic Labelings

Let $G = (V, E)$ be a graph with $|V| = n$, $|E| = m$, and let $r = n + m$. For our convenience, we define a fixed ordering on the set $V \cup E$ by the numbers $1, 2, \ldots, r$, i.e., each number of $\{1, 2, \ldots, r\}$ represents an edge or a vertex of the graph.

2.1 IP Based Algorithm

For the IP we consider integer variables w_i for $1 \leq i \leq r$. We define

$$w_i \; = \; j, \quad \text{if edge/vertex } i \text{ receives label } j$$

for $1 \leq i, j \leq r$. This definition leads to the following linear constraint:

$$1 \ \leq w_i \ \leq r \quad \text{for } i = 1, 2, \ldots, r.$$

To ensure that there is a one-to-one mapping between the edges/vertices and the labels, we introduce further 0-1 variables x_{ij}, which are defined as follows:

$$x_{i,j} = \begin{cases} 1, & \text{if the label of edge/vertex } i \\ & \text{is larger than the label of edge/vertex } j \\ 0, & \text{otherwise} \end{cases}$$

for $1 \leq i \neq j \leq r$. This can be expressed by the following linear constraint:

$$w_i - w_j \geq 1 + (x_{ij} - 1) \cdot r \quad \text{for } 1 \leq i \neq j \leq r.$$

The correctness of this constraint is based on the fact that w_i and w_j do not differ by more than $r - 1$.

Clearly, exactly one of the labels w_i and w_j is larger than the other one, i.e.,

$$x_{ij} + x_{ji} = 1 \quad \text{for } 1 \leq i < j \leq r.$$

Finally, we have to add linear constraints which ensure that the conditions of EML/VML/TML are fulfilled. For this purpose, we distinguish the cases that the weight of each single edge is constant and that the weight of each single vertex is constant, where in the case of TML both conditions have to be fulfilled.

– **EML/TML**: Let an arbitrary edge $e = (v, w)$ be given, where e, v, w are represented by the labels $w_{i_1}, w_{i_2}, w_{i_3}$. This leads to the following linear constraint:

$$w_{i_1} + w_{i_2} + w_{i_3} = k.$$

– **VML/TML**: Let an arbitrary vertex v be given with incident edges $e_1, e_2, \ldots, e_l$ with $1 \leq l \leq n - 1$, where $v, e_1, e_2, \ldots, e_l$ are represented by the labels $w_{i_1}, w_{i_2}, \ldots, w_{i_l}, w_{i_{l+1}}$. We receive the following linear constraint:

$$\sum_{t=1}^{l+1} w_{i_t} = h.$$

Remark 1. In total we have $\frac{3}{2}\left((n+m)^2 - (n+m)\right) + sn + tm$ constraints, where we have for VML $s = 1, t = 0$, for EML $s = 0, t = 1$, and for TML $s = 1, t = 1$.

Remark 2. This IP based model can easily be transformed to a model, where the magic constants h and k are integer variables of the IP. This variant of the IP algorithm searches for magic labelings with arbitrary magic constants. We denote this variant as ARB-IP-MAGIC.

2.2 SAT Based Algorithm

For the encoding we use r^2 Boolean variables $x_{i,j}$ with $1 \leq i, j \leq r$, where we set

$$x_{i,j} = \begin{cases} \textsc{True}, & \text{if edge/vertex } i \text{ receives label } j \\ \textsc{False}, & \text{otherwise} \end{cases}.$$

Labeling Clauses. For receiving a feasible labeling we need the following conditions.

First, each edge/vertex needs to have exactly one label. This leads to the condition:

$$\text{For } i = 1, 2, \ldots, r \text{ exactly one } j \in \{1, 2, \ldots, r\} \text{ exists with } x_{i,j} = \textsc{True}. \quad (1)$$

Second, each label has to be used by exactly one edge/vertex. This leads to the condition:

$$\text{For } j = 1, 2, \ldots, r \text{ exactly one } i \in \{1, 2, \ldots, r\} \text{ exists with } x_{i,j} = \textsc{True}. \quad (2)$$

All $2r$ restrictions of (1) and (2) have the same structure, namely that exactly one of the r involved Boolean variables is set to True and the rest to False. To represent this, we introduce $2r^2$ auxiliary variables $y_1, y_2, \ldots, y_{2r^2}$ with r y's for one restriction. W.l.o.g., consider the first restriction, which contains the Boolean variables $x_{1,1}, x_{1,2}, \ldots, x_{1,r}$, and the corresponding auxiliary variables $y_1, y_2, \ldots, y_r$. For $1 \leq i \leq r$ we use y_i to represent that at least one of $x_{1,1}, x_{1,2}, \ldots, x_{1,i}$ is True. Precisely, the y variables are defined as follows.

- $y_1 = x_{1,1}$ or equivalently $(\neg x_{1,1} \vee y_1) \wedge (x_{1,1} \vee \neg y_1)$,
- $y_i = x_{1,i} \vee y_{i-1}$ or equivalently $(y_i \vee \neg x_{1,i}) \wedge (y_i \vee \neg y_{i-1}) \wedge (\neg y_i \vee x_{1,i} \vee y_{i-1})$ for $i = 2, 3, \ldots, r$.

In addition, we need to enforce that no more than one $x_{1,i}$ with $1 \leq i \leq r$ can be True. This means, if $x_{1,i}$ is True, none of the x_{1j} for $1 \leq j < i \leq r$ can be True. This is formulated as

- $\neg y_{i-1} \vee \neg x_{1i}$ for $i = 2, \ldots, r$.

Finally, y_r must be True.

Magic Clauses. Furthermore we have to add clauses which ensure that the conditions of EML/VML/TML are fulfilled. We consider the parameters $c \in \mathbb{N}$ for the magic constants h and k, respectively, and $l \in \mathbb{N}$, which captures the number of summands for each single condition.

– **EML/TML**: Set $c = k$ and $l = 2$. For a given edge the sum of $l + 1$ labels (namely the label of the edge and of its l endpoint vertices) equals c.
– **VML/TML**: Set $c = h$. For a given vertex with degree $l \in \mathbb{N}$ the sum of $l+1$ labels (namely the label of the vertex and of its l incident edges) equals c.

Observe that both conditions have the following structure: For given constants $c, l \in \mathbb{N}$ the sum of $l+1$ labels equals c. For $l \in \mathbb{N}$ let W be the set containing all possible l-tuples $\overrightarrow{w} = (w_1, w_2, \ldots, w_l)$ with $w_i \in \{1, 2, \ldots, r\}$ for $1 \le i \le l$ and $w_i \neq w_j$ for $1 \le i < j \le l$.

As an example instance, consider $l = 3$ and $r = 4$. Then it holds $|W| = 24$ and

$$\begin{aligned} W = \{&(1,2,3),(1,3,2),(2,1,3),(2,3,1),(3,1,2),(3,2,1),(1,2,4),(1,4,2),\\ &(2,1,4),(2,4,1),(4,1,2),(4,2,1),(1,3,4),(1,4,3),(3,1,4),(3,4,1),\\ &(4,1,3),(4,3,1),(2,3,4),(2,4,3),(3,2,4),(3,4,2),(4,2,3),(4,3,2)\}. \end{aligned}$$

Now let an edge or vertex f be given with corresponding $c, l \in \mathbb{N}$, i.e., if f is an edge, then $c = k$ and $l = 2$, and otherwise, $c = h$ and l is the degree of f. We want to fulfill the condition that f has constant weight c. This means that the sum of the label of f and of its neighboring elements is c. Let $f_1, f_2, \ldots, f_l$ be the neighboring elements of f. For this l compute the set W and choose an arbitrary element $\overrightarrow{w} \in W$ with $w := \sum_{i=1}^{l} w_i$. Then label $f_1, f_2, \ldots, f_l$ by $w_1, w_2, \ldots, w_l$, and consider four cases:

Case 1: $i \in \{1, 2, \ldots, l\}$ exists with $c - w = w_i$,
Case 2: $w \ge c$,
Case 3: $w < c - r$,
Case 4: Otherwise.

As all labels are different and are contained in the set $\{1, 2, \ldots, r\}$, it is clear that for the Cases 1, 2, or 3 no labeling of f exists such that the sum of the labels of $f, f_1, f_2, \ldots, f_l$ is c. In these cases we add the clause

$$\neg x_{f_1,w_1} \vee \neg x_{f_2,w_2} \vee \cdots \vee \neg x_{f_l,w_l}$$

meaning that labeling $f_1, f_2, \ldots, f_l$ by $w_1, w_2, \ldots, w_l$ is not possible. For Case 4 such a labeling is possible, but only if f is labeled with $c - w$. This leads to the following clause:

$$\neg x_{f_1,w_1} \vee \neg x_{f_2,w_2} \vee \cdots \vee \neg x_{f_l,w_l} \vee x_{f,c-w}.$$

Thus for each $\overrightarrow{w} \in W$ we have an additional clause.

Note that if we consider VMLs or TMLs with vertices of large degree, the cardinality of $|W|$ and therefore the number of magic clauses becomes exponential. Thus for dense graphs, the resulted algorithm is expected to have bad performance, which is confirmed by the results presented in Section 3.

3 Experimental Results

As already mentioned, no effective algorithm is known for general instances of EML, VML, and TML, respectively. In this section we compare the algorithms of Section 2, which are called IP-MAGIC, and SAT-MAGIC. All algorithms have been implemented in C++, where we make use of the IP solver

Cplex, version 12 [24], and a Sat solver implemented by Eén and Sörensson, called MiniSat [6,23]. The experiments were carried out on a SunOS system with a 386 CPU with 2613 MHz clock rate and 32192 MB of RAM. We test random graphs with number of vertices $n = 10, 15$, where $p = 10\%, 20\%, 30\%, 40\%$ edges are chosen randomly and uniformly distributed from all possible $n\cdot(n-1)/2$ ones. Note that the following lower and upper bounds for possible magic constants can easily be computed: Consider the following cases:

– Lower bound for k:
  Let l be the number of vertices with degree 0. Label these vertices with $m+n-l+1, m+n-l+2, \dots, m+n$. Label the remaining vertices with $1, 2, \dots, n-l$ and the edges with $n-l+1, n-l+2, \dots, m+n-l$. Let the vertices $i = 1, 2, \dots, n-l$ with degree $\neq 0$ be ordered by decreasing degree and let $\deg(i)$ be their degrees. This leads to the lower bound

$$\left(\sum_{i=1}^{n-l} \deg(i)\cdot i + \sum_{i=1}^{m} (n-l+i)\right)/m,$$

  where l is the number of isolated vertices.
– Upper bound for k:
  Let l be the number of vertices with degree 0. Label these vertices with $1, 2, \dots, l$. Label the remaining vertices with $m+l+1, m+l+2, \dots, m+n$ and the edges with $l+1, l+2, \dots, m+l$. Let the vertices $1, 2, \dots, n-l$ with degree $\neq 0$ be ordered by increasing degree and let $\deg(i)$ be their degrees. This leads to the upper bound

$$\left(\sum_{i=1}^{n-l} (\deg(i)\cdot(m+l+i)) + \sum_{i=1}^{m} (l+i)\right)/m,$$

  where l is the number of isolated vertices.
– Lower bound for h:
  Label the m edges with the labels $1, 2, \dots, m$ and the n vertices with $m+1$, $m+2$, …, $m+n$. This leads to the lower bound

$$\left(\sum_{i=1}^{m} (2\cdot i) + \sum_{i=1}^{n} (m+i)\right)/n.$$

– Upper bound for h:
  Label the m edges with the labels $n+1, n+2, \dots, n+m$ and the n vertices with $1, 2, \dots, n$. This leads to the upper bound

$$\left(\sum_{i=1}^{n} i + \sum_{i=1}^{m} 2\cdot(n+i)\right)/n.$$

For each single instance of the tests this lower bound $lb \in \mathbb{N}$ and this upper bound $ub \in \mathbb{N}$ are computed. Then we choose $\min\{10, ub - lb + 1\}$ values of the interval $[lb, ub]$ and for each value receive a single instance of the forms EML or VML. Thus we have $16 = 2 \cdot 2 \cdot 4$ test classes, where each test class consists of up to 10 single instances.

For both algorithms and for each test class, Table 1 and Table 2 give the percentage of successes in short time. More precisely, the given value is the percentage of how many instances of this test class can be solved within 1 hour.

The results show that Sat-Magic performs better for EML. On the other hand, as expected, Sat-Magic behaves badly for VMLs with large density.

Table 1. Comparison of IP-Magic and Sat-Magic for random graphs with 10 vertices

Size 10	EML				VML			
Density p (%)	10	20	30	40	10	20	30	40
IP-Magic (%)	100	70	80	0	100	100	100	10
Sat-Magic (%)	100	100	100	100	100	100	0	0

Table 2. Comparison of IP-Magic and Sat-Magic for random graphs with 15 vertices

Size 15	EML				VML			
Density p (%)	10	20	30	40	10	20	30	40
IP-Magic (%)	30	0	0	0	100	100	20	0
Sat-Magic (%)	70	70	0	0	100	0	0	0

4 Applications to the Theory of Magic Labelings

Wallis' book [22] contains a summary over the most known results in the theory of magic labeling. Moreover, he lists more than 30 open research problems, where mostly for specific classes of graphs the characteristics of EML, VML, TML are investigated. To show their potential we have applied our algorithms to those open problems and received solutions for five open problems. All these solutions are negative results, i.e., counterexamples for the corresponding claims. However, we expect that the algorithms also make contributions to positive results in the future, as the analysis of concrete labelings found by the algorithms can lead to new theoretical ideas. The solved research problems are described in the following.

4.1 Research Problem 2.4

Problem: For $n \in \mathbb{N}$ define the graph $K_{2n\backslash n}$ as the graph with $2n$ vertices, which contains all possible edges $\{i, j\}$ with $1 \leq i < j \leq 2n$ except those with $n+1 \leq i < j \leq 2n$. Does $K_{2n\backslash n}$ have an EML for all $n \in \mathbb{N}$?

Solution: Consider $n = 3$. The algorithm Arb-IP-Magic proves in 223.81 seconds that the graph $K_{6\backslash 3}$ (6 vertices and 12 edges) has no EML.

4.2 Research Problem 2.5a)

Problem: Let n be odd. Does a cycle C_n with n vertices have an EML, where for the magic constant k holds $\frac{5n+3}{2} \leq k \leq \frac{7n+3}{2}$?

Solution: Consider $n = 5$. Then we have $14 \leq k \leq 19$. The algorithms Sat-Magic (IP-Magic) proves in 0.06 (4.52) seconds for $k = 15$ and in 0.05 (5.88) seconds for $k = 18$ that the graph C_5 (5 vertices and 5 edges) has no EML. Note that for the remaining values $k = 14, 16, 17, 19$ the graph C_5 does have an EML.

4.3 Research Problem 2.9

Problem: For $m, n, p \in \mathbb{N}$ consider the complete tripartite graph $K_{m,n,p}$ consisting of three sets with the number of vertices m, n, p, respectively, where edges appear only between these three sets, but not inside. Does $K_{m,n,p}$ always have an EML?

Solution: Consider $m = n = p = 2$. The algorithm Arb-IP-Magic proves in 217.89 seconds that the graph $K_{2,2,2}$ (6 vertices and 12 edges) has no EML.

4.4 Research Problem 2.16

Problem: For $n \in \mathbb{N}$ consider the graph nK_4 consisting of n copies of the complete graph K_4, where no edges appear between these copies. Does nK_4 always have an EML, if n is even.

Solution: Consider $n = 2$. The algorithm Arb-IP-Magic proves in 18963.25 seconds that the graph $2K_4$ (8 vertices and 12 edges) has no EML.

4.5 Research Problem 3.3

Problem: For $m \in \mathbb{N}$ consider the complete bipartite graph $K_{m,m+1}$ consisting of two sets with the number of vertices m, $m+1$, respectively, where edges appear only between these two sets, but not inside. Does $K_{m,m+1}$ always have a VML, where for the magic constant h holds $\frac{(m+1)^2(m+2)}{2} \leq h \leq \frac{(m+1)(m^2+4m+2)}{2}$?

Solution: Consider $m = 2$. Then we have $18 \leq h \leq 21$. The algorithms Sat-Magic (IP-Magic) proves in 0.18 (14.55) seconds for $h = 21$ that the graph $K_{2,3}$ (5 vertices and 6 edges) has no VML. Note that for the remaining values $h = 18, 19, 20$ the graph $K_{2,3}$ does have a VML.

5 Future Work

It would be interesting to find specialized IP or Sat encodings for graphs like cycles, wheels, paths and complete graphs etc. In particular, symmetry conditions of these graphs should be helpful to specialize the general encoding.

A further question is whether it is possible to create a Sat encoding of polynomial size also for finding VMLs and TMLs in dense graphs?

There is another open research question from [22], namely about the very rare TMLs. In [7,22] it has been shown by an exhaustive computer search that only 6 graphs with number of vertices smaller than 11 exist which have a TML. In [1] this search has been extended so that it can be applied to the $1,018,997,864$ non-isomorphic graphs with 11 vertices (for details see [1]). However, it has turned out that still some candidate graphs have to be considered, and for these graphs, in particular the not connected graphs, the algorithms of this work are a reasonable method.

References

1. Arnold, F.: Totally Magic Graphs – A Complete Search On Small Graphs. Master Thesis, Clausthal University of Technology, Germany (2013)
2. Baker, A., Sawada, J.: Magic Labelings on Cycles and Wheels. In: Yang, B., Du, D.-Z., Wang, C.A. (eds.) COCOA 2008. LNCS, vol. 5165, pp. 361–373. Springer, Heidelberg (2008)
3. Bertault, F., Feria-Purón, R., Miller, M., Pérez-Rosés, H., Vaezpour, E.: A Heuristic for Magic and Antimagic Graph Labellings. In: Proc. VII Spanish Congress on Metaheuristics and Evolutive and Bioinspired Algorithms, Valencia, Spain (2010)
4. Bloom, G.S., Golomb, S.W.: Applications of Numbered Undirected Graphs. Proc. IEEE 65(4), 562–570 (1977)
5. Bloom, G.S., Golomb, S.W.: Numbered Complete Graphs, Unusual Rulers, and Assorted Applications. In: Theory and Applications of Graphs. Lecture Notes in Mathematics, vol. 642, pp. 53–65 (1978)
6. Eén, N., Sörensson, N.: An Extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
7. Exoo, G., Ling, A.C.H., McSorley, J.P., Philipps, N.C., Wallis, W.D.: Totally Magic Graphs. Discrete Math. 254(1-3), 103–113 (2002)
8. Gallian, J.A.: A Dynamic Survey of Graph Labelings. Electron. J. Combin. 15, DS6 (2008)
9. Gent, I.P., Lynce, I.: A Sat Encoding for the Social Golfer Problem. In: 19th International Joint Conference on Artificial Intelligence (IJCAI), Workshop on Modelling and Solving Problems with Constraints (2005)
10. Gomes, C.P., Kautz, H., Sabharwal, A., Selman, B.: Satisfiability Solvers. In: van Harmelen, F., Lifschitz, V., Porter, B. (eds.) Handbook of Knowledge Representation. Foundations of Artificial Intelligence, vol. 3, pp. 89–134. Elsevier (2008)
11. Hoos, H.H.: Sat-Encodings, Search Space Structure, and Local Search Performance. In: Proc. 16th International Joint Conference on Artificial Intelligence (IJCAI), pp. 296–303. Morgan Kaufmann (1999)
12. Jäger, G., Zhang, W.: An Effective Algorithm for and Phase Transitions of the Directed Hamiltonian Cycle Problem. J. Artificial Intelligence Res. 39, 663–687 (2010)

13. Kalantari, B., Khosrovshahi, G.B.: Magic Labeling in Graphs: Bounds, Complexity, and an Application to a Variant of TSP. Networks 28(4), 211–219 (1996)
14. Lynce, I., Marques-Silva, J., Prestwich, S.D.: Boosting Haplotype Inference with Local Search. Constraints 13(1-2), 155–179 (2008)
15. Lynce, I., Ouaknine, J.: Sudoku as a Sat Problem. In: Proc. 9th International Symposium on Artificial Intelligence and Mathematics, AIMATH (2006)
16. MacDougall, J.A., Miller, M., Slamin, Wallis, W.D.: Vertex-magic Total Labelings of Graphs. Util. Math. 61, 3–21 (2002)
17. MacDougall, J.A., Miller, M., Wallis, W.D.: Vertex-magic Total Labelings of Wheels and Related Graphs. Util. Math. 62, 175–183 (2002)
18. Prestwich, S.D.: Sat Problems with Chains of Dependent Variables. Discrete Appl. Math. 130(2), 329–350 (2003)
19. Sun, G.C., Guan, J., Lee, S.-M.: A Labeling Algorithm for Magic Graph. Congr. Numer. 102, 129–137 (1994)
20. Vanderbei, R.J.: Linear Programming: Foundations and Extensions, 3rd edn. International Series in Operations Research & Management Science, vol. 114. Springer (2008)
21. Velev, M.N., Gao, P.: Efficient Sat Techniques for Absolute Encoding of Permutation Problems: Application to Hamiltonian Cycles. In: Proc. 8th Symposium on Abstraction, Reformulation and Approximation (SARA), pp. 159–166 (2009)
22. Wallis, W.D.: Magic Graphs. Birkhäuser, Boston (2001)
23. Source Code of [6] (MiniSat), `http://minisat.se/`
24. Homepage of IP solver Cplex, `http://www.ilog.com/products/optimization/archive.cfm`
25. International Sat Competition, `http://www.satcompetition.org/`

An Optimal Algorithm for Computing All Subtree Repeats in Trees

Tomáš Flouri[1,⋆], Kassian Kobert[1,⋆⋆], Solon P. Pissis[1,2,⋆⋆⋆], and Alexandros Stamatakis[1,3]

[1] Heidelberg Institute for Theoretical Studies, Germany
[2] Florida Museum of Natural History, University of Florida, USA
[3] Karlsruhe Institute of Technology, Institute for Theoretical Informatics, Postfach 6980, 76128 Karlsruhe

Abstract. Given a labeled tree T, our goal is to group repeating subtrees of T into equivalence classes with respect to their topologies and the node labels. We present an explicit, simple, and time-optimal algorithm for solving this problem for unrooted unordered labeled trees, and show that the running time of our method is linear with respect to the size of T. By unordered, we mean that the order of the adjacent nodes (children/neighbors) of any node of T is irrelevant. An unrooted tree T does not have a node that is designated as root and can also be referred to as an undirected tree. We show how the presented algorithm can easily be modified to operate on trees that do not satisfy some or any of the aforementioned assumptions on the tree structure; for instance, how it can be applied to rooted, ordered or unlabeled trees.

1 Introduction

Tree data structures are among the most common and well-studied of all combinatorial structures. Tree structures are present in a wide range of applications, such as, in the implementation of functional programming languages [12], term-rewriting systems [11], programming environments [2], code optimization in compiler design [1], code selection [8], theorem proving [13], and computational biology [14].

Thus, efficiently extracting the repeating patterns in a tree structure, represents an important computational problem. Recently, Christou *et al.* [5] presented a linear-time algorithm for computing all subtree repeats in *rooted ordered unlabeled* trees. In [4], Christou *et al.* extended this algorithm to compute all subtree repeats in *rooted ordered labeled* trees in linear time *and* space.

The limitation of the aforementioned results is that they cannot be applied to *unrooted* or *unordered* trees. By unrooted, we mean that the input tree does not have a dedicated root node; and, by unordered, we mean that the order of

[⋆] Supported by the DFG grant STA 860/4.
[⋆⋆] Supported by a PhD scholarship from institutional funding at HITS.
[⋆⋆⋆] Supported by the NSF–funded iPlant Collaborative (NSF grant #DBI-0735191).

© Springer-Verlag Berlin Heidelberg 2013

the descendant nodes (children) of any node of the tree is irrelevant. Such trees are a generalization of rooted ordered trees, and, hence, they arise naturally in a broader range of real-life applications. For instance, unrooted unordered trees are used in the field of (molecular) phylogenetics [7,16].

The field of molecular phylogenetics deals with inferring the evolutionary relationships among species using molecular sequencing technologies and statistical methods. Phylogenetic inference methods typically return unrooted unordered labeled trees that represent the evolutionary history of the organisms under study. These trees depict evolutionary relationships among the molecular sequences of extant organisms (living organisms) that are located at the tips (leaves) of those trees and hypothetical common ancestors at the inner nodes of the tree. With the advent of so-called next-generation sequencing technologies, large-scale multi-national sequencing projects such as, for instance, 1KITE[1] (1000 insect transcriptome sequencing project) emerge. In these projects, large phylogenies that comprise thousands of species and massive amounts of whole-transcriptome or even whole-genome molecular data need to be reconstructed.

Provided a fixed multiple sequence alignment (MSA) of the sequences—representing species—under study, the goal of phylogenetic inference is to find *the* tree topology that best explains the underlying data, using a biologically reasonable optimality criterion—a scoring function for the trees. One such optimality criterion is *Maximum Likelihood* (ML) [6]. Finding the optimal tree under ML is known to be NP-complete [3]. Note that the number of possible unrooted tree topologies for n species, located at the tips, grows super-exponentially with n. Therefore, widely-used tools for ML-based inference of phylogenies, such as RAxML [15] and PHYML [9], rely on heuristic search strategies for exploring the immense tree space.

The likelihood of each candidate tree topology T is calculated by computing the conditional likelihoods at each inner node of T. The conditional likelihoods are computed independently for each *site* (column in the MSA). The conditional likelihoods are computed via a post-order traversal of T starting from a virtual root. Note that, as long as the statistical model of evolution is time-reversible (i.e. evolution occurred in the same way if followed forward or backward in time) the likelihood score is invariant with respect to where in T the virtual root has been placed.

In phylogenetic inference software, a common technique for optimizing the likelihood function, which typically consumes $\approx 95\%$ of total execution time, is to eliminate duplicate *sites* (equivalent columns in the MSA). This is achieved by compressing identical sites into site patterns and assigning them a corresponding weight. This can be done because duplicate sites yield exactly the same likelihood *iff* they evolve under the same statistical model of evolution. When two sites are identical, this means that the leaves of the tree are labeled equally. Consider a forest of trees with the same topology, where, for each tree, the labels are defined by the molecular data stored at a particular site of the MSA and the position of the tips. Knowing equivalent subtrees within such a forest would allow someone

[1] `http://www.1kite.org/`

to minimize the number of operations required to compute the likelihood of a phylogenetic tree. This can be seen as a generalization of the site compression technique.

Our Contribution. In this article, we extend the series of results presented in [5] and [4] by introducing an algorithm that computes all subtree repeats in *unrooted unordered labeled* trees in linear time and space. The importance of our contribution is underlined by the fact that the presented algorithm can be easily modified to work on trees that do not satisfy some or any of the above assumptions on the tree structure; e.g. it can be applied to rooted, ordered or unlabeled trees.

2 Preliminaries

2.1 Basic Definitions

An unrooted unordered tree is an undirected unordered acyclic connected graph $T = (V, E)$ where V is the set of nodes and E the set of edges such that $E \subset V \times V$ with $|E| = |V| - 1$. The number of nodes of a tree T is denoted by $|T| := |V|$. An *alphabet* Σ is a finite, non-empty set whose elements are called *symbols.* A *string* over an alphabet Σ is a finite, possibly empty, string of symbols of Σ. The length of a string x is denoted by $|x|$, and the concatenation of two strings x and y by xy. A tree T is *labeled* if every node of T is labeled by a symbol from some alphabet Σ. Different nodes may have the same label.

A *tree center* of an unrooted tree $T = (V, E)$ is the set of all vertices such that the greatest node distance to any leaf is minimal. An unrooted tree T has either one node that is a tree center, in which case it is called a *central tree*, or two adjacent nodes that are tree centers, in which case it is called a *bicentral tree* [10]. Let $\hat{T}(T) = (\hat{V}, \hat{A})$ be the rooted tree on $\hat{V} = V \cup \{r\}$, where $\hat{A}$ is defined such that $|\hat{A}|$ is minimal with $(u, v) \in \hat{A}$ only if $\{u, v\} \in E$ and each node other than r is reachable from one central point. If T is a bicentral tree, we add the additional root node r to V and add two edges to $\hat{A}$, namely (r, v) and (r, u), where v and u are the central points of T. Otherwise, if T is a central tree, with tree center u, we set $r := u$ and thus $\hat{V} = V$.

We call $u \in V$ a *child* of v *iff* $(v, u) \in \hat{A}$. In this case, we call v the *parent* of u and define *parent*$(u) := v$. We call u and u' *siblings iff* there exists a node $v \in \hat{V}$ such that (v, u), $(v, u') \in \hat{A}$. Note that under this definition two central points of a bicentral tree are siblings of each other.

The (rooted) subtree that is obtained by removing edge $\{v, u\}$ and contains node v as its root node is denoted by $\hat{T}(v, u)$. We consider only *full subtrees*, i.e. subtrees which contain all nodes and edges that can be reached from v when only the edge $\{v, u\}$ is removed from the tree. The special case $\hat{T}(v, v)$ denotes the tree containing all nodes that is rooted in v. For simplicity, we refer to $\hat{T}(v, \mathit{parent}(v))$ as $\hat{T}(v)$. The *diameter* of an unrooted tree T is denoted by $d(T)$ and is defined as the number of edges of the longest path between any two *leafs* (nodes with degree 0) of T. The *height of a rooted (sub)tree* $\hat{T}(v, u)$ of some tree

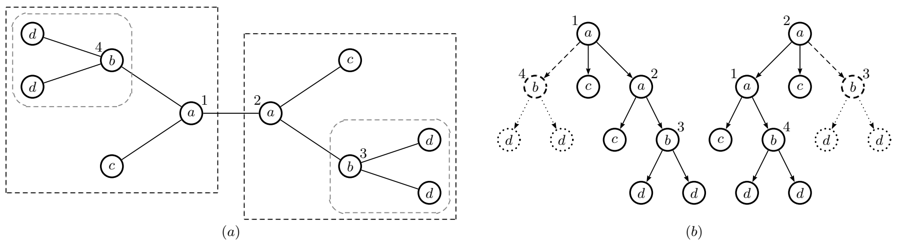


Fig. 1. (a) An *unrooted tree* T consisting of 10 nodes; a *non-overlapping subtree repeat* $R = \{(3,2),(4,1)\}$ is marked with dashed rounded rectangles; another *non-overlapping subtree repeat* containing the trees $\hat{T}(1,2), \hat{T}(2,1)$ is marked with dashed rectangles (b) An *overlapping subtree repeat* $R = \{(2,3),(1,4)\}$ of T resulting from the deletion of the dashed edge and its corresponding dotted subtree. This is an overlapping subtree repeat since nodes 1 and 2–and the node labeled by c–are contained in both subject trees. A *total repeat* $R = \{(1,1),(2,2)\}$ of T can be obtained by keeping all the edges and rooting T in node 1 ($\hat{T}(1,1)$) and 2 ($\hat{T}(2,2)$), respectively

T, denoted by $h(v,u)$, is defined as the number of edges on the longest path from the root v to some leaf of $\hat{T}(v,u)$. The *height of a node* v, denoted by $h(v)$, is defined as the length of the longest path from v to some leaf in $\hat{T}(T)$.

For simplicity, in the rest of the text, we denote: a rooted unordered labeled tree by $\hat{T}$; an unrooted unordered labeled tree by T; and the rooted (directed) version of T by $\hat{T}(T)$, as defined above.

2.2 Subtree Repeats

Two trees $\hat{T}_1 = (V_1, A_1)$ and $\hat{T}_2 = (V_2, A_2)$ are *equal* ($\hat{T}_1 = \hat{T}_2$) if there exists a bijective mapping $f : V_1 \to V_2$ such that the following two properties hold

$$(v_1, v_2) \in A_1 \Leftrightarrow (f(v_1), f(v_2)) \in A_2$$

$$\mathit{label}(v) = \mathit{label}(f(v)), \forall v \in V_1.$$

A *subtree repeat* R in a tree T is a set of node tuples $(u_1, v_1), \ldots, (u_{|R|}, v_{|R|})$, such that $\hat{T}(u_1, v_1) = \ldots = \hat{T}(u_{|R|}, v_{|R|})$. We call $|R|$ the repetition *frequency* of R. If $|R| = 1$ we say that the particular subtree does not repeat. An *overlapping* subtree repeat is a subtree repeat R, where at least one node v is contained in all $|R|$ trees. If no such v exists, we call it a *non-overlapping* subtree repeat. A *total repeat* R is a subtree repeat that contains all nodes in T, that is, $R = \{(u_1, u_1), \ldots, (u_{|R|}, u_{|R|})\}$. See Fig. 1 in this regard.

In the following, we consider the problem of computing all such subtree repeats of an unrooted tree T.

3 Algorithm

The algorithm works in two stages: the forward/non-overlapping stage and the backward/overlapping stage. The forward stage finds all non-overlapping subtree repeats of some tree T. The backward stage uses the identifiers assigned during the forward stage to detect all overlapping subtree repeats, including total repeats.

3.1 The Forward/Non-overlapping Stage

We initially present a brief description of the algorithmic steps. Thereafter, we provide a formal description of each step in Algorithm 1.

In the following, we identify each node in the tree by a unique integer in the range of 1 to $|T|$. Such a unique integer labeling can be obtained, for instance, by a pre- or post-order tree traversal.

The basic idea of the algorithm can be explained by the following steps:

1. Partition nodes by height.
2. Assign a unique identifier to each label in Σ.
3. For each height level starting from 0 (the leaves).
   - i For each node v of the current height level construct a string containing the identifier of the label of v and the identifiers of the subtrees that are attached to v.
   - ii For each such string, sort the identifiers within the string.
   - iii Lexicographically sort the strings (for the current height level).
   - iv Find non-overlapping subtree repeats as identical adjacent strings in the lexicographically sorted sequence of strings.
   - v Assign unique identifiers to each set of repeating subtrees (equivalence class).

We will explain each step by referring to the corresponding lines in Algorithm 1.

Partitioning the nodes according to their height requires time linear with respect to the size of the tree, and is described in line 2 of Algorithm 1. This is done using an array H of queues, where $H[i]$, for all $0 \leq i \leq \lfloor d(T)/2 \rfloor$, contains all nodes of height i. Thereafter, we assign a unique identifier to each label in Σ in lines 3-7. The main loop of the algorithm starts at line 8 and processes the nodes at each height level starting bottom-up from the leaves towards the central points. The main loop consists of four steps. First, a string is constructed for each node v which comprises the identifier for the label at v followed by the identifiers assigned to $u_1, u_2, \ldots, u_{c_v}$. The identifiers $u_1, u_2, \ldots, u_{c_v}$ represent the subtrees $\hat{T}(u_1), \hat{T}(u_2), \ldots, \hat{T}(u_{c_v})$, where $u_1, u_2, \ldots, u_{c_v}$ are the children of v (lines 11-16). Assume that this particular step constructs k strings $s_1, s_2, \ldots, s_k$.

In the next step, we sort the identifiers within each string. To obtain this sorting in linear time, we first need to remap individual identifiers contained as letters in those strings to the range $[1, m]$. Here, m is the number of unique identifiers in the strings constructed for this particular height, and the following property holds: $m \leq \sum_{i=1}^{k} |s_i|$. We then apply a bucket sort to these remapped identifiers and reconstruct the ordered strings $r_1, r_2, \ldots, r_k$ (lines 17-20).

Algorithm 1. Forward-Stage

```
   Input  : Unrooted tree $T = (V, E)$ labeled from $\Sigma$
   Output: Sets $\mathcal{R}_{reps}$ of non-overlapping subtree repeats of $T$
1  ▷ Partition tree nodes by height
2  for all $v \in V$ do Enqueue($H[h(v)], v$)
3  $cnt \leftarrow 0$
4  ▷ Assign a number from 1 to $|\Sigma|$ to each label
5  for all labels $\ell \in \Sigma$ do
6      $cnt \leftarrow cnt + 1$
7      $L[\ell] \leftarrow cnt$
8  ▷ Compute subtree repeats
9  $reps \leftarrow 0$
10 for $i \leftarrow 0$ to $\lfloor d(T)/2 \rfloor$ do
11     $S \leftarrow \emptyset$
12     ▷ Construct a string of numbers for each node $v$ and its children
13     foreach $v \in H[i]$ do
14         Let $children(v) = \{\, u \mid \{u, v\} \in E \,\} \setminus \{parent(v)\}$ and $c_v = |children(v)|$
15         $s_v \leftarrow L[label(v)]K[u_1]K[u_2] \ldots K[u_{c_v}]$, if $children(v) = \{u_1, u_2, \ldots, u_{c_v}\}$
16         $S \leftarrow S \cup \{s_v\}$
17     ▷ Remap numbers $[1, |T| + |\Sigma|)$ to $[1, |H[i]| + \sum_{v \in H[i]} c_v]$
18     $R \leftarrow$ Remap($S$)
19     ▷ Bucket sort strings
20     Bucket sort the (unique) numbers of all strings in R.
21     Let $R'$ be the set of individually sorted strings that have been extracted
       from the respective sorted list from the previous step.
22     Lexicographically sort the strings in $R'$ using radix sort and obtain a sorted
       list $R''$ of strings $r_1, r_2, \ldots, r_{|R''|}$.
23     Let each $r_i$ be of the form $k_1^i k_i^2 \ldots k_{|r_i|}^i$ and the corresponding, original
       unsorted string $s_i$ of the form $L[v_1^i]K[v_2^i] \ldots K[v_{|r_i|}^i]$.
24     $reps \leftarrow reps + 1$
25     $\mathcal{R}_{reps} \leftarrow \{(v_1^1, parent(v_1^1))\}$
26     $K[v_1^1] \leftarrow reps + cnt$
27     for $j \leftarrow 2$ to $k$ do
28         if $r_j = r_{j-1}$ then
29             $\mathcal{R}_{reps} \leftarrow \mathcal{R}_{reps} \cup \{(v_1^j, parent(v_1^j))\}$
30         else
31             $reps \leftarrow reps + 1$
32             $\mathcal{R}_{reps} \leftarrow \{(v_1^j, parent(v_1^j))\}$
33         $K[v_1^j] \leftarrow reps + cnt$
```

The next step for the current height level is to find the subtree repeats as identical strings. To achieve this, we lexicographically sort the ordered strings $r_1, r_2, \ldots, r_k$ (line 22), and check neighboring strings for equivalence (lines 23-33). For each equivalence class $\mathcal{R}_i$ we choose a new, unique identifier, that is assigned to the root nodes of all the subtrees in that class (lines 26 and 33). Finally, each set $\mathcal{R}_i$ contains exactly the tuples of those nodes that are the roots of a particular non-overlapping subtree repeat of T and their respective parents.

Remapping from $\mathcal{D}_1 = [1, |T| + |\Sigma|)$ to $\mathcal{D}_2 = [1, |H[i]| + \sum_{v \in H[i]} c_v]$ can be done using an array A of size $|T| + |\Sigma|$, a counter m, and a queue Q. We read the numbers of the strings one by one. If a number x from domain $\mathcal{D}_1$ is read for the first time, we increase the counter m by one, set $A[x] := m$, and place m in Q. Subsequently, we replace x by m in the string. In case a number x has already been read, that is, $A[x] \neq 0$, we replace x by $A[x]$ in the string. When the remapping step is completed, only the altered positions in array A will be cleaned up, by traversing the elements of Q.

Theorem 1 (Correctness). *Given an unrooted tree T, Algorithm 1 correctly computes all non-overlapping subtree repeats.*

Proof. First note that if any two subtrees $\hat{T}_1$ and $\hat{T}_2$ are repeats of each other, they must, by definition, be of the same height. So the algorithm is correct in only comparing trees of the same height. Additionally, non-overlapping subtrees repeats of a tree T can only be of height $\lfloor d(T)/2 \rfloor$ or less, where $d(T)$ is the diameter of T. Therefore, the algorithm is correct in stopping after processing all $\lfloor d(T)/2 \rfloor + 1$ height classes, in order to extract all the non-overlapping subtree repeats. Since the algorithm only extracts non-overlapping repeats, we define repeats to mean non-overlapping repeats for the rest of this proof. In addition, for simplicity, we consider the rooted version of T for the rest of this proof.

We show that the algorithm correctly computes all repeats for a tree of any height by induction. For the base case we consider an arbitrary tree of height 1 (trees with height 0 are trivial). Any tree of height 1 only has the root node and any number of leafs attached to it. At the root we can never find a subtree repeat, so we only need to consider the next lower (height) level, that is, the leaf nodes. Any two leafs with identical labels will, by construction of the algorithm, be assigned the same identifiers and thus be correctly recognized as repeats of each other.

Now, assume that all (sub)trees of height $m - 1$ have correctly been assigned with identifiers by the algorithm *and* that they are identical for two (sub)trees *iff* they are unordered repeats of each other.

Consider an arbitrary tree of height $m+1$. The number of repeats for the tree spanned from the root (node r) is always one (the whole tree). Now consider the subtrees of height m. The root of any subtree of height m must be a child of r. For any child of r that induces a tree of height smaller than m, all repeats have already been correctly calculated according to our assumption.

Two (sub)trees are repeats of each other *iff* the two roots have the same label and there is a one-to-one mapping from subtrees induced by children of the root of one tree to topologically equivalent subtrees induced by children of the root of the second tree. By the induction hypothesis, all such topologically equivalent subtrees of height $m - 1$ or smaller have already been assigned identifiers that are unique for each equivalence class. Thus, deciding whether two subtrees are repeats of each other can be done by comparing the root labels and the corresponding identifiers of their children, which is exactly the process described in the algorithm. The approach used in the algorithm correctly identifies identically labeled strings since the order of identifiers has been sorted for a given height class. Thus the algorithm finds all repeats of height m (and $m+1$ at the root). □

Theorem 2 (Complexity). *Algorithm 1 runs in time and space $\mathcal{O}(|T|)$.*

Proof. We prove the linearity of the algorithm by analyzing each of the steps in the outline of the algorithm. Steps 1 and 2 are trivial and can be computed in $|T|$ and $|\Sigma|$ steps, respectively. Notice that $|\Sigma| \leq |T|$.

The main for loop visits each node of T once. For each node v a string s_v is constructed which contains the identifier of the label of v and the identifiers assigned to the child nodes of v. Thus, each node is visited at most twice: once as parent and once as child. This leads to $2n-1$ node traversals, where n is the number of nodes of T, since the root node is the only node that is visited exactly once. The constructed strings for a height level i are composed of the nodes in $H[i]$ and their respective children. In total we have $c(i) := \sum_{v \in H[i]} c_v$ child nodes at a height level i, where c_v is the number of children of node v. Therefore, the total size of all constructed strings for a particular height level i is $|H[i]| + c(i)$. Step 3ii runs in linear time with respect to the number of nodes at each height level i *and* their children. This is because the remapping is computed in linear time with respect to $|H[i]|+c(i)$. By the remapping, we ensure that the identifiers in each string are within the range of 1 to $|H[i]| + c(i)$. Using bucket sort we can then sort the remapped identifiers in time $|H[i]| + c(i)$ for each height level i. Consequently, the identifiers in each string can be sorted in time $|H[i]| + c(i)$ by traversing the sorted list of identifiers and positioning the respective identifier in the corresponding string on a first-read-first-place basis. This requires additional space $|H[i]| + c(i)$ to keep track which remapped identifier corresponds to which strings.

After remapping and sorting the strings, finding identical strings as repeats requires a lexicographical sorting of the strings. Strings that are identical form classes of repeats. Lexicographical sorting (using radix sort) requires time $\mathcal{O}(|H[i]| + c(i))$ and at most space for storing $|T| + |\Sigma|$ elements since the identifiers are in the range of 1 to $|T| + |\Sigma|$. This memory space needs to be allocated only once. Moreover, the elements that have been used are cleared/cleaned-up at each step via a queue as explained for the remapping function.

By summing over all height levels we obtain $\sum_{i=0}^{\lfloor d(T)/2 \rfloor} (|H[i]| + c(i)) = 2n-1$. Thus the total time over all height levels for each step described in the loop is $\mathcal{O}(|T|)$. The overall time and space complexity of the algorithm is thus $\mathcal{O}(|T|)$. □

We conclude this section with an example demonstrating Algorithm 1. Consider the tree T from Fig. 2. The superscript indices denote the number associated with each node, which, in this particular example, correspond to a pre-order traversal of $\hat{T}(T)$ by designating node 1 as the root. Lines 1-2 partition the nodes of T in $\lfloor d(T)/2 \rfloor + 1$ sets according to their height. The sets $H[0] =$

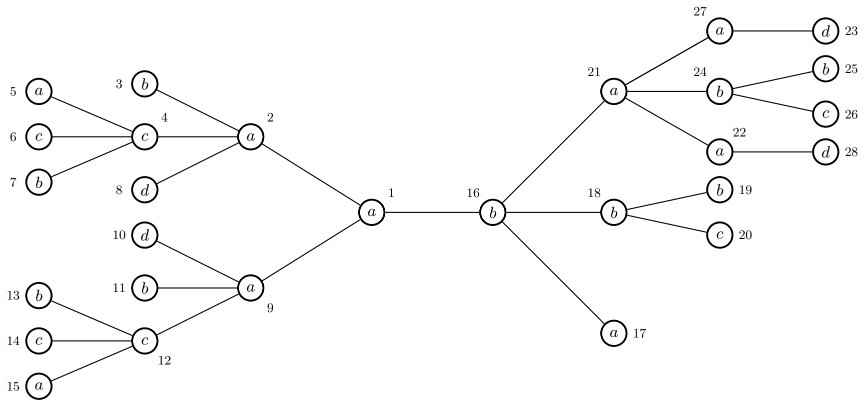


Fig. 2. Graphical representation of tree T. The superscript indices denote the unique identifier assigned to each node by traversing T

$\{3, 5, 6, 7, 8, 10, 11, 13, 14, 15, 17, 19, 20, 23, 25, 26, 28\}$, $H[1] = \{4, 12, 18, 22, 24, 27\}$, $H[2] = \{2, 9, 21\}$ and $H[3] = \{1, 16\}$ are created. Lines 5-7 create a mapping between labels and numbers. $L[a] = 1$, $L[b] = 2$, $L[c] = 3$, and $L[d] = 4$. Table 1 shows the state of lists S, R, R', R'' during the computation of the main loop of Algorithm 1 for each height level, where S is the list of string identifiers, R is

Table 1. State of lists S, R, R', R'' for each height level and resulting sets $\mathcal{R}_{reps}$ of non-overlapping subtree repeats

Height	Step	Process	Repeats
0	Strings: S	$2, 1, 3, 2, 4, 4, 2, 2, 3, 1, 1, 2, 3, 4, 2, 3, 4$	$\mathcal{R}_1 = \{3, 7, 11, 13, 19, 25\}$
	Remapping: R	$1, 2, 3, 1, 4, 4, 1, 1, 3, 2, 2, 1, 3, 4, 1, 3, 4$	$\mathcal{R}_2 = \{5, 15, 17\}$
	Sorting: R'	$1, 2, 3, 1, 4, 4, 1, 1, 3, 2, 2, 1, 3, 4, 1, 3, 4$	$\mathcal{R}_3 = \{6, 14, 20, 26\}$
	Repeats: R''	$\underbrace{1, 1, 1, 1, 1, 1}_{5}, \underbrace{2, 2, 2}_{6}, \underbrace{3, 3, 3, 3}_{7}, \underbrace{4, 4, 4, 4}_{8}$	$\mathcal{R}_4 = \{8, 10, 23, 28\}$
1	Strings: S	$3\ 6\ 7\ 5, 3\ 5\ 7\ 6, 2\ 5\ 7, 1\ 8, 2\ 5\ 7, 1\ 8$	
	Remapping: R	$1\ 2\ 3\ 4, 1\ 4\ 3\ 2, 5\ 4\ 3, 6\ 7, 5\ 4\ 3, 6\ 7$	$\mathcal{R}_7 = \{22, 27\}$
	Sorting: R'	$1\ 2\ 3\ 4, 1\ 2\ 3\ 4, 3\ 4\ 5, 6\ 7, 3\ 4\ 5, 6\ 7$	$\mathcal{R}_5 = \{4, 12\}$
	Repeats: R''	$\underbrace{1\ 2\ 3\ 4, 1\ 2\ 3\ 4}_{9}, \underbrace{3\ 4\ 5, 3\ 4\ 5}_{10}, \underbrace{6\ 7, 6\ 7}_{11}$	$\mathcal{R}_6 = \{18, 24\}$
2	Strings: S	$1\ 5\ 9\ 8, 1\ 8\ 5\ 9, 1\ 11\ 10\ 11$	
	Remapping: R	$1\ 2\ 3\ 4, 1\ 4\ 2\ 3, 1\ 5\ 6\ 5$	$\mathcal{R}_8 = \{2, 9\}$
	Sorting: R'	$1\ 2\ 3\ 4, 1\ 2\ 3\ 4, 1\ 5\ 5\ 6$	$\mathcal{R}_9 = \{21\}$
	Repeats: R''	$\underbrace{1\ 2\ 3\ 4, 1\ 2\ 3\ 4}_{12}, \underbrace{1\ 5\ 5\ 6}_{13}$	
3	Strings: S	$2\ 6\ 10\ 13, 1\ 12\ 12$	
	Remapping: R	$1\ 2\ 3\ 4, 5\ 6\ 6$	$\mathcal{R}_{10} = \{16\}$
	Sorting: R'	$1\ 2\ 3\ 4, 5\ 6\ 6$	$\mathcal{R}_{11} = \{1\}$
	Repeats: R''	$\underbrace{1\ 2\ 3\ 4}_{14}, \underbrace{5\ 6\ 6}_{15}$	

the list of remapped identifiers, R' is the list of individually sorted remapped identifiers, and R'' is the list R' lexicographically sorted. Fig. 3 depicts tree T with the respective identifiers for each node as assigned by Algorithm 1.

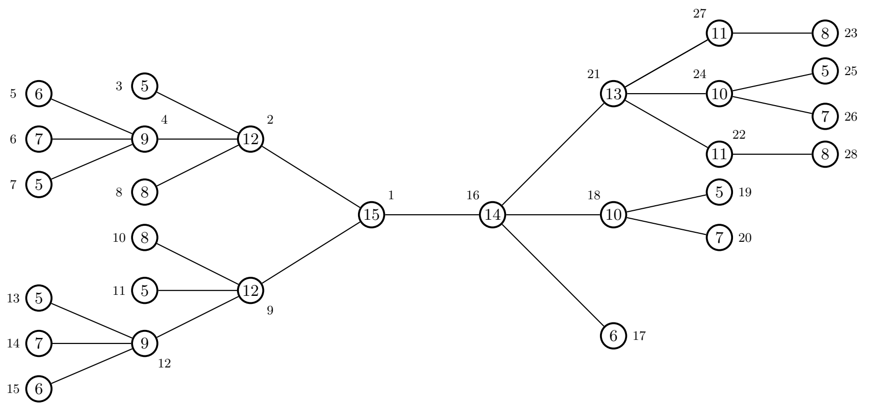


Fig. 3. Graphical representation of tree T with the associated identifier for each node as assigned by Algorithm 1

3.2 The Backward/Overlapping Stage

Definition 1 (Sibling repeat). *Given an unrooted tree T, two equal subtrees of $\hat{T}(T)$ whose roots have the same parent are called a* sibling repeat*.*

Definition 2 (Child repeat). *Given an unrooted tree T, two subtrees of $\hat{T}(T)$ whose root's have the same identifiers and whose root's respective parents are roots of trees in the same sibling or child repeat, are called a* child repeat*.*

Note that with these definitions we get that two trees with roots u and v respectively, are child or sibling repeats of each other *iff* the unique path between nodes u and v is symmetrical with respect to the node labels of the nodes traversed on the path. Also note, that child repeats and sibling repeats can occur in the same repeat class; it is merely a property shared between two (or more) trees.

The two following lemmas illustrate why it is necessary and sufficient to know the identifiers from the forward stage to compute all overlapping subtree repeats.

Lemma 1 (Sufficient conditions). *Let r be the parent of u and v, where u and v are roots of a sibling repeat. Then the trees $\hat{T}(u, u)$ and $\hat{T}(v, v)$ are elements of the same total repeat. The trees $\hat{T}(r, u)$ and $\hat{T}(r, v)$ are elements of the same overlapping subtree repeat.*

Let u and v be roots of a child repeat. Further let r_u and r_v be the parents of u and v, respectively. Then the trees $\hat{T}(u, u)$ and $\hat{T}(v, v)$ are elements of the same total repeat, and the trees $\hat{T}(r_u, u)$ and $\hat{T}(r_v, v)$ are elements of the same overlapping subtree repeat.

Proof. Trivial, by inspection; see Fig. 2. □

In Fig. 2, the trees $\hat{T}(2,1)$ and $\hat{T}(9,1)$ form a sibling repeat, thus the trees $\hat{T}(4,2)$ and $\hat{T}(12,9)$ form a child repeat. From the sibling repeat, we get that $\hat{T}(2,2)$ and $\hat{T}(9,9)$ are elements of a total repeat, while $\hat{T}(1,2)$ and $\hat{T}(1,9)$ are within the same overlapping repeat. Analogously, for the child repeat we get the trees $\hat{T}(4,4)$ and $\hat{T}(12,12)$ as total repeats and $\{(2,4),(9,12)\}$ as an overlapping repeat.

Note that Lemma 1 implies that all nodes of a subtree that is element of an overlapping subtree repeat with repetition frequency $|R|$ are roots of trees in overlapping repeat classes of frequency at least $|R|$.

Lemma 2 (Necessary conditions). *Any two trees that are elements of a total repeat must have been assigned the same identifiers at their respective roots during the forward stage, and be rooted in roots of either sibling or child repeats.*

Any two trees that are elements of an overlapping subtree repeat, but not of a total repeat, must have been assigned the same identifiers at their respective roots during the forward stage, and be rooted in parents of roots of either sibling or child repeats.

Proof. We first look at the case of total repeats. Let $\hat{T}(u,u) = \hat{T}(v,v)$. We now consider the unique path p between u and v. Obviously, for equality among these two trees to hold, the path must be symmetrical, which by recursion implies that u and v are roots of either sibling or child repeats; see Fig. 4.

The case of other overlapping subtree repeats works just the same. Let $\hat{T}(r_u,u) = \hat{T}(r_v,v)$ not be total, but overlapping subtree repeat. These trees are obtained by removing a single edge from the tree: (r_u,u) and (r_v,v), respectively. Let p be the path between u and v. Since the trees are elements of an overlapping subtree repeat, r_u and r_v must lie on this path. Additionally, since r_u and r_v are on the path from u to v, $h(v) = h(u)$, and any tree is acyclic, then r_u and r_v must be closer to the central points than u and v, respectively. Since there is an edge connecting r_u with u and r_v with v this means that r_u and r_v are parents of u and v, respectively. Again, the path p is symmetrical with respect to the node labels of nodes along the path, so u and v are roots of either sibling or child repeats. □

Given these two lemmas, we can compute all overlapping subtree repeats by checking for sibling and child repeats. This can be done by comparing the identifiers assigned to nodes in the forward stage. The actual procedure of computing all overlapping subtree repeats is described in Algorithm 2. Algorithm 2 takes as input an unrooted tree T that has been processed by Algorithm 1; i.e. each node of tree T has already been assigned an identifier according to its non-overlapping repeat class.

First, the algorithm considers the rooted version of T, that is $\hat{T}(T)$. This is done since many operations and definitions rely on $\hat{T}(T)$. Next, we define a queue Q, whose elements are sets of nodes. Initially, Q contains only the set containing the root node of $\hat{T}(T)$ (line 2). Processing Q is done by dequeuing a single set of nodes at a time (lines 5-16). For a given set U of Q, the algorithm creates a set I containing the identifiers of children of all the nodes in U. Then, the algorithm remaps these identifiers to the range of $[1,|I|]$ constructing a new set

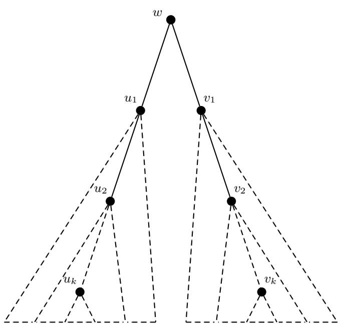


Fig. 4. $T(v_2, v_k) = T(u_2, u_k)$ is an overlapping repeat *iff* $T(u_k, u_2) = T(v_k, v_2)$ is a child repeat, which is true *iff identifier*(u_k) = *identifier*(v_k), *identifier*(u_2) = *identifier*(v_2), *identifier*(u_1) = *identifier*(v_1)

I' (line 12). Then, we construct a list C of tuples, such that each tuple contains the remapped identifier of a child and the corresponding node. Therefore, we can use bucket sort to sort these tuples by the remapped identifiers in time linear in the cardinality of I.

We are now in a position to apply Lemmas 1 and 2. By Lemma 2, finding sibling and child repeats is done by creating sets of nodes with equivalent identifiers in C (line 18). This can be easily done due to the sorting part of the algorithm. These sets are then enqueued in Q, and, by Lemma 1 and 2, all resulting subtree repeats (overlapping and total) are, thus, created (lines 21-22). Hence we immediately obtain the following result.

Theorem 3 (Correctness). *Given an unrooted tree T with identifiers assigned by Algorithm 1, Algorithm 2 correctly computes all overlapping subtree repeats, including total repeats.*

Algorithm 2 enqueues each node of T once. For each enqueued node, a constant number of operations is performed. Therefore we get the following result.

Theorem 4 (Complexity). *Algorithm 2 runs in time and space $\mathcal{O}(|T|)$.*

4 Final Remarks

We presented a simple and time-optimal algorithm for computing all full subtree repeats in unrooted unordered labeled trees; and showed that the running time of our method is linear with respect to the size of the input tree.

The presented algorithm can easily be modified to operate on trees that do not satisfy some or any of the aforementioned assumptions on the tree structure.

- *Rooted trees*: In a rooted tree $\hat{T}$, only non-overlapping repeats can occur. Therefore it is sufficient to apply Algorithm 1 with the following modifications: first, we define $\hat{T}(\hat{T}) := \hat{T}$; second, the main for loop must iterate over the height of $\hat{T}$, instead of depending on its diameter.

Algorithm 2. Backward-Stage

```
Input  : Unrooted tree $T = (V, E)$ labeled from $\Sigma$ with identifiers assigned by
         Algorithm 1
Output: Sets $\mathcal{R}'_{reps}$ of overlapping subtree repeats of $T$
1  ▷ Initialize queue $Q$ with the root node $r$ of $\hat{T}(T)$
2  Enqueue($Q, \{r\}$)
3  ▷ Compute overlapping subtree repeats
4  while Queue-Not-Empty($Q$) do
5      $U \leftarrow$ Dequeue($Q$)
6      ▷ Get the identifiers of the children of the nodes in $U$
7      Let $cod(U)$ be the cumulated out degree of all the nodes in $U$
8      Let $children(U) = \{u_1, u_2, \dots, u_{cod(U)}\}$ be the children of the nodes in $U$
9      Let $ids(children(U)) = \{i_1, i_2, \dots, i_{cod(U)}\}$ be the identifiers of
       $\{u_1, u_2, \dots, u_{cod(U)}\}$
10     $I \leftarrow ids(children(U))$
11     ▷ Remap numbers $[1, |T| + |\Sigma|)$ to $[1, |I|]$
12     $I' \leftarrow$ Remap($I$)
13     Let $I' = \{i'_1, i'_2, \dots, i'_{cod(U)}\}$ be the remapped identifiers of
       $\{u_1, u_2, \dots, u_{cod(U)}\}$
14     Let $C =< i'_1, u_1 >, < i'_2, u_2 >, \dots, < i'_{cod(U)}, u_{cod(U)}) >$ be a list of tuples
15     ▷ Bucket sort the remapped identifiers
16     Bucket sort the list $C$ by $i'_1, i'_2, \dots, i'_{cod(U)}$.
17     ▷ Extract the equivalence classes
18     foreach $E = \{v_1, v_2, \dots, v_k\}$ of nodes with equivalent identifiers in $C$ do
19         Enqueue($Q, E$)
20         for $i \leftarrow 1$ to $k$ do
21             $\mathcal{R}'_{reps} \leftarrow \mathcal{R}'_{reps} \cup \{(parent(v_i), v_i)\}$
22             $\mathcal{R}'_{reps+1} \leftarrow \mathcal{R}'_{reps+1} \cup \{(v_i, v_i)\}$
23         $reps \leftarrow reps + 2$
```

- *Ordered trees*: If for a node the order of its adjacent nodes is relevant, i.e. the tree is ordered, the bucket sort procedures in Algorithms 1 and 2 must be omitted. Additionally, sibling repeats must not be merged in line 19 of Algorithm 2 but rather be enqueued separately.
- *Unlabeled trees*: Trivially, an unlabeled tree can be seen as a labeled tree with a single uniform symbol assigned to all nodes.

Algorithm 1 can also be used to compute subtree repeats over a *forest* of rooted unordered trees. The method is the same as for the case of a single tree. The method reports all subtree repeats by clustering the identifiers of equal subtrees from all trees in the forest into an equivalence class. The correctness of this approach can be trivially obtained by connecting the roots of all trees in the forest with a virtual root node, and applying the algorithm to this single tree.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: principles, techniques, and tools, 2nd edn. Addison-Wesley (2006)
2. Barstow, D.R., Shrobe, H.E., Sandewall, E.: Interactive Programming Environments. McGraw-Hill, Inc. (1984)
3. Chor, B., Tuller, T.: Finding a maximum likelihood tree is hard. Journal of ACM 53(5), 722–744 (2006)
4. Christou, M., Crochemore, M., Flouri, T., Iliopoulos, C.S., Janoušek, J., Melichar, B., Pissis, S.P.: Computing all subtree repeats in ordered trees. Information Processing Letters 112(24), 958–962 (2012)
5. Christou, M., Crochemore, M., Flouri, T., Iliopoulos, C.S., Janoušek, J., Melichar, B., Pissis, S.P.: Computing all subtree repeats in ordered ranked trees. In: Grossi, R., Sebastiani, F., Silvestri, F. (eds.) SPIRE 2011. LNCS, vol. 7024, pp. 338–343. Springer, Heidelberg (2011)
6. Felsenstein, J.: Evolutionary trees from DNA sequences: a maximum likelihood approach. Journal of Molecular Evolution 17(6), 368–376 (1981)
7. Felsenstein, J.: Inferring phylogenies. Sinauer Associates (2003)
8. Ferdinand, C., Seidl, H., Wilhelm, R.: Tree automata for code selection. Acta Inf. 31, 741–760 (1994)
9. Guindon, S., Dufayard, J.F., Lefort, V., Anisimova, M., Hordijk, W., Gascuel, O.: New Algorithms and Methods to Estimate Maximum-Likelihood Phylogenies: Assessing the Performance of PhyML 3.0. Systematic Biology 59(3), 307–321 (2010)
10. Harary, F.: Graph Theory. Addison Wesley Publishing Company (1994)
11. Hoffmann, C.M., O'Donnell, M.J.: Programming with equations. ACM Trans. Program. Lang. Syst. 4, 83–112 (1982)
12. Hudak, P.: Conception, evolution, and application of functional programming languages. ACM Computing Surveys 21, 359–411 (1989)
13. Knuth, D.E., Bendix, P.B.: Simple word problems in universal algebra. In: Leech, J. (ed.) Computational Problems in Abstract Algebra, pp. 263–297. Pergamon Press (1970)
14. Mauri, G., Pavesi, G.: Algorithms for pattern matching and discovery in RNA secondary structure. Theoretical Computer Science 335(1), 29–51 (2005)
15. Stamatakis, A.: RAxML-VI-HPC: maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models. Bioinformatics 22(21), 2688–2690 (2006)
16. Yang, Z.: Computational Molecular Evolution. Oxford University Press, Oxford (2006)

Approximation Bounds on the Number of Mixedcast Rounds in Wireless Ad-Hoc Networks*

Sang Hyuk Lee and Tomasz Radzik

Department of Informatics, King's College London, United Kingdom
{sang_hyuk.lee,tomasz.radzik}@kcl.ac.uk

Abstract. We consider the following type of Maximum Network Lifetime problems. For a wireless network N with given capacities of node batteries, and a specification of a communication task which is to be performed periodically in N, find a maximum-size feasible collection of routing topologies for this task. Such a collection of routing topologies defines the maximum number of rounds this task can be performed before the first node in the network dies due to battery depletion. The types of communication tasks which we consider are unicast, broadcast, convergecast and mixedcast. The mixedcast is the requirement that some fixed number of tasks of the basic types (unicast, broadcast, convergecast) are periodically performed. We show that one can compute in polynomial time the number k of mixedcast rounds which is at least $\lfloor k_{opt}/5 \rfloor$, for the single-topology variant of the problem, and at least $\lfloor k_{opt}/6 \rfloor$, for the multiple-topology variant, improving the previous bounds.

Keywords: Network Lifetime, Wireless Networks, Approximation Algorithm, Broadcast, Convergecast.

1 Introduction

Unicast, *broadcast* and *convergecast* are the fundamental communication tasks in wireless ad-hoc networks. Unicast is one-to-one communication, where information held in one node (the *source*) is transmitted to another node (the *destination*), possibly via intermediate nodes. Broadcast is one-to-all communication, where information held in one node (the source) is transmitted to all other nodes. Convergecast can be viewed as the opposite to broadcast: information held in every node is transmitted to one specified node (the sink, or the destination). Many network operations and services such as information dissemination and data collection rely on these three communication tasks.

The nodes of a wireless ad-hoc network are often battery-powered, but are intended to operate over a long period of time. Typical applications for such

* Research partially supported by the EPSRC grant EP/J006300/1 "Random walks on computer networks" and the 2012 SAMSUNG GRO project "Fast Low Cost Methods to Learn Structure of Large Networks."

© Springer-Verlag Berlin Heidelberg 2013

networks include environmental monitoring and military surveillance, where replacing or recharging the batteries may not be easy, or even not possible at all. Therefore, an important design objective for communication algorithms is to optimise the energy efficiency, so that the network lifetime is maximised.

A wide range of optimization problems modelling the energy efficiency in adhoc wireless networks have been proposed. One general approach is to focus on a *single session*, aiming to minimize the energy used to complete one specified communication task [11, 23, 25]. The example of this approach is the Minimum Energy Broadcast problem [2, 22, 23, 25]. The other general approach is to consider *multiple sessions* with the aim of maximising the *lifetime* of the network, which could mean, for example, maximising the number of times that specified communication tasks can be repeated until the first node depletes its all energy [5, 16–20]. This second approach is typically employed in continuous monitoring applications, where periodic data gathering (convergecast) or reporting (unicast) have to be performed. For such applications, the first, "greedy" approach of optimizing only the current session may give sub-optimal solutions. This is because the network lifetime does not solely depend on the energy spent while performing a specified communication task, but also on the remaining battery capacity of the individual nodes.

In this paper we follow the second, "global" approach to energy efficient communication, and consider a class of *Maximum Network Lifetime* (MNL) problems [5, 17, 18]. A problem of this class is given by a specification of a network (node-to-node connections, communication costs, initial capacities of node batteries, etc.) and a specification of a communication task (e.g., a broadcast from a given node). This communication task is to be executed periodically, as many times as possible. We refer to one execution of this task as one communication round. The output is a collection of *routing topologies* such that each routing topology defines one execution of the specified communication task (one round). The objective is to maximise the number of communication rounds, that is, to maximise the network lifetime. The constraints are that every node must have sufficient battery capacity to participate in all rounds.

The structure of a routing topology depends on the type of a communication task. For broadcast and convergecast, a routing topology is a rooted directed spanning tree, with the edges directed away from the root for broadcast, or towards the root for convergecast. Such trees are referred to as *broadcast trees* and *convergecast trees.* Each communication round is defined by one routing tree. A routing topology for unicast is a simple path from the source to the destination.

The MNL problems can be categorized according to various characteristics. For example, both *fractional* and *discrete* MNL problems have been considered [5, 19]. In the fractional variant, a message is allowed to be divided into smaller messages, which can be transmitted separately and possibly along different routes. Whereas in the discrete variant, each message has to be sent in one transmission. The discrete variant seems to reflect better the existing network protocols, which often treat data packets as unsplitable units of transmission.

Another categorization of MNL problems is related to the type of antenna used for communication. In the *omnidirectional* model, each node has a 360-degree coverage. This means that if a node u transmits a message with enough power to reach another node at distance d, then all nodes which are located within distance d in any direction are also able to receive the same message. In the *directional* antenna model, each node has a limited angular coverage. Only the nodes which are located within distance d in that direction are able to receive the message. The special case of the directional antenna model is the *unidirectional* antenna model, called also the *single-recipient* model [16–18, 20]: each transmission is directed to a single node.

In this paper we consider the discrete variant of the MNL problems under the unidirectional antenna model and with full *aggregation* of messages. This last condition means that each node of a convergecast tree T waits until it receives messages from all its children. Then it combines all messages and sends them together as one message in one transmission to the parent. We focus on *mixedcast*, which is a combination of unicast, broadcast and convergecast. The mixedcast MNL problem was introduced in [17] as a problem of designing the maximum number of communication rounds such that each round consists of τ broadcasts and γ convergecasts, where τ and γ are given non-negative integers. We will follow this definition, but note that our method can be also applied to a *generalized mixedcast*, when all three types of communication tasks can be combined, and more than one task of each type can be specified. For example, we might require that each round consists of two broadcasts from each of the source nodes $r_1, r_2, \ldots, r_q$ and one convergecast to the destination node r_0.

The MNL problems can be also categorised into *single topology* and *multiple topology* problems [18], which are distinguished by different output requirements. In the single topology variant, the same routing topology is used for the same task in all communication rounds (for example, the same broadcast tree is used in each round to broadcast from the source node r_1). In the multiple topology variant, the routing topologies can be different in different rounds.

We consider both single and multiple topology MNL problems. Thus the following eight problems (and their acronyms) are relevant in this paper: Single Topology and Multiple Topology Unicast (STU and MTU), Broadcast (STB and MTB), Convergecast (STC and MTC) and Mixedcast (STM and MTM).

1.1 Previous Work

The previous studies of the topic of maximising the lifetime of ad-hoc wireless networks have been considering mainly the omnidirectional communication model [9, 12, 13, 18, 19, 24]. Kang and Poovendran [9] investigate the fractional variants of the maximum network lifetime problem for broadcast communication, proposing a polynomial time algorithm for the STB problem and some heuristics for the MTB problem. Orda and Yassour [18] improve the time complexity of the STB and STU problem, prove that the MTB problem is NP-hard, and propose additional MTB heuristics. Segal [20] further improves the running time of the STB problem. Additional results related to the maximum network lifetime problem under broadcast communication can be found in [3, 14, 19]. Kalpakis *et al.* [8] consider the fractional variants of the MTC problem with full aggregation, giving a polynomial time algorithm, but the polynomial bound is of high-degree. For the same problem, Standford and Tongngam [21] give $(1-\epsilon)$-approximation algorithm with a considerably faster running time. Wu *et al.* [24] consider the convergecast problem with full aggregation and propose an online approximation algorithm, which is based on Fürer and Raghavachari [6] algorithm for the minimum-degree Steiner tree problem. Lin *et al.* [13] extend [24] to a more general model with adjustable transmission power levels. Liang and Liu [12] propose heuristics for the online maximum network problem for convergecast.

Orda and Yassour [18] were the first to consider the complexity of the MNL problem in *unidirectional* communication model. Under this model, they show that the fractional variant of the STB problem is NP-hard, and propose a polynomial time algorithm for the fractional variant of the MTB problem.

We consider now the MNL problems in the unidirectional discrete model. It is not difficult to show that in this model the STU and STC problems can be solved in polynomial time. Segal [20] shows that we can actually get a linear time algorithm for the STC problem (and a similar algorithm works for the STU problem). Bodlaender *et al.* [1] show that the decision variant of the multiple topology unicast (MTU) problem is *strongly* NP-complete and APX-hard. A simple reduction from MTU to MTC implies that MTC is also strongly NP-complete and APX-hard. Elkin *et al.* [5] show that the broadcast problems STB and MTB are NP-hard. Actually, the special case of the MNL broadcast problems which asks whether a given network allows one broadcast is already NP-complete (a simple reduction from the Hamiltonian path problem).

Elkin *et al.* [5] give also an $\Omega(1/\log n)$-approximation algorithm for the STB problem, under the assumption that k_{opt} (the maximal number of rounds) is appropriately large. Nutov [16] shows a constant-ratio approximation algorithm for the MTU problem, and Nutov and Segal [17] show constant-ratio approximation algorithms for the STB, MTB and MTC problems, if k_{opt} is appropriately large. They also show that the MTC problem admits a 1/31-approximation polynomial time algorithm. In [10], we further improve approximation ratios for the STB, MTB and MTC problems (and a simple reduction from MTU to MTC implies also an improved approximation ratio for the MTU problem). The previous results for the discrete variants of the MNL problems under the unidirectional model are summarized in Table 1. The values in the table are the lower bounds on the computed number of rounds. If an algorithm computes at least $\lfloor k_{opt}/\beta \rfloor$ rounds, then we refer to the value β as the *approximation factor* of this algorithm.

A simple reduction shows that the broadcast problems STB and MTB can be viewed as special cases of the mixedcast problems STM and MTM, implying that the mixedcast problems are NP-hard. Nutov and Segal [17] show constant-ratio approximation algorithms for both STM and MTM problems. These approximation ratios can be improved by combining the mixedcast approximation framework from [17] with the approximation ratios for the broadcast and convergecast problems shown in [10]. The exact values of approximation ratios for the mixedcast problems are given in Table 2.

1.2 Our Contribution

We study the discrete variant of the *Maximum Network Lifetime* problems for unicast, broadcast, convergecast and mixedcast, under the unidirectional antenna model. We consider both the single and multiple topology variants of these problems. We improve the approximation factors for the mixedcast problems shown in, or implied by, [10, 16, 17] to the values given in the theorem below (see also Table 2).

Theorem 1. *For each of the single and multiple topology mixedcast problems STM and MTM, there exists a polynomial time algorithm, which finds a solution with k rounds, such that $k \geq \lfloor k_{opt}/\beta \rfloor$, where:*

- *$\beta = 5$, for the STM problem;*
- *$\beta = 6$, for the MTM problem.*

The approximation algorithms for the mixedcast problems proposed in [17] are based on approximation algorithms for the broadcast and convergecast problems. First the battery capacity at each node is split into two parts, in the same fixed ratio, with one part designated to support broadcasts and the other convergecasts. Then broadcast and convergecast trees are computed separately using approximation broadcast and convergecast algorithms. If β_B and β_C are the approximation factors of the algorithms for the MNL broadcast and convergecast problems, respectively, then the resulting algorithm for the MNL mixedcast problem has the approximation factor of $\beta_B + \beta_C$. For example, the approximation factor of 10 for the MTM problem in the column "[17] + [10]" in Table 2 is the sum of the approximation factors 6 and 4 for the MTB and MTC problems.

We improve the $\beta_B + \beta_C$ approximation factor for the mixedcast problems by replacing the fixed split of the battery capacities with a computed adaptive split. In our algorithm a node which turns out to be more important for broadcasts than for convergecasts will have more battery capacity (possibly the whole capacity) designated for broadcasts. The approximation factor of our mixedcast algorithm is equal to $\max\{\beta_B, \beta_C\}$. As mentioned earlier, our algorithm extends to the generalized mixedcast problem.

2 Notation and Preliminaries

Graph Preliminaries. For a directed graph $G = (V, E)$ and a node $v \in V$, let $\delta_G^{out}(v) = \delta_E^{out}(v)$ be the set of edges out-going from v, and let $\delta_G^{in}(v) = \delta_E^{in}(v)$ be the set of edges in-coming to v. For $F \subseteq E, \delta_F^{out}(v)$ is the set of edges in F out-going from v. We define $\delta_F^{in}(v)$ analogously. For $S \subseteq V$, let $\delta_G^{out} = \delta_E^{out}(S)$ be the set of edges outgoing from the set S, that is,

$$\delta_G^{out}(S) = \{(v, u) : v \in S \text{ and } u \in V \backslash S\}.$$

Table 1. Lower bounds on the number of rounds computed by the previous poly-time approximation algorithms for the discrete MNL problems

	unicast	convergecast	broadcast
Single topology	k_{opt}	k_{opt}	$\lfloor k_{opt}/25 \rfloor$, [17] $\lfloor k_{opt}/5 \rfloor$, [10]
Multiple Topology	$\lfloor k_{opt}/16 \rfloor$, [16] $(1/31)k_{opt}$, [16] $\lfloor k_{opt}/4 \rfloor$, [10]	$\lfloor k_{opt}/16 \rfloor$, [17] $(1/31)k_{opt}$, [17] $\lfloor k_{opt}/4 \rfloor$, [10]	$\lfloor k_{opt}/36 \rfloor$, [17] $\lfloor k_{opt}/6 \rfloor$, [10]

Table 2. Previous results and new contributions for the mixedcast problems

	[17]	[17] + [10]	this paper
Single Topology Mixedcast	$\lfloor k_{opt}/36 \rfloor$	$\lfloor k_{opt}/6 \rfloor$	$\lfloor k_{opt}/5 \rfloor$
Multiple Topology Mixedcast	$\lfloor k_{opt}/100 \rfloor$	$\lfloor k_{opt}/10 \rfloor$	$\lfloor k_{opt}/6 \rfloor$

A directed graph G is said to be *k-edge-outconnected from a node r*, if there are k-edge-disjoint paths from r to each node in G. A directed graph G is said to be *k-edge-inconnected to a node r*, if there are k-edge-disjoint paths from every node to r. It is well known that there are k edge-disjoint paths from a node r to each nodes in G (from each node in G to r), if and only if, $\delta_G^{in}(S) \geq k$ ($\delta_G^{out}(S) \geq k$), for every subset $\emptyset \neq S \subseteq V \backslash \{r\}$.

An *out-arborescence* (broadcast tree) T_{out} is a directed spanning tree that has a unique path from a root r to every node in V. An *in-arborescence* (convergecast tree) T_{in} is a directed spanning tree that has a path from every node to the root r. An *arborescence* refers to either out-arborescence or in-arborescence, depending on the context.

Model. We consider a wireless ad-hoc network N consisting of n stationary nodes. Each node v is equipped with a unidirectional antenna, which only permits a single node to receive a transmitted message. Each node v has a finite amount of battery capacity. Let $\mathbb{R}_+$ denote the set of non-negative real numbers.

Definition 1. *A (static) wireless ad-hoc network $N = (V, E, w, B)$ is a weighted, directed graph (V, E), where V is a set of nodes with $|V| = n$, $E \subseteq V \times V$ is a set of directed edges, $w : E \to \mathbb{R}_+$ is an edge-weight function representing energy cost of transmissions, and $B : V \to \mathbb{R}_+$ is a battery capacity function.*

In the network N, a directed edge (u, v) exists, if node u is able to directly transmit a message to node v. The edge-weight $w(u, v)$ of this edge denotes the amount of energy consumed to transmit one message from node u to node v. The edge weights are part of the input and we do not assume that they are related to distances between the nodes (the network is not embedded in any geometric space). The battery capacity $B(v)$ denotes the current battery power of node v. We allow the initial battery capacities to be different at different nodes.

In our model, we take into account only the energy consumption of transmissions, assuming that in wireless networks the radio frequency transmission dominates the energy usage. In particular, we do not consider energy consumption for receiving and processing data. We assume that all nodes share the same frequency band and the MAC layer is based on a "collision-free" protocol, so transmissions do not interfere with each other. For the convergecast problem, we additionally assume that the messages from different nodes can always be aggregated into one message. and then aggregate them into one message and send The properties of our model imply that it suffices to consider trees as routing topologies for broadcast and convergecast, and paths for unicast.

Definitions of the Problems. The *Maximum Network Lifetime* (MNL) problem for unicast, broadcast and convergecast is defined as follows. The input to the problems is a network $N = (V, E, w, B)$, and a node $r \in V$ (for broadcast and convergecast) or two nodes $r, s \in V$ (for unicast). The output is a collection of routing topologies $\mathcal{R} = \{R_1, ..., R_k\}$ for the given communication task, which satisfy the following energy constraints.

$$\sum_{i=1}^{k} w(\delta_{R_i}^{out}(v)) \;=\; \sum_{i=1}^{k} \sum_{e \in \delta_{R_i}^{out}(v)} w(e) \;\le\; B(v), \qquad \text{for all } v \in V. \tag{1}$$

The left-hand side of (1) is the total energy used by node v over k communication rounds, when the ith round is done according to the routing topology R_i. The objective of the problem is to maximise k. In the single topology variant, the same routing topology R is employed for all k rounds, i.e. $R_i = R$ for all $i \le k$. In this case, the constraints (1) simplify to the following constraints.

$$k \cdot \sum_{e \in \delta_{R}^{out}(v)} w(e) \;\le\; B(v), \quad \text{for all } v \in V. \tag{2}$$

The MNL *mixedcast* problem is defined in the following way. We are given two positive integer parameters τ and γ, and the objective is to find the maximum integer k such that τk broadcast trees and γk convergecast trees can be performed whilst the energy constraints are satisfied. More formally, the input to this problem is a network $N = (V, E, w, B)$, two nodes $r_b, r_c \in V$, and two integers $\tau, \gamma \ge 1$. The output is a maximum integer k, a collection of τk broadcast trees $\mathcal{T}_{out} = \{T'_1, ..., T'_{\tau k}\}$ rooted at node r_b and a collection of γk convergecast trees $\mathcal{T}_{in} = \{T''_1, ..., T''_{\gamma k}\}$ rooted at node r_c, that satisfy the following energy constraints.

$$\sum_{i=1}^{\tau k} \sum_{e \in \delta_{T'_i}^{out}(v)} w(e) \;+\; \sum_{i=1}^{\gamma k} \sum_{e \in \delta_{T''_i}^{out}(v)} w(e) \;\le\; B(v), \qquad \text{for all } v \in V. \tag{3}$$

In the single topology variant, we need to find one convergecast tree T_{in} and one broadcast tree T_{out} that are feasible for γk convergecast rounds and τk broadcast rounds. Hence, the energy constraints (3) simplify to:

$$\tau k \cdot \sum_{e \in \delta^{out}_{T_{out}}(v)} w(e) \; + \; \gamma k \cdot \sum_{e \in \delta^{out}_{T_{in}}(v)} w(e) \; \leq \; B(v), \qquad \text{for all } v \in V. \qquad (4)$$

3 Decision Versions of the MNL Problems

Nutov and Segal [17] proposed approximation algorithms for the MNL problems which first find a good value of k using binary search. When a good value of k is identified, a full solution (a collection of routing topologies) is computed using Nutov's bi-criteria algorithm for a certain type of network design problems [15]. We considered the same approach in [10], simplifying the analysis and deriving better approximation factors for the STB, MTB and MTC problems.

We continue discussion focusing on the multiple topology broadcast, convergecast and mixedcast problems. The single topology case is similar, with simpler details. The decision versions MTB(k) and MTC(k) of the multiple topology broadcast and convergecast problems are to compute feasible solutions (collections of trees) for the given number of rounds k. Some help in solving this decision problems comes from Edmonds' theorem [4] stated below.

Theorem 2. *[4] Let $H = (V, E)$ be a directed graph with a specified root $r \in V$. The graph H contains k edge-disjoint spanning out-arborescences (in-arborescences) rooted at r, if and only if, H is k-edge-outconnected from r (k-edge-inconnected to r). Moreover, there is a polynomial time algorithm that computes such k disjoint arborescences, if they exist.*

In the context of the MTB(k) and MTC(k) problems, Edmonds' theorem is used in the following way. Since the routing topologies used in different rounds do not have to be edge-disjoint, we consider a multigraph G_k instead of the input graph G. The multigraph $G_k = (V, E_k)$ is obtained from G by replacing each edge with its k copies. If we are solving MTB(k), then we first find a k-edge-outconnected subgraph H of G_k which satisfies energy constraints (the way to do this is discussed later). Then we apply Edmonds' theorem to H to retrieve k out-arborescences from H. These will be k broadcast trees in G, which satisfy the energy constraints.

As mentioned in Section 2, a graph H is k-edge-outconnected (contains k edge-disjoint paths from node r to each node), if and only if, $\delta^{in}_H(S) \geq k$ for every subset $\emptyset \neq S \subseteq V \backslash \{r\}$. Therefore, the MTB($k$) problem reduces (via Edmonds' theorem) to the following integer program $P^{MTB}_{IP}(k, B)$ with variables $x(e), e \in E$, where E here is the edge set E_k of the multigraph G_k:

$$\begin{aligned} x(\delta^{in}_E(S)) &\geq k, && \text{for all } \emptyset \neq S \subseteq V \backslash \{r\}, && (C) \\ \sum_{e \in \delta^{out}_E(v)} x(e) w(e) &\leq B(v), && \text{for all } v \in V, && (W) \\ x(e) &\in \{0, 1\}, && \text{for all } e \in E. && (B) \end{aligned}$$

In the above formulation, for a subset of edges $F \subseteq E$, $x(F) = \sum_{e \in F} x(e)$. The MTC($k$) problem leads to a similar integer program, but with the cut constraints (C) replaced with

$$x(\delta_E^{out}(S)) \geq k, \qquad \text{for all } \emptyset \neq S \subseteq V \backslash \{r\}.$$

We refer to the resulting integer program as $P_{IP}^{MTC}(k, B)$.

Nutov [15] showed a polynomial-time approximation algorithm for a type of network design problems, which includes these two integer programs. This is summarised in the theorem below.

Theorem 3. *[15] For each integer program $P_{IP}^{MTB}(k, B)$ and $P_{IP}^{MTC}(k, B)$, there exists a polynomial time algorithm, which computes one of the following two outcomes.*

1. *The algorithm correctly determines that the corresponding LP polytope (the polytope of the LP relaxation) is empty.*
2. *If the LP-polytope is not empty, then the algorithm finds a k-edge-outconnected spanning subgraph H (for the MTB IP problem), or a k-edge-inconnected spanning subgraph H (for the MTC IP problem), which violates the energy constraints (1) by at most a factor of β, that is,*

$$\sum_{e \in \delta_H^{out}(v)} w(e) \;\leq\; \beta \cdot B(v), \qquad \textit{for all } v \in V, \tag{5}$$

where $\beta = 6$, for the MTB IP problem, and $\beta = 4$, for the MTC IP problem.

4 Mixedcast Problems

In this section, we discuss the mixedcast problems. Nutov and Segal [17] proposed solving the mixedcast problem by splitting the battery capacity at each node into two parts in proportion $\beta_B : \beta_C$, where β_B and β_C are the approximation factors of algorithms for the broadcast and convergecast problems, respectively. One part is used for the computation of broadcast trees and the other for convergecast trees. The approximation factor of the resulting algorithm is $\beta_B + \beta_C$. This approach, together with the approximation factors β_B and β_C derived in [10] (see Table 1) gives the following approximation bounds.

Lemma 1. *For the mixedcast problems, we can find in polynomial time solutions with values $k \geq \lfloor k_{opt}/\beta_M \rfloor$, where $\beta_M = 6$, for the STM problem, and $\beta_M = 10$, for the MTM problem.*

Proof. For the sake of simplicity, we give a proof only for the multiple topology mixedcast (MTM) problem, but note that the single topology (STM) problem can also be proved in a similar fashion.

Let β_B and β_C denote the values of β for the multiple topology broadcast and convergecast problems, that is, $\beta_B = 6$ and $\beta_C = 4$. Let B_M denote the node's battery capacity function of the MTM problem and let k^* denote our solution for the mixedcast problem. The algorithm works as follows. We apply the MTB algorithm [10] for the root node r_b and the battery capacity function

$$B_B \;=\; \frac{\beta_B}{\beta_B+\beta_C} B_M.$$

As a result, we compute a collection of broadcast trees $\mathcal{T}_\mathcal{B} = \{T_1, T_2, ..., T_{k_B^*}\}$ rooted at node r_b, which use at most $B_B(v)$ amount of energy at each node v. Similarly, we apply the MTC algorithm [10] for the root node r_c and the battery capacity function

$$B_C \;=\; \frac{\beta_C}{\beta_B+\beta_C} B_M.$$

We get a collection of convergecast trees $\mathcal{T}_\mathcal{C} = \{T_1, T_2, ..., T_{k_C^*}\}$ rooted at node r_c, which use at most $B_C(v)$ amount of energy at each node v. Hence, the total energy used by the k_B^* broadcast trees and k_C^* convergecast trees is bounded by the initial battery capacity B_M, i.e, all these trees together sastify the energy constraints (1). Our solution to the mixedcast problem MTM is

$$k^* = \min\left\{ \left\lfloor \frac{k_B^*}{\tau} \right\rfloor, \left\lfloor \frac{k_C^*}{\gamma} \right\rfloor \right\},$$

and the first $T_1, T_2, ..., T_{\tau k^*}$ broadcast trees from $\mathcal{T}_\mathcal{B}$ and the first $T_1, T_2, ..., T_{\gamma k^*}$ convergecast trees from $\mathcal{T}_\mathcal{C}$.

Now we show that $k^* \geq \lfloor k_{opt}/\beta_M \rfloor$, where k_{opt} is the optimal value of k for the multiple topology mixedcast (MTM) problem. Note that $\beta_M = \beta_B + \beta_C$. We note that k is feasible for the MTB (MTC) problem, if and only if, the integer program $P_{IP}^{MTB}(k, B)$ $\left(P_{IP}^{MTC}(k, B)\right)$ is not empty.

The algorithm for the multiple topology mixedcast problem always returns a feasible solution: either $k^* = 0$, or $k^* > 0$, and a collection of broadcast trees and convergecast trees that are feasible for k^* rounds. This implies that if $k_{opt} = 0$, then $k^* = 0$. Assume now that $k_{opt} \geq 1$. This implies that τk_{opt} broadcast and γk_{opt} convergecast rounds can be performed within the battery capacity function B_M. This implies that the integer programs $P_{IP}^{MTC}(\gamma k_{opt}, B_M)$ and $P_{IP}^{MTB}(\tau k_{opt}, B_M)$ are not empty.

Lemma 2 (given below) implies that the following polytope

$$P_{LP}^{MTB}\left(\left\lfloor \frac{\tau \cdot k_{opt}}{\beta_B+\beta_C} \right\rfloor, \frac{B_M}{\beta_B+\beta_C} \right)$$

is not empty, so,

$$k_B^* \;\geq\; \left\lfloor \frac{\tau \cdot k_{opt}}{\beta_B+\beta_C} \right\rfloor \;\geq\; \tau \left\lfloor \frac{k_{opt}}{\beta_B+\beta_C} \right\rfloor.$$

Similarly, the following polytope

$$P_{LP}^{MTC}\left(\left\lfloor \frac{\gamma \cdot k_{opt}}{\beta_B+\beta_C} \right\rfloor, \frac{B_M}{\beta_B+\beta_C} \right),$$

is not empty, so,

$$k_C^* \ \geq \ \left\lfloor \frac{\gamma \cdot k_{opt}}{\beta_B + \beta_C} \right\rfloor \ \geq \ \gamma \left\lfloor \frac{k_{opt}}{\beta_B + \beta_C} \right\rfloor .$$

Hence,

$$k^* \ = \ \min\left\{ \left\lfloor \frac{k_C^*}{\gamma} \right\rfloor, \left\lfloor \frac{k_B^*}{\tau} \right\rfloor \right\} \ \geq \ \left\lfloor \frac{k_{opt}}{\beta_B + \beta_C} \right\rfloor = \left\lfloor \frac{k_{opt}}{\beta_M} \right\rfloor .$$

The following lemma is intuitive, though the formal proof (given in [10]) has to deal with the technicality that for different values of k, polytopes $P_{LP}^X(k, B)$ are defined on different (multi) graphs G_k.

Lemma 2. *If the polytope $P_{LP}^{MTB}(k, B)$ is not empty, then the polytope $P_{LP}^{MTB}(\lfloor k/\beta \rfloor, B/\beta)$ is also not empty, for any $\beta \geq 1$. The analogous property holds for the P_{LP}^{MTC} and P_{LP}^{STB} polytopes.*

To prove the better approximation factors given in Theorem 1, we need a new approach. We replace the fixed split of the battery capacities with a computed, more efficient partitioning, which does not have to be the same at all nodes. We discuss here only the multiple topology problem MTM.

Theorem 1. *(The MTM problem) For multiple topology mixedcast problem MTM, there exists a polynomial time algorithm, which finds a solution with k rounds, such that $k \geq \lfloor k_{opt}/\beta \rfloor$, where $\beta = 6$.*

Proof. Let $\beta_M = \max\{\beta_B, \beta_C\}$. Let $P_{IP}^{MTM}(k, B)$ be the following integer program, with variables $x(e)$ and $y(e), e \in E$, where E is the set E_k of edges in the multigraph $G_{\overline{k}}$, and $\overline{k} = \tau k + \gamma k$.

$$\begin{aligned}
x(\delta_E^{in}(S)) &\geq \tau k, && \text{for all } \emptyset \neq S \subseteq V \backslash \{r\}, \\
y(\delta_E^{out}(S)) &\geq \gamma k, && \text{for all } \emptyset \neq S \subseteq V \backslash \{r\}, \\
\sum_{e \in \delta_E^{out}(v)} (x(e) + y(e)) w(e) &\leq B(v), && \text{for all } v \in V, \\
x(e), y(e) &\in \{0, 1\}, && \text{for all } e \in E.
\end{aligned}$$

The discussion in Section 3 implies that there is a k-round feasible solution for MTM, if and only if, this integer program is feasible. We denote the LP-relaxation of the above IP by $P_{LP}^{MTM}(k, B)$. Let k' be the largest integer k such that the polytope $P_{LP}^{MTM}(k, B/\beta_M)$ is not empty.

The algorithm for MTM first finds k' using binary search, solving in each iteration a linear relaxation of the current integer program. If $k' = 0$, we output the empty collections of convergecast and broadcast trees. If $k' \geq 1$, then the algorithm also finds a feasible solution $\langle \overline{x}(e) \rangle_{e \in E}$ and $\langle \overline{y}(e) \rangle_{e \in E}$ for the polytope $P_{LP}^{MTM}(k', B/\beta_M)$. Let $B_{\overline{x}(v)}$ (resp., $B_{\overline{y}(v)}$) be the fraction of the battery capacity $B(v)/\beta_M$ which is used by the broadcast part $\langle \overline{x}(e) \rangle_{e \in E}$ of the solution (resp., by the convergecast part $\langle \overline{y}(e) \rangle_{e \in E}$ of the solution). That is,

$$B_{\overline{x}}(v) = \sum_{e \in \delta_E^{out}(v)} \overline{x}(e) w(e), \qquad B_{\overline{y}}(v) = \sum_{e \in \delta_E^{out}(v)} \overline{y}(e) w(e).$$

Now we apply the approximation algorithms of Theorem 3 to (nonempty) sets $P_{IP}^{MTB}(\tau k', B_{\overline{x}})$ and $P_{IP}^{MTC}(\gamma k', B_{\overline{y}})$. This way we compute a collection of broadcast trees $\mathcal{T}_{\mathcal{B}} = \{T'_1, .., T'_{\tau k'}\}$ rooted at node r_b, which use at most $\beta_B B_{\overline{x}}(v)$ energy at each node v, and a collection of convergecast trees $\mathcal{T}_{\mathcal{C}} = \{T''_1, .., T''_{\gamma k'}\}$ rooted at node r_c, which use at most $\beta_C B_{\overline{y}}(v)$ energy at each node v. The output of our MTM algorithm is $(k', \mathcal{T}_{\mathcal{B}}, \mathcal{T}_{\mathcal{C}})$. This output is feasible, because the energy usage at each node v is at most

$$\beta_B B_{\overline{x}}(v) \;+\; \beta_C B_{\overline{y}}(v) \;\le\; \beta_M (B_{\overline{x}}(v) \;+\; B_{\overline{y}}(v)) \;\le\; \beta_M (B(v)/\beta_M) \;=\; B(v).$$

We now show that

$$k' \ge \lfloor k_{opt}/\beta_M \rfloor. \tag{6}$$

If $k_{opt} = 0$, then $P_{LP}^{MTM}(k, B/\beta_M)$ is empty for each $k \ge 1$, so $k' = 0$ and (6) holds. Assume now that $k_{opt} \ge 1$. The set $P_{IP}^{MTM}(k_{opt}, B)$ is not empty, so the LP polytope $P_{LP}^{MTM}(k_{opt}, B)$ is not empty. Lemma 3 (below) implies that the LP polytope $P_{LP}^{MTM}(\lfloor k_{opt}/\beta_M \rfloor, B/\beta_M)$ is not empty, and k' has been computed as the largest integer k such that $P_{LP}^{MTM}(k, B/\beta_M)$ is not empty, so (6) holds also in this case.

The following lemma is analogous to Lemma 2.

Lemma 3. *If the polytope $P_{LP}^{MTM}(k, B)$ is not empty, then the polytope $P_{LP}^{MTM}(\lfloor k/\beta \rfloor, B/\beta)$ is also not empty, for any $\beta \ge 1$. The analogous property holds for the P_{LP}^{STM} polytopes.*

It should be clear that our approach can be extended to more general mixedcast problems mentioned in Section 1. If, for example, we require that convergecast, broadcast and unicast tasks are periodically performed in proportion $\gamma : \tau : \eta$, then we extend the integer program $P_{IP}^{MTM}(k, B)$ by adding variables $z(e)$, $e \in E$, and the cut constraints appropriate for the unicast problem. The approximation factor of the obtained algorithm is equal to $\max\{\beta_B, \beta_C, \beta_U\}$, which currently is equal to 6 (see Table 1). The mixedcast can also contain a number of different communication tasks of the same type. Since in our model we may have at most n different broadcast tasks, n different convergecast tasks and n^2 different unicast tasks, then also in this general case we have a polynomial time algorithm with the same approximation factor $\max\{\beta_B, \beta_C, \beta_U\}$.

We conclude with some remarks regarding the running times of the MNL algorithms discussed in this paper. We have described these algorithms in terms of the multigraph G_k, which may have exponential size, because the values of k may be exponential. To get polynomial running times, the multigraph G_k has to be replaced with an appropriate capacitated graph G, and the capacitated versions of Edmonds' and Nutov's theorems have to be used. More specifically, for example in the integer program $P_{IP}^{MTM}(k, B)$ used in the proof of Theorem 1, we would have one integral variable $x(e)$ for each edge e of the original graph G, with the bounds $0 \le x(e) \le k$. The capactitated version of Nutov's algorithm gives a directed capacitated spanning subgraph, in which for every node v there is an integral flow of value k from the root node r to v (if the corresponding polytope $P_{LP}^{MTM}(k, B)$ is not empty). Now Edmonds' theorem for uncapaciated graphs discussed in Section 3 cannot be applied. Instead, Gabow's algorithm [7] can be used to extract k trees from a directed capacitated graph in polynomial time.

Another technical issue is that the MNL algorithms have to solve linear programs, which have polynomial number of variables, but exponential number of constraints. Those LP's have, however, polynomial-time separation oracles, so they can be solved in polynomial times by the ellipsoid method. For example, for the current values $(x(e), y(e))_{e \in E}$ for $P_{LP}^{MTM}(k, B/\beta_M)$, finding a set $\emptyset \neq S \subseteq V \setminus \{r\}$ such that $x(\delta_E^{out}(S)) < \gamma k$ (a violated cut constraint) can be done in polynomial time by computing minimum cuts (P_v, Q_v) in G separating the root node $r_c \in P_v$ from a node $v \in Q_v$, for each node $v \in V$.

5 Conclusions

We considered the single and multiple topology discrete Maximum Network Lifetime (MNL) problems for unicast, broadcast, convergecast and mixedcast, under the unidirectional model. We have shown improved approximation algorithms for the problem of computing the maximum number of rounds in the mixedcast problems. Our approach also extends to more general mixedcast problems than considered previously.

We conclude with suggestions for some possible further research. One direction is to investigate whether the current approximation factors of MNL algorithms can be further improved. The approach which we follow is an iterative process, where in each iteration the LP-relaxation of an appropriate IP problem has to be solved, giving good (polynomial) theoretical running times, but high practical computational costs for larger networks. Hence, development of practical algorithms or heuristics for the MNL problems is worth further investigations. Another direction is to solve these problems by distributed algorithms. In this study we have considered only a centralized approach, in which a prior knowledge of the full network topology is assumed.

References

1. Bodlaender, H.L., Tan, R.B., van Dijk, T.C., van Leeuwen, J.: Integer maximum flow in wireless sensor networks with energy constraint. In: Gudmundsson, J. (ed.) SWAT 2008. LNCS, vol. 5124, pp. 102–113. Springer, Heidelberg (2008)
2. Clementi, A.E.F., Crescenzi, P., Penna, P., Rossi, G., Vocca, P.: On the complexity of computing minimum energy consumption broadcast subgraphs. In: Ferreira, A., Reichel, H. (eds.) STACS 2001. LNCS, vol. 2010, pp. 121–131. Springer, Heidelberg (2001)
3. Deng, G., Gupta, S.K.S.: maximising broadcast tree lifetime in wireless ad hoc networks. In: GLOBECOM (2006)
4. Edmonds, J.: Edge-disjoint branchings. In: Rustin, B. (ed.) Combinatorial Algorithms, pp. 91–96. Academic Press (1973)

5. Elkin, M., Lando, Y., Nutov, Z., Segal, M., Shpungin, H.: Novel algorithms for the network lifetime problem in wireless settings. Wireless Networks 17(2), 397–410 (2011)
6. Fürer, M., Raghavachari, B.: Approximating the minimum-degree steiner tree to within one of optimal. J. Algorithms 17(3), 409–423 (1994)
7. Gabow, H.N., Manu, K.S.: Packing algorithms for arborescences (and spanning trees) in capacitated graphs. Math. Program. 82, 83–109 (1998)
8. Kalpakis, K., Dasgupta, K., Namjoshi, P.: Efficient algorithms for maximum lifetime data gathering and aggregation in wireless sensor networks. Computer Networks 42(6), 697–716 (2003)
9. Kang, I., Poovendran, R.: maximising network lifetime of broadcasting over wireless stationary ad hoc networks. MONET 10(6), 879–896 (2005)
10. Lee, S.-H., Radzik, T.: Improved Approximation Bounds for Maximum Lifetime Problems in Wireless Ad-Hoc Network. In: Li, X.-Y., Papavassiliou, S., Ruehrup, S. (eds.) ADHOC-NOW 2012. LNCS, vol. 7363, pp. 14–27. Springer, Heidelberg (2012)
11. Liang, W.: Constructing minimum-energy broadcast trees in wireless ad hoc networks. In: MobiHoc, pp. 112–122 (2002)
12. Liang, W., Liu, Y.: Online data gathering for maximising network lifetime in sensor networks. IEEE Trans. Mob. Comput. 6(1), 2–11 (2007)
13. Lin, H.C., Li, F.J., Wang, K.Y.: Constructing maximum-lifetime data gathering trees in sensor networks with data aggregation. In: ICC, pp. 1–6 (2010)
14. Maric, I., Yates, R.D.: Cooperative multicast for maximum network lifetime. IEEE Journal on Selected Areas in Communications 23(1), 127–135 (2005)
15. Nutov, Z.: Approximating directed weighted-degree constrained networks. In: Goel, A., Jansen, K., Rolim, J.D.P., Rubinfeld, R. (eds.) APPROX and RANDOM 2008. LNCS, vol. 5171, pp. 219–232. Springer, Heidelberg (2008)
16. Nutov, Z.: Approximating maximum integral flows in wireless sensor networks via weighted-degree constrained k-flows. In: DIALM-POMC, pp. 29–34 (2008)
17. Nutov, Z., Segal, M.: Improved approximation algorithms for maximum lifetime problems in wireless networks. In: Dolev, S. (ed.) ALGOSENSORS 2009. LNCS, vol. 5804, pp. 41–51. Springer, Heidelberg (2009)
18. Orda, A., Yassour, B.A.: Maximum-lifetime routing algorithms for networks with omnidirectional and directional antennas. In: MobiHoc, pp. 426–437 (2005)
19. Park, J., Sahni, S.: Maximum lifetime broadcasting in wireless networks. In: AICCSA, p. 8. IEEE Computer Society (2005)
20. Segal, M.: Fast algorithm for multicast and data gathering in wireless networks. Inf. Process. Lett. 107(1), 29–33 (2008)
21. Stanford, J., Tongngam, S.: Approximation algorithm for maximum lifetime in wireless sensor networks with data aggregation. In: SNPD, pp. 273–277 (2006)
22. Wan, P.J., Li, X.Y., Frieder, O.: Minimum energy cost broadcasting in wireless networks. In: Encyclopedia of Algorithms (2008)
23. Wieselthier, J.E., Nguyen, G.D., Ephremides, A.: On the construction of energy-efficient broadcast and multicast trees in wireless networks. In: INFOCOM, pp. 585–594 (2000)
24. Wu, Y., Fahmy, S., Shroff, N.B.: On the construction of a maximum-lifetime data gathering tree in sensor networks: Np-completeness and approximation algorithm. In: INFOCOM, pp. 356–360 (2008)
25. Zagalj, M., Hubaux, J.P., Enz, C.C.: Minimum-energy broadcast in all-wireless networks: Np-completeness and distribution issues. In: MOBICOM, pp. 172–182 (2002)

Maximum Spectral Radius of Graphs with Connectivity at Most k and Minimum Degree at Least δ

Hongliang Lu[1] and Yuqing Lin[2]

[1] Department of Mathematics
Xi'an Jiaotong University, Xi'an 710049, P.R. China
[2] School of Electrical Engineering and Computer Science
The University of Newcastle, Newcastle, Australia

Abstract. Li, Shiu, Chan and Chang [On the spectral radius of graphs with connectivity at most k, J. Math. Chem., 46 (2009), 340-346] studied the spectral radius of graphs of order n with $\kappa(G) \leq k$ and showed that among those graphs, the maximum spectral radius is obtained uniquely at K_k^n, which is the graph obtained by joining k edges from k vertices of K_{n-1} to an isolated vertex. In this paper, we study the spectral radius of graphs of order n with $\kappa(G) \leq k$ and minimum degree $\delta(G) \geq k$. We show that among those graphs, the maximum spectral radius is obtained uniquely at $K_k + (K_{\delta-k+1} \cup K_{n-\delta-1})$.

Keywords: connectivity, spectral radius.

1 Introduction

Let G be a simple graph of order n with vertex set $V(G) = \{v_1, v_2, \ldots, v_n\}$. We denote by $\delta(G)$ the minimum degree of vertices of G. We use $N_G(v)$ to denote the set of neighbors of vertex v, $d_G(v) = |N_G(v)|$ to denote the degree of vertex v of graph G.Let x and y be two nonadjacent vertices of G. $G + xy$ is obtained from G by adding an edge xy. Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two graphs. The union $G_1 \cup G_2$ is defined to be $G_1 \cup G_2 = (V_1 \cup V_2, E_1 \cup E_2)$. The join $G_1 + G_2$ is obtained from $G_1 \cup G_2$ by adding all the edges joining a vertex of G_1 to a vertex of G_2.

The adjacency matrix of the graph G is defined to be a matrix $A(G) = [a_{ij}]$ of order n, where $a_{ij} = 1$ if v_i is adjacent to v_j, and $a_{ij} = 0$ otherwise. Since $A(G)$ is symmetric and real, the eigenvalues of $A(G)$, also referred to as the eigenvalues of G, can be arranged as: $\lambda_n(G) \leq \lambda_{n-1}(G) \leq \cdots \leq \lambda_1(G)$. The largest eigenvalue $\lambda_1(G)$ is called spectral radius and also denoted by $\rho(G)$. Let $\pi = (V_1, \ldots, V_r)$ be a partition of $V(G)$. The partition is *equitable* if the number of neighbors in V_j of a vertex u in V_i is a constant b_{ij}, independent of u. The entries of the adjacency matrix of this quotient are given by

$$A(G/\pi)_{ij} = b_{ij}.$$

© Springer-Verlag Berlin Heidelberg 2013

For $k \geq 1$, we say that a graph G is k-connected if either G is a complete graph K_{k+1}, or else it has at least $k+2$ vertices and contains no $(k-1)$-vertex cut. The *connectivity* $\kappa(G)$ of G is the maximum value of k for which G is k-connected.

When G is connected, $A(G)$ is irreducible and by the Perron-Frobenius Theorem, the spectral radius is simple and there is an unique positive unit eigenvector. We shall refer to such an eigenvector as the Perron vector of G. For more details on the Perron-Frobenius Theorem for nonnegative matrices, see Chapter 8 of [3].

The eigenvalues of a graph are related to many of its properties and key parameters. The most studied eigenvalues have been the spectral radius $\rho(G)$ (in connection with the chromatic number, the independence number and the clique number of the graph [10, 12]). Brualdi and Solheid [2] proposed the following problem concerning spectral radius:

Given a set of graphs $\mathscr{C}$, find an upper bound for the spectral radius of graphs in $\mathscr{C}$ and characterize the graphs in which the maximal spectral radius is attained.

If $\mathscr{C}$ is the set of all connected graphs on n vertices with k cut vertices, Berman and Zhang [1] solved this problem. Liu et al. [9] studied this problem for $\mathscr{C}$ to be the set of all graphs on n vertices with k cut edges. Wu et al. [13] studied this problem for $\mathscr{C}$ to be the set of trees on n vertices with k pendent vertices. Feng, Yu and Zhang [4] studied this problem for $\mathscr{C}$ to be the set of graphs on n vertices with matching number β.

Li, Shiu, Chan and Chang studied this question for graphs with n vertices and connectivity at most k, and obtained the following result.

Theorem 1 (Li, Shiu, Chan and Chang [11]). *Among all the graphs with connectivity at most k, the maximum spectral radius is obtained uniquely at $K_k + (K_1 \cup K_{n-k-1})$.*

Let $G_{k,\delta,n} = K_k + (K_{\delta-k+1} \cup K_{n-\delta-1})$. We denote by $\mathcal{V}_{k,\delta,n}$ the set of graphs of order n with $\kappa(G) \leq k \leq n-1$ and $\delta(G) \geq k$. Clearly, $\mathcal{V}_{k,\delta+1,n} \subseteq \mathcal{V}_{k,\delta,n}$. In this paper, we investigate the problem stated above for the graphs in $\mathcal{V}_{k,\delta,n}$. We show that among all those graphs, the maximal spectral radius is obtained uniquely at $G_{k,\delta,n}$.

In our arguments, we need the following technical lemmas.

Theorem 2 (Li and Feng [8]). *Let G be a connected graph, and G' be a proper subgraph of G. Then $\rho(G') < \rho(G)$.*

Theorem 3 (Haemers, [5]). *Let G be a graph and $\pi = (V_1, \ldots, V_r)$ be a partition of $V(G)$ with quotient matrix $Q = A(G/\pi)$. Then*

$$\lambda_1(G) \geq \lambda_1(Q), \tag{1}$$

with equality if the partition is equitable.

2 Main Results

Theorem 4. *Let G be a connected graph. Let $\{v_1, \ldots, v_k\} \subseteq N_G(v)$ and $\{v_{k+1}, \ldots, u_{k+l}\} \subseteq V(G) - N_G(v)$. Suppose $x = (x_1, x_2, \ldots, x_n)^T$ is the Perron vector* *of G, where x_i corresponds to the vertex v_i $(1 \le i \le n)$. Let G^* be the graph obtained from G by deleting the edges vv_i $(1 \le i \le k)$ and adding the edges vv_i $(k{+}1 \le i \le k{+}l)$. If $\sum_{i=1}^{k} x_i \le \sum_{i=k+1}^{k+l} x_i$, then $\rho(G) \le \rho(G^*)$. Furthermore, if $\sum_{i=1}^{k} x_i < \sum_{i=k+1}^{k+l} x_i$, then $\rho(G) < \rho(G^*)$.*

Proof. We have

$$\begin{aligned} x^T(A(G^*) - A(G))x &= -x_v \sum_{i=1}^{k} x_i + x_v(\sum_{j=k+1}^{k+l} x_j - \sum_{i=1}^{k} x_i) + x_v \sum_{j=k+1}^{k+l} x_j \\ &= 2x_v(\sum_{j=k+1}^{k+l} x_j - \sum_{i=1}^{k} x_i) \ge 0. \end{aligned}$$

So we have

$$\rho(G^*) = \max_{\|y\|=1} y^T A(G^*)y \ge x^T A(G^*)x \ge x^T A(G)x = \rho(G). \tag{2}$$

If $\sum_{i=k}^{k} x_i < \sum_{i=k+1}^{k+l} x_i$, then we have

$$\rho(G^*) = \max_{\|y\|=1} y^T A(G^*)y \ge x^T A(G^*)x > x^T A(G)x = \rho(G). \tag{3}$$

Hence $\rho(G^*) > \rho(G)$. This completes the proof. □

With above proof, we obtain the following result.

Corollary 5 *Suppose G^* in Theorem 4 is connected, and $y = (y_1, y_2, \ldots, y_n)^T$ is the Perron vector of G^*, then $\sum_{i=k+1}^{k+l} y_i \ge \sum_{i=1}^{k} y_i$. Furthermore, if $\sum_{i=k+1}^{k+l} x_i > \sum_{i=1}^{k} x_i$, then $\sum_{i=k+1}^{k+l} y_i > \sum_{i=1}^{k} y_i$.*

Proof. Suppose that $\sum_{i=k+1}^{k+l} y_i < \sum_{i=1}^{k} y_i$, by Theorem 4, we have $\rho(G^*) < \rho(G)$, a contradiction.

Since $\sum_{i=k+1}^{k+l} x_i > \sum_{i=1}^{k} x_i$, by Theorem 4, we have $\rho(G^*) > \rho(G)$. If $\sum_{i=k+1}^{k+l} y_i \le \sum_{i=1}^{k} y_i$, by Theorem 4 then we have $\rho(G^*) \le \rho(G)$, a contradiction. This completes the proof. □

Lemma 1. *Let G be a graph. If the minimum degree of G is no less than $\frac{n+k-1}{2}$, then G is $(k+1)$-connected.*

Theorem 6. *Let n, k and δ be three positive integers. Among all the connected graphs of order n with connectivity at most k and minimum degree δ, the maximal spectral radius is obtained uniquely at $G_{k,\delta,n}$.*

Proof. By Lemma 1, we have $2\delta \le n{+}k{+}2$. If $n = k{+}1$, then K_{k+1} is an unique k-connected graph with order n. So we can assume that $n \ge k+2$. Now we have to prove that for every $G \in \mathcal{V}_{k,\delta,n}$, then $\rho(G) \le \rho(G_{k,\delta,n})$, where the equality holds if and only if $G = G_{k,\delta,n}$. Let $G^* \in \mathcal{V}_{k,\delta,n}$ with $V(G^*) = \{v_1, \ldots, v_n\}$ be the graph with maximum spectral radius in $\mathcal{V}_{k,\delta,n}$, that is, $\rho(G) \leq \rho(G^*)$ for all $G \in \mathcal{V}_{k,\delta,n}$.

Denote the Perron vector by $x = (x_1, \ldots, x_n)$, where x_i corresponding to v_i for $i = 1, \ldots, n$. Since $G^* \in \mathcal{V}_{k,\delta,n}$ and it is not a complete graph, then G^* has a k-vertex cut, say $S = \{v_1, \ldots, v_k\}$. In the following, we will prove the following three claims.

Claim 1. $G^* - S$ contains exactly two components.

Suppose contrary that $G^* - S$ contains three components G_1, G_2 and G_3. Let $u \in G_1$ and $v \in G_2$. It is obvious that S is also an k-vertex cut of $G^* + uv$; i.e. $G^* + uv \in \mathcal{V}_{k,\delta,n}$. By Theorem 2, we have $\rho(G^*) < \rho(G^* + uv)$. This contradicts the definition of G^*.

Therefore, $G^* - S$ has exactly two components G_1 and G_2.

Claim 2. Each subgraph of G^* induced by vertices $V(G_i) \cup S$, for $i = 1, 2$, is a clique.

Suppose contrary that there is a pair of non-adjacent vertices $u, v \in V(G_i) \cup S$ for $i = 1$ or 2. Again, $G^* + uv \in \mathcal{V}_{k,\delta,n}$. By Theorem 2, we have $\rho(G^*) < (G^* + uv)$. This contradicts the definition of G^*.

From Claim 2, it is clear that all G_1 and G_2 are cliques too. Then we write K_{n_i} instead of G_i, for $i = 1, 2$, in the rest of the proof, where $n_i = |G_i|$. Since $\delta(G) \geq k$, we have $n_i \geq \delta - k + 1$ for $i = 1, 2$.

Claim 3. Either $n_1 = \delta - k + 1$ or $n_2 = \delta - k + 1$.

Otherwise, we have $n_1 > \delta - k + 1$ and $n_2 > \delta - k + 1$. Let $v \in G_1$ and $u \in G_2$. Suppose

$$N_{G^*}(v) = \{v_1, v_2, \ldots, v_{n_1 - 1}, v_1, v_2, \ldots, v_k\}$$

and

$$N_{G^*}(u) = \{u_1, u_2, \ldots, u_{n_2 - 1}, v_1, v_2, \ldots, v_k\}.$$

Partition the vertex set of G^* into three parts: the vertices of S; the vertices of G_1; the vertices of G_2. This is an equitable partition of G with quotient matrix

$$Q = \begin{pmatrix} k-1 & n_1 & n_2 \\ k & n_1 - 1 & 0 \\ k & 0 & n_2 - 1 \end{pmatrix}$$

By Perron-Frobenius Theorem, Q has a Perron-vector $x = (x_1, x_2, x_3)$. Now we show that $x_2 < x_3$ if $n_1 < n_2$. Let $\rho(Q)$ denote the largest eigenvalue of Q. Then we have

$$kx_1 + (n_1 - 1)x_2 = \rho(Q)x_2 \tag{4}$$

$$kx_1 + (n_2 - 1)x_3 = \rho(Q)x_3 \tag{5}$$

By (4) and (5), we have

$$(n_2 - 1)x_3 - (n_1 - 1)x_2 = \rho(Q)(x_3 - x_2).$$

Hence

$$(\rho(Q) - n_2 + 1)(x_3 - x_2) = (n_2 - n_1)x_2 > 0.$$

By Theorem 3, $\rho(Q)$ is also the largest eigenvalue of G^*, so we have $\rho(Q) > \rho(K_{n_2}) = n_2 - 1$. Hence $x_3 > x_2$. The eigenvector x can be extended to an eigenvector of $A(G^*)$, say

$$y = (x_{11}, \ldots, x_{1k}, x_{21}, \ldots, x_{2n_1}, x_{31}, \ldots, x_{3n_2}),$$

where $x_{i1} = \ldots = x_{in_i} = x_i$ for $i = 2, 3$ and $x_{11} = \cdots = x_{1k} = x_1$. Let $z = \frac{1}{\sqrt{kx_1^2 + n_1 x_2^2 + n_3 x_3^2}} y$. We have $zz^T = 1$ and so z is a Perron-vector of G^*. Let $H = G^* - \{vv_1, vv_2, \ldots, vv_{n_1-1}\} + \{vu_1, \ldots, vu_{n_2}\}$ and we have $H \in \mathcal{V}_{k,\delta,n}$. Since $n_2 x_3 > (n_1 - 1)x_2$, by Theorem 4, $\rho(G^*) < \rho(H)$, which is a contradiction. This completes claim 3.

By Claim 3, we can say $n_2 \geq n_1 = \delta + 1 - k$. Hence $G^* = G_{k,\delta,n}$. This completes the proof. □

Theorem 7. *Spectral radius of $G_{k,\delta,n}$ is the largest root of the following equation $x^3 + (3-n)x^2 + (n\delta - \delta^2 - n - kn + k + k\delta + 2 - 2\delta)x + (kn\delta + k^2 + n\delta + k^2\delta - k\delta - k^2 n - k\delta^2 - 2\delta - \delta^2) = 0$.*

Proof. Let G_1 be the subgraph of $G_{k,\delta,n}$ induced by the k vertices of degree $n-1$ of $G_{k,\delta,n}$, G_2 be the subgraph induced by the $\delta - k + 1$ vertices of degree δ and G_3 be the subgraph induced by the remaining $n - \delta - 1$ vertices of degree $n - \delta - 1 + k$. Also, let G_{ij} be the bipartite subgraph induced by $V(G_i)$ and $V(G_j)$ and let e_{ij} be the size of G_{ij}. The quotient matrix Q corresponding to partition $\pi = (V(G_1), V(G_2), V(G_3))$ is the following

$$Q = \begin{pmatrix} \frac{2e_1}{n_1} & \frac{e_{12}}{n_1} & \frac{e_{13}}{n_1} \\ \frac{e_{21}}{n_2} & \frac{2e_2}{n_2} & \frac{e_{23}}{n_2} \\ \frac{e_{31}}{n_3} & \frac{e_{32}}{n_3} & \frac{2e_{33}}{n_3} \end{pmatrix} = \begin{pmatrix} k-1 & \delta - k + 1 & n - \delta - 1 \\ k & \delta - k & 0 \\ k & 0 & n - \delta - 2 \end{pmatrix}.$$

After the calculation, we obtain

$$\begin{aligned} det(xI - Q) = & x^3 + (3-n)x^2 + (n\delta - \delta^2 - n - kn + k + k\delta + 2 - 2\delta)x + \\ & (kn\delta + k^2 + n\delta + k^2\delta - k\delta - k^2 n - k\delta^2 - 2\delta - \delta^2). \end{aligned}$$

By Theorem 3, we get

$$\lambda_1(G_{k,\delta,n}) \geq \lambda_1(Q), \tag{6}$$

with the equality if the partition is equitable. Note that the partition is equitable, so the equality hold. This completes the proof. □

References

1. Berman, A., Zhang, X.D.: On the spectral radius of graphs with cut vertices. J. Combin. Theory Ser. B. 83, 233–240 (2001)
2. Brualdi, R.A., Solheid, E.S.: On the spectral radius of complementary acyclic matrices of zeros and ones. SIAM J. Algebra Discret. Method. 7, 265–272 (1986)
3. Carl, M.: Matrix analysis and applied linear algebra. SIAM (2000) ISBN 0-89871-454-0
4. Feng, L., Yu, G., Zhang, X.: Spectral radius of graphs with given matching number. Linear Algebra Appl. 422, 133–138 (2007)
5. Haemers, W.: Interlacing eigenvalues and graphs. Linear Algebra Appl. 226-228, 593–616 (1995)
6. Godsil, C., Royle, G.: Algebraic Graph Theory. Springer, New York (2001)
7. Guo, J.: The effect on the Laplacian spectral radius of a graph by adding or grafting edges. Linear Algebra Appl. 413, 59–71 (2006)
8. Li, Q., Feng, K.: On the largest eigenvalue of graphs. Acta Math. Appl. Sinica. 2, 167–175 (1979)
9. Liu, H., Lu, M., Tian, F.: On the spectral radius of graphs with cut edges. Linear Algebra Appl. 389, 139–145 (2004)
10. Nikiforov, V.: Some inequalities for the largest eigenvalue of a graph. Combin. Probab. Comput. 11, 179–189 (2001)
11. Li, Shiu, Chan, Chang: On the spectral radius of graphs with connectivity at most k. J. Math. Chem. 46, 340–346 (2009)
12. Wilf, H.: Spectral bounds for the clique and independence number of graphs. J. Combin. Theory Ser. B 40, 113–117 (1986)
13. Wu, B., Xiao, E., Hong, Y.: The spectral radius of trees on k pendant vertices. Linear Algebra Appl. 395, 343–349 (2005)
14. Zhou, B., Trinajstić, N.: On the largest eigenvalue of the distance matrix of a connected graph. Chem. Phys. Lett. 447, 384–387 (2007)

Degree Sequences of PageRank Uniform Graphs and Digraphs with Prime Outdegrees

Nacho López and Francesc Sebé

Dept. of Mathematics, Universitat de Lleida,
C.Jaume II, 69, E-25001, Spain
{nlopez,fsebe}@matematica.udl.cat

Abstract. A *PageRank uniform digraph* is a digraph whose vertices have all the same PageRank score. These digraphs are interesting in the scope of privacy preserving release of digraph data in environments where a dishonest analyst may have previous structural knowledge about the PageRank score of some vertices. In this paper we first characterize PageRank uniform graphs (viewed as symmetric digraphs) and their degree sequence. Next, given a sequence of prime integers S, we give necessary and sufficient conditions for S to be the outdegree sequence of a PageRank uniform digraph.

1 Introduction

A sequence which is the degree sequence of some graph is called a *graphical sequence*. A necessary and sufficient condition for a sequence to be graphical was found independently by Havel [7] and Hakimi [5]. A different characterization was given by Erdös and Gallai (see [1]). Since then, the study of graphical sequences has been focused on graphs with some specific properties. These studies include graphical sequences of self-complementary, planar, bipartite and hamiltonian graphs, graphs with a prescribed vertex connectivity or graphs containing a hamiltonian path (see [6] for a complete survey).

The degree sequence of a directed graph (digraph) contains ordered pairs with the outdegree and the indegree of each of its vertices. A sequence of pairs of nonnegative integers is said to be *digraphical* if it is the degree sequence of some digraph. Degree sequences of digraphs have been widely studied. Their main characterization was found independently by Fulkerson [4] and Ryser [14]. As it happens with graphs, digraphical sequences have also been studied for digraphs with prescribed properties. Research on digraphical sequences has also been carried out for indegree and outdegree sequences separately. For instance, Nash and Williams [12] found sufficient conditions for the indegree and the outdegree sequences of a digraph that guarantee the hamiltonicity of that digraph.

In this paper, we study the outdegree sequence of PageRank uniform digraphs, *i.e.* digraphs whose vertices have all the same PageRank score. More precisely, we provide a characterization of the degree sequence of PageRank uniform symmetric digraphs (graphs) and the outdegree sequence of PageRank uniform digraphs with prime outdegrees.

© Springer-Verlag Berlin Heidelberg 2013

1.1 Applications to Privacy Preserving Release of Social Network Data

Online platforms (like Facebook[1]) in which people can create an account and establish relations among them are becoming very popular. The arising social network can be modeled by means of (directed) graphs in which each user/account corresponds to a vertex and the relations between pairs of users are the edges (for symmetric relations like 'be friends') or the arcs (for asymmetric relations like 'trust in'). These platforms manage very sensitive personal data whose leak would endanger users'privacy.

Social network data [15] are a source of valuable information for several research areas like psychology, sociology and economics, but due to their private content, their release for scientific analysis requires a pre-processing for ensuring no private information about people will be disclosed. The measures to be taken, including identifiers removal, depend on the previous structural information a dishonest analyst is assumed to have [19]. Some proposals [3,9,17,18,20] apply variants of the k-anonymity model to achieve *vertex privacy* (it has to be difficult for an attacker to link identities of network members to the vertices of the released anonymous graph), so that they perturb the graph until the groups of vertices sharing a given structural parameter have at least size k.

In a recent proposal [10], a solution is given in which the n-confusion model [16] is applied to directed graph data in scenarios where an attacker may have previous knowledge about the PageRank score of some vertices. In this context, releasing a perturbed digraph whose vertices have all the same PageRank score would provide perfect privacy since its vertices are indistinguishable with respect to that score.

Before being able to perturb a digraph so that it becomes PageRank uniform, we need a characterization of such digraphs. This paper provides a step towards that objective by studying two particular cases: symmetric digraphs (graphs) and digraphs with prime outdegrees.

2 Preliminaries

This section presents the used notation together with an introduction to PageRank.

2.1 Terminology and Notation

A *digraph* $D = (V, A)$ is a finite nonempty set V of objects called *vertices* and a set A of ordered pairs of vertices called *arcs*. The *order* of D is the cardinality of its set of vertices V. If (u, v) is an arc, it is said that u is *adjacent to* v and also that v is *adjacent from* u. The set of vertices that are adjacent from [to] a given vertex v is denoted by $N^+(v)$ [resp. $N^-(v)$] and its cardinality is the *outdegree* of v, $d^+(v)$ [resp. *indegree* of v, $d^-(v)$]. If $d^+(v) = k$ [resp. $d^-(v) = k$], for all

[1] `http://www.facebook.com/`

$v \in V$, then D is said to be *outregular* [resp. *inregular*] of degree k. A digraph D is called *regular of degree* k if $d^+(v) = d^-(v) = k$ for every vertex v of D. If $d^+(v) = 0$ for all $v \in V$, then D is said to be the *empty* digraph.

A *graph* G can be viewed as a symmetric digraph where every symmetric pair of arcs (a, b) and (b, a) is identified as a non ordered pair $\{a, b\}$ called *edge*. A *walk* of length h from a vertex u to a vertex v ($u - v$ walk) in G is a sequence of vertices $u = u_0, u_1, \ldots, u_{h-1}, u_h = v$ such that each pair $\{u_{i-1}, u_i\}$ is an edge of G. A graph G is *connected* if there is a $u - v$ walk for every pair of vertices u and v of G. The reader is referred to Chartrand and Lesniak [1] for additional concepts on digraphs and graphs.

2.2 The PageRank Vector of a Digraph

The PageRank [8,13] algorithm was designed as a method for assigning a relevance score to webpages. This score, together with some other criteria, is then considered for determining the order in which the results of search engine queries are shown to the user. The PageRank algorithm first creates a directed graph representing web pages (the vertices) and the hyperlinks among them (the arcs).

Given a directed graph $D = (V, A)$ of order n, the $n \times n$ *normalized link matrix* $\mathbf{P} = (p_{ij})$ is defined so that, for each pair of vertices $v_i, v_j \in V$,

$$p_{ij} = \begin{cases} \frac{1}{d^+(v_j)} & \text{if } d^+(v_j) > 0 \text{ and } (v_j, v_i) \in A, \\ 0 & \text{if } d^+(v_j) > 0 \text{ and } (v_j, v_i) \notin A, \\ \frac{1}{n} & \text{if } d^+(v_j) = 0. \end{cases}$$

By considering a surfer riding the digraph vertices, each coefficient p_{ij} of $\mathbf{P}$ corresponds to the probability that the surfer jumps to vertex v_i after having reached vertex v_j, assuming the next movement is taken uniformly at random among the arcs emanating from v_j. If the random surfer falls in a vertex with no outgoing arcs, the navigation is restarted from a randomly chosen vertex. This random restart behaviour is allowed at any moment (with a small probability $1 - \alpha$), by creating matrix $\mathbf{P}(\alpha)$,

$$\mathbf{P}(\alpha) = \alpha \mathbf{P} + (1 - \alpha)\frac{1}{n}\mathbf{J}, \tag{1}$$

where $\mathbf{J}$ denotes the order n all ones square matrix. By construction, $\mathbf{P}(\alpha)$ is a positive matrix [11], hence, it has a unique dominant eigenvalue. The *PageRank vector* is defined to be the (positive) eigenvector $\mathcal{P} = (p_0, \ldots, p_{n-1})$ with $\sum_i p_i = 1$ (the Perron vector of $\mathbf{P}(\alpha)$) associated to this eigenvalue. The value of α (*dumping factor*) is usually chosen to be $\alpha = 0.85$. The relevance score assigned by PageRank to vertex v_j is p_j. This value represents the long-run fraction of time the surfer would spend at vertex v_j.

This algorithm can be applied to provide a relevance score to the vertices of any digraph. For instance, by applying it to a digraph representing scientific publications and the 'cite' relations among them, we could determine the most influent publications.

Example 1. Let us consider a small digraph (Figure 1) $D = (V, A)$, with $V = \{v_0, v_1, v_2, v_3, v_4\}$ and

$$A = \{(v_0, v_1), (v_0, v_4), (v_1, v_2), (v_2, v_1), (v_3, v_0), (v_3, v_1), (v_3, v_2), (v_3, v_4)\}.$$

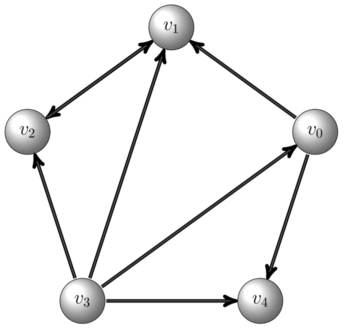


Fig. 1. Digraph having v_1 as its most relevant vertex, while v_3 is the least one

Applying the PageRank algorithm to digraph D, the following scores vector is obtained: $\mathcal{P} = (p_0, p_1, p_2, p_3, p_4) = (0.0515, 0.4222, 0.4104, 0.0425, 0.0734)$. Hence, v_1 is the most relevant vertex while v_3 is the least one (no vertex points to it). In this particular example, we took $\alpha = 0.85$. Matrix $\mathbf{P}$ is as follows,

$$\mathbf{P} = \begin{pmatrix} 0 & 0 & 0 & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{2} & 0 & 1 & \frac{1}{4} & \frac{1}{5} \\ 0 & 1 & 0 & \frac{1}{4} & \frac{1}{5} \\ 0 & 0 & 0 & 0 & \frac{1}{5} \\ \frac{1}{2} & 0 & 0 & \frac{1}{4} & \frac{1}{5} \end{pmatrix}.$$

3 Characterization of PageRank Uniform Graphs

A digraph D is called to be *PageRank uniform*[2] when its PageRank vector $\mathcal{P}$ is uniform, that is, all the vertices of D have the same PageRank score.

In [2], it was proven that the value given to the dumping factor α does not influence the PageRank uniformity of a digraph. Hence, we can focus our attention on matrix $\mathbf{P}$ (instead of $\mathbf{P}(\alpha)$). By definition, the normalized link matrix $\mathbf{P}$ of an order n digraph without self-loops nor zero outdegree vertices, satisfies:

a) All the elements in the main diagonal of $\mathbf{P}$ are zero, that is $p_{ii} = 0$.

b) Given $p_{ij} = \dfrac{1}{k} \Rightarrow \exists\, i_1, \cdots, i_k$ such that $p_{i'j} = \begin{cases} \frac{1}{k} \text{ if } i' \in \{i_1, \cdots, i_k\}, \\ 0 \text{ otherwise.} \end{cases}$

[2] These digraphs were called *PageRank regular* in [2]. Nevertheless, we prefer the term *uniform* so as to avoid confusion with the term *regular* in the classical sense.

In particular, each column of $\mathbf{P}$ sums 1. In addittion, a digraph is PageRank uniform if and only if each row of $\mathbf{P}$ also sums 1, as it is stated in the next proposition whose proof was given in [2].

Proposition 1. *Let D be a digraph and let $\mathbf{P}$ be its normalized link matrix. D is a PageRank uniform digraph if and only if each row of $\mathbf{P}$ sums 1, that is, $\sum_j p_{ij} = 1$, for each i.*

The condition $\sum_j p_{ij} = 1$ is equivalent to saying that $\left(1, \frac{1}{n}\mathbf{j}\right)$ is an eigenpair for $\mathbf{P}$ ($\mathbf{j}$ denotes the length n all ones vector). The next proposition states that regular digraphs are PageRank uniform:

Proposition 2. *Let D be a regular digraph. Then, D is a PageRank uniform digraph.*

Proof. If D is the empty digraph of order n, then $\mathbf{P} = \mathbf{P}(\alpha) = \frac{1}{n}\mathbf{J}$ and $\left(1, \frac{1}{n}\mathbf{j}\right)$ is its Perron eigenpair so that D is PageRank uniform.

Let D be a regular digraph of degree $k > 0$. The adjacency matrix $\mathbf{A}$ of D satisfies $\mathbf{A^T} = k\mathbf{P}$. Besides, for any regular vector $\mathbf{v}$, we have $\mathbf{A^T v} = k\mathbf{v}$, since the elements of each column of $\mathbf{A}$ sum k (every vertex of D has indegree k). By chosing $\mathbf{v} = \frac{1}{n}\mathbf{j}$, we get $k\mathbf{Pv} = k\mathbf{v}$, that is, $\mathbf{Pv} = \mathbf{v}$. So, $(1, \frac{1}{n}\mathbf{j})$ is an eigenpair for $\mathbf{P}$ and, by Proposition 1, we conclude that D is a PageRank uniform digraph. □

The converse of Proposition 2 is not true in general. Nevertheless, if we restrict our attention to graphs (symmetric digraphs), then the condition of being regular turns out to be sufficient to achieve PageRank uniformity for connected graphs. By considering a non-connected graph G as the union of its connected components, the next theorem provides a characterization for PageRank uniform graphs.

Theorem 1. *Let $G = (V, E)$ be a graph. The following statements are equivalent,*

- *G is a PageRank uniform graph.*
- *$G = H_1 \cup \cdots \cup H_s$, where each H_i is a connected regular graph of order ≥ 2, or G is the empty graph.*

Proof. If G is the empty graph, we can consider it as the empty (symmetric) digraph. Since it is regular then it is PageRank uniform by Proposition 2. Let us assume $G = H_1 \cup \cdots \cup H_s$, with each H_i being a connected regular graph of order ≥ 2. First of all, every H_i is PageRank uniform by Proposition 2. Let $\mathbf{P_1}, \ldots, \mathbf{P_s}$ be the normalized link matrices of $H_1, \ldots, H_s$, respectively. The normalized link matrix $\mathbf{P}$ of G is block diagonal, being $\mathbf{P_i}$ its diagonal blocks, that is,

$$\mathbf{P} = \begin{pmatrix} \mathbf{P_1} & 0 & \cdots & 0 \\ 0 & \mathbf{P_2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{P_s} \end{pmatrix}.$$

Hence, $\mathbf{P}$ satisfies Proposition 1 if and only if each $\mathbf{P_i}$ does, which is the case.

Conversely, let us assume G is PageRank uniform. We distinguish two cases:

- *G is connected.* Let us consider a minimum degree vertex $u \in V$, with $d(u) = k > 0$ and let $\{v_1, v_2, \ldots, v_k\}$ be the neighbors of u. The sum of the row of $\mathbf{P}$ corresponding to vertex u is $\sum_{i=1}^{k} \frac{1}{d(v_i)}$. This sum is 1 by the assumed PageRank uniformity. Now, since $k \leq d(v_i)$ for any $v_i \in V$, we get,

$$1 = \sum_{i=1}^{k} \frac{1}{d(v_i)} \leq \sum_{i=1}^{k} \frac{1}{k} = 1,$$

  which is only possible if $d(v_i) = k$ for each $1 \leq i \leq k$. In this way, all the neighbors of a minimum degree vertex have also minimum degree. By recursively applying this reasoning and taking into account that G is connected, we conclude that all the vertices in G have the same minimum degree k, that is, G is regular.
- *G is non-connected.* Then, $G = H_1 \cup \cdots \cup H_s$ where each H_i is a connected graph of order ≥ 2. Since G is assumed to be PageRank uniform, every H_i is also PageRank uniform. Now, we can apply the previous argument to each connected component and derive that each H_i is regular.
  If some H_i had order 1, then G would have at least one isolated vertex. The digraph surfer considered by the PageRank algorithm only visits isolated vertices as a result of a random restart, while connected vertices can be visited after a random restart or by reaching them from a neighbor vertex. Hence, non-isolated vertices receive a higher PageRank score than isolated ones. In this way, G would be PageRank uniform only if all the other vertices were also isolated in which case G would be empty.

□

4 PageRank Uniformity and Degree Sequences

In this section we study the degree sequence of PageRank uniform digraphs. We provide necessary and sufficient conditions for a sequence of nonnegative integers to be the degree sequence of a PageRank uniform graph. Regarding directed graphs, we give necessary and sufficient conditions for a sequence of *prime* integers to be the outdegree sequence of a PageRank uniform digraph.

4.1 PageRank Uniform Graphical Sequences

A nondecreasing sequence $S : d_1, d_2, \cdots, d_n$ of nonnegative integers is called a *PageRank uniform graphical* sequence if there is a PageRank uniform graph G of order n whose degree sequence is S.

For short, a degree sequence will be denoted by $S : d_1^{n_1}, d_2^{n_2}, \ldots, d_s^{n_s}$, where $d_i^{n_i}$ denotes that value d_i appears n_i times in S. It is well known that a degree sequence can correspond to a regular graph if and only if it is of the form $S : d^n$ with $n \geq d + 1$ and the product dn is even. By proposition 1, every PageRank uniform graph G is either the empty graph or the union of regular connected components of order ≥ 2. As a consequence, the characterization of PageRank uniform graphical sequences is immediately determined.

Proposition 3. *Let $S : d_1^{n_1}, d_2^{n_2}, \ldots, d_s^{n_s}$ be a nondecreasing sequence of nonnegative integers. Then, S is a PageRank uniform graphical sequence if and only if $d_i \geq 1$, $n_i \geq d_i + 1$ and $d_i n_i$ is even, for each $1 \leq i \leq s$; or $d_1 = 0$ and $s = 1$.*

4.2 PageRank Uniform Outdigraphical Sequences Composed of Prime Numbers

A nondecreasing sequence $S : d_1, d_2, \cdots, d_n$ of nonnegative integers, is called a *PageRank uniform outdigraphical sequence* if there exists a PageRank uniform digraph D of order n such that S is the outdegree sequence of D.

In this paper, we focus on the particular case in which $d_1, d_2, \cdots, d_n$ are all prime numbers. PageRank uniform digraphs with prime outdegrees were studied in [2]. The authors proved that, in a PageRank uniform digraph with prime outdegrees, the inneighbors of any vertex v have all the same (prime) outdegree which coincides with the indegree of v. This property can be expressed in terms of the normalized link matrix $\mathbf{P}$ of D as follows.

Lemma 1. *Let $\mathbf{P}$ be the normalized link matrix of a digraph D having prime outdegrees. Then, D is PageRank uniform if and only if all the non-null elements of each row of $\mathbf{P}$ are equal and their sum is 1.*

If a PageRank uniform digraph D has a vertex v with prime outdegree q, its corresponding column of $\mathbf{P}$ contains the value $\frac{1}{q}$ exactly q times. Hence, value $\frac{1}{q}$ appears in at least q different rows of $\mathbf{P}$. Let i be such a row. By Lemma 1, the non-null elements of row i must all be $\frac{1}{q}$. Since each row of $\mathbf{P}$ sums 1, there must be exactly q elements with value $\frac{1}{q}$ in row i. This observation gives a necessary condition for a sequence to be PageRank uniform outdigraphical when all its elements are prime.

Proposition 4. *Let $S : q_1^{n_1}, q_2^{n_2}, \ldots, q_s^{n_s}$ be a PageRank uniform outdigraphical sequence of prime numbers. Then, $n_i \geq q_i$ for each $1 \leq i \leq s$.*

If we strenghten the necessary condition of Proposition 4 by requiring $n_i \geq q_i + 1$, for each $1 \leq i \leq s$, we transform it into a sufficient condition. To see that, let us denote by $\mathbb{Z}_n$ the additive group of integers modulo n. For any set I of integers, the *circulant digraph* $\mathrm{Cay}(\mathbb{Z}_n; I)$ has vertex set $\mathbb{Z}_n$, and there is an arc from u to $u + a$, for every $u \in \mathbb{Z}_n$ and $a \in I$. When $I = \{1, 2, \ldots, d\}$ with $d \leq n - 1$, we have that $\mathrm{Cay}(\mathbb{Z}_n; I)$ is a PageRank uniform digraph whose outdegree sequence is $S : d^n$. Moreover, by constructing a digraph as the union of order n_i circulant digraphs whose vertices have all outdegree q_i, with $n_i \geq q_i + 1$, we construct a PageRank uniform digraph whose outdegree sequence is $S : q_1^{n_1}, q_2^{n_2}, \ldots, q_s^{n_s}$. That is, the digraph $D = \cup_{i=1}^{s} \mathrm{Cay}(\mathbb{Z}_{n_i}; \{1, 2, \ldots, q_i\})$ is PageRank uniform and its outdegree sequence is S. As we will next see, circulant digraphs play a prominent role in the characterization of PageRank uniform digraphical sequences composed of prime numbers.

Let $\mathbf{v}$ be a dimension n vector. The *circulant matrix* $\mathbf{C_v}$ with *generating vector* $\mathbf{v}$ is a $n \times n$ matrix such that its first column is $\mathbf{v}$ and the remaining columns of $\mathbf{C_v}$ are cyclic permutations of $\mathbf{v}$ with offset equal to the column index. For our purposes, a given vector $(a_1, \ldots, a_1, a_2, \ldots, a_2, \ldots, a_s, \ldots, a_s)$ in which a_1 appears b_1 times, a_2 appears b_2 times, and so on, will be denoted by $[a_1^{b_1}, a_2^{b_2}, \ldots, a_s^{b_s}]$. Note that the normalized link matrix of $\mathrm{Cay}(\mathbb{Z}_n; \{1, 2, \ldots, d\})$ is a circulant matrix with generating vector $[0^1, (\frac{1}{d})^d, 0^{n-d-1}]$ (see Figure 2).

Given the sequence $S : q_1^{n_1}, q_2^{n_2}, \ldots, q_s^{n_s}$ with $n_i \geq q_i + 1$, we can construct a PageRank uniform digraph whose outdegree sequence is S by considering circulant matrices. For each i, we just need to take vector $\mathbf{v_i} = [0^1, (\frac{1}{q_i})^{q_i}, 0^{n_i - q_i - 1}]$ and construct matrix $\mathbf{P}$ as follows,

$$\mathbf{P} = \begin{pmatrix} \mathbf{C_{v_1}} & 0 & \cdots & 0 \\ 0 & \mathbf{C_{v_2}} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{C_{v_s}} \end{pmatrix}$$

where $\mathbf{C_{v_i}}$ is the circulant matrix with generating vector $\mathbf{v_i}$. In fact, $\mathbf{P}$ is the normalized link matrix of $D = \cup_{i=1}^{s} \mathrm{Cay}(\mathbb{Z}_{n_i}; \{1, 2, \ldots, q_i\})$. Next, a characterization of the PageRank uniform outdigraphical sequences having prime outdegrees is given.

Theorem 2. *Let $S : q_1^{n_1}, q_2^{n_2}, \ldots, q_s^{n_s}$ be a sequence of prime numbers given in nondecreasing order. Then, S is a PageRank uniform outdigraphical sequence if and only if the following conditions hold:*

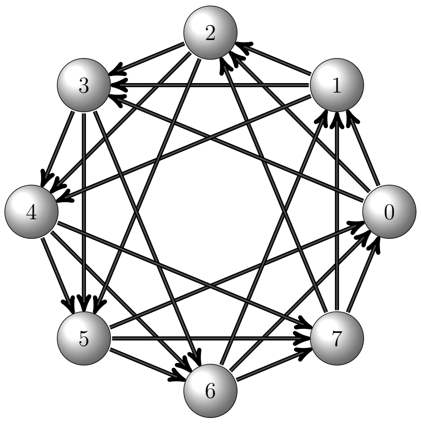
i) $n_i \geq q_i$*, for all* $1 \leq i \leq s$*, and*

ii) *if* $n_s = q_s$ *then* $n_s \leq \lfloor \frac{n}{2} \rfloor$*, where* $n = \sum_{i=1}^{s} n_i$.

Proof. The necessity of condition (i) was stated in Proposition 4. To see the necessity of (ii), let S be a PageRank uniform outdigraphical sequence such that $n_s = q_s$ and let $\mathbf{P}$ be the normalized matrix of a digraph D having S as its outdegree sequence.

We assert that, if $n_s > \lfloor \frac{n}{2} \rfloor$, then there exist integers $i < j$ and $i' > j'$, such that $p_{ij} = p_{i'j'} = \frac{1}{q_s}$. That is, matrix $\mathbf{P}$ has $\frac{1}{q_s}$ values at both sides of its main diagonal.

Assume to the contrary that $p_{ij} = \frac{1}{q_s}$ only when $i < j$ (p_{ij} is above the main diagonal of $\mathbf{P}$). Let p_{ij} be such a value. Since D is PageRank uniform and q_s is prime, there must be q_s elements taking value $\frac{1}{q_s}$ at row i and q_s elements taking that value at column j, each of them located above the main diagonal of $\mathbf{P}$. Row i has $(n - i)$ positions above the main diagonal and column j has $(j - 1)$



$$\mathbf{P} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ \frac{1}{3} & 0 & 0 & 0 & 0 & 0 & \frac{1}{3} & \frac{1}{3} \\ \frac{1}{3} & \frac{1}{3} & 0 & 0 & 0 & 0 & 0 & \frac{1}{3} \\ \frac{1}{3} & \frac{1}{3} & \frac{1}{3} & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} & 0 \end{pmatrix}$$

Fig. 2. The circulant digraph $\mathrm{Cay}(\mathbb{Z}_8; \{1, 2, 3\})$ and its normalized link matrix **P**. Matrix **P** is circulant with generating vector $[0^1, (\frac{1}{3})^3, 0^4]$.

so that we need,

$$\begin{cases} j - 1 \geq q_s, \\ n - i \geq q_s. \end{cases}$$

Besides, since $q_s = n_s > \lfloor \frac{n}{2} \rfloor$, we get,

$$1 + \lfloor \frac{n}{2} \rfloor < j < i < n - \lfloor \frac{n}{2} \rfloor,$$

so that $n > 1 + 2\lfloor \frac{n}{2} \rfloor$ which is impossible. Hence our assertion is true. As a consequence, every row and column of **P** containing a $\frac{1}{q_s}$ value should have in addition at least one zero (the one located in the main diagonal of **P**), so that $n_s > q_s$ and our assumption is contradicted.

Now, we will see that (i) and (ii) are sufficient conditions by constructing the matrix **P** of a PageRank uniform digraph having S as its outdegree sequence. We start by putting an $n_s \times n_s$ circulant matrix $\mathbf{C_{v_s}}$ into the right upper corner of **P**. This matrix contains all the $\frac{1}{q_s}$ values of **P** together with (maybe) some zeroes. Matrix **P** is constructed from $\mathbf{C_{v_s}}$ and an $(n - n_s) \times (n - n_s)$ matrix **Q** as follows,

$$\mathbf{P} = \begin{pmatrix} 0 & \mathbf{C_{v_s}} \\ \mathbf{Q} & 0 \end{pmatrix}.$$

Besides, matrix **Q** has the following form:

$$\mathbf{Q} = \begin{pmatrix} \mathbf{C_{v_1}} & 0 & \cdots & 0 \\ 0 & \mathbf{C_{v_2}} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{C_{v_{s-1}}} \end{pmatrix}$$

where each $\mathbf{C_{v_i}}$ is a circulant order n_i matrix containing the $\frac{1}{q_i}$ values of **P**. Notice that the main diagonal of **P** (it contains zeroes since self-loops are not allowed) crosses through either $\mathbf{C_{v_s}}$, $\mathbf{Q}$ or none of them, depending on the particular value of n_s.

Now, we define the generating vectors $\mathbf{v_i}$ of $\mathbf{C_{v_i}}$ by distinguishing two cases.

a) *Case* $n_s > \lfloor \frac{n}{2} \rfloor$*:* The generating vector $\mathbf{v_s}$ of matrix $\mathbf{C_{v_s}}$ is defined to be,

$$\mathbf{v_s} = [(\frac{1}{q_s})^{n-n_s}, 0^{n_s-q_s}, (\frac{1}{q_s})^{q_s+n_s-n}].$$

Notice that $\mathbf{v_s}$ contains at least one zero since $n_s \geq q_s + 1$ by (ii). Moreover, $n_s > \lfloor \frac{n}{2} \rfloor$ implies that the main diagonal of $\mathbf{P}$ crosses through $\mathbf{C_{v_s}}$, so that some null diagonal of $\mathbf{C_{v_s}}$ is part of the main diagonal of $\mathbf{P}$. Now, matrix $\mathbf{Q}$ has order $n - n_s$ and it is not crossed by the main diagonal of $\mathbf{P}$. Hence, we can define the generating vectors $\mathbf{v_i}$ of matrices $\mathbf{C_{v_i}}$ $(i \neq s)$ as follows,

$$\mathbf{v_i} = \begin{cases} [(\frac{1}{q_i})^{q_i}, 0^{n_i-q_i}] & \text{if } n_i > q_i, \\ [(\frac{1}{q_i})^{q_i}] & \text{if } n_i = q_i. \end{cases} \tag{2}$$

b) *Case* $n_s \leq \lfloor \frac{n}{2} \rfloor$*:* The generating vector $\mathbf{v_s}$ of matrix $\mathbf{C_{v_s}}$ is defined to be,

$$\mathbf{v_s} = \begin{cases} [(\frac{1}{q_s})^{q_s}, 0^{n_s-q_s}] & \text{if } n_s > q_s, \\ [(\frac{1}{q_s})^{q_s}] & \text{if } n_s = q_s. \end{cases}$$

In this case, the main diagonal of $\mathbf{P}$ falls below $\mathbf{C_{v_s}}$ and it may cross through $\mathbf{Q}$. The circulant matrices $\mathbf{C_{v_i}}$ that are not crossed by the main diagonal of $\mathbf{P}$, are defined as we did in the previous case (Eq. 2). By construction of $\mathbf{Q}$, if the main diagonal of $\mathbf{P}$ crosses through some $\mathbf{C_{v_i}}$ $(i \neq s)$ then $n_i \geq n_s$. But since $n_s \geq q_s > q_i$, we get $n_i > q_i$ and vector $\mathbf{v_i}$ contains at least one zero so that we can arrange its elements in such way that some null diagonal of $\mathbf{C_{v_i}}$ accomodates the main diagonal of $\mathbf{P}$.

In any case $\mathbf{P}$ is the normalized link matrix of a PageRank uniform digraph whose outdegree sequence is S.

□

Acknowledgements. The authors are partially funded by the Spanish Ministry of Economy and Competitiveness under project TIN2010-18978 and by the Government of Catalonia under Grant 2009SGR-442.

References

1. Chartrand, G., Lesniak, L.: Graphs & Digraphs, 4th edn. CRC Press, Boca Raton (2004)
2. Conde, J., López, N., Sebé, F.: PageRank regular digraphs with prime out-degrees. In: Proc. of 5th Intl. Workshop on Optimal Network Topologies (in press, 2013)
3. Cheng, J., Fu, A.W.-C., Liu, J.: K-Isomorphism: privacy preserving network publication against structural attacks. In: Proc. of SIGMOD 2010 (2010)

4. Fulkerson, D.R.: Zero-one matrices with zero traces. Pacific J. Math. 10, 831–836 (1960)
5. Hakimi, S.L.: On the realizability of a set of integers as degrees of the vertices of a graph. J. SIAM Appl. Math. 10, 496–506 (1962)
6. Hakimi, S.L., Schmeichel, E.F.: Graphs and their degree sequences: a survey. Theory and applications of graphs, pp. 225–235 (1976)
7. Havel, V.: A remark on the existence of finite graphs. Časopis Pěst. Mat. 80, 477–480 (1955)
8. Langville, A.N., Meyer, C.D.: Deeper inside pagerank. Internet Mathematics 1(3), 335–380 (2004)
9. Liu, K., Terzi, E.: Towards identity anonymization on graphs. In: Proc. of SIGMOD 2008 (2008)
10. López, N., Sebé, F.: Privacy-preserving release of blogosphere data in the presence of search engines. Inf. Processing and Management 49, 833–851 (2013)
11. Meyer, C.D.: Matrix analysis and applied linear algebra. SIAM: Society for Industrial and Applied Mathematics (2001)
12. Nash-Williams, C.S.J.A.: Hamilton circuits in graphs and digraphs. In: The Many Facets of Graph Theory, pp. 237–243. Springer (1968)
13. Page, L., Brin, S., Motwani, R., Winograd, T.: The pagerank citation ranking: bringing order to the web. Technical Report, Stanford InfoLab (1998)
14. Ryser, H.: Combinatorial Mathematics. Wiley, New York (1963)
15. Scott, J.: Social network analysis handbook. Sage Publications Inc. (2000)
16. Stokes, K., Torra, V.: n-Confusion: a generalization of k-anonymity. In: Proc. of 5th Intl. Workshop on Privacy and Anonymity in the Information Society (2012)
17. Wu, W., Xiao, Y., Wang, W., He, Z., Wang, Z.: K-Symmetry model for identity anonymization in social networks. In: Proc. of EDBT 2010 (2010)
18. Zhou, B., Pei, J.: Preserving privacy in social networks against neighborhood attacks. In: Proc. of ICDE 2008, pp. 506–515 (2008)
19. Zhou, B., Pei, J., Luk, W.-S.: A brief survey on anonymization techniques for privacy preserving publishing of social network data. ACM SIGKDD Explorations Newsletter 10(2), 12–22 (2008)
20. Zou, L., Chen, L., Öszu, M.T.: K-Automorphism: a general framework for privacy preserving network publication. In: Proc. of VLDB 2009 (2009)

On the Maximum Independent Set Problem in Subclasses of Subcubic Graphs*

Vadim Lozin[1], Jérôme Monnot[3,2], and Bernard Ries[2,3]

[1] DIMAP and Mathematics Institute, University of Warwick,
Coventry, CV4 7AL, UK
`V.Lozin@warwick.ac.uk`
[2] PSL, Université Paris-Dauphine, 75775 Paris Cedex 16, France
[3] CNRS, LAMSADE UMR 7243
`{monnot,ries}@lamsade.dauphine.fr`

Abstract. It is known that the maximum independent set problem is NP-complete for subcubic graphs, i.e. graphs of vertex degree at most 3. Moreover, the problem is NP-complete for H-free subcubic graphs whenever H contains a connected component which is not a tree with at most 3 leaves. We show that if every connected component of H is a tree with at most 3 leaves and at most 7 vertices, then the problem can be solved for H-free subcubic graphs in polynomial time.

Keywords: Independent set, Polynomial-time algorithm, Subcubic graph.

1 Introduction

In a graph, an *independent set* is a subset of vertices no two of which are adjacent. The maximum independent set problem consists in finding in a graph an independent set of maximum cardinality. This problem is generally NP-complete [3]. Moreover, it remains NP-complete even under substantial restriction, for instance, for planar graphs or subcubic graphs (i.e. graphs of vertex degree at most 3). In the present paper, we focus on subcubic graphs in the attempt to identify further restrictions which may lead to polynomial-time algorithms to solve the problem. One such restriction is known to be a bound on the chordality, i.e. on the length of a largest chordless cycle. Graphs of bounded degree and bounded chordality have bounded tree-width [2], and hence the problem can be solved in polynomial time for such graphs. In terms of forbidden induced subgraphs bounded chordality means excluding large chordless cycles, i.e. cycles $C_k, C_{k+1}, \ldots$ for a constant k. More generally, it was recently shown in [6] that excluding large apples (all definitions can be found in the end of the introduction) together with bounded degree leads to a polynomial-time algorithm to solve the problem. In both cases (i.e. for graphs without large cycles and for graphs

* The first author gratefully acknowledges support from DIMAP - the Center for Discrete Mathematics and its Applications at the University of Warwick, and from EPSRC, grant EP/I01795X/1.

© Springer-Verlag Berlin Heidelberg 2013

without large apples) the restrictions are obtained by excluding infinitely many graphs. In the present paper, we study subclasses of subcubic graphs obtained by excluding *finitely many* graphs. A necessary condition for polynomial-time solvability of the problem in such classes was given in [1] and can be stated as follows: the maximum independent set problem can be solved in polynomial time in the class of graphs defined by a *finite* set Z of forbidden induced subgraphs *only if* Z contains a graph every connected component of which is a tree with at most three leaves. In other words, for polynomial-time solvability of the problem we must exclude a graph every connected component of which has the form $S_{i,j,k}$ represented in Figure 1. Whether this condition is sufficient for polynomial-time solvability of the problem is a big open question.

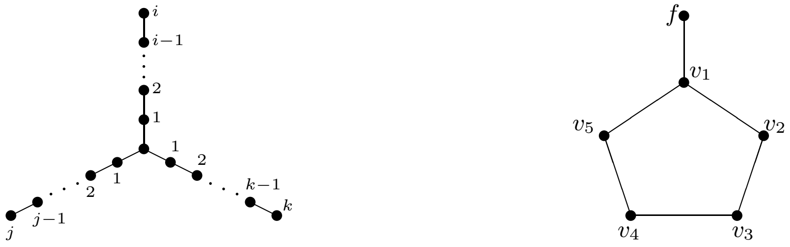


Fig. 1. Graphs $S_{i,j,k}$ (left) and A_5 (right)

Without the restriction on vertex degree, polynomial-time solvability of the problem in classes of $S_{i,j,k}$-free graphs was shown only for very small values of i, j, k. In particular, the problem can be solved for $S_{1,1,1}$-free (claw-free) graphs [11], $S_{1,1,2}$-free (fork-free) graphs [5], and $S_{0,1,1}+S_{0,1,1}$-free ($2P_3$-free) graphs [7]. The complexity of the problem in $S_{0,2,2}$-free (P_5-free) graphs remains an open problem in spite of the multiple partial results on this topic (see e.g. [4,8,9,10]).

With the restriction on vertex degree, we can do much better. In particular, we can solve the problem for $S_{1,j,k}$-free graphs of bounded degree for any j and k, because by excluding $S_{1,j,k}$ we exclude large apples. However, nothing is known about classes of $S_{i,j,k}$-free graphs of bounded degree where all three indices i, j, k are at least 2. To make a progress in this direction, we consider best possible restrictions of this type, i.e. we study $S_{2,2,2}$-free graphs of vertex degree at most 3, and show that the problem is solvable in polynomial time in this class. More generally, we show that the problem is polynomial-time solvable in the class of H-free subcubic graphs, where H is a graph every connected component of which is isomorphic to $S_{2,2,2}$ or to $S_{1,j,k}$.

The organization of the paper is as follows. In the rest of this section, we introduce basic definitions and notations. In Section 2 we prove a number of preliminary results. Finally, in Section 3 we present a solution.

All graphs in this paper are simple, i.e. undirected, without loops and multiple edges. The vertex set and the edge set of a graph G are denoted by $V(G)$ and $E(G)$, respectively. For a vertex $v \in V(G)$, we denote by $N(v)$ the neighborhood of v, i.e., the set of vertices adjacent to v, and by $N[v]$ the closed neighbourhood of v, i.e. $N[v] = N(v) \cup \{v\}$. For $v, w \in V(G)$, we set $N[v,w] = N[v] \cup N[w]$.

The *degree* of v is the number of its neighbors, i.e., $d(v) = |N(v)|$. The subgraph of G induced by a set $U \subseteq V(G)$ is obtained from G by deleting the vertices outside of U and is denoted $G[U]$. If no induced subgraph of G is isomorphic to a graph H, then we say that G is H-free. Otherwise we say that G contains H. If G contains H, we denote by $[H]$ the subgraph of G induced by the vertices of H and all their neighbours. As usual, by C_p we denote a chordless cycle of length p. Also, an *apple* A_p, $p \geq 4$, is a graph consisting of a cycle C_p and a vertex f which has exactly one neighbour on the cycle. We call vertex f the *stem* of the apple. See Figure 1 for the apple A_5. The size of a maximum independent set in G is called the *independence number* of G and is denoted $\alpha(G)$.

2 Preliminary Results

We start by quoting the following result from [6].

Theorem 1. *For any positive integers d and p, the maximum independent set problem is polynomial-time solvable in the class of $(A_p, A_{p+1}, \ldots)$-free graphs with maximum vertex degree at most d.*

We solve the maximum independent set problem for $S_{2,2,2}$-free subcubic graphs by reducing it to subcubic graphs without large apples.

Throughout the paper we let G be an $S_{2,2,2}$-free subcubic graph and $K \geq 1$ a large fixed integer. If G contains no apple A_p with $p \geq K$, then the problem can be solved for G by Theorem 1. Therefore, from now on we assume that G contains an induced apple A_p with $p \geq K$ formed by a chordless cycle $C = C_p$ of length p and a stem f. We denote the vertices of C by $v_1, \ldots, v_p$ (listed along the cycle) and assume without loss of generality that the only neighbour of f on C is v_1 (see Figure 1 for an illustration).

If v_1 is the only neighbour of f in G, then the deletion of v_1 together with f reduces the independence number of G by exactly 1. This can be easily seen and also is a special case of a more general reduction described in Section 2.1. The deletion of f and v_1 destroys the apple A_p. The idea of our algorithm is to destroy all large apples by means of other simple reductions that change the independence number by a constant. Before we describe the reductions in Section 2.1, let us first characterize the local structure of G in the case when the stem f has a neighbor different from v_1.

Lemma 1. *If f has a neighbor g different from v_1, then g has at least one neighbor on C and the neighborhood of g on C is of one of the 8 types represented in Figure 2.*

Proof. First observe that g must have a neighbor among $\{v_{p-1}, v_p, v_2, v_3\}$, since otherwise we obtain an induced $S_{2,2,2}$. If g has only 1 neighbor on C, then clearly we obtain configuration (1) or (2).

Now assume that g has two neighbors on C. Suppose first that g is adjacent neither to v_2 nor to v_p. Then g must be adjacent to at least one of v_{p-1}, v_3.

Without loss of generality, we may assume that g is adjacent to v_{p-1} and denote the third neighbor of g by v_j. If $2 < j < p-3$, then we clearly obtain an induced $S_{2,2,2}$ centered at g. Otherwise, we obtain configuration (3) or (4).

Now assume g is adjacent to one of v_2, v_p, say to v_p, and again denote the third neighbor of g by v_j. If $j \in \{p-2, p-1\}$, then we obtain configuration (5) or (6). If $j \in \{2, 3\}$, then we obtain configuration (7) or (8). If $3 < j < p-2$, then G contains an $S_{2,2,2}$ induced by $\{v_{j-2}, v_{j-1}, v_j, v_{j+1}, v_{j+2}, g, f\}$. □

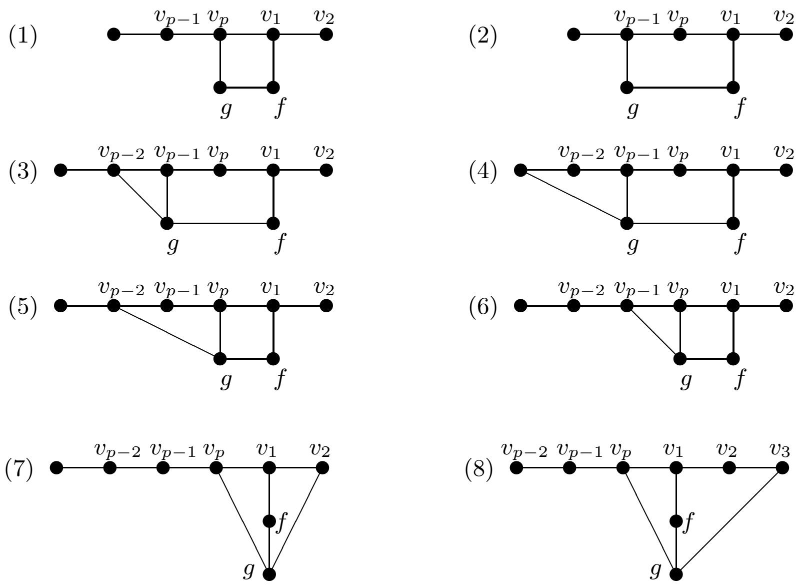

(1)
v_{p-1} v_p v_1 v_2
g f
(2)
v_{p-1} v_p v_1 v_2
g f
(3)
v_{p-2} v_{p-1} v_p v_1 v_2
g f
(4)
v_{p-2} v_{p-1} v_p v_1 v_2
g f
(5)
v_{p-2} v_{p-1} v_p v_1 v_2
g f
(6)
v_{p-2} v_{p-1} v_p v_1 v_2
g f
(7)
v_{p-2} v_{p-1} v_p v_1 v_2
f
g
(8)
v_{p-2} v_{p-1} v_p v_1 v_2 v_3
f
g

Fig. 2. $A_p + g$

2.1 Graph Reductions

***H*-subgraph reduction** Let H be an induced subgraph of G.

Lemma 2. *If $\alpha(H) = \alpha([H])$, then $\alpha(G - [H]) = \alpha(G) - \alpha(H)$.*

Proof. Since any independent set of G contains at most $\alpha([H])$ vertices in $[H]$, we know that $\alpha(G - [H]) \geq \alpha(G) - \alpha([H])$. Now let S be an independent set in $G - [H]$ and A an independent set of size $\alpha(H)$ in H. Then $S \cup A$ is an independent set in G and hence $\alpha(G) \geq \alpha(G - [H]) + \alpha(H)$. Combining the two inequalities together with $\alpha(H) = \alpha([H])$, we conclude that $\alpha(G - [H]) = \alpha(G) - \alpha(H)$. □

The deletion of $[H]$ in the case when $\alpha(H) = \alpha([H])$ will be called the *H-subgraph reduction.* For instance, if a vertex v has degree 1, then the deletion of v together with its only neighbour is the H-subgraph reduction with $H = \{v\}$.

***Φ*-Reduction.** Let us denote by Φ the graph represented on the left of Figure 3. The transformation replacing Φ by Φ' as shown in Figure 3 will be called *Φ-reduction.*

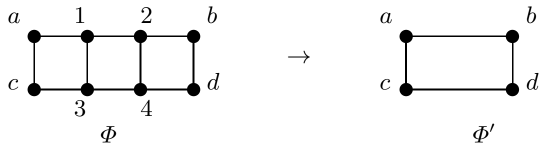


Fig. 3. Φ-reduction

Lemma 3. *By applying the Φ-reduction to an $S_{2,2,2}$-free subcubic graph G, we obtain an $S_{2,2,2}$-free subcubic graph G' such that $\alpha(G') = \alpha(G) - 2$.*

Proof. Let S be an independent set in G. Clearly it contains at most two vertices in $\{a, b, c, d\}$ and at most two vertices in $\{1, 2, 3, 4\}$. Denote $X = S \cap \{1, 2, 3, 4\}$. If the intersection $S \cap \{a, b, c, d\}$ contains at most one vertex or one of the pairs $\{a, d\}$, $\{b, c\}$, then $S - X$ is an independent set in G' of size at least $\alpha(G) - 2$. If $S \cap \{a, b, c, d\} = \{a, b\}$, then X contains at most one vertex and hence $S - (X \cup \{b\})$ is an independent set in G' of size at least $\alpha(G) - 2$. Therefore, $\alpha(G') \geq \alpha(G) - 2$.

Now let S' be an independent set in G'. Then the intersection $S' \cap \{a, b, c, d\}$ contains at most two vertices. If $S' \cap \{a, b, c, d\} = \{a, d\}$, then $S' \cup \{2, 3\}$ is an independent set of size $\alpha(G') + 2$ in G. Similarly, if $S' \cap \{a, b, c, d\}$ contains at most one vertex, then G contains an independent set of size at least $\alpha(G') + 2$. Therefore, $\alpha(G) \geq \alpha(G') + 2$. Combining the two inequalities, we conclude that $\alpha(G') = \alpha(G) - 2$.

Now let us show that G' is an $S_{2,2,2}$-free subcubic graph. The fact that G' is subcubic is obvious. Assume to the contrary that it contains an induced subgraph H isomorphic to $S_{2,2,2}$. If H contains none of the edges ab and cd, then clearly H is also an induced $S_{2,2,2}$ in G, which is impossible. If S contains both edges ab and cd, then it contains $C_4 = (a, b, c, d)$, which is impossible either. Therefore, H has exactly one of the two edges, say ab. If vertex b has degree 1 in H, then by replacing b by vertex 1 we obtain an induced $S_{2,2,2}$ in G. By symmetry, a also is not a vertex of degree 1 in H. Therefore, we may assume, without loss of generality, that a has degree 3 and b has degree 2 in H. Let us denote by x the only neighbour of b in H. Then $(H - \{b, x\}) \cup \{1, 2\}$ is an induced $S_{2,2,2}$ in G. This contradiction completes the proof. □

***AB*-Reduction.** The AB-reduction deals with two graphs A and B represented in Figure 4. We assume that the vertices v_i belong to the cycle $C = C_p$, and the vertices p_j are outside of C.

Lemma 4. *If G contains an induced subgraph isomorphic to A, then*

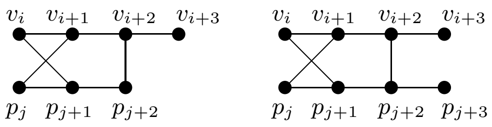


Fig. 4. Induced subgraphs A (left) and B (right)

– *either A can be extended to an induced subgraph of G isomorphic to B in which case p_{j+2} can be deleted without changing $\alpha(G)$*
– *or the deletion of $N[v_i] \cup N[p_j]$ reduces the independence number by 2.*

Proof. Assume first that A can be extended to an induced B (by adding vertex p_{j+3}). Consider an independent set S containing vertex p_{j+2}. Then S contains neither p_{j+1} nor p_{j+3} nor v_{i+2}. If neither p_j nor v_i belongs to S, then p_{j+2} can be replaced by p_{j+1} in S. Now assume, without loss of generality, that v_i belongs to S. Then $v_{i+1} \notin S$ and therefore we may assume that $v_{i+3} \in S$, since otherwise p_{j+2} can be replaced by v_{i+2} in S. If p_{j+3} has one more neighbour x in S (different from p_{j+2}), then vertices $v_i, v_{i+2}, v_{i+3}, p_{j+1}, p_{j+2}, p_{j+3}$ and x induce an $S_{2,2,2}$ in G (because the 3 endpoints are in S and the internal vertices have degree 3 in A). Therefore, we conclude that p_{j+2} is the only neighbour of p_{j+3} in S, in which case p_{j+2} can be replaced by p_{j+3} in S. Thus, for any independent S in G containing vertex p_{j+2}, there is an independent set of size $|S|$ which does not contain p_{j+2}. Therefore, the deletion of p_{j+2} does not change the independence number of G.

Now let us assume that A cannot be extended to B. Clearly, every independent set S in $G - N[v_i, p_j]$ can be extended to an independent set of size $|S| + 2$ in G by adding to S vertices v_i and p_j. Therefore, $\alpha(G) \geq \alpha(G - N[v_i, p_j]) + 2$.

Conversely, consider an independent set S in G. If it contains at most 2 vertices in $N[v_i, p_j]$, then by deleting these vertices from S we obtain an independent set of size at least $|S| - 2$ in $G - N[c_i, p_j]$.

Suppose now that S contains more than 2 vertices in $N[v_i, p_j]$. Let us show that in this case it must contain exactly three vertices in $N[v_i, p_j]$, two of which are v_{i+1} and p_{j+1}. Indeed, $N[v_i, p_j]$ contains at most 6 vertices: $v_{i-1}, v_i, v_{i+1}, p_j$, p_{j+1} and possibly some vertex x. Moreover, if x exists, then it is adjacent to v_{i-1}, since otherwise an $S_{2,2,2}$ arises induced either by vertices $x, p_j, p_{j+1}, p_{j+2}, v_{i+2}$, v_{i-1}, v_i (if p_{j+2} is not adjacent to v_{i-1}) or by vertices $p_j, v_{i+1}, v_{i+2}, v_{i+3}, v_{i+4}$, v_{i-1}, p_{j+2} (if p_{j+2} is adjacent to v_{i-1}). Therefore, S cannot contain more than three vertices in $N[v_i, p_j]$, and if it contains tree vertices, then two of them are v_{i+1} and p_{j+1}. As a result, S contains neither v_{i+2} nor p_{j+2}. If each of v_{i+2} and p_{j+2} has one more neighbour in S (different from v_{i+1} and p_{j+1}), then A can be extended to B, which contradicts our assumption. Therefore, we may assume without loss of generality that p_{j+1} is the only neighbour of p_{j+2} in S. In this case, the deletion from $N[v_i, p_j]$ of the three vertices of S and adding to it vertex p_{j+2} results in an independent set of size $|S| - 2$ in $G - N[v_i, p_j]$.

Therefore, $\alpha(G - N[v_i, p_j]) \geq \alpha(G) - 2$. Combining with the inverse inequality, we conclude that $\alpha(G - N[v_i, p_j]) = \alpha(G) - 2$. □

Other Reductions. Two other reductions that will be helpful in the proof are the following.

– The A^*-*reduction* applies to an induced A^* (Figure 5) and consists in deleting vertex p_{j+2}.
– The *House-reduction* applies to an induced *House* (Figure 5) and consists in deleting the vertices of the triangle $v_{i+2}, v_{i+3}, p_{j+2}$.

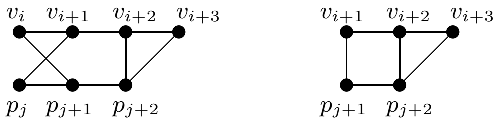


Fig. 5. Induced subgraphs A^* (left) and *House* (right)

Lemma 5. *The A^*-reduction does not change the independence number, and the House-reduction reduces the independence number by exactly 1.*

Proof. Assume G contains an induced A^* and let S be an independent set containing p_{j+2}. If S does not contain v_{i+1}, then p_{j+2} can be replaced by v_{i+2}, and if S contains v_{i+1}, then p_{j+2} can be replaced by p_{j+1}. Therefore, G has an independent set of size $|S|$ which does not contain p_{j+2} and hence the deletion of p_{j+2} does not change the independence number.

Assume G contains an induced *House* and let S be a maximum independent set in G. Then obviously at most one vertex of the triangle $v_{i+2}, v_{i+3}, p_{j+2}$ belongs to S. On the other hand, S must contain at least one vertex of this triangle. Indeed, if none of the three vertices belong to S, then each of them must have a neighbour in S (else S is not maximum), but then both v_{i+1} and p_{j+1} belong to S, which is impossible. Therefore, every maximum independent set contains exactly one vertex of the triangle, and hence the deletion of the triangle reduces the independence number by exactly 1. □

3 Solving the Problem

In the subgraph of G induced by the vertices having at least one neighbor on $C = C_p$, every vertex has degree at most 2 and hence every connected component in this subgraph is either a path or a cycle. Let F be the component of this subgraph containing the stem f. In what follows we analyze all possible cases for F and show that in each case the apple A_p can be destroyed by means of graph reductions described above or by other simple reductions.

Lemma 6. *If F is a cycle, then A_p can be destroyed by graph reductions that change the independence number by a constant.*

Proof. If F is a triangle, then, according to Lemma 1, the neighbors of F in C are three consecutive vertices of C. In this case, F together with two consecutive vertices of C form a House and hence the deletion of F reduces the independence number of G by exactly one.

Assume F is a cycle of length 4 induced by vertices f_1, f_2, f_3, f_4. With the help of Lemma 1 it is not difficult to see that the neighbors of F in C must be consecutive vertices, say $v_i, \ldots, v_{i+3}$, and the only possible configuration, up to symmetry, is this: v_i is a neighbor of f_1, v_{i+1} is a neighbor of f_2, v_{i+2} is a neighbor of f_4, v_{i+3} is a neighbor of f_3. In this case, the deletion of vertex v_{i+1} does not change the independence number of G. To show this, consider an independent set S containing vertex v_{i+1}. Then S does not contain $2, v_i, v_{i+2}$. If $f_4 \in S$, then $f_1, f_3 \notin S$, in which case v_{i+1} can be replaced by f_2 in S. So, assume $f_4 \notin S$. If $f_3 \notin S$, then we can assume that $v_{i+3} \in S$ (else v_{i+1} can be replaced by f_3 in S), in which case v_{i+1}, v_{i+3} can be replaced by v_{i+2}, f_3. So, assume $f_3 \in S$, and hence $v_{i+3} \notin S$. But now v_{i+1} can be replaced by v_{i+2} in S. This proves that for every independent set S containing v_{i+1}, there is an independent set of the same size that does not contain v_{i+1}. Therefore, the deletion of v_{i+1} does not change the independence number of G.

Assume F is a cycle of length 5 induced by vertices f_1, f_2, f_3, f_4, f_5. With the help of Lemma 1 it is not difficult to verify that the neighbors of F in C must be consecutive vertices, say $v_i, \ldots, v_{i+4}$, and the only possible configuration, up to symmetry, is this this: f_1 is adjacent to v_i, f_2 is adjacent to v_{i+1}, f_3 is adjacent to v_{i+3}, f_4 is adjacent to v_{i+4}, f_5 is adjacent to v_{i+2}. But then the vertices $f_2, f_3, f_4, f_5, v_{i+2}, v_{i+4}, v_{i+5}$ induce an $S_{2,2,2}$.

If F is a cycle of length more than 5, then an induced $S_{2,2,2}$ can be easily found. □

Lemma 7. *If F is a path with at least 5 vertices, then A_p can be destroyed by graph reductions that change the independence number by a constant.*

Proof. Assume F has at least 5 vertices $f_1, \ldots, f_5$. Denote the neighbour of f_3 on C by v_i. Assume v_{i-1} has a neighbour in $\{f_1, f_5\}$, say f_1 (up to symmetry). By Lemma 1, f_2 is adjacent either to v_{i-2} or v_{i+1}.

Let first f_2 be adjacent to v_{i+1}. Then either f_1 is not adjacent to v_{i-2}, in which case the vertices $v_{i-2}, \ldots, v_{i+1}, f_1, f_2, f_3$ induce an A, or f_1 is adjacent to v_{i-2}, in which case f_4 is adjacent to v_{i+2} (by Lemma 1) and hence the vertices $v_i, \ldots, v_{i+3}, f_2, f_3, f_4$ induce an A. In either case, we can apply Lemma 4.

Suppose now that f_2 is adjacent to v_{i-2}. Then f_1 is not adjacent to v_{i+1}, since otherwise f_4 is adjacent to v_{i+2} (by Lemma 1), in which case the vertices $v_{i+1}, \ldots, v_{i+4}, f_1, f_3, f_4$ induce an $S_{2,2,2}$. As a result, vertices $v_{i-2}, \ldots, v_{i+1}$, f_1, f_2, f_3 induce an A and we can apply Lemma 4.

The above discussion shows that v_{i-1} has no neighbour in $\{f_1, f_5\}$. By symmetry, v_{i+1} has no neighbour in $\{f_1, f_5\}$. Then each of v_{i-1} and v_{i+1} has a neighbour in $\{f_2, f_4\}$, since otherwise $f_1, \ldots, f_5, v_i$ together with v_{i-1} or with

v_{i+1} induce an $S_{2,2,2}$. Up to symmetry, we may assume that v_{i-1} is adjacent to f_2, while v_{i+1} is adjacent to f_4.

If f_1 is adjacent to v_{i-2} or f_5 is adjacent to v_{i+2}, then an induced Φ arises, in which case we can apply the Φ-reduction. Therefore, we can assume that f_1 is adjacent to v_{i-3}, while f_5 is adjacent to v_{i+3}.

We may assume that vertex v_{i-2} has no neighbour x different from v_{i-3}, v_{i-1}, since otherwise x must be adjacent to f_1 (else vertices $x, v_{i-2}, v_{i-1}, v_i, v_{i+1}, f_1, f_2$ induce an $S_{2,2,2}$), in which case $v_{i-3}, \ldots, v_i, x, f_1, f_2$ induce an A and we can apply the AB-reduction. Similarly, we may assume that vertex f_1 has no neighbour x different from v_{i-3}, f_2. But then $d(f_1) = d(v_{i-2}) = 2$ and we can apply the H-subgraph reduction with $H = \{v_{i-2}, f_1\}$. □

Lemma 8. *If F is a path with 4 vertices, then A_p can be destroyed by graph reductions that change the independence number by a constant.*

Proof. Let F be a path (f_1, f_2, f_3, f_4). Without loss of generality we assume that f_2 is adjacent to v_i and f_3 to v_j with $j > i$. By Lemma 1, $j = i + 1$ or $j = i + 2$.

Case $j = i+1$. Assume f_1 is adjacent to v_{i+2}. Then vertices $v_i, v_{i+1}, v_{i+2}, v_{i+3}, f_1, f_2, f_3$ induce either the graph A (if f_1 is not adjacent to v_{i+3}) or the graph A^* (if f_1 is adjacent to v_{i+3}), in which case we can apply either Lemma 4 or Lemma 5. Therefore, we may assume that f_1 is not adjacent to v_{i+2}, and by symmetry, f_4 is not adjacent to v_{i-1}. Then by Lemma 1, f_1 must have a neighbour in $\{v_{i-2}, v_{i-1}\}$ and f_4 must have a neighbour in $\{v_{i+2}, v_{i+3}\}$.

Assume that f_4 is adjacent to v_{i+3}. If v_{i+2} has a neighbour x outside of the cycle C, then x is not adjacent to f_4 (else F has more than 4 vertices) and hence $v_{i-1}, v_i, v_{i+1}, v_{i+2}, x, f_3, f_4$ induce an $S_{2,2,2}$. Therefore, the degree of v_{i+2} in G is 2. Similarly, the degree of f_4 in G is two. But now we can apply the H-subgraph reduction with $H = \{v_{i+2}, f_4\}$. This allows us to assume that f_4 is not adjacent to v_{i+3}, and by symmetry, f_1 is not adjacent to v_{i-2}. But then f_1 is adjacent to v_{i-1} and f_4 is adjacent to v_{i+2}, in which case we can apply the Φ-reduction to the subgraph of G induced by $v_{i-1}, v_i, v_{i+1}, v_{i+2}, f_1, f_2, f_3, f_4$.

Case $j = i + 2$. If f_1 or f_4 is adjacent to v_{i+1}, then an induced graph A arises, in which case we can apply Lemma 4. Then f_1 must be adjacent to v_{i-1}, since otherwise it adjacent to v_{i-2} (by Lemma 1), in which case vertices $v_{i-2}, f_1, f_2, f_3, f_4, v_i, v_{i+1}$ induce an $S_{2,2,2}$. By symmetry, f_4 is adjacent to v_{i+3}.

If f_1 is adjacent to v_{i-2}, then we can apply the *House*-reduction to the subgraph of G induced by $v_{i-2}, v_{i-1}, v_i, f_1, f_2$, and if f_1 is adjacent to v_{i-3}, then vertices $v_{i-3}, f_1, f_2, f_3, f_4, v_i, v_{i+1}$ induce an $S_{2,2,2}$. Therefore, we may assume by Lemma 1 that f_1 has degree 2 in G. By symmetry, f_4 has has degree 2. Also, to avoid an induced $S_{2,2,2}$, we conclude that v_{i+1} has degree 2. But now we apply the H-subgraph reduction with $H = \{f_1, v_i, v_{i+2}, f_4\}$, which reduces the independence number of G by 4. □

Lemma 9. *If F is a path with 3 vertices, then A_p can be destroyed by graph reductions that change the independence number by a constant.*

Proof. Assume F is a path (f_1, f_2, f_3). Without loss of generality let f_2 be adjacent to v_1. Since G is $S_{2,2,2}$-free, each of f_1 and f_3 must have at least one neighbor in $\{v_{p-1}, v_p, v_2, v_3\}$. Denote $L = \{v_{p-1}, v_p\}$ and $R = \{v_2, v_3\}$.

Case (a): f_1 and f_3 have both a neighbor in R. Due to the symmetry, we may assume without loss of generality that f_1 is adjacent to v_2, while f_3 is adjacent to v_3. Then we may further assume that f_1 is adjacent to v_4, since otherwise vertices $v_1, v_2, v_3, v_4, f_1, f_2, f_3$ induced either an A (if f_3 is not adjacent to v_4) or an A^* (if f_3 is adjacent to v_4), in which case we can apply either Lemma 4 or Lemma 5. But now the deletion of f_3 does not change the independence number of G. Indeed, let S be an independent set containing f_3. If $f_1 \in S$, then f_3 can be replaced by v_3. If $f_1 \notin S$, then we can assume that $v_1 \in S$ (else f_3 can be replaced by f_2), in which case f_3, v_1 can be replaced by f_2, v_2.

The above discussion allows us to assume, without loss of generality, that f_1 has no neighbor in R, while f_3 has no neighbor in L.

Case (b): f_3 is adjacent to v_3. Then we may assume that f_3 is not adjacent to v_2, since otherwise we can apply the *House*-reduction to the subgraph of G induced by v_1, v_2, v_3, f_3, f_2. Let us show that in this case

– *the degree of v_2 is 2.* Assume to the contrary v_2 has a third neighbour x. Then x is not adjacent to v_{p-1}, since otherwise G contains an $S_{2,2,2}$ induced either by $v_{p-1}, x, v_2, v_1, f_2, v_3, v_4$ (if x is not adjacent to v_4) or by $v_{p-2}, v_{p-1}, x, v_2, v_1, v_4, v_5$ (if x is adjacent to v_4). This implies that x is adjacent to v_p, since otherwise $x, v_2, v_1, f_2, f_3, v_p, v_{p-1}$ induced an $S_{2,2,2}$. As a result, f_1 is adjacent to v_{p-1}. Due to the degree restriction, x may have at most one neighbour in $\{v_{p-3}, v_{p-2}, v_4, v_5\}$. By symmetry, we may assume without loss of generality that x has no neighbour in $\{v_4, v_5\}$. Also, f_3 has no neighbour in $\{v_4, v_5\}$, since otherwise this neighbour together with $v_{p-1}, f_1, f_2, f_3, v_1, v_2$ would induce an $S_{2,2,2}$. But now $x, v_2, v_3, v_4, v_5, f_3, f_2$ induce an $S_{2,2,2}$. This contradiction complete the proof of the claim.

If f_3 also has degree two, then we can apply the H-subgraph reduction with $H = \{v_3, f_3\}$. Therefore, may assume that f_3 has one more neighbour, which must be, by Lemma 1, either v_4 or v_5. If f_3 is adjacent to f_5, then $f_1, f_2, f_3, v_5, v_6, v_3, v_2$ induce an $S_{2,2,2}$. Therefore, f_3 is adjacent to v_4. But now v_3 can be deleted without changing the independence number. Indeed, let S be an independent set containing v_3. If S does not contain v_1, then v_3 can be replaced by v_2, and if S contains v_1, then v_1, v_3 can be replaced by v_2, f_3.

Cases (a) and (b) reduce the analysis to the situation when f_1 is adjacent to v_p and non-adjacent to v_{p-1}, while f_3 is adjacent to v_2 and non-adjacent to v_3. If f_3 is adjacent to v_4, then vertices $v_p, v_1, v_2, v_3, v_4, f_1, f_2, f_3$ induced the graph Φ, in which case we can apply Lemma 3. Therefore, we can assume by Lemma 1 that the degree of f_3 is 2, and similarly the degree of f_1 is 2. But now we can apply the H-subgraph reduction with $H = \{f_1, v_1, f_3\}$, which reduces the independence number of G by 3. □

Lemma 10. *If F is a path with 2 vertices, then A_p can be destroyed by graph reductions that change the independence number by a constant.*

Proof. If F is a path with 2 vertices, we deal with the eight cases represented in Figure 2. It is easy to see that in cases (1) and (7), every maximum independent set must contain exactly one of f, g and thus by deleting f, g we reduce the independence number by exactly 1.

In case (5), the deletion of f, g also reduces the independence number by exactly 1. Indeed, let S be a maximum independent set containing neither f nor g. Since S is maximum it must contain v_1, v_{p-2} and hence it does not contain v_p, v_{p-1}. But then $(S \setminus \{v_1\}) \cup \{v_p, f\}$ is an independent set larger than S, contradicting the choice of S. Therefore, every maximum independent set contains exactly one of f and g and hence $\alpha(G - \{f, g\}) = \alpha(G) - 1$.

In case (2), the deletion of the set $X = \{v_{p-1}, v_p, v_1, f, g\}$ reduces the independence number of the graph by exactly 2. Indeed, any independent set of G contains at most two vertices in X, and hence $\alpha(G - X) \geq \alpha(G) - 2$. Assume now that S is a maximum independent set in $G - X$. If $v_2 \notin S$, then $S \cup \{v_1, g\}$ is an independent set in G of size $\alpha(G - X) + 2$. Now assume $v_2 \in S$. By symmetry, $v_{p-2} \in S$. Assume v_p has a neighbour x in S. Then x is adjacent neither to v_{p-2} nor to v_2, as all three vertices belong to S. Also, x cannot be adjacent to both v_{p-3} and v_3, since otherwise an induced $S_{2,2,2}$ can be easily found. But if x is not adjacent, say, to v_3, then $x, v_p, v_1, v_2, v_3, f, g$ induce an $S_{2,2,2}$. This contradiction shows that v_p has no neighbours in S. Therefore, $S \cup \{v_p, f\}$ is an independent set in G of size $\alpha(G - X) + 2$, and hence $\alpha(G) \geq \alpha(G - X) + 2$. Combining the two inequalities, we conclude that $\alpha(G - X) = \alpha(G) - 2$.

In case (3), we may delete g without changing the independence number, because in any independent set S containing g, vertex g can be replaced either by v_{p-1} (if S does not contain v_p) or by f (if S contains v_p). In case (6), we apply the *House*-reduction.

In cases (4) and (8), we find another large apple A' whose stem f' belongs to a path F' with at least 3 vertices. In case (4), A' is induced by the cycle $v_1, \ldots, v_{p-3}, g, f$ with stem $f' = v_{p-1}$, and in case (8) the apple is induced by the cycle $v_3, \ldots, v_p, g$ with stem $f' = v_1$. In both cases, the situation can be handled by one of the previous lemmas. □

Theorem 2. *Let H be a graph every connected component of which is isomorphic either to $S_{2,2,2}$ or to $S_{1,j,k}$. The maximum independent set problem can be solved for H-free graphs of maximum vertex degree at most 3 in polynomial time.*

Proof. First, we show how to solve the problem in the case when $H = S_{2,2,2}$. Let $G = (V, E)$ be an $S_{2,2,2}$-free subcubic graph and let K be a large fixed constant. We start by checking if G contains an apple A_p with $p \geq K$. To this end, we detect every induced $S_{1,k,k}$ with $k = K/2$, which can be done in time n^K. If G is $S_{1,k,k}$-free, then it is obviously A_p-free for each $p \geq K$. Assume a copy of $S_{1,k,k}$ has been detected and let x, y be the two vertices of this copy at distance k from the center of $S_{1,k,k}$. We delete from G all vertices of $V(S_{1,k,k}) - \{x, y\}$ and all their neighbours, except x and y, and determine if in the resulting graph there is a path connecting x to y. It is not difficult to see that this procedure can be implemented in polynomial time.

Assume G contains an induced apple A_p with $p \geq K$. If the stem of the apple has degree 1 in G, we delete it together with its only neighbour, which destroys the apple and reduces the independence number of G by exactly one. If the stem has degree more than 1, we apply one of the lemmas of Section 3 to destroy A_p and reduce the independence number of G. It is not difficult to see that all the reductions used in the lemmas can be implemented in polynomial time.

Thus in polynomial time we reduce the problem to a graph G' which does not contain any apple A_p with $p \geq K$, and then we find a maximum independent set in G' with the help of Theorem 1. This also shows that in polynomial time we can compute $\alpha(G)$, since we know the difference between $\alpha(G)$ and $\alpha(G')$. To find a maximum independent set in G, we take an arbitrary vertex $v \in V(G)$. If $\alpha(G - v) = \alpha(G)$, then there is a maximum independent set in G that does not contain v and hence v ignored (deleted). Otherwise, v belongs to every maximum independent set in G and hence it must be included in the solution. Therefore, in polynomial time we can find a maximum independent set in G. This completes the proof of the theorem in the case when $H = S_{2,2,2}$.

By Theorem 1 we also know how to solve the problem in the case when $H = S_{1,j,k}$. Now we assume that H contains $s > 1$ connected components. Denote by S any of the components of H and let H' be the graph obtained from H by deleting S. Consider an H-free graph G. If G does not contain a copy of S, the problem can be solved for G by the first part of the proof. So, assume G contains a copy of S. By deleting from G the vertices of $[S]$ we obtain a graph G' which is H'-free and hence the problem can be solved for G' by induction on s. The number of vertices in $[S]$ is bounded by a constant independent of $|V(G)|$ (since $|V(S)| < |V(H)|$ and every vertex of S has at most three neighbours in G), and hence the problem can be solved for G in polynomial time as well, which can be easily seen by induction on the number of vertices in $[S]$. $\square$

4 Conclusion

Unless $P = NP$, the maximum independent set problem can be solved in polynomial time for H-free subcubic graphs *only if* every connected component of H has the form $S_{i,j,k}$ represented in Figure 1. Whether this condition is sufficient for polynomial-time solvability of the problem is a challenging open question. In this paper, we contributed to this topic by solving the problem in the case when every connected component of H is isomorphic either to $S_{2,2,2}$ or to $S_{1,j,k}$. Our poof also shows that, in order to answer the above question, one can be restricted to H-free subcubic graphs where H is connected. In other words, one can consider $S_{i,j,k}$-free, or more generally, $S_{k,k,k}$-free subcubic graphs. We believe that the answer is positive for all values of k and hope that our solution for $k = 2$ can base a foundation for algorithms for larger values of k.

References

1. Alekseev, V.E.: On the local restrictions effect on the complexity of finding the graph independence number. In: Combinatorial-Algebraic Methods in Applied Mathematics, pp. 3–13. Gorkiy University Press (1983) (in Russian)
2. Bodlaender, H.L., Thilikos, D.M.: Treewidth for graphs with small chordality. Discrete Appl. Math. 79, 45–61 (1997)
3. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guie to the Theory of NP-Completeness, 5th edn. W.H. Freeman (1979) ISBN 0-7167-1045-5
4. Gerber, M.U., Hertz, A., Schindl, D.: P_5-free augmenting graphs and the maximum stable set problem. Discrete Applied Mathematics 132, 109–119 (2004)
5. Lozin, V., Milanič, M.: A polynomial algorithm to find an independent set of maximum weight in a fork-free graph. J. Discrete Algorithms 6, 595–604 (2008)
6. Lozin, V.V., Milanic, M., Purcell, C.: Graphs Without Large Apples and the Maximum Weight Independent Set Problem. Graphs and Combinatorics, doi:10.1007/s00373-012-1263-y
7. Lozin, V.V., Mosca, R.: Maximum regular induced subgraphs in $2P_3$-free graphs. Theoret. Comput. Sci. 460, 26–33 (2012)
8. Lozin, V.V., Mosca, R.: Maximum independent sets in subclasses of P_5-free graphs. Information Processing Letters 109, 319–324 (2009)
9. Maffray, F.: Stable sets in k-colorable P_5-free graphs. Information Processing Letters 109, 1235–1237 (2009)
10. Mosca, R.: Some results on maximum stable sets in certain P_5-free graphs. Discrete Applied Mathematics 132, 175–183 (2003)
11. Minty, G.J.: On maximal independent sets of vertices in claw-free graphs. J. Combin. Theory Ser. B 28, 284–304 (1980)

Construction Techniques for Digraphs with Minimum Diameter

Mirka Miller[1,2,3,4], Slamin[5], Joe Ryan[6], and Edy Tri Baskoro[4,6]

[1] School of Mathematical and Physical Sciences, University of Newcastle, Australia
[2] Department of Mathematics, University of West Bohemia, Pilsen, Czech Republic
[3] Department of Informatics, King's College London, United Kingdom
[4] Department of Mathematics, ITB Bandung, Indonesia
[5] School of Information Systems, University of Jember, Indonesia
[6] School of Electrical Engineering and Computer Science, University of Newcastle, Australia

{mirka.miller,joe.ryan}@newcastle.edu.au, slamin@unej.ac.id, ebaskoro@gmail.com

Abstract. We consider the so-called *order/degree problem*, that is, to determine the smallest diameter of a digraph given order and maximum out-degree. There is no general efficient algorithm known for the construction of such optimal digraphs but various construction techniques for digraphs with minimum diameter have been proposed. In this paper, we survey the known techniques.

1 Introduction

In communication network design, there are several factors which should be considered. For example, each processing element should be directly connected to a limited number of other processing elements in such a way that there always exists a connection route from one processing element to another. Furthermore, in order to minimise the communication delay between processing elements, the directed connection route must be as short as possible. Informally, a communication network can be modelled as a *digraph*, where each processing element is represented by a *vertex* and the directed connection between two processing elements is represented by an *arc*. The number of vertices (processing elements) is called the *order* of the digraph and the number of arcs (directed connections) incident from a vertex is called the *out-degree* of that vertex. The *diameter* is defined to be the largest of the shortest paths (directed connection routes) between any two vertices of the digraph.

In graph-theoretical terms, the problems in communication network design can be modelled as optimal digraph problems. The example described above corresponds to the so-called *order/degree problem*: Construct digraphs with the smallest possible diameter $K(n, d)$ for a given order n and maximum out-degree d.

Let $\mathcal{G}(n, d, k)$ denotes the set of all digraphs G, not necessarily diregular, of order n, maximum out-degree d, and diameter k. Then we have the following well known optimisation problem for digraphs,

© Springer-Verlag Berlin Heidelberg 2013

The *degree/diameter problem*: determine the largest order $N(d,k)$ of a digraph given maximum out-degree d and diameter at most k.

A related but less well known problem is

The *order/degree problem*: determine the smallest diameter $K(n,d)$ of a digraph given order n and maximum out-degree d.

For the degree/diameter problem, there is a natural upper bound on the number of vertices of the digraph given maximum out-degree d and diameter at most k, namely,

$$N(d,k) = \sum_{i=0}^{k} n_i \leq 1 + d + d^2 + \ldots + d^k = \frac{d^{k+1}-1}{d-1} \tag{1}$$

The right-hand side of (1) is called the *Moore bound* and is denoted by $M(d,k)$. A digraph of (necessarily constant) out-degree d, diameter k and order equal to $M(d,k)$ is called a *Moore digraph.* It is well known that Moore digraphs exist only in the trivial cases when $d = 1$ (directed cycles of length $k+1$, C_{k+1}, for any $k \geq 1$) or $k = 1$ (complete digraphs of order $d+1$, K_{d+1}, for any $d \geq 1$) [10,2].

With regards to the order/degree problem, we can derive a lower bound for the diameter by performing a log operation and taking into account that for $d > 1$ and $k > 1$ the Moore bound is not attainable. Then we obtain

$$K(n,d) \geq \lceil \log_d(n(d-1)+d) \rceil - 1 \tag{2}$$

where $1 < d \leq n-1$ and $\lceil x \rceil$ is the smallest integer larger than x.

The known results on this problem are the generalised deBruijn digraphs of order n, $N(d,k-1) < n \leq d^k$ [6] and the generalised Kautz digraphs of order n, $N(d,k-1) < n \leq d^k$ and order $n = d^k + d^{k-b}$, for odd b, [7]. Using the line digraph technique, it is shown in [5] that it is possible to construct digraphs on $n = d^k + d^{k-b}$ vertices and diameter k with b odd or even. More general results are the digraphs of order n, $N(d,k-1) < n \leq d^k + d^{k-1}$. These digraphs can be obtained by several constructions such as, generalised digraphs on alphabets [4], partial line digraphs [3], and the construction based on deletion of some vertices whose out-neighbourhoods are identical [9]. In the case $d = 2$ and $k \geq 4$, the best known results are the digraphs of order n, $N(2,k-1) < n \leq 25 \times 2^{k-4}$ [3,9] (here the upper bound is obtained by the Alegre digraph and its line digraphs). In the next section we describe several construction techniques for large digraphs. In Section 3 we present the characteristics and classifications of the construction techniques.

2 Construction Techniques

In this section we describe the construction techniques for large digraphs with minimum diameter: the generalised deBruijn digraphs, generalised Kautz digraphs, line digraphs, digon reduction, vertex deletion scheme, and voltage assignment technique.

2.1 Generalised deBruijn Digraphs

Imase and Itoh [6] constructed digraphs for given arbitrary order n and out-degree d, $1 < d < n$, by the following procedure. Let the vertices of the digraphs be labeled by $0, 1, ..., n-1$. A vertex u is adjacent to v, if

$$v \equiv du + i \pmod{n},\ i = 0, 1, ..., d-1 \tag{3}$$

For example, Figure 1 shows the digraph of order $n = 8$, out-degree $d = 2$ and diameter $k = 3$ which is obtained from this construction.

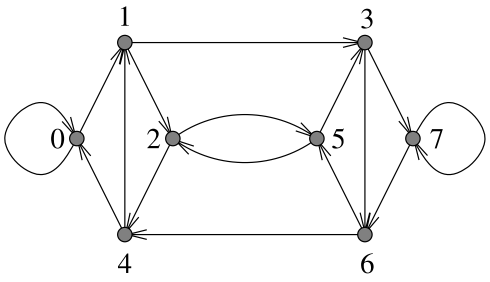


Fig. 1. The generalised deBruijn digraph $G \in \mathcal{G}(8, 2, 3)$

We note that when $n = d^k$, the digraphs obtained from this construction are isomorphic to the deBruijn digraphs of degree d and diameter k (see, for example, the digraph in Figure 1).

2.2 Generalised Kautz Digraphs

Imase and Itoh [7] presented a second construction of digraphs for given arbitrary order n and out-degree d, $1 < d < n$, by the following procedure. Let the vertices of digraphs be labeled by $0, 1, ..., n-1$. A vertex u is adjacent to v, if

$$v \equiv -du - i \pmod{n},\ i = 1, 2, ..., d \tag{4}$$

For example, Figure 2 shows the digraph of $n = 9$, out-degree $d = 2$ and diameter $k = 3$ that is obtained by this construction.

Note that when $n = d^k + d^{k-1}$, the digraphs obtained from this construction are isomorphic to the Kautz digraphs of degree d and diameter k.

Fiol, Llado and Villar [4] constructed generalisations of Kautz digraphs using their representation as digraphs on alphabets, that is, digraphs whose vertices are represented by words from a given alphabet and whose arcs are defined by an adjacency rule that relates pairs of words.

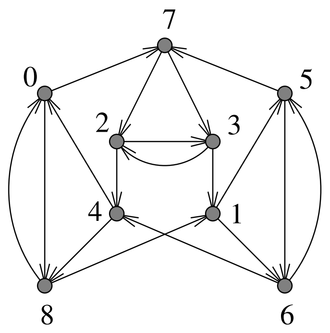


Fig. 2. The generalised Kautz digraph $G \in \mathcal{G}(9, 2, 3)$

2.3 Line Digraphs

Let $G = (V, A)$ and let N be the multiset of all walks of length 2 in G. The *line digraph* of a digraph G, $L(G) = (A, N)$, that is, the set of vertices of $L(G)$ is equal to the set of arcs of G and the set of arcs of $L(G)$ is equal to the set of walks of length 2 in G. This means that a vertex uv of $L(G)$ is adjacent to a vertex wx if and only if $v = w$. As an illustration, Figure 3 shows an example of a digraph and its line digraph.

The order of the line digraph $L(G)$ is equal to the number of arcs in the digraph G. For a diregular digraph G of out-degree $d \geq 2$, the sequence of line digraph iterations

$$L(G), L^2(G) = L(L(G)), ..., L^i(G) = L(L^{i-1}(G)), ...$$

is an infinite sequence of diregular digraphs of degree d.

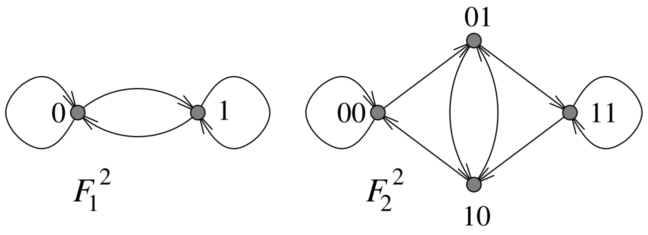


Fig. 3. The digraph F_1^2 and its line digraph F_2^2

Fiol, Yebra, and Alegre [5] constructed the digraphs with minimum diameter using line digraph iterations applied to a modification of the so-called F_k^d *digraph.*

Another interesting construction for digraphs with minimum diameter was presented by Fiol and Llado [3], based on their concept of partial line digraph.

2.4 Digon Reduction

A *digon* $\{u, v\}$ is a pair of arcs (u, v) and (v, u) in a digraph G. Alternatively, we can say that a digon is $\boldsymbol{K}_2$, a directed clique on two vertices in G. Miller and Fris [8] gave a construction technique for digraphs of degree 2 with minimum diameter using digon reduction scheme which they applied together with line digraph iterations. The construction procedure is as follows.

Let $G \in \mathcal{G}(n, d, k)$ be a digraph of order n, out-degree $d = 2$ and diameter k which contains p digons. Then $G' \in \mathcal{G}(n-1, d, k')$, for $k' \leq k$, can be obtained from G by 'gluing' two vertices which share a digon. This procedure can be repeated as many time as there are digons in the base digraph. For example, Figure 4(b) shows the digraph G' which is obtained from digraph G (Figure 4(a)) by gluing two vertices y and z.

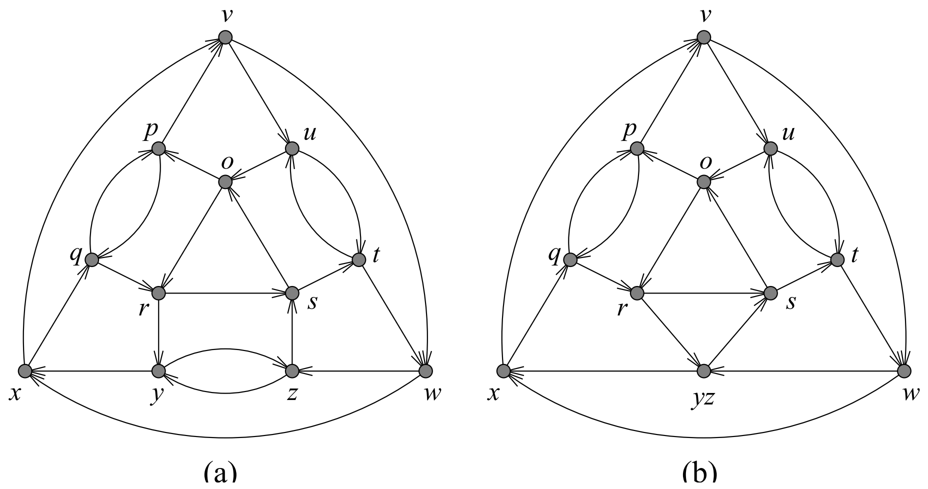


Fig. 4. The digraphs $G \in \mathcal{G}(12, 2, 3)$ and $G' \in \mathcal{G}(11, 2, 3)$

A generalisation of the digon reduction is $\boldsymbol{K}_d$-reduction, that is, 'gluing together' all the vertices of a directed clique $\boldsymbol{K}_d$.

Another generalisation can be obtained by gluing together a subdigraph H of G to form a single vertex h, if the multiset of the out-neighbours of $V(H)$ within $V(G) \backslash V(H)$ contains at most d vertices. If the original digraph G had diameter k then the digraph $G - H + \{h\}$ has diameter at most k.

2.5 Vertex Deletion Scheme

Vertex deletion scheme was introduced in [9] as described in the following procedure.

Let $G \in \mathcal{G}(n, d, k)$ and $N^+(u) = N^+(v)$ for two vertices $u, v \in G$, Then $G_1 \in \mathcal{G}(n-1, d, k')$, $k' \leq k$ is derived from G by deleting vertex u together with its outgoing arcs and reconnecting the incoming arcs of u to the vertex v. Figure 5(a) shows an example of digraph $G \in \mathcal{G}(12, 2, 3)$ with the property that some vertices have identical out-neighbourhoods. For example, $N^+(7) = N^+(12)$. Deleting vertex 12 together with its outgoing arcs and then reconnecting its incoming arcs to vertex 7 (since $N^+(7) = N^+(12)$), we obtain a new digraph $G_1 \in \mathcal{G}(11, 2, 2)$ as shown in Figure 5(b).

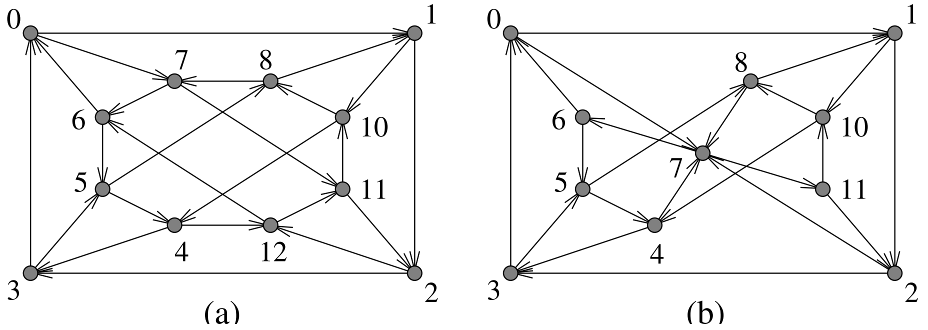


Fig. 5. The digraph $G \in \mathcal{G}(12, 2, 3)$ and $G_1 \in \mathcal{G}(11, 2, 3)$ obtained from G

Note that line digraphs are one class of digraphs that is suitable as base digraphs for a vertex deletion scheme since line digraphs contain pairs of vertices with the same out-neighbourhoods. However, line digraphs are not the only possible base digraphs for this scheme. Notably, the Alegre digraph which is not a line digraph but contains 10 pairs of vertices with the same out-neighbourhoods can be used for vertex deletion.

2.6 Voltage Assignments

One of the most successful construction techniques is voltage assignment [1]. Interestingly, it was first introduced for digraphs but the subsequent modification for undirected graphs has proved to be the single most successful technique currently in use. So far this has not been the case for directed graphs. However, we include the technique here for completeness and also because there is a good chance that it could be used to produce new optimal digraphs in the future. The technique can be described as follows.

Let G be a digraph and $A(G)$ be the set of arcs of G. Let Γ be an arbitrary group. Any mapping $\alpha : A(G) \to \Gamma$ is called a *voltage assignment*. The *lift* of G by α, denoted by G^α, is the digraph defined as follows: the vertex-set $V(G^\alpha) = V(G) \times \Gamma$, the arc-set $A(G^\alpha) = A(G) \times \Gamma$, and there is an arc (a, f) in G^α from (u, g) to (v, h) if and only if $f = g$, a is an arc from u to v, and $h = g\alpha(a)$.

For example, Figure 6 shows the digraph G and its lift G^α with $\Gamma = Z_3$ and the voltage assignment $\alpha(a) = \alpha(e) = 2$, $\alpha(b) = \alpha(d) = 1$, $\alpha(c) = \alpha(f) = \alpha(g) = \alpha(h) = 0$.

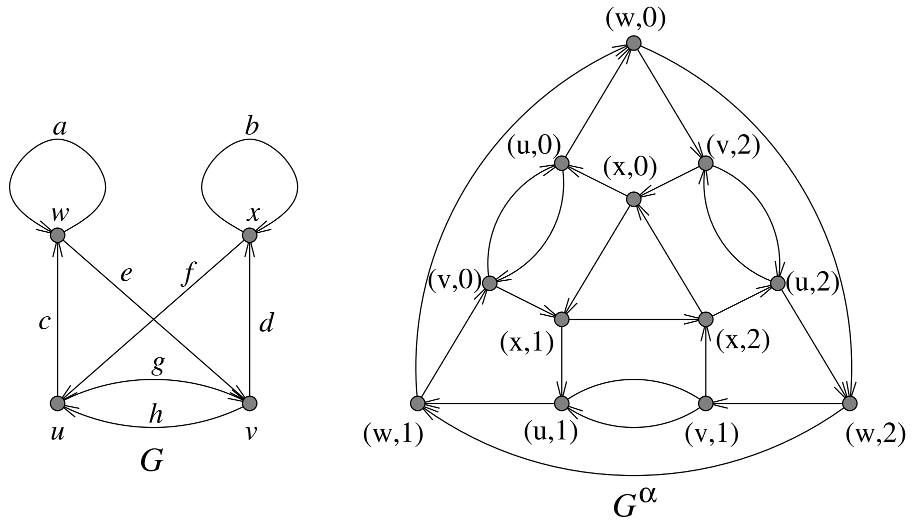


Fig. 6. The digraph G and its lift G^α

3 Characterisation and Classification

In this section, we discuss the characteristics of the various construction techniques.

Generalised deBruijn digraphs: Given order n and maximum out-degree d, we can construct a digraph directly using the formula given in (3). The resulting digraphs are always diregular of degree d. The minimum diameter is achieved when the order $N(d, k-1) < n \leq d^k$.

Generalised Kautz digraphs: Given order n and maximum out-degree d, we can construct a digraph directly using the formula given in (4). The generated digraphs are always diregular of degree d. The minimum diameter is achieved when the order $N(d, k-1) < n \leq d^k$ or $n = d^k + d^{k-b}$, for odd b.

Line digraphs: Given order n and maximum out-degree d, we can construct a digraph based on another digraph with smaller order and the same maximum out-degree. The generated digraphs are always diregular of degree d. The minimum diameter is achieved when the order $N(d, k-1) < n \leq d^k$ or $n = d^k + d^{k-b}$, for any $1 \leq b \leq k-1$.

Digon reduction: Given order n and maximum out-degree $d = 2$, we can construct a digraph based on another digraph with larger order that contains digons or based on another digraph with smaller order by applying line digraph iteration. The generated digraphs are always diregular of degree d. The minimum diameter is achieved when the order $N(d, k-1) < n \leq d^k + d^{k-1}$, for $d = 2$.

Generalised digraphs on alphabets: Given order n and maximum out-degree d, we can construct a digraph directly using given formulas. The generated digraphs are very likely to be non-diregular. The minimum diameter is achieved when the order $N(d, k-1) < n \leq d^k + d^{k-1}$.

Partial line digraphs: Given order n and maximum out-degree d, we can construct a digraph based on another digraph with smaller order and the same maximum out-degree. The generated digraphs are very likely to be non-diregular. The minimum diameter is achieved when the order $N(d, k-1) < n \leq d^k + d^{k-1}$ and $N(d, k-1) < n \leq d^k + d^{k-1} + d^{k-4}$, for $d = 2$ and $k \geq 4$.

Vertex deletion scheme: Given order n and maximum out-degree d, we can construct a digraph based on another digraph with larger order and the same maximum out-degree that contains vertices with the identical out-neighbourhoods. The generated digraphs are very likely to be non-diregular. The minimum diameter is achieved when the order $N(d, k-1) < n \leq d^k + d^{k-1}$, for $d \geq 3$ and $N(d, k-1) < n \leq d^k + d^{k-1} + d^{k-4}$, for $d = 2$ and $k \geq 4$.

Voltage assignments: Given order n and maximum out-degree d, we can construct a digraph based on another digraph with smaller order and the same maximum out-degree. The generated digraphs are diregular.

Table 1. The range of order n when minimum diameter is achieved by different techniques

Technique	**Order**
Generalised de Bruijn digraphs	$N(d, k-1) < n \leq d^k$
Generalised Kautz digraphs	(a) $N(d, k-1) < n \leq d^k$
	(b) $n = d^k + d^{k-b}$ for odd $0 < b < k$
Line digraphs	(a) $N(d, k-1) < n \leq d^k$
	(b) $n = d^k + d^{k-b}$ for any $0 < b < k$
Digon reduction	$N(2, k-1) < n \leq 2^k + 2^{k-1}$
Generalised digraphs on alphabets	$N(d, k-1) < n \leq d^k + d^{k-1}$
Partial line digraphs	$N(d, k-1) < n \leq d^k + d^{k-1}$ for $d \geq 3$
	$N(2, k-1) < n \leq 25 \cdot 2^{k-4}$ for $k \geq 4$
Vertex deletion scheme	$N(d, k-1) < n \leq d^k + d^{k-1}$ for $d \geq 3$
	$N(2, k-1) < n \leq 25 \cdot 2^{k-4}$ for $k \geq 4$

In Table 1, we summarise the range of order n of the digraphs with out-degree $d \geq 2$ when the minimum diameter k, for $k \geq 2$, is achieved by different construction techniques.

We conclude with some open problems.

Problem 1. Find other construction techniques that generate diregular digraphs with minimum diameter for the same or better range of order n as given by these three constructions.

Problem 2. Find digraphs which are larger than Kautz digraphs for any out-degree $d > 2$ and diameter $k > 1$.

References

1. Baskoro, E.T., Branković, L., Miller, M., Plesník, J., Ryan, J., Siráň, J.: Large digraphs with small diameter: A voltage assignment approach. JCMCC 24, 161–176 (1997)
2. Bridges, W.G., Toueg, S.: On the impossibility of directed Moore graphs. J. Combinatorial Theory Series B29, 339–341 (1980)
3. Fiol, M.A., Llado, A.S.: The partial line digraph technique in the design of large interconnection networks. IEEE Trans. on Computers 41(7), 848–857 (1992)
4. Fiol, M.A., Llado, A.S., Villar, J.L.: Digraphs on alphabets and the (d, N) digraph problem. Ars Combinatoria 25C, 105–122 (1988)
5. Fiol, M.A., Yebra, J.L.A., Alegre, I.: Line digraph iterations and the (d, k) digraph problem. IEEE Transactions on Computers C-33, 400–403 (1984)
6. Imase, M., Itoh, M.: Design to minimize a diameter on building block network. IEEE Trans. on Computers C-30, 439–442 (1981)
7. Imase, M., Itoh, M.: A design for directed graphs with minimum diameter. IEEE Trans. on Computers C-32, 782–784 (1983)
8. Miller, M., Fris, I.: Minimum diameter of diregular digraphs of degree 2. Computer Journal 31, 71–75 (1988)
9. Miller, M.: On the monotonicity of minimum diameter with respect to order and maximum out-degree. In: Du, D.-Z., Eades, P., Sharma, A.K., Lin, X., Estivill-Castro, V. (eds.) COCOON 2000. LNCS, vol. 1858, pp. 193–201. Springer, Heidelberg (2000)
10. Plesník, J., Znám, Š.: Strongly geodetic directed graphs. In: Acta F. R. N. Univ. Comen. - Mathematica XXIX, pp. 29–34 (1974)

Appendix

In Table 2, we compare the values of diameter obtained by each construction technique for order $1 \leq n \leq 50$ and out-degree $d = 2$.

Technique:

GBD	Generalised deBruijn digraphs
GKD	Generalised Kautz digraphs
LD	Line digraphs
DR	Digon reduction
GDA	Generalised digraphs on alphabets
PLD	Partial line digraphs
VDS	Vertex deletion scheme
VA	Voltage assignments

Table 2. The diameter of digraphs obtained by various technique for $d = 2$ and $n \leq 50$

n	GBD	GKD	LD	DR	GDA	PLD	VDS	VA
1	0	0	0	0	0	0	0	N/A
2	**1**	**1**	**1**	**1**	**1**	**1**	**1**	N/A
3	**1**	**1**	**1**	**1**	**1**	**1**	**1**	N/A
4	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**
5	3	3	**2**	**2**	**2**	**2**	**2**	N/A
6	3	3	**2**	**2**	**2**	**2**	**2**	**2**
7	**3**	**3**	**3**	**3**	**3**	**3**	**3**	N/A
8	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**
9	4	**3**	**3**	**3**	**3**	**3**	**3**	**3**
10	4	4	**3**	**3**	**3**	**3**	**3**	**3**
11	4	4	4	**3**	**3**	**3**	**3**	N/A
12	4	**3**	**3**	**3**	**3**	**3**	**3**	**3**
13	**4**	**4**	**4**	**4**	**4**	**4**	**4**	N/A
14	**4**	**4**	**4**	**4**	**4**	**4**	**4**	**4**
15	**4**	**4**	**4**	**4**	**4**	**4**	**4**	**4**
16	**4**	**4**	**4**	**4**	**4**	**4**	**4**	**4**
17	5	5	5	5	**4**	**4**	**4**	N/A
18	5	**4**	**4**	**4**	**4**	**4**	**4**	**4**
19	5	5	5	**4**	**4**	**4**	**4**	N/A
20	5	5	**4**	**4**	**4**	**4**	**4**	**4**
21	5	5	5	**4**	**4**	**4**	**4**	**4**
22	5	5	5	**4**	**4**	**4**	**4**	**4**
23	5	5	5	**4**	**4**	**4**	**4**	N/A
24	5	**4**	**4**	**4**	**4**	**4**	**4**	**4**
25	5	5	5	5	5	**4**	**4**	**4**
26	5	5	5	5	5	5	5	5
27	5	5	5	5	5	5	5	5
28	5	5	5	5	5	5	5	5
29	**5**	**5**	**5**	**5**	**5**	**5**	**5**	N/A
30	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**
31	**5**	**5**	**5**	**5**	**5**	**5**	**5**	N/A
32	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**
33	6	6	6	6	**5**	**5**	**5**	**5**
34	6	6	**5**	6	**5**	**5**	**5**	**5**
35	6	6	6	6	**5**	**5**	**5**	**5**
36	6	**5**	**5**	**5**	**5**	**5**	**5**	**5**
37	6	6	6	6	**5**	**5**	**5**	N/A
38	6	6	6	**5**	**5**	**5**	**5**	**5**
39	6	6	6	**5**	**5**	**5**	**5**	**5**
40	6	6	**5**	**5**	**5**	**5**	**5**	**5**
41	6	6	6	**5**	**5**	**5**	**5**	N/A
42	6	6	6	**5**	**5**	**5**	**5**	**5**
43	6	6	6	**5**	**5**	**5**	**5**	N/A
44	6	6	6	**5**	**5**	**5**	**5**	**5**
45	6	6	6	**5**	**5**	**5**	**5**	**5**
46	6	6	6	**5**	**5**	**5**	**5**	**5**
47	6	6	6	**5**	**5**	**5**	**5**	N/A
48	6	6	6	**5**	**5**	**5**	**5**	**5**
49	6	6	6	6	6	**5**	**5**	**5**
50	6	6	6	6	6	**5**	**5**	**5**

Suffix Tree of Alignment: An Efficient Index for Similar Data

Joong Chae Na[1], Heejin Park[2], Maxime Crochemore[3], Jan Holub[4], Costas S. Iliopoulos[3], Laurent Mouchard[5], and Kunsoo Park[6,⋆]

[1] Sejong University, Korea
[2] Hanyang University, Korea
[3] King's College London, UK
[4] Czech Technical University in Prague, Czech Republic
[5] University of Rouen, France
[6] Seoul National University, Korea
`kpark@snu.ac.kr`

Abstract. We consider an index data structure for similar strings. The generalized suffix tree can be a solution for this. The generalized suffix tree of two strings A and B is a compacted trie representing all suffixes in A and B. It has $|A|+|B|$ leaves and can be constructed in $O(|A|+|B|)$ time. However, if the two strings are similar, the generalized suffix tree is not efficient because it does not exploit the similarity which is usually represented as an alignment of A and B.

In this paper we propose a space/time-efficient *suffix tree of alignment* which wisely exploits the similarity in an alignment. Our suffix tree for an alignment of A and B has $|A|+l_d+l_1$ leaves where l_d is the sum of the lengths of *all* parts of B different from A and l_1 is the sum of the lengths of *some* common parts of A and B. We did not compromise the pattern search to reduce the space. Our suffix tree can be searched for a pattern P in $O(|P| + occ)$ time where occ is the number of occurrences of P in A and B. We also present an efficient algorithm to construct the suffix tree of alignment. When the suffix tree is constructed from scratch, the algorithm requires $O(|A| + l_d + l_1 + l_2)$ time where l_2 is the sum of the lengths of other common substrings of A and B. When the suffix tree of A is already given, it requires $O(l_d + l_1 + l_2)$ time.

Keywords: Indexes for similar data, suffix trees, alignments.

1 Introduction

The *suffix tree* of a string S is a compacted trie representing all suffixes of S [18,22]. Over the years, the suffix tree has not only been a fundamental data structure in the area of string algorithms but also it has been used for many applications in engineering and computational biology. The suffix tree can be

⋆ Corresponding author.

© Springer-Verlag Berlin Heidelberg 2013

constructed in $O(|S|)$ time for a constant alphabet [18,21] and an integer alphabet [8], where $|S|$ denotes the length of S. The suffix tree has $|S|$ leaves and requires $O(|S|)$ space.

We consider storing and indexing multiple data which are very similar. Nowadays, tons of new data are created every day. Some data are totally original and substantially different from existing data. Others are, however, created by modifying some existing data and thus they are similar to the existing data. For example, a new version of a source code is a modification of its previous version. Today's backup is almost the same as yesterday's backup. An individual Genome is more than 99% identical to the Human reference Genome (the 1000 genome project [1]). Thus, storing and indexing similar data in an efficient way is becoming more and more important.

Similar data are usually stored efficiently: When new data are created, they are aligned with the existing ones. Then, the resulting alignment shows the common and different parts of the new data. By only storing the different parts of the new data, the similar data can be stored efficiently.

When it comes to indexing, however, neither the suffix tree nor any variant of the suffix tree uses this similarity or alignment to index similar data efficiently. Consider the *generalized suffix tree* [2,10] for two similar strings $A =$ `aaatcaaa` and $B =$ `aaatgaaa`. Three common suffixes `aaa`, `aa`, `a` are stored twice in the generalized suffix tree. Moreover, two similar suffixes `aaatcaaa` and `aaatgaaa` are stored in distinct leaves even though they are very similar. Thus, the generalized suffix tree has $|A| + |B|$ leaves, most of which are redundant.

Recently, there have been some studies concerning efficient indexes for similar strings. Mäkinen et al. [16,17] first proposed an index for similar (repetitive) strings using run-length encoding, a suffix array, and BWT [5]. Huang et al. [11] proposed an index of size $O(n + N \log N)$ bits where n is the total length of common parts in one string, N is the total length of different parts in all strings. Their basic approach is building separately data structures for common parts and ones for different parts between strings. A self-index based on LZ77 compression [23] has been also developed due to Kreft and Navarro [13]. Another index based on Lemple-Ziv compression scheme is due to Do et al. [7]. They compressed sequences using a variant of the relative Lempel-Ziv (RLZ) compression scheme [14] and used a number of auxiliary data structures to support fast pattern search. Navarro [19] gave a short survey on some of these indexes.

Although these studies assume slightly different models on similar strings, most of them adopt classical compressed indexes to utilize the similarity among strings, that is, they focus on how to efficiently represent or encode common (repetitive) parts in strings. However, none of them support linear-time pattern search. Moreover, their pattern search time do not depend on only the pattern length but also the text length, and some indexes require (somewhat complicated) auxiliary data structures to improve pattern search time. In short, those data structures achieve smaller indexes by sacrificing pattern search time.

In this paper, we propose an efficient index for similar strings without sacrificing the pattern search time. It is a novel data structure for similar strings, named *suffix tree of alignment.* We assume that strings (texts) are aligned with each others, e.g., two strings A and B can be represented as $\alpha_1\beta_1 \dots \alpha_k\beta_k\alpha_{k+1}$ and $\alpha_1\delta_1 \dots \alpha_k\delta_k\alpha_{k+1}$, respectively, where α_i's are common chunks and β_i's and δ_i's are chunks different from the other string. (We note that the given alignment is not required to be optimal.) Then, our suffix tree for A and B has the following properties. (It should be noted that our index and algorithms can be generalized to three or more strings, although we only describe our contribution for two strings for simplicity.)

- **Space Reduction**: Our suffix tree has $|A| + l_d + l_1$ leaves where l_d is the sum of the lengths of *all* chunks of B different from A (i.e., $\Sigma_{i=0}^{k}|\delta_i|$) and l_1 is the sum of the lengths of *some* common chunks of A and B. More precisely, l_1 is $\Sigma_{i=0}^{k}|\alpha_i^*|$ where α_i^* is the longest suffix of α_i appearing at least twice in A or in B. The value of α_i^* is $O(\log\max(|A|, |B|))$ on average for random strings [12]. Furthermore, the values of l_d and l_1 are very small in practice. For instance, consider two human genome sequences from two different individuals. Since they are more than 99% identical, l_d is very small compared to $|B|$. We have computed α_i^* for human genome sequences and found out α_i^* is very close to $\log\max(|A|, |B|)$, even though human genome sequences are not random. Hence, our suffix tree is space-efficient for similar strings. Note that the space of our index can be further reduced in the form of compressed indexes such as the compressed suffix tree [9,20]. Our index is an important building block (rather than a final product) towards the goal of efficient indexing for highly similar data.
- **Pattern Search**: Our index is achieved without compromising the linear-time pattern search. That is, using our suffix tree, one can search a pattern P in $O(|P| + occ)$ time, where occ is the number of occurrences of P in A and B. In addition to the linear-time pattern search, we believe that our index supports the most of suffix tree functionalities, e.g., regular expression matchings, matching statistics, approximate matchings, substring range reporting, and so on [3,4,10], because our index is a kind of suffix trees.

We also present an efficient algorithm to construct the suffix tree of alignment. One naïve method to construct our suffix tree is constructing the generalized suffix tree and deleting unnecessary leaves. However, it is not time/space-efficient.

- When our suffix tree for the strings A and B is constructed from scratch, our construction algorithm requires $O(|A| + l_d + l_1 + l_2)$ time where l_2 is the sum of the lengths of other parts of common chunks of A and B. More precisely, l_2 is $\Sigma_{i=1}^{k+1}|\hat{\alpha}_i|$ where $\hat{\alpha}_i$ is the longest prefix of α_i such that $d_i\alpha_i$ appears at least twice in A and B (d_i is the character preceding α_i in B. Likewise with l_1, the value of l_2 is also very small compared to $|A|$ or $|B|$ in practice.
- Our algorithm is incremental, i.e., we construct the suffix tree of A and then transform it to the suffix tree of the alignment. Thus, when the suffix tree of A is already given, it requires $O(l_d + l_1 + l_2)$ time. $O(l_d + l_1 + l_2)$ is the minimum time required to make our index a kind of suffix tree so that linear-time pattern search is possible on both A and B. Furthermore, our

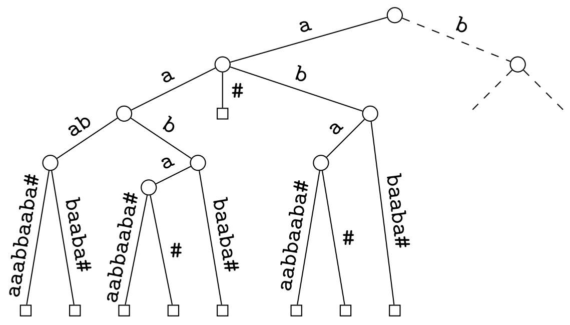


Fig. 1. The suffix tree of string `aaabaaabbaaba#`

algorithm can be applied to the case when some strings are newly inserted or deleted.

– Our algorithm uses constant-size extra working space except for our suffix tree itself. Thus, it is space-efficient compared to the naïve method.

The space/time-efficiency of our construction algorithm becomes large when handling many strings. The efficiency is feasible when the alignment has been computed in advance, which is the case in some applications. For instance, in the Next-Generation Sequencing, the reference genome sequence is given and the genome sequence of a new individual is obtained by aligning against the reference sequence. So, when a string (a new genome sequence) is obtained, the alignment is readily available. Moreover, since our index does not require that the given alignment is optimal, we can use a near-optimal alignment instead of the optimal alignment if the time to compute an alignment is an important issue. Since the given strings are assumed to be highly similar, a near-optimal alignment can be computed fast from exact string matching instead of dynamic programming requiring much time.

2 Preliminaries

2.1 Suffix Trees

Let S be a string over a fixed alphabet Σ. A substring of S beginning at the first position of S is called a *prefix* of S and a substring ending at the last position of S is called a *suffix* of S. We denote by $|S|$ the length of S. We assume that the last character of S is a special symbol $\# \in \Sigma$, which occurs nowhere else in S.

The *suffix tree* of a string S is a compacted trie with $|S|$ leaves, each of which represents each suffix of S. Figure 1 shows the suffix tree of a string `aaabaaabbaaba#`. For formal descriptions, the readers are referred to [6,10]. McCreight [18] proposed a linear-time construction algorithm using auxiliary links called *suffix links* and also an algorithm for an *incremental editing*, which transform the suffix tree of $S = \alpha\beta\gamma$ to that of $S' = \alpha\delta\gamma$ for some (possibly empty) string α, β, δ, γ.

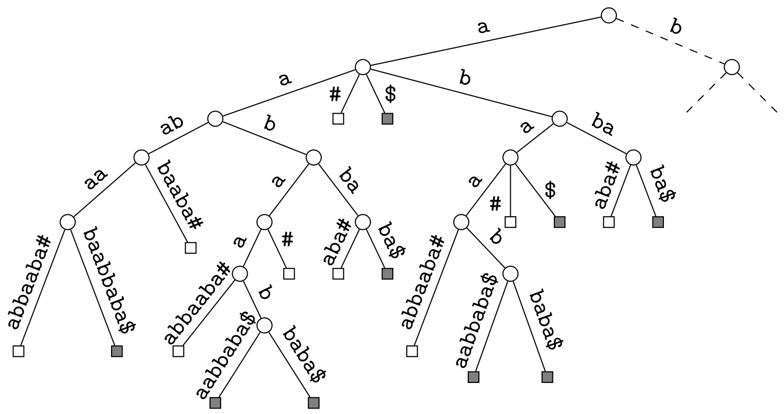


Fig. 2. The generalized suffix tree of two strings A = `aaabaaabbaaba#` and B = `aaabaabaabbaba$`. Leaves denoted by white squares and gray squares represent suffixes of A and B, respectively.

The *generalized suffix tree* of two strings A and B is a suffix tree representing all suffixes of the two strings. It can be obtained by constructing the suffix tree of the concatenated string AB where it is assumed that the last characters of A and B are distinct [2,10]. Thus, the generalized suffix tree has $|A| + |B|$ leaves and can be constructed in $O(|A| + |B|)$ time. Figure 2 shows the generalized suffix tree of two strings `aaabaaabbaaba#` and `aaabaabaabbaba$`.

2.2 Alignments

Given two strings A and B, an alignment of A and B is a mapping between the two strings that represents how A can be transform to B by replacing substrings of A into those of B. For example, let $A = \alpha\beta\gamma$ and $B = \alpha\delta\gamma$ for some strings α, β, γ, and δ ($\neq \beta$). Then, we can get B from A by replacing β with δ. We denote this replacement by alignment $\alpha(\beta/\,\delta)\gamma$.

More generally, an alignment of two strings $A = \alpha_1\beta_1 \ldots \alpha_k\beta_k\alpha_{k+1}$ and $B = \alpha_1\delta_1 \ldots \alpha_k\delta_k\alpha_{k+1}$, for some $k \geq 1$, can be denoted by $\alpha_1(\beta_1/\,\delta_1) \ldots \alpha_k(\beta_k/\,\delta_k)$ α_{k+1}. For simplicity, we assume that both A and B end with the special symbol $\# \in \Sigma$, which is contained in α_{k+1}. Without loss of generality, we assume the following conditions are satisfied for every $i = 1, \ldots, k$.

- α_{i+1} is not empty (α_1 can be empty).
- Either β_i or δ_i can be empty.
- The first characters of $\beta_i\alpha_{i+1}$ and $\delta_i\alpha_{i+1}$ are distinct.

Note that these conditions are satisfied for the optimal alignments by most of popular distance measures such as the edit distance [15]. Moreover, alignments unsatisfying the conditions can be easily converted to satisfy the conditions. If α_{i+1} ($i = 1, \ldots, k-1$) is empty, β_i and β_{i+1} (δ_i and δ_{i+1}) can be merged. (Note that α_{k+1} cannot be empty since $\#$ is contained in α_{k+1}.) If both β_i and δ_i are empty, α_i and α_{i+1} can be merged. Finally, if the first characters of $\beta_i\alpha_{i+1}$ and $\delta_i\alpha_{i+1}$ are identical (say c), we include c in α_i instead of $\beta_i\alpha_{i+1}$ and $\delta_i\alpha_{i+1}$.

3 Suffix Tree of Simple Alignments

In this section, we define the suffix tree of a simple alignment ($k = 1$) and present how to construct the suffix tree.

3.1 Definitions

For some strings α, β, γ, and δ, let $\alpha(\beta/\,\delta)\gamma$ be an alignment of two strings $A = \alpha\beta\gamma$ and $B = \alpha\delta\gamma$. We define suffixes of the alignment, called *alignment-suffixes* (for short *a-suffixes*). Let α^a and α^b be the longest suffixes of α which occur at least twice in A and B, respectively, and let α^* be the longer of α^a and α^b. That is, α^* is the longest suffix of α which occurs at least twice in A or in B. Then, there are 4 types of a-suffixes as follows.

1. a suffix of γ,
2. a suffix of $\alpha^*\beta\gamma$ longer than γ,
3. a suffix of $\alpha^*\delta\gamma$ longer than γ,
4. $\alpha'(\beta/\,\delta)\gamma$ where α' is a suffix of α longer than α^*. (Note that an a-suffix of this type represents two normal suffixes derived from A and B.)

For example, consider an alignment `aaabaa(abba/baabb)aba#`. Then, α^a and α^b are `baa` and `aabaa`, respectively, and α^* is `aabaa`. Since α^* is `aabaa`, `ba#`, $\texttt{abaa}\underline{\texttt{abba}}\texttt{aba\#}$, $\underline{\texttt{aabb}}\texttt{aba\#}$, and $\texttt{aaabaa(}\underline{\texttt{abba/baabb}}\texttt{)aba\#}$ are a-suffixes of type 1, 2, 3, and 4, respectively (underlined strings denote symbols in β and δ). The reason why we divide a-suffixes longer than $(\beta/\,\delta)\gamma$ into ones longer than $\alpha^*(\beta/\,\delta)\gamma$ (type 4) and the others (types 2 and 3), or why α^* becomes the division point, has to do with properties of suffix trees and we explain the reason later.

The *suffix tree of alignment* $\alpha(\beta/\,\delta)\gamma$ is a compacted trie representing all a-suffixes of the alignment. Formally, the suffix tree T for the alignment is a rooted tree satisfying the following conditions.

1. Each nonterminal arc is labeled with a nonempty substring of A or B.
2. Each terminal arc is labeled with a nonempty suffix of $\beta\gamma$ or $\delta\gamma$, or with $\alpha'(\beta/\,\delta)\gamma$, where α' is a nonempty suffix of α.
3. Each internal node v has at least two children and the labels of arcs from v to its children begin with distinct symbols.

Figure 3 shows the suffix tree of the alignment `aaabaa(abba/baabb)aba#`.

The differences from classic suffix trees of strings (including generalized suffix trees) are as follows. To reduce space, we represent common suffixes of A and B with one leaf. For example, there exists one leaf representing `aba#` in Figure 3 because `aba#` is common suffixes of A and B (type 1). However, suffixes of A and B longer than γ derived from $(\beta/\delta)\gamma$ are not common and thus we deal with these suffixes separately (types 2 and 3). For suffixes longer than $(\beta/\delta)\gamma$,

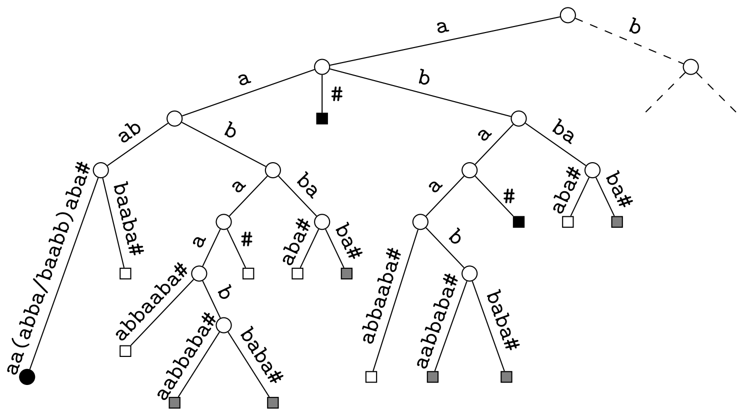


Fig. 3. The suffix tree of an alignment aaabaa(abba/baabb)aba#. Leaves denoted by black squares, white squares, gray squares, and black circles represent a-suffixes of types 1, 2, 3, and 4, respectively.

we have two cases. First, consider an a-suffix $\alpha'(\beta/\,\delta)\gamma$ (type 4) such that α' is a suffix of α longer than α^*, e.g., aaabaa(abba/baabb)aba#. Due to the definition of α^*, α' appears only once in each of A and B (at the same position) and we can represent $\alpha'(\beta/\,\delta)\gamma$ with one leaf by considering the terminal arc connected to the leaf is labeled with an alignment not a string, e.g., the leftmost (black circle) leaf in Figure 3. However, it cannot be applicable to an a-suffix $\alpha''(\beta/\,\delta)\gamma$ such that α'' is a suffix of α^*, e.g., abaa(abba/baabb)aba#. Since α'' appears at least twice in A or in B, $(\beta/\,\delta)$ may not be contained in the label of one arc. Thus, we represent the a-suffix by two leaves, one of which represents $\alpha''\beta\gamma$ (type 2) and the other $\alpha''\delta\gamma$ (type 3), e.g., leaf x representing abaaabbaaba# and leaf y representing abaabaabbaba# in Figure 3.

Pattern search can be solved using the suffix tree of alignment in the same way as using suffix trees of strings except for handling terminal arcs labeled with alignments. When we meet a terminal arc labeled with an alignment $\alpha'(\beta/\,\delta)\gamma$ during search, we first compare α' with the pattern and then decide which of β and δ we compare with the pattern by checking the first symbols of $\beta\gamma$ and $\delta\gamma$. This comparison is in fact similar to branching at nodes.

3.2 Construction

We describe how to construct the suffix tree T for an alignment. We assume the suffix tree T^A of string A is given. (T^A can be constructed in $O(|A|)$ time [18,21].) To transform T^A into the suffix tree T of the alignment, we should insert the suffixes of B into T^A. We divide the suffixes of B into three groups: suffixes of γ, suffixes of $\alpha^*\delta\gamma$ longer than γ, and suffixes of $\alpha\delta\gamma$ longer than $\alpha^*\delta\gamma$, which correspond to a-suffixes of types 1, 3, and 4, respectively. First, we do not have to do anything for a-suffixes of type 1. The suffixes of type 1 (suffixes of γ) already exist in T^A because these are common suffixes in A and B.

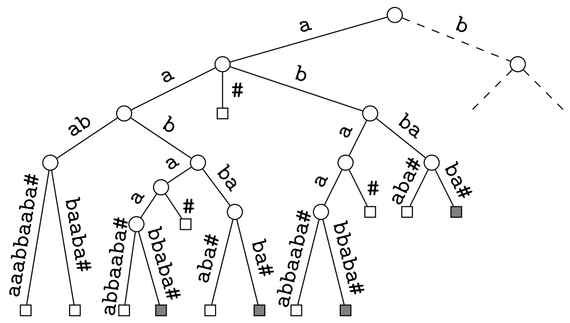


Fig. 4. The tree when Step A is applied to the suffix tree of A in Figure 1

Inserting the suffixes of B longer than γ consists of three steps. We *explicitly* insert the suffixes shorter than or equal to $\alpha^*\delta\gamma$ (type 3) in Steps A and B, and *implicitly* insert the suffixes longer than $\alpha^*\delta\gamma$ (type 4) in Step C as follows.

A. Find α^a and insert the suffixes of $\alpha^a\delta\gamma$ longer than γ.
B. Find α^* and insert the suffixes of $\alpha^*\delta\gamma$ longer than $\alpha^a\delta\gamma$.
C. Insert *implicitly* the suffixes of $\alpha\delta\gamma$ longer than $\alpha^*\delta\gamma$.

In Step A, we first find α^a in T^A using the doubling technique in incremental editing of [18] as follows. For a string χ, we call a leaf a χ-*leaf* if the suffix represented by the leaf contains χ as a prefix. Then, if and only if a string χ occurs at least twice in A, there are at least two χ-leaves in T^A. To find α^a, we check for some suffixes α' of α whether or not there are at least two α'-leaves in T^A. Let $\alpha_{(j)}$ be the suffix of α of length j. We first check whether or not there are at least two $\alpha_{(j)}$-leaves in increasing order of $j = 1, 2, 4, 8, \ldots, |\alpha|$. Suppose $\alpha_{(h)}$ is the shortest suffix among these $\alpha_{(j)}$'s such that there is only one $\alpha_{(j)}$-leaf. (Note that $h/2 \leq |\alpha^a| < h$.) Then, α^a can be found by checking whether or not there are at least two $\alpha_{(j)}$-leaves in decreasing order of $j = h-1, h-2, \ldots$, which can be done efficiently using suffix links [18].

After finding α^a, we insert the suffixes from the longest $\alpha^a\delta\gamma$ to the shortest $d\gamma$ where d is the last character of $\alpha\delta$: Inserting the longest suffix is done by traversing down the suffix tree from the root and inserting the other suffixes can be done efficiently using suffix links [18,21]. Figure 4 shows the tree when Step A is applied to the suffix tree of A in Figure 1.

Let T' be the tree when Step A is finished. In Step B, we first find α^* using T' and insert into T' the suffixes longer than $\alpha^a\delta\gamma$ from the longest $\alpha^*\delta\gamma$ to the shortest in the same way as we did in Step A. Unlike Step A, however, we have the following difficulties for finding α^* because T' is an incomplete suffix tree: i) suffixes of B longer than $\alpha^a\delta\gamma$ are not represented in T', ii) both suffixes of A and suffixes of B are represented in one tree T', and iii) some suffixes of B (a-suffixes of type 1) share leaves with suffixes of A but some suffixes (a-suffixes of type 3) of B do not.

But we show that T' has sufficient information to find α^*. (Recall α^* is the longest suffix of α occurring at least twice in A or in B.) Notice that our goal in

Step B is finding α^* but not α^b. If $|\alpha^b| \leq |\alpha^a|$, for no suffix α' of α longer than α^a, there are at least two α'-leaves in T', in which case $\alpha^* = \alpha^a$. Thus, we do not need to consider suffixes of α shorter than or equal to α^a.

Lemma 1. *For a suffix α' of α longer than α^a, if and only if α' occurs at least twice in B, there are at least two α'-leaves in T'.*

Proof. (If) We first show that there is an α'-leaf in T' due to the occurrence occ_1 of α' as a suffix of α. Since α is common in A and B, occ_1 appears in both A and B as prefixes of $\alpha'\beta\gamma$ and $\alpha'\delta\gamma$, respectively. Note that $\alpha'\delta\gamma$ is not represented in T' but $\alpha'\beta\gamma$ is. Hence, there is an α'-leaf f_1 in T' due to occ_1.

Next, we show that there is another α'-leaf in T' due to an occurrence occ_2 of α' other than occ_1 in B. Let p_1 and p_2 be the start positions of occ_1 and occ_2 in B, respectively, and let p_a be the start position of the suffix $\alpha^a\delta\gamma$ in B. We first prove by contradiction that occ_2 is contained in $\alpha^a\delta\gamma$. Suppose otherwise, that is, p_2 precedes p_a. We have two cases according to which of p_1 and p_2 precedes. First consider the case that p_2 precedes p_1, In this case, occ_2 is properly contained in α, which means that α' appears at least twice in α and also in A. This contradicts with the definition of α^a since α' is longer than α^a. Consider the case that p_1 precedes p_2. Let α'' be the suffix of α starting at p_2. Then, $|\alpha'| > |\alpha''| > |\alpha^a|$ and α'' is a prefix of occ_2. Furthermore, α'' is also a prefix of $\alpha'\delta\gamma$. It means that α'' occurs twice in α as a proper prefix of α' and a proper suffix of α'. This contradicts with the definition of α^a since α'' is longer than α^a. Therefore, p_2 does not precede p_a, which means occ_2 is contained in $\alpha^a\delta\gamma$.

Now we show that there is an α'-leaf f_2 in T' due to occ_2 and f_2 is distinct from f_1. Let η be the suffix of B starting at position p_2. Then, η is a proper suffix of $\alpha^a\delta\gamma$ since p_2 follows p_a. Because T' represents all suffixes of $\alpha^a\delta\gamma$, there exists an α'-leaf f_2 representing η in T'. Moreover, suffixes of A and B share leaves in T' only if they are suffixes of γ. Since the suffix $\alpha'\beta\gamma$ represented by f_1 is longer than γ, f_1 and f_2 are distinct.

(Only if) We prove by contradiction the converse, i.e., if α' occurs only once in B, there is only one α'-leaf in T'. Suppose there are two α'-leaves in T'. Since α' occurs only once in B, no suffix of B except for $\alpha'\delta\gamma$ contains α' as a prefix. Moreover, there is no leaf representing $\alpha'\delta\gamma$ in T' because $|\alpha'\delta\gamma| > |\alpha^a\delta\gamma|$ and no suffix of B longer than $\alpha^a\delta\gamma$ is represented in T'. Thus, no α'-leaf in T' represents a suffix of B and the two α'-leaves in T' represent two suffixes of A. It means α' occurs twice in A, which contradicts with the definition of α^a that α^a is the longest suffix of α occurring at least twice in A since $|\alpha'| > |\alpha^a|$. Therefore, there is only one α'-leaf in T' if α' occurs only once in B. □

Corollary 1. *For a suffix α' of α longer than α^a, if and only if α' occurs at least twice in A or in B, there are at least two α'-leaves in T'.*

By Corollary 1, we can find α^* by checking for some suffixes α' of α longer than α^a whether or not there are at least two α'-leaves in T'. It can be done in $O(|\alpha^*|)$ using the way similar to Step A. When Step B is applied to the tree in Figure 4, the resulting tree is the same as the tree in Figure 3 except that the terminal arc connected to the leftmost leaf (black circle) is labeled with suffix `aaabbaaba#` of string A but not with a-suffix `aa(abba/baabb)aba#` of the alignment.

In Step C, for every suffix α' of α longer than α^*, we *implicitly* insert the suffix $\alpha'\delta\gamma$ of B. Since the suffix $\alpha'\delta\gamma$ of B and the suffix $\alpha'\beta\gamma$ of A (a-suffixes of type 4) should be represented by one leaf, we do not insert a new leaf but convert the leaf representing $\alpha'\beta\gamma$ to represent the a-suffix $\alpha'(\beta/\,\delta)\gamma$. It can be done by replacing *implicitly* every β properly contained in labels of terminal arcs with $(\beta/\,\delta)$. Consequently, we explicitly do nothing in Step C, and these implicit changes are already reflected in the given alignment. For example, the suffix tree in Figure 3 is obtained by replacing implicitly the label `aa`$\underline{\texttt{abba}}$`aba#` of the terminal arc connected to the leftmost leaf (black circle) with a-suffix `aa(abba/baabb)aba#` of the alignment.

We consider the time complexity of our algorithm. In step A, finding α^a takes $O(|\alpha^a|)$ time and inserting suffixes takes $O(|\alpha^a\delta\hat{\gamma}|)$ time, where $\hat{\gamma}$ is the longest prefix of γ such that $d\hat{\gamma}$ occurs at least twice in A and B (where d is the character preceding γ). For detailed analysis, the readers are referred to [18]. In step B, similarly, finding α^* takes $O(|\alpha^*|)$ time and inserting suffixes takes $O(|\alpha^*\delta\hat{\gamma}|)$ time. Step C takes no time since it is implicitly done. Thus, we get the following theorem.

Theorem 1. *Given an alignment $\alpha(\beta/\,\delta)\gamma$ and the suffix tree of string $\alpha\beta\gamma$, the suffix tree of $\alpha(\beta/\,\delta)\gamma$ can be constructed in $O(|\alpha^*| + |\delta| + |\hat{\gamma}|)$ time.*

4 Suffix Tree of General Alignments

We extend the definitions and the construction algorithm into more general alignments. Let $\alpha_1(\beta_1/\,\delta_1)\ldots\alpha_k(\beta_k/\,\delta_k)\alpha_{k+1}$ be an alignment of two strings $A = \alpha_1\beta_1\ldots\alpha_k\beta_k\alpha_{k+1}$ and $B = \alpha_1\delta_1\ldots\alpha_k\delta_k\alpha_{k+1}$. For $1 \le i \le k+1$, let α_i^a and α_i^b be the longest suffixes of α_i occurring at least twice in A and B, respectively, and let α_i^* be the longer of α_i^a and α_i^b. That is, α_i^* is the longest suffix of α_i which occurs at least twice in A or in B. Moreover, let $\hat{\alpha}_i$ be the longest prefix of α_i such that $d_i\hat{\alpha}_i$ occurs at least twice in A and B where d_i is the character preceding α_i in B.

The suffix tree of the alignment is a compacted trie that represents the following a-suffixes of the alignment.

1. a suffix of α_{k+1},
2. a suffix of $\alpha_i^*\beta_i\alpha_{i+1}\ldots\alpha_{k+1}$ longer than $\alpha_{i+1}\ldots\alpha_{k+1}$,
3. a suffix of $\alpha_i^*\delta_i\alpha_{i+1}\ldots\alpha_{k+1}$ which is longer than $\alpha_{i+1}\ldots\alpha_{k+1}$,
4. $\alpha_i'(\beta_i/\,\delta_i)\ldots\alpha_{k+1}$, where α_i' is a suffix of α_i longer than α_i^*.

Given the suffix tree of A, the suffix tree of the alignment can be constructed as follows (the details are omitted).

A1. Find α_i^a using the suffix tree of A for each i $(1 \le i \le k)$.
A2. Insert the suffixes of $\alpha_i^a\delta_i\alpha_{i+1}\ldots\alpha_{k+1}$ longer than $\alpha_{i+1}\ldots\alpha_{k+1}$ for each i.
B1. Find α_i^* for each i.

B2. Insert the suffixes of $\alpha_i^* \delta_i \ldots \alpha_{k+1}$ longer than $\alpha_i^a \delta_i \ldots \alpha_{k+1}$ for each i.

C. Insert *implicitly* the suffixes of $\alpha_i \delta_i \ldots \alpha_{k+1}$ longer than $\alpha_i^* \delta_i \ldots \alpha_{k+1}$ for each i.

Theorem 2. *Given an alignment $\alpha_1(\beta_1/\,\delta_1)\ldots\alpha_k(\beta_k/\,\delta_k)\alpha_{k+1}$ and the suffix tree of string $\alpha_1\beta_1\ldots\alpha_k\beta_k\alpha_{k+1}$, the suffix tree of the alignment can be constructed in time at most linear to the sum of the lengths of α_i^*, δ_i, $\hat{\alpha}_{i+1}$ for $1 \le i \le k$.*

Our definitions and algorithms can be also extended into alignments of more than two strings. For example, consider an alignment $\alpha(\beta/\,\delta/\,\vartheta)\gamma$ of three strings $A = \alpha\beta\gamma$, $B = \alpha\delta\gamma$, and $C = \alpha\vartheta\gamma$ such that the first characters of $\beta\gamma$, $\delta\gamma$, and $\vartheta\gamma$ are distinct. We define α^a, α^b, and α^c as the longest suffix of α which occurs at least twice in A, B, and C, respectively, and α^* as the longest of α^a, α^b, and α^c. Then, there are 5 types of a-suffixes. (Suffixes of $\alpha^*\vartheta\gamma$ longer than γ are added as a new type of a-suffixes.) The suffix tree of the alignment can be defined similarly and constructed as follows: From the suffix tree of $\alpha\beta\gamma$, we construct the suffix tree of $\alpha(\beta/\,\delta)\gamma$ by inserting some suffixes of B, and then convert into the suffix tree of $(\beta/\,\delta/\,\vartheta)\gamma$ by inserting suffixes of C (and some suffixes of B occasionally). We omit the details.

Acknowledgements. Joong Chae Na was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education, Science and Technology(2012-0003214), and by the IT R&D program of MKE/KEIT [10038768, The Development of Supercomputing System for the Genome Analysis]. Heejin Park was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education, Science and Technology(2012-0006999), by Seoul Creative Human Development Program (HM120006), by the Proteogenomics Research Program through the National Research Foundation of Korea funded by the Korean Ministry of Education, Science and Technology, and by the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT & Future Planning(2012-054452). Jan Holub has been partially supported by the Czech Science Foundation (GAČR) as project No. 13-03253S. Kunsoo Park was supported by National Research Foundation of Korea-Grant funded by the Korean Government(MSIP) (2012K1A3A4A07030483).

References

1. The 1000 Genomes Project Consortium. A map of human genome variation from population-scale sequencing 467(7319), 1061–1073 (2010)
2. Amir, A., Farach, M., Galil, Z., Giancarlo, R., Park, K.: Dynamic dictionary matching. J. Comput. Syst. Sci. 49, 208–222 (1994)
3. Baeza-Yates, R.A., Gonnet, G.H.: Fast text searching for regular expressions or automaton searching on tries. J. ACM 43(6), 915–936 (1996)
4. Bille, P., Gørtz, I.L.: Substring range reporting. In: Giancarlo, R., Manzini, G. (eds.) CPM 2011. LNCS, vol. 6661, pp. 299–308. Springer, Heidelberg (2011)

5. Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, California (1994)
6. Crochemore, M., Rytter, W.: Jewels of Stringology. World Scientific Publishing, Singapore (2002)
7. Do, H.H., Jansson, J., Sadakane, K., Sung, W.-K.: Fast relative lempel-ziv self-index for similar sequences. In: Snoeyink, J., Lu, P., Su, K., Wang, L. (eds.) FAW-AAIM 2012. LNCS, vol. 7285, pp. 291–302. Springer, Heidelberg (2012)
8. Farach-Colton, M., Ferragina, P., Muthukrishnan, S.: On the sorting-complexity of suffix tree construction. J. ACM 47(6), 987–1011 (2000)
9. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. SIAM J. Comput. 35(2), 378–407 (2005)
10. Gusfield, D.: Algorithms on Strings, Tree, and Sequences. Cambridge University Press, Cambridge (1997)
11. Huang, S., Lam, T.W., Sung, W.K., Tam, S.L., Yiu, S.M.: Indexing similar dna sequences. In: Chen, B. (ed.) AAIM 2010. LNCS, vol. 6124, pp. 180–190. Springer, Heidelberg (2010)
12. Karlin, S., Ghandour, G., Ost, F., Tavare, S., Korn, L.J.: New approaches for computer analysis of nucleic acid sequences. Proc. Natl. Acad. Sci. 80(18), 5660–5664 (1983)
13. Kreft, S., Navarro, G.: On compressing and indexing repetitive sequences. Theor. Comput. Sci. (to appear)
14. Kuruppu, S., Puglisi, S.J., Zobel, J.: Relative lempel-ziv compression of genomes for large-scale storage and retrieval. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 201–206. Springer, Heidelberg (2010)
15. Levenshtein, V.: Binary codes capable of correcting deletions, insertions and reversals. Soviet Physics Doklady 10(8), 707–710 (1966)
16. Mäkinen, V., Navarro, G., Sirén, J., Välimäki, N.: Storage and retrieval of individual genomes. In: Batzoglou, S. (ed.) RECOMB 2009. LNCS, vol. 5541, pp. 121–137. Springer, Heidelberg (2009)
17. Mäkinen, V., Navarro, G., Sirén, J., Välimäki, N.: Storage and retrieval of highly repetitive sequence collections. J. Comput. Bio. 17(3), 281–308 (2010)
18. McCreight, E.M.: A space-economical suffix tree construction algorithm. J. ACM 23(2), 262–272 (1976)
19. Navarro, G.: Indexing highly repetitive collections. In: Arumugam, S., Smyth, B. (eds.) IWOCA 2012. LNCS, vol. 7643, pp. 274–279. Springer, Heidelberg (2012)
20. Sadakane, K.: Compressed suffix trees with full functionality. Theor. Comput. Sci. 41(4), 589–607 (2007)
21. Ukkonen, E.: On-line construction of suffix trees. Algorithmica 14(3), 249–260 (1995)
22. Weiner, P.: Linear pattern matching algorithms. In: Proc. of the 14th IEEE Symp. on Switching and Automata Theory, pp. 1–11 (1973)
23. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. on Information Theory 23(3), 337–343 (1977)

Fitting Voronoi Diagrams to Planar Tesselations*

Greg Aloupis[1], Hebert Pérez-Rosés[2,3], Guillermo Pineda-Villavicencio[4,5], Perouz Taslakian[6], and Dannier Trinchet-Almaguer[7]

[1] Chargé de Recherches FNRS, Université Libre de Bruxelles, Belgium
`aloupis.greg@gmail.com`
[2] Department of Mathematics, University of Lleida, Spain
[3] Conjoint Fellow, University of Newcastle, Australia
`hebert.perez@matematica.udl.cat`
[4] Center for Informatics and Applied Optimization, University of Ballarat, Australia
[5] Department of Mathematics, Ben-Gurion University of the Negev, Israel
`work@guillermo.com.au`
[6] School of Science and Engineering, American University of Armenia
`ptaslakian@aua.am`
[7] AlessTidyCraft Software Solutions, Havana, Cuba
`trinchet@gmail.com`

Abstract. Given a tesselation of the plane, defined by a planar straight-line graph G, we want to find a minimal set S of points in the plane, such that the Voronoi diagram associated with S 'fits' G. This is the Generalized Inverse Voronoi Problem (GIVP), defined in [12] and rediscovered recently in [3]. Here we give an algorithm that solves this problem with a number of points that is linear in the size of G, assuming that the smallest angle in G is constant.

Keywords: Voronoi diagram, Dirichlet tesselation, planar tesselation, inverse Voronoi problem.

1 Introduction

Any planar straight-line graph (PSLG) subdivides the plane into cells, some of which may be unbounded. The Voronoi diagram (also commonly referred to as *Dirichlet tesselation*, or *Thiessen polygon*) of a set S of n points is a PSLG with n cells, where each cell belongs to one point from S and consists of all points in the plane that are closer to that point than to any other in S.

Let G be a given PSLG, whose cells can be considered bounded and convex for all practical purposes. Indeed, if some cell is not convex, it can always be partitioned into convex subcells, thus yielding a finer tesselation. The asymptotic size complexity of the PSLG remains the same by this 'convexification' operation.

The *Inverse Voronoi Problem* (IVP) consists of deciding whether G coincides with the Voronoi diagram of some set S of points in the plane, and if so, finding S. This problem was first studied by Ash and Bolker [1]. Subsequently, Aurenhammer presented a more efficient algorithm [2], which in turn was improved by

* Mathematics Subject Classification: 52C45, 65D18, 68U05.

© Springer-Verlag Berlin Heidelberg 2013

Hartvigsen, with the aid of linear programming [5], and later by Schoenberg, Ferguson and Li [8]. Yeganova also used linear programming to determine the location of S [13,14].

In the IVP, the set S is limited to have one point per cell; a generalized version of this problem (GIVP) allows more than one point per cell. In this case, new vertices and edges may be added to G, but the original ones must be kept, as shown in Figure 1. With this relaxation the set S always exists, hence we are interested in minimizing its size.

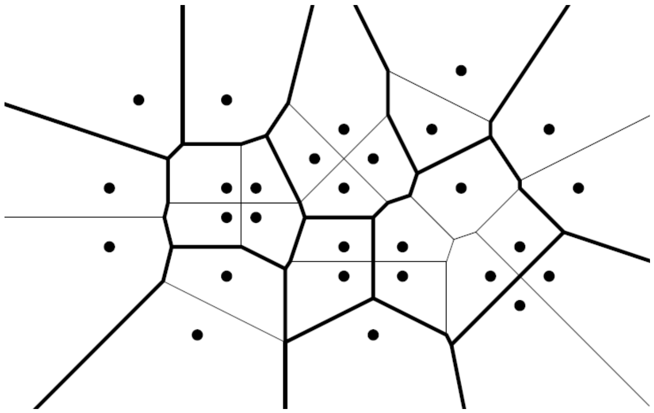


Fig. 1. GIVP: Thick edges represent the original input tesselation

The GIVP in $\mathbb{R}^2$ was indirectly mentioned in [13,14], in the context of set separation. It was formally stated and discussed in the III Cuban Workshop on Algorithms and Data Structures, held in Havana in 2003, where an algorithm for solving the problem in $\mathbb{R}^2$ was sketched by the current authors. However, the manuscript remained dormant for several years, and the algorithm was only published in Spanish in 2007 [12]. Recently, the problem was revisited in [3], where another algorithm for the GIVP in $\mathbb{R}^2$ is given, and the special case of a rectangular tesselation is discussed in greater detail. The authors of [3] were unaware of [12], however the two algorithms turn out to have certain common aspects.

This paper is an expanded and updated English version of [12]. It contains a description and analysis of the aforementioned algorithm for solving the GIVP in $\mathbb{R}^2$. This is followed by the description of an implementation of the algorithm, which was used to make a first (if only preliminary) experimental study of the algorithm's performance. Our algorithm generates $\mathcal{O}(E)$ sites in the worst case, where E is the number of edges of G (provided that the smallest angle of G is constant). This bound is asymptotically optimal for tesselations with such angular constraints.

In comparison, the analysis given for the algorithm in [3] states that $\mathcal{O}(V^3)$ sites are generated, where V is the size of a refinement of G such that all faces are triangles with acute angles. Given an arbitrary PSLG, there does not appear to be any known polynomial upper bound on the size of its associated acute triangulation. Even though it seems to us that the analysis in [3] should have given a tighter upper bound in terms of V, even a linear bound would not make much of a difference, given that V can be very large compared to the size of G. The analysis in [3] is purely theoretical, so it would be interesting to perform an experimental study to shed some light on the algorithm's performance in practice.

This paper is organized as follows: In Section 2 we describe the algorithm and discuss its correctness and performance. In Section 3 we derive some variants of the general strategy, and deal with several implementation issues of each variant. Section 4 is devoted to an experimental analysis of the algorithm's performance. Finally, in Section 5 we summarize our results and discuss some open problems arising as a result of our work.

2 The Algorithm

First we establish some notation and definitions. In that respect we have followed some standard texts, such as [4].

Let p and q be points of the plane; as customary, $\overline{pq}$ is the segment that joins p and q, and $|\overline{pq}|$ denotes its length. B_{pq} denotes the bisector of p and q, and H_{pq} is the half-plane determined by B_{pq}, containing p. For a set S of points in the plane, $\mathrm{Vor}(S)$ denotes the *Voronoi diagram* generated by S. The points in S are called *Voronoi sites* or *generators.*

If $p \in S$, $V(p)$ denotes the cell of $\mathrm{Vor}(S)$ corresponding to the site p. For any point q, $C_S(q)$ is the largest empty circle centered at q, with respect to S (the subscript S can be dropped if it is clear from the context). Two points $p, q \in S$ are said to be *(strong) neighbors* (with respect to S) if their cells share an edge in $\mathrm{Vor}(S)$; in this case E_{pq} denotes that edge.

We will make frequent use of the following basic property of Voronoi diagrams:

Lemma 1 ([4], Thm. 7.4, p. 150). *The bisector B_{pq} defines an edge of the Voronoi diagram if, and only if, there exists a point x on B_{pq} such that $C_S(x)$ contains both p and q on its boundary, but no other site. The (open) edge in question consists of all points x with that property.*

The technique used by the algorithm is to place pairs of points (*sentinels*) along each edge e of the PSLG (each pair is placed so that it is bisected by e) in order to 'guard' or 'protect' e. The number of sentinels required to protect e depends on its length and the relative positions of its neighboring edges. Each pair of sentinels meant to guard e is placed on the boundary of some circle, whose center lies on e. Furthermore this circle will not touch any other edge. The only exception is when the circle is centered on an endpoint of e, in which case it is allowed to touch all other edges sharing that endpoint. More formally, we have the following.

Definition 1. *Let G be a PSLG, and let e be an edge of G. Let S be a set of points, and $p, q \in S$. The pair of points p, q is said to be a* ***pair of sentinels*** *of*

e if they are strong neighbors with respect to S, and E_{pq} is a subsegment of e. In this case, e (or more precisely, the segment E_{pq}) is said to be ***guarded*** *by p and q.*

The algorithm works in two stages: First, for each vertex v of G we draw a circle centered on v. This is our set of *initial circles* (this is described in more detail below). Then we proceed to cover each edge e of G by non-overlapping *inner circles*, whose centers lie on e, and which do not intersect any other edges of G.

Let u be a given vertex of G, and let λ be the length of the shortest edge of G incident to u. We denote as $\xi_G(u)$ the initial circle centered at u, which will be taken as the largest circle with radius $\rho_0 \leq \lambda/2$ that does not intersect any edge of G, except those that are incident to u. Once we have drawn $\xi_G(u)$, for each edge e incident to u we can choose a pair of sentinels p, q, placed on $\xi_G(u)$, one on each side of e, at a suitably small distance ϵ from e, as in Figure 2. Later in this section we discuss how to choose ϵ appropriately.

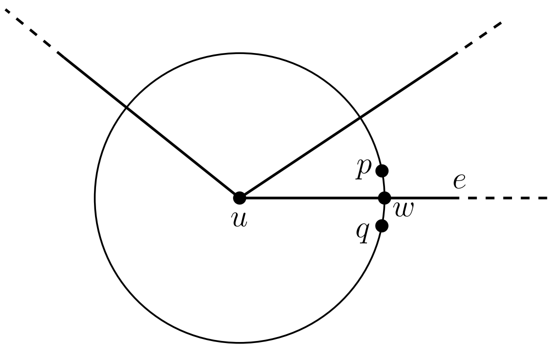


Fig. 2. Initial circle $\xi_G(u)$ for vertex u and sentinels of e

Let w be the point of intersection between $\xi_G(u)$ and e; now p and q guard the segment $\overline{uw}$ of e, which means that $\overline{uw}$ will appear in the Voronoi diagram that will be constructed, provided that we do not include any new points inside $\xi_G(u)$ (see Lemma 1).

Let $e = \overline{uv}$ be an edge of G, and w_1, w_2 the intersection points of $\xi(u)$ and $\xi(v)$ with e, respectively.[1] The segments $\overline{uw_1}$ and $\overline{uw_2}$ are now guarded, whereas the (possibly empty) segment $\overline{w_1w_2}$ still remains unguarded. In order to guard $\overline{w_1w_2}$ it suffices to cover that segment with circles centered on it, not intersecting with any edge other than e, and not including any sentinel belonging to another circle. Then we can choose pairs of sentinels on each covering circle, each sentinel being at distance ϵ from e, as shown in Figure 3.

As a consequence of Lemma 1, e will be guarded in all its length, provided that no new point is later included inside one of the circles centered on e. To ensure

[1] For convenience, we have dropped the subscript G.

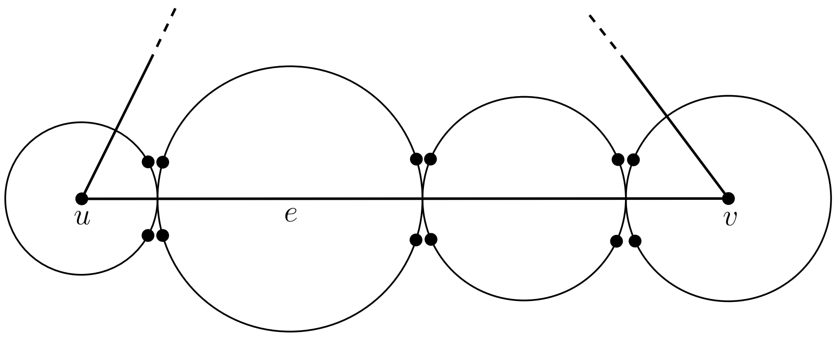


Fig. 3. Edge covered by circles

this, we will not allow an inner circle of e to get closer than ϵ to another edge f, because then a sentinel of f might fall inside the circle. With this precaution, the sentinels guarding e will not interfere with other edges, since they will not be included in any circle belonging to another edge.

In summary, an outline of the algorithm is:

1. For each vertex $u \in G$, draw initial circle $\xi_G(u)$ centered on u.
2. Choose a suitable value of ϵ.
3. For each vertex u and for each edge e incident to u, place a pair of sentinels on $\xi_G(u)$, symmetric to one another with respect to e, at distance ϵ from e.
4. For each edge $e \in G$, cover the unguarded segment of e with inner circles centered on e, and then place pairs of sentinels on each circle.

This algorithm is a general strategy that leads to several variants when Step 4 is specified in more detail, as will be seen in Section 3. In order to prove that the algorithm works it suffices to show that:

1. The algorithm terminates after constructing a finite number of circles (and sentinels).
2. After termination, every edge of G is guarded (see the discussion above).

In order to show that the algorithm terminates we will establish some facts. Let $\rho_0 > 0$ be the radius of the smallest initial circle. Now let α be the smallest angle formed by any two incident edges of G, say e and f. By taking $\epsilon \leq \rho_0 \sin \frac{\alpha}{2}$ we make sure that any sentinel will be closer to the edge that it is meant to guard than to any other edge. This is valid for all initial circles.

After all initial circles have been constructed, together with their corresponding sets of sentinels, for every edge e there may be a *middle segment* that remains unguarded. This segment must be covered by a finite number of inner circles. Take one edge, say e, with middle unguarded segment of length δ. If we use circles of radius ϵ to cover the unguarded segment, then we can be sure that these circles will not intersect any circle belonging to another edge. Exactly $\lfloor \delta/2\epsilon \rfloor + 1$

such circles will suffice to cover the middle segment, where the last one may have a radius ϵ' smaller than ϵ. For this last circle, the sentinels could be placed at distance $\epsilon' < \epsilon$ from e (c.f. Figure 4).

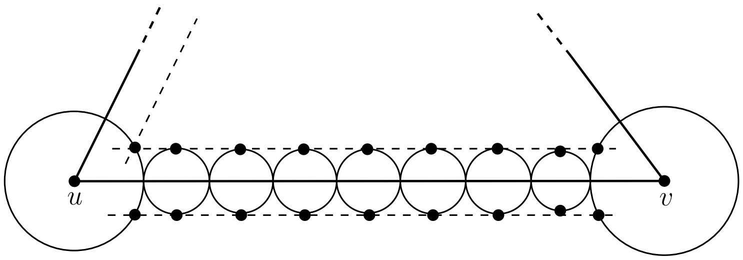


Fig. 4. Covering the middle segment of edge $\overline{uv}$ by inner circles of radius ϵ

Using circles of radius ϵ is, among all the possible variants mentioned here, the one that yields the largest number of circles, and hence the largest number of sentinels (generators of the Voronoi diagram). Now let e be the longest edge of G, with length Δ. In the worst case, the number of inner circles that cover e will be $\lfloor(\Delta - 2\rho_0)/2\epsilon\rfloor + 1$, and the number of sentinels will be twice that number plus four (corresponding to the sentinels of both initial circles). Therefore, the algorithm generates a number of points that is linear in E, the number of edges, which is asymptotically optimal, since a lower bound for the number of points is the number of faces in G.

Note that by letting G become part of the problem instance, the number of generators becomes a function of α, and it is no longer linear in E. In practice, however, screen resolution and computer arithmetic impose lower bounds on α. Under such constraints, the above analysis remains valid. This leads to our main result:

Theorem 1. *Let G be a planar straight-line graph, whose smallest angle α is larger than a fixed constant. Then, the corresponding Generalized Inverse Voronoi Problem can be solved with $\mathcal{O}(E)$ generators, where E is the number of edges of G.*

3 Implementation

In step 4 of the algorithm given in the previous section, the method to construct the inner circles was left unspecified. Taking the circles with radius ϵ, as suggested in the preceding analysis, is essentially a brute-force approach, and may easily result in too many sentinels being used. In this section we discuss two different methods for constructing the inner circles.

First let us note that in order to reduce the number of sentinels in our construction we may allow two adjacent circles on the same edge to overlap a little, so that they can share a pair of sentinels (see Figure 5). This observation is valid for all variants of the algorithm.

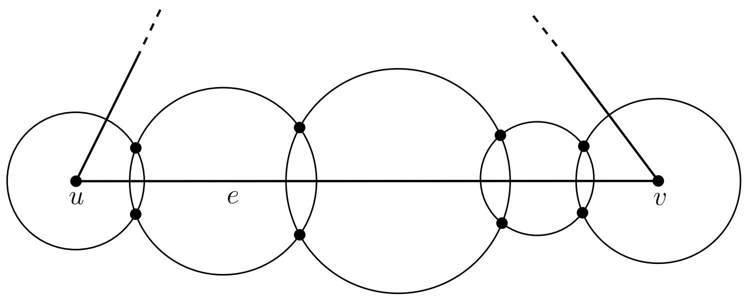


Fig. 5. Adjacent circles share a pair of sentinels

The first variant for the construction of the inner circles along an edge is to place them sequentially (iteratively), letting them grow as much as possible, provided that they do not enter the ϵ-wide 'security area' of another edge. Obviously, this greedy heuristic must yield a smaller number of Voronoi generators than the naive approach of taking all circles with radius ϵ.

Suppose we want to construct an inner circle χ for edge e, adjacent to another circle on e that has already been fixed and on which we have already placed two sentinels: $a = (x_a, y_a)$, and $b = (x_b, y_b)$. Let f be the first edge that will be touched by χ as it grows, while constrained to have its center on e and a, b on its boundary. Let f' be a straight line parallel to f, at distance ϵ from f, and closer to χ than f. Let e be defined by the equation $y = mx+n$, and f' by the equation $Ax + By + C = 0$.[2] The distance of any point (x, y) to f' is given by $\frac{|Ax+By+C|}{\sqrt{A^2+B^2}}$. The radius of χ must be equal to this distance. Hence the x-coordinate of the center satisfies the following quadratic equation:

$$(A^2 + B^2)((x_a - x)^2 + (y_a - (mx + n))^2) = (Ax + B(mx + n) + C)^2$$

or

$$\begin{gathered}-(A^2 + 2ABm + B^2m^2 - D(m^2 + 1))x^2 \\ -2(A(Bn + C) + B^2mn + BCm + D(x_a - m(n - y_a)))x \\ -B^2n^2 - 2BCn - C^2 + D(n^2 - 2ny_a + x_a^2 + y_a^2) = 0\end{gathered}$$

where $D = A^2 + B^2$.

Our second variant for constructing inner circles is also based on the principle of letting them grow until they come within distance ϵ of some edge. Yet, instead of growing the circles sequentially along the edge that is to be covered, we center the first inner circle on the midpoint of the unguarded middle segment.

[2] The equation of f' can be obtained easily after the initial circles have been constructed and their sentinels placed.

This will yield at most two smaller disjoint unguarded segments, on which we recurse. In the worst case, a branch of the recursion will end when an unguarded segment can be covered by a circle of radius ϵ. The advantage of this approach is that the coordinates of the center can be determined with much less computation, thus avoiding potential roundoff errors. Additionally, this variant is more suitable for parallel implementation than the previous one. On the other hand, we need an extra data structure to handle the unguarded segments.

We end this discussion with a word about the choice of ϵ. On one hand, ϵ must be sufficiently small for the construction to be carried out. On the other hand, for the sake of robustness to numerical errors, it is convenient to take ϵ as large as possible. That is why we defer the actual choice of ϵ until the initial circles have been drawn. A different approach might be to use a variable-sized ϵ, which would lead to a more complicated, yet (hopefully) more robust algorithm.

A final remark: For the sake of simplicity we have assumed throughout the whole discussion that the cells of the input tesselation are convex, but our algorithm could be easily generalized to accept tesselations with non-convex cells.

4 Experimental Analysis

From the analysis in Section 2 we know that the number of sites generated by our algorithm is linear in the size of the input, provided that the smallest angle α is constant. However, we would like to get a more precise idea about the algorithm's performance, and the difference between the two strategies we have suggested for Step 4. For that purpose, we have implemented the algorithm and carried out a set of experiments.

Our experimental workbench consists of a Graphical User Interface, which can generate a tesselation on a random point set, store it in a DCEL data structure, and then apply one of the two variants of the algorithm for solving the GIVP, described in Section 2.

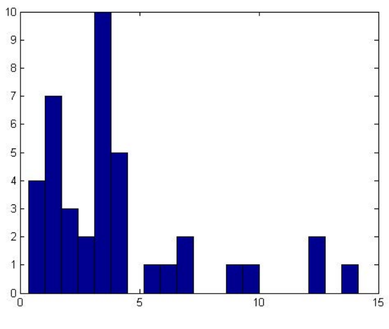


Fig. 6. Histogram of α

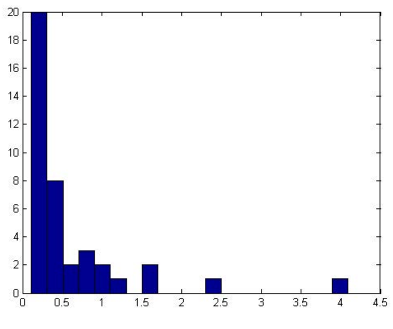


Fig. 7. Histogram of ϵ

Table 1. Experimental results

Exp. num.	**Tesselation**			**Num. of sites generated**		**Smallest angle (degrees)**	**ϵ-neigh. (pixels)**
	Vertices	**Edges**	**Regions**	**Recursive version**	**Sequential version**		
1	72	142	66	1 020	852	1.63	1.20
2	117	206	91	916	870	4.09	12.30
3	194	252	60	1 468	1 296	1.07	9.61
4	274	376	105	1 672	1 596	1.60	12.26
5	229	429	202	2 400	2 148	3.38	0.91
6	314	441	129	2 208	2 020	0.56	5.70
7	336	472	138	2 656	2 374	0.47	4.03
8	339	475	138	3 098	2 618	3.95	0.18
9	344	480	138	3 140	2 720	0.23	4.48
10	357	493	138	2 844	2 530	0.13	7.24
11	339	501	164	2 580	2 364	0.38	3.81
12	390	568	180	2 680	2 520	8.92	0.60
13	438	637	281	3 320	3 028	0.21	6.34
14	403	641	240	3 838	3 382	0.16	7.07
15	472	684	214	3 432	3 144	0.25	2.56
16	397	721	319	4 244	3 718	0.11	3.16
17	421	784	365	5 112	4 406	1.30	0.12
18	564	826	264	4 092	3 790	0.70	0.11
19	504	986	463	4 148	4 020	2.37	14.16
20	512	999	472	4 276	4 134	1.52	3.68
21	552	1 056	506	4 048	4 796	0.25	4.40
22	574	1 107	535	4 856	4 689	3.68	0.75
23	601	1 166	567	5 240	5 009	0.80	3.43
24	645	1 256	613	5 852	5 521	0.77	3.62
25	672	1 292	622	5 720	5 992	0.25	3.44
26	738	1 311	575	6 124	5 832	0.34	1.84
27	724	1 399	677	6 440	6 194	0.23	3.23
28	815	1 441	628	6 832	6 478	1.20	0.30
29	763	1 479	718	6 960	6 599	2.14	0.19
30	772	1 495	725	6 900	6 610	2.54	0.29
31	855	1 522	669	7 684	7 158	1.43	0.31
32	894	1 607	712	9 062	8 685	0.36	0.23
33	898	1 615	716	8 152	7 580	1.19	0.33
34	963	1 750	789	9 637	9 045	0.88	0.29
35	1 006	1 842	838	9 236	8 582	1.85	0.34
36	1 018	1 874	858	10 144	9 228	1.09	0.27
37	984	1 902	920	7 924	7 792	4.40	0.25
38	1 015	1 962	949	8 396	8 198	3.34	0.31
39	1 066	1 973	909	10 392	9 492	0.63	0.30
40	1 019	1 999	982	8 952	8 616	1.49	0.45
MED	**558**	**1 027.5**	**489**	**4 566**	**4 547.5**	**3.4**	**0.3**
AVG	**590**	**1 054**	**467**	**5 192**	**4 891**	**0.59**	**4.06**
STD	**282.7**	**571.24**	**292.73**	**2 715.5**	**2 600**	**3.36**	**0.74**

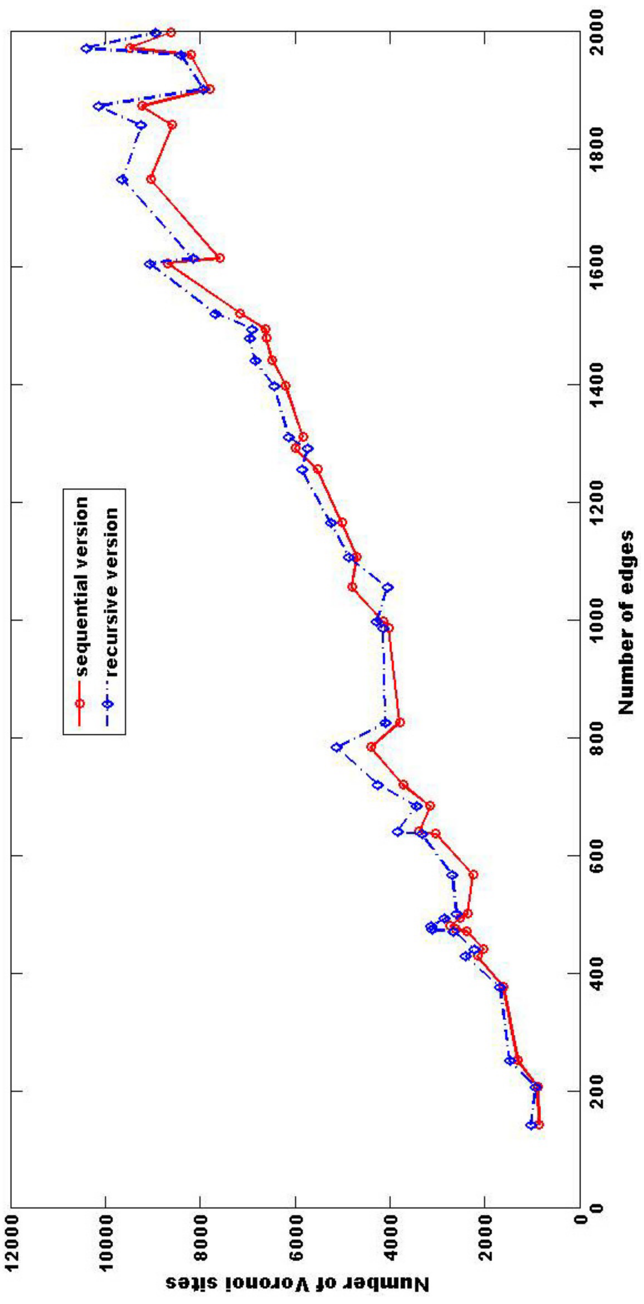


Fig. 8. Plot of the results in Table 1

The GUI is described in more detail in [11,12], and a beta Windows version can be downloaded from `https://www.researchgate.net/publication/239994361_Voronoi_data`. The file 'Voronoi data.rar' contains the Windows executable and a few DCEL files, consisting of sample tesselations. The user can generate additional tesselations randomly, and apply either variant of the algorithm on them.

The tesselations are generated as follows: First, the vertex set of G is randomly generated from the uniform distribution in a rectangular region. Then, pairs of vertices are chosen randomly to create edges. If a new edge intersects existing edges, then the intersection points are added as new vertices, and the intersecting edges are decomposed into their non-intersecting segments. Finally, some edges are added to connect disjoint connected components and dangling vertices, so as to make the PSLG biconnected.

Table 1 displays some statistics about 40 such randomly generated tesselations: Number of vertices, number of edges, number of regions, number of Voronoi sites with the recursive version of Step 4, number of Voronoi sites with the sequential version of Step 4, the smallest angle α, and the width ϵ of the security area. The tesselations have been listed in increasing order of the number of edges. For each parameter, the table also provides the median (MED), the mean value (AVG), and the standard deviation (STD).

From the tabulated data we can also get empirical estimates about the correlation among different parameters, especially α and ϵ, and about the distribution of their values. The parameters α and ϵ show a weak negative correlation with the number of edges, of -0.548 and -0.358 respectively. In turn they are positively correlated with one another, with a correlation of 0.67. These empirical findings agree with intuition.

Figures 6 and 7 display the histograms of α and ϵ with 20 bins. They can be well approximated by Poisson distributions, with $\lambda = 4.06$ and $\lambda = 0.59$, respectively, and with 95% confidence intervals $[3.437; 4.686]$ and $[0.375; 0.8784]$.

The comparison between the two variants of the algorithm is shown in Figure 8. We can see that the sequential variant is slightly better than the recursive variant, as it generates a smaller number of sites in most cases. However, the difference between both variants is not significant. Indeed, the linear regression fits have very similar slopes: The linear fit for the sequential variant is $y = 4.4831x + 158.59$, whereas the linear fit in the recursive case is $y = 4.6241x + 318.51$.

5 Conclusions and Open Problems

Our results show that the Generalized Inverse Voronoi Problem can be solved with a number of generators that is linear in the size of the input tesselation, provided that we enforce a lower bound on the size of the smallest angle. On the other hand, the algorithm described in [3] produces $\mathcal{O}(V^3)$ generators, where V is the number of vertices of an acute triangulation of G. As the performance of the two algorithms is given as a function of different parameters, a theoretical comparison between them is not straightforward. An experimental study could be helpful, but that would require an implementation of the algorithm in [3]. In practice, our algorithm generates approximately $4.48E + 159$ Voronoi sites, where E is the number of edges of the input tesselation.

In any case, the number of generators produced by both algorithms may still be too large, and it may be possible to reduce it to a number closer to F, the number of faces of the tesselation, which is the trivial lower bound. This lower bound can only be achieved if the tesselation is a Voronoi tesselation. In the more general case, how close to F can we get?

In particular, our algorithm still has plenty of room for improvement. In Section 3 we have already mentioned several strategies that can decrease the number of Voronoi sites produced. The design of a parallel version, and a version that is robust against degenerate cases and numerical roundoff errors, are other issues to consider. Roundoff errors have long been an important concern in Computational Geometry in general, and in Voronoi diagram computation, in particular (see e.g. [10]).

Other practical questions have to do with the experimental analysis of our algorithms. We have devised a method to generate a PSLG on a random point set, but we have not analyzed how this compares to generating such graphs uniformly from the set of all PSLGs that can be defined on a given point set. Regarding certain properties of our generated graphs (expected number of vertices, edges, and faces, expected area of the faces, distribution of the smallest angle, etc.), we have not attempted a theoretical analysis, but we have estimated some of these parameters empirically. A more comprehensive set of experiments will reveal how these tesselations compare with those generated by other methods.

As a final remark, we point out that our algorithm could also be generalized to other metrics, continuous or discrete, including graph metrics. Potential applications include image representation and compression, as described in [6], and pattern recognition (e.g. given a partition of some sample space, we could select a set of 'representatives' for each class). In the case of graphs, Voronoi partitions can be used to find approximate shortest paths (see [9,7], for instance). In social networks, node clustering around a set of 'representative' nodes, or 'super-vertices', is a popular technique for network visualization and/or anonymization [15].

Acknowledgements. Janos Pach contributed some key ideas for the algorithm, at the early stages of this work. Hebert Pérez-Rosés was partially supported by the Spanish Ministry of Economy and Competitiveness, under project TIN2010-18978. Guillermo Pineda-Villavicencio was supported by a postdoctoral fellowship funded by the Skirball Foundation, via the Center for Advanced Studies in Mathematics at the Ben-Gurion University of the Negev, Israel, and by an ISF grant.

References

1. Ash, P., Bolker, E.D.: Recognizing Dirichlet Tesselations. Geometriae Dedicata 19, 175–206 (1985)
2. Aurenhammer, F.: Recognising Polytopical Cell Complexes and Constructing Projection Polyhedra. J. Symbolic Computation 3, 249–255 (1987)

3. Banerjee, S., Bhattacharya, B.B., Das, S., Karmakar, A., Maheshwari, A., Roy, S.: On the Construction of a Generalized Voronoi Inverse of a Rectangular Tesselation. In: Procs. 9th Int. IEEE Symp. on Voronoi Diagrams in Science and Engineering, pp. 132–137. IEEE, New Brunswick (2012)
4. de Berg, M., Cheong, O., van Kreveld, M., Overmars, M.: Computational Geometry. Algorithms and Applications, 3rd edn. Springer, Berlin (2008)
5. Hartvigsen, D.: Recognizing Voronoi Diagrams with Linear Programming. ORSA J. Comput. 4, 369–374 (1992)
6. Martínez, A., Martínez, J., Pérez-Rosés, H., Quirós, R.: Image Processing using Voronoi diagrams. In: Procs. 2007 Int. Conf. on Image Proc., Comp. Vision, and Pat. Rec., pp. 485–491. CSREA Press (2007)
7. Ratti, B., Sommer, C.: Approximating Shortest Paths in Spatial Social Networks. In: Procs. 2012 ASE/IEEE Int. Conf. on Social Computing and 2012 ASE/IEEE Int. Conf. on Privacy, Security, Risk and Trust, pp. 585–586. IEEE Comp. Soc. (2012)
8. Schoenberg, F.P., Ferguson, T., Li, C.: Inverting Dirichlet Tesselations. The Computer J. 46, 76–83 (2003)
9. Sommer, C.: Approximate Shortest Path and Distance Queries in Networks. PhD Thesis, Department of Computer Science, The University of Tokyo, Japan (2010)
10. Sugihara, K., Iri, M.: Construction of the Voronoi Diagram for 'One Million' Generators in Single-Precision Arithmetic. Procs. IEEE 80, 1471–1484 (1992)
11. Trinchet-Almaguer, D.: Algorithm for Solving the Generalized Inverse Voronoi Problem. Honour's Thesis, Department of Computer Science, University of Oriente, Cuba (2005) (in Spanish)
12. Trinchet-Almaguer, D., Pérez-Rosés, H.: Algorithm for Solving the Generalized Inverse Voronoi Problem (in Spanish). Revista Cubana de Ciencias Informaticas 1(4), 58–71 (2007)
13. Yeganova, L., Falk, J.E., Dandurova, Y.V.: Robust Separation of Multiple Sets. Nonlinear Analysis 47, 1845–1856 (2001)
14. Yeganova, L.E.: Robust linear separation of multiple finite sets. Ph.D. Thesis, George Washington University (2001)
15. Zhou, B., Pei, J., Luk, W.-S.: A brief survey on anonymization techniques for privacy preserving publishing of social network data. ACM SIGKDD Explorations Newsletter 10, 12–22 (2008)

Partial Information Network Queries

Ron Y. Pinter and Meirav Zehavi

Department of Computer Science, Technion - Israel Institute of Technology,
Haifa 32000, Israel
{pinter,meizeh}@cs.technion.ac.il

Abstract. We present a new pattern matching problem, the *partial information query (PIQ)* problem, which includes as special cases two problems that have important applications in bioinformatics: the *alignment query (AQ)* problem and the *topology-free query (TFQ)* problem. In both problems we have a pattern $\mathcal{P}$ and a graph H, and we seek a subgraph of H that *resembles* $\mathcal{P}$. AQ requires knowing the topology of $\mathcal{P}$, while TFQ ignores it. PIQ fits the scenario where partial information is available on the topology of $\mathcal{P}$. Our main result is a parameterized algorithm for PIQ, which can handle inputs where $\mathcal{P}$ is a set of trees. It significantly improves the best known running time in solving TFQ. We also improve the best known running times in solving two special cases of AQ.

Keywords: parameterized algorithm, pattern matching, partial information query, alignment query, topology-free query.

1 Introduction

Algorithms for the *alignment query (AQ)* and the *topology-free query (TFQ)* problems provide means to study the function and evolution of biological networks. Given a pattern $\mathcal{P}$ and a host graph H, these queries seek a subgraph of H that resembles $\mathcal{P}$. Similar queries for sequences have been studied and used extensively in the past four decades. Today, with the increasing amount of information we have on biological networks, they are relevant to them as well.

TFQ requires only the connectivity of the solution, while AQ requires resemblance between the *topology* of $\mathcal{P}$ and the solution. A user having partial information on the topology of $\mathcal{P}$ can either run an alignment query for each possible topology for $\mathcal{P}$, given this partial information, or run a topology-free query. The first method is inefficient, while the second may output undesirable results that contradict the partial information on $\mathcal{P}$. We present a generalization of AQ and TFQ, that we call the *partial information query (PIQ)* problem, which fits the scenario where only partial information is available on $\mathcal{P}$.

Parameterized algorithms are an approach to solve NP-hard problems by confining the combinatorial explosion to a parameter k. More precisely, a problem is *fixed-parameter tractable (FPT)* with respect to a parameter k if an instance of size n can be solved in $O^*(f(k))$ time for some function f [15].[1] In this paper we present parameterized algorithms for NP-hard special cases of PIQ.

[1] O^* hides factors polynomial in the input size.

© Springer-Verlag Berlin Heidelberg 2013

Notation: Given a graph G, $V(G)$ and $E(G)$ denote its node set and edge set, respectively. Given $U \subseteq V(G)$, $G[U]$ denotes the subgraph of G induced by U. Denote the label of a node v by $l(v)$, and its neighbor set by $N(v)$. Given a set of tuples A, $i \in \mathbb{N}$ and an element e, $A[(i,e)]$ denotes the set of tuples in A such that e appears in their i^{th} position.

1.1 Problem Statement

Roughly speaking, given a graph H and a set of graphs $\mathcal{P}$, PIQ asks if H has a connected subgraph that can be partitioned into subgraphs, such that each resembles a different graph in $\mathcal{P}$.

Formally, the input for PIQ consists of

- L - A set of labels.
- $\Delta : L \times L \rightarrow \mathbb{R}$ - A label-to-label similarity score table.
- $\mathcal{P}$ - A set of labeled graphs $P_1, P_2, \ldots, P_t$.
- $H, w : E(H) \rightarrow \mathbb{R}$ - An edge-weighted labeled graph.
- $W \in \mathbb{R}$ - Minimum score.

We need to decide if there is a connected subgraph S of H, a partition of $V(S)$ into the subsets $\{V_S^1, \ldots, V_S^t\}$, and an isomorphism m_i between $S[V_S^i]$ and P_i, for all $1 \le i \le t$, such that

- $\sum_{1\le i\le t} \sum_{v \in V_S^i} \Delta(l(v), l(m_i(v))) + \sum_{e \in E(S)} w(e) \ge W$.
- Any cycle in S is completely contained in $S[V_S^i]$, for some $1 \le i \le t$.[2]

Denote $V(\mathcal{P}) = \bigcup_{1\le i\le t} V(P_i)$, and $k = |V(\mathcal{P})|$.

Special Cases of PIQ: AQ is the special case where $t = 1$, and all the edge-weights are 0. Moreover, TFQ is the special case where $t = k$, and $\Delta(l, l') \in \{-\infty, 0\}$ for all $l, l' \in L$.

1.2 Related Work

AQ – Related Work: Even the special case of AQ where P_1 is a simple path is NP-hard since it generalizes the Hamiltonian path problem [9].

Pinter et al. [16] gave an $O^*(1)$ time algorithm for AQ, which can handle inputs where both P_1 and H are trees. This algorithm was used to perform inter-species and intra-species alignments of metabolic pathways [17], and a pathway evolution study [14]. It was recently extended to a certain family of DAGs [18].

Another approach, based on color coding [1], enables H to be a general graph, and provides parameterized algorithms with parameter k. This approach is used by QPath [19] to perform simple path queries in $O^*(5.437^k)$ time. QNet [6] extends QPath by allowing P_1 to be a graph whose treewidth tw is bounded. Its running time is $O^*(8.155^k|V(H)|^{tw+1})$. PADA1 [3] is an alternative to QNet

[2] We thus avoid solving a generalization of the W[1]-hard Clique problem [7].

Table 1. Parameterized algorithms for TFQ. The first two algorithms require W and the edge-weights of H to be nonnegative integers, and their running times depend on the numeric value of W (which is exponential in its length).

Reference	Running Time	Method
Guillemot et al. [10]	$O^*(4^kW^2)$	multilinear detection [12]
Pinter et al. [18]	$O^*(2^kW)$	narrow sieves [2]
Bruckner et al. [4]	$O^*(k!3^k)$	color coding [1]
This paper	$\mathbf{O^*(20.25^{k+O(\log^2 k)})}$	**randomized divide-and-conquer [5]**

that bounds the size fvs of the feedback vertex set of P_1 instead of its treewidth. Its running time is $O^*(8.155^k|V(H)|^{fvs})$. Hüffner et al. [11] reduce the running time of QPath to $O^*(4.314^k)$. All of these algorithms are randomized.

We note that there is a variety of problems related to AQ that have applications in bioinformatics, and refer the reader to the surveys [8,21] for information.

TFQ – Related Work: Unweighted TFQ (i.e., TFQ restricted to inputs where $w(e) = 0$, for all $e \in E(H)$) was introduced by Lacroix et al. [13], and TFQ was introduced by Bruckner et al. [4]. Lacroix et al. [13] proved that unweighted TFQ is NP-hard even if H is a tree.

On the positive side, TFQ parameterized by k is in FPT. Table 1 presents known parameterized algorithms for TFQ. All of them are randomized.

We note that there are several problems related to TFQ that have applications in bioinformatics, and refer the reader to the survey [20] for information.

1.3 Our Contribution

Our first algorithm, AQ-Alg, can handle inputs for PIQ where H is a general graph and $\mathcal{P}$ is a set of one tree. AQ-Alg runs in $O^*(6.75^k)$ time, which improves QNet and PADA1 for inputs where P_1 is a tree. If P_1 is a path, AQ-Alg runs in $O^*(4^k)$ time, which further improves QPath. Our algorithm uses the randomized divide-and-conquer method [5].

Our main result is the second algorithm, PIQ-Alg, which is a modification of AQ-Alg. PIQ-Alg can handle inputs for PIQ where H is a general graph and $\mathcal{P}$ is a set of trees. It runs in $O^*(6.75^{k+O(\log^2 k)}3^t)$ time. In particular, it solves TFQ in $O^*(20.25^{k+O(\log^2 k)})$ time (since then $t = k$), which significantly improves the running time of the previous best algorithm by Bruckner et al. [4].

Due to space constraints, proofs are omitted.

2 AQ-Alg: An Algorithm for AQ

We start by presenting an algorithm, that we call AQ-Alg, for the special case of PIQ where $\mathcal{P}$ is a set of one tree (i.e., $t = 1$ and P_1 is a tree).

2.1 Overview

We use the randomized divide-and-conquer method [5]. AQ-Alg randomly divides the problem into two smaller subproblems that it recursively solves, and then combines the answers. The following overview is illustrated in Fig. 1.

Each recursive stage concerns a rooted subtree R of P_1, $U \subseteq V(H)$ and a set *Solved* of rooted subtrees of R. The subgraph of R induced by the nodes in R that are not in any tree in *Solved* and the roots of the trees in *Solved* is a tree. We next denote this subgraph by R'.

Each tree in *Solved* has several pairs. Such a pair consists of a node $h \in V(H)$ and a score s, and it concerns an isomorphism of score s between the tree and a subtree of H that maps the root of the tree to h. These pairs and isomorphisms between R' and subtrees of H that map the nodes of R' (excluding its root) to nodes in U help finding scores of isomorphisms between R and subtrees of H.

In the base cases of the recursion, R' has at most two nodes. Otherwise we divide R' into two subtrees that have a common node, and randomly divide U into two subsets. We use the first subset to map the first subtree. Then we use the results and the second subset to map the second subtree.

2.2 Preliminaries

Each definition presented in this section is preceded by an explanation of its relevance, and is illustrated in Fig. 1.

Choose $p_3 \in V(P_1)$, and add $p_1, p_2, \{p_1, p_2\}$ and $\{p_2, p_3\}$, that are new nodes and edges, to P_1. Add a new node h^* to H and connect it to all the nodes in $V(H)$ by edges of weight 0. Thus we avoid a special treatment of the first call to the recursive procedure AQ-Rec, that is the main part of AQ-Alg.

Root P_1 at p_1, and use a preorder to denote its nodes by $p_1, p_2, \ldots, p_{|V(P_1)|}$. For nodes p and $n_p \in N(p)$, $T(p, n_p)$ is the subtree induced by p, its children whose indexes are greater than that of n_p, and the descendants of these children.

Each stage of AQ-Rec concerns a subtree of P_1 of the form $T(r, n_r)$, for nodes r and $n_r \in N(r)$, a set $U \subseteq V(H)$, and a set *Solved* of disjoint subtrees of $T(r, n_r)$. The trees in *Solved* are of the form $T(p, n_p)$, and thus represented by pairs (p, n_p). Definition 1 concerns this set of trees.

Definition 1. *Given Solved* $\subseteq \{(p, n_p) : p \in V(P_1), n_p \in N(p)\}$, $r \in V(P_1)$ *and* $n_r \in N(r)$, *we say that Solved is an* (r, n_r)*-subtree set if its trees are disjoint subtrees of* $T(r, n_r)$ *and one of them is rooted at* r *(i.e., Solved*$[(1, r)] \neq \emptyset$*).*

Each tree $T(p, n_p)$ in *Solved* has several scores. Each score corresponds to its mapping to a subtree T of H. We only know the root of T, and it belongs to U iff $p \neq r$. Moreover, no tree has different scores for isomorphisms that map its root to the same node in $V(H)$. We use a tuple (p, n_p, h, s) to represent an isomorphism of score s between $T(p, n_p)$ and a subtree of H that maps p to h. Definition 2 concerns these tuples. We note that PS stands for Partial Solutions.

Definition 2. *Let* $PS \subseteq \{(p, n_p, h, s) : p \in V(P_1), n_p \in N(p), h \in V(H), s \in \mathbb{R}\}$, $r \in V(P_1), n_r \in N(r)$ *and* $U \subseteq V(H)$. PS *is an* (r, n_r, U)*-set if*

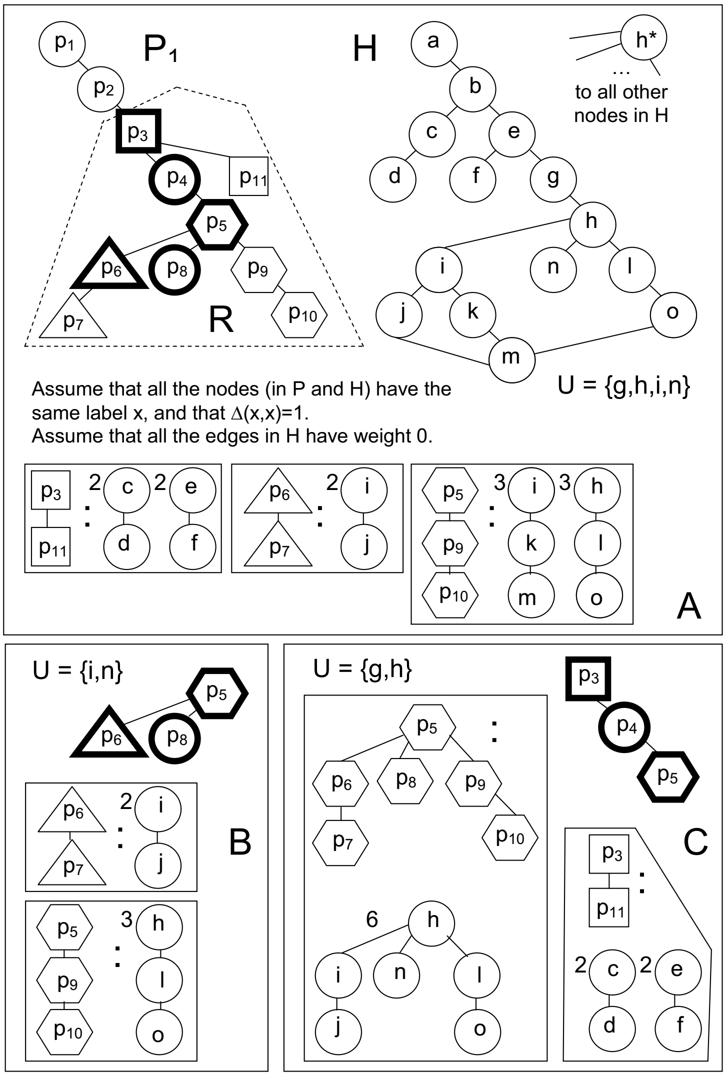

P1
H
p1
p2
p3
p4
p11
p5
p6
p8
p9
p7
R
p10
a
b
c
e
d
f
g
h
i
n
l
j
k
o
m
h*
...
to all other nodes in H
U = {g,h,i,n}
Assume that all the nodes (in P and H) have the same label x, and that Δ(x,x)=1.
Assume that all the edges in H have weight 0.
A
U = {i,n}
B
U = {g,h}
C
2
3
6

Fig. 1. An illustration for Section 2.1. Part A presents the input for AQ-Alg and a recursive stage. *Solved* contains the squares, triangles and hexagons trees. R' is the subtree induced by the bold nodes. Each tree in *Solved* has information on several isomorphisms that are represented by pairs (e.g., the pairs of the squares tree: $(c, 2)$ and $(e, 2)$). We divide the problem of Part A into the subproblems of Parts B and C. We solve Part B and use its answer (i.e., the isomorphism of the hexagons tree in Part C) to solve Part C. Examples for the definitions of Section 2.2 (see Part A): $T(p_3, p_2) = R$, $Solved = \{(p_3, p_4), (p_6, p_5), (p_5, p_8)\}$ is a (p_3, p_2)-subtree set, $PS = \{(p_3, p_4, c, 2), (p_3, p_4, e, 2), (p_6, p_5, i, 2), (p_5, p_8, i, 3), (p_5, p_8, h, 3)\}$ is a (p_3, p_2, U)-set, $T(PS) = R'$, $mid(PS) = \{(p_3, p_2, 4, 0),\ (p_3, p_4, 0, 4),\ (p_3, p_{11}, 0, 4),\ (p_4, p_3, 3, 1),\ (p_4, p_5, 0, 4),\ (p_5, p_4, 2, 2),\ (p_5, p_6, 1, 3),\ (p_5, p_8, 0, 4),\ (p_5, p_9, 0, 4),\ (p_6, p_5, 0, 4), (p_6, p_7, 0, 4), (p_8, p_5, 0, 4)\}$.

1. $\{(p, n_p) : PS[(1, p), (2, n_p)] \neq \emptyset\}$ *is an* (r, n_r)*-subtree set.*
2. $\forall(p, n_p, h, s) \in PS$*:* $\forall s'[(p, n_p, h, s') \in PS \to s = s']$ *and* $(p \neq r \leftrightarrow h \in U)$*.*

Suppose we have an (r, n_r, U)-set PS. We find the best options (corresponding to different mappings of r) to map the roots of the subtrees of $T(r, n_r)$ in PS and the nodes in $T(r, n_r)$ which do not belong to these subtrees to subtrees whose nodes (excluding the mappings of r) are in U. Thus we map all $T(r, n_r)$ and use only nodes in U and nodes that we have already used for computing PS. Definition 3 concerns this set of nodes in $T(r, n_r)$ which we want to map.

Definition 3. *Given an* (r, n_r, U)*-set* PS*,* $T(PS)$ *is the subtree of* P_1 *induced by* $\{v \in V(T(r, n_r)) : \nexists(p, n_p, h, s) \in PS \text{ s.t. } v \in V(T(p, n_p)) \setminus \{p\}\}$*.*

We divide our problem into two smaller subproblems. We achieve this by finding a node $m \in V(T(PS))$ and a neighbor $n_m \in N(m)$ that divide $T(PS)$ into two smaller subtrees: $P_1[V(T(PS)) \cap V(T(m, n_m))]$ and $P_1[V(T(PS)) \setminus V(T(m, n_m))] \cup \{m\}$. Definition 4 concerns our division options.

Definition 4. *Given an* (r, n_r, U)*-set* PS*, we define:*
$mid(PS) = \{(m, n_m, size_L, size_R) : m \in V(T(PS)), n_m \in N(m), size_L = |V(T(PS)) \cap V(T(m, n_m))| - 1, size_R = |V(T(PS))| - size_L - 1\}$*.*

We seek a tuple $(m, n_m, size_L, size_R) \in mid(PS)$ that minimizes $\max\{size_L, size_R\}$. Then, as the following lemma implies, our new subproblems are small.

Lemma 1. *Given a rooted tree* T *s.t.* $v_1, v_2, \ldots, v_n$ *is a preorder of* $V(T)$ *and* $n \geq 3$*, there are* $v_i \in V(T)$ *and* $v_j \in N(v_i)$ *s.t.* $\max\{2, \lceil\frac{n}{3}\rceil\} \leq |V(T(v_i, v_j))| \leq \lfloor\frac{2n}{3}\rfloor$*. If* T *is a path, there are* $v_i \in V(T)$ *and* $v_j \in N(v_i)$ *s.t.* $|V(T(v_i, v_j))| = \lceil\frac{n}{2}\rceil$*.*

2.3 The Algorithm

First note that an input for AQ-Rec is of the form (r, n_r, U, PS), where $r \in V(P_1)$, $n_r \in N(r)$, $U \subseteq V(H)$, and PS is $\emptyset$ or an (r, n_r, U)-set. The output SOL is $\emptyset$ or an (r, n_r, U)-set s.t. $SOL[(1, r), (2, n_r)] = SOL$ (i.e., the tuples in SOL represent mappings of $T(r, n_r)$).

```
AQ-Alg($\mathcal{P}, H, \Delta, W$):

1. Add elements to the input as noted in Section 2.2.
2. $SOL \Leftarrow$ AQ-Rec($p_2, p_1, V(H) \setminus \{h^*\}, \{(p_2, p_3, h^*, 0)\}$).
3. Accept iff ($SOL \neq \emptyset \wedge \max_{(p,n_p,h,s) \in SOL}\{s\} \geq W$).
```

Now we present the pseudocode of AQ-Rec.

```
AQ-Rec($r, n_r, U, PS$):

1. If $PS = \emptyset \vee |V(T(PS))| = 1$: Return $PS$.
```

We handle two base cases. $PS = \emptyset$ implies that we could not map some subtree of $T(r, n_r)$ in previous computations, and thus we return $\emptyset$.

```
2. If $|V(T(PS))| = 2$:
   (a) Denote by $v$ the node in $V(T(PS))$ which is not $r$.
   (b) If $PS[(1,v)] = \emptyset$: Return $\bigcup_{h \text{ s.t. } U \cap N(h) \neq \emptyset, s \text{ s.t. } (r,v,h,s) \in PS} \{(r, n_r, h, \max_{h' \in U \cap N(h)} \{s + w(\{h,h'\}) + \Delta(l(v), l(h'))\})\}$.
   (c) Return $\bigcup_{h \text{ s.t. } \exists h' \in N(h)[PS[(1,v),(3,h')] \neq \emptyset], s \text{ s.t. } (r,v,h,s) \in PS} \{(r, n_r, h, \max_{h' \in N(h), s' \text{ s.t. } (v,r,h',s') \in PS} \{s + s' + w(\{h,h'\})\})\}$.
```

We handle the two remaining base cases. They correspond to whether or not v is a root of a tree in PS. In both, for each mapping of r, we find the best legal mapping of v to a node h' in U.

```
3. $SOL \Leftarrow \emptyset$.
```

SOL will hold tuples that represent the best mappings we find for $T(r, n_r)$.

```
4. Choose $(m, n_m, size_L, size_R) \in mid(PS)$ that minimizes $\max\{size_L, size_R\}$.
```

We find the best nodes m and n_m to divide our problem of mapping $T(PS)$ into the two smaller subproblems of mapping $V(T(PS)) \cap V(T(m, n_m))$ and $(V(T(PS)) \setminus V(T(m, n_m))) \cup \{m\}$.

```
5. $prob_L \Leftarrow \frac{size_L}{|V(T(PS))|-1}$ and $prob_R \Leftarrow 1 - prob_L$.
6. Repeat $\frac{1}{(1-1/e)^2 prob_L^{size_L} prob_R^{size_R}}$ times:
   (a) $U_L \Leftarrow \emptyset$ and $U_R \Leftarrow U$.
   (b) ForEach $h \in U$: With probability $prob_L$ move $h$ from $U_R$ to $U_L$.
```

We randomly partition U into two sets, U_L and U_R, which we use in the first and second subproblems, respectively: the nodes in $(V(T(PS)) \cap V(T(m, n_m))) \setminus \{m\}$ are mapped to nodes in U_L, and then the other nodes in $V(T(PS)) \setminus \{r\}$ are mapped to nodes in U_R. The probability $prob_L$ of a node to be in U_L and the number of executions of Step 6 guarantee that with good probability the solutions to our subproblems allow solving our problem of mapping $T(PS)$.

```
6. (c) $SOL_L \Leftarrow \emptyset$ and $SOL_R \Leftarrow \emptyset$.
```

SOL_L and SOL_R will hold the solutions we find for our subproblems.

```
6. (d) $PS_L \Leftarrow \{(p, n_p, h, s) \in PS : p \in V(T(m, n_m)), p \neq m \leftrightarrow h \in U_L\}$.
   (e) If $PS[(1,m)] = \emptyset$:
       i. If $U_R = \emptyset$: Next iteration.
       ii. Add $\bigcup_{n_p \in N(m) \text{ s.t. } V(T(m,n_p)) = \{m\}, h \in U_R} \{(m, n_p, h, \Delta(l(m), l(h)))\}$ to $PS_L$.
   (f) If $\exists p \in V(T(m, n_m))[PS[(1,p)] \neq \emptyset \wedge PS_L[(1,p)] = \emptyset]$: Next iteration.
```

PS_L holds the tuples in PS that are relevant to mapping $T(m, n_m)$. If it does not have a tree rooted at m, we add all the options for mapping the tree that contains only m to a node in U_R (if $U_R = \emptyset$, we skip the iteration). If we lost the tuples representing all the mappings in PS of a tree that is relevant to PS_L, we skip the iteration.

```
6. (g) $SOL_L \Leftarrow \mathsf{AQ\text{-}Rec}(m, n_m, U_L, PS_L)$.
```

We solve our first subproblem.

```
6. (h) $PS_R \Leftarrow SOL_L \cup \{(p, n_p, h, s) \in PS : p \notin V(T(m, n_m)), h \notin U_L\}$.
   (i) If $SOL_L = \emptyset \vee \exists p \notin V(T(m, n_m))[PS[(1, p)] \neq \emptyset \wedge PS_R[(1, p)] = \emptyset]$:
       Next iteration.
```

PS_R holds SOL_L and the tuples of PS that are relevant to our second subproblem. If $SOL_L = \emptyset$ or we lost the tuples representing all the mappings in PS of a tree that is relevant to our second subproblem, we skip the iteration.

```
6. (j) $SOL_R \Leftarrow \mathsf{AQ\text{-}Rec}(r, n_r, U_R, PS_R)$.
   (k) For each $h, s$ s.t. $(r, n_r, h, s) \in SOL_R \wedge \nexists s'[(r, n_r, h, s') \in SOL \wedge s \leq s']$:
       $SOL \Leftarrow (SOL \cup \{(r, n_r, h, s)\}) \setminus SOL[(1, r), (2, n_r), (3, h)]$.
7. Return $SOL$.
```

We solve our second subproblem. Then we update SOL to hold the tuples representing the best mapping of $T(r, n_r)$ we have found so far.

Theorem 1. AQ-Alg *solves inputs for PIQ where $\mathcal{P}$ is a set of one tree in $O(6.75^{k+O(\log k)}|E(H)|)$ time and $O(|V(H)| \log^2 k)$ space. If P_1 is a path, it runs in $O(4^{k+O(\log k)}|E(H)|)$ time.*

3 PIQ-Alg: An Algorithm for PIQ

We now present an algorithm for PIQ that can handle inputs where $\mathcal{P}$ is a set of trees (i.e., P_i is a tree for all $1 \leq i \leq t$). PIQ-Alg is a modification of AQ-Alg.

3.1 Overview

The overview is illustrated in Fig. 2. Each recursive stage concerns a rooted subtree R of a tree in $\mathcal{P}$, $U \subseteq V(H)$, a set *Solved* of rooted trees and $size \in \mathbb{N}$. Each tree in *Solved* has several triples (AQ-Alg only needs pairs). Each triple consists of a set of trees $\hat{\mathcal{P}} \subseteq \mathcal{P}$, $h \in V(H)$ and a score s. It concerns a subtree S of H, a partition of $V(S)$ into the subsets $\{V_S^1, \ldots, V_S^{|\hat{\mathcal{P}}|+1}\}$, an isomorphism m_1 between the tree in *Solved* and $S[V_S^1]$ that maps the root of the tree to h, and an isomorphism m_i between $S[V_S^i]$ and a different tree in $\hat{\mathcal{P}}$ for all $2 \leq i \leq |\hat{\mathcal{P}}| + 1$, such that $\sum_{1 \leq i \leq |\hat{\mathcal{P}}|+1} \sum_{v \in V_S^i} \Delta(l(v), l(m_i(v))) + \sum_{e \in E(S)} w(e) = s$. Note that such a triple can be considered as a *partial solution*.

Using the triples, we map sets of *size* nodes to subtrees of H. Each node (excluding the root of R) is mapped to a node in U, such that neighbors are mapped to neighbors. Such a set contains the nodes of R', that is the subtree of R induced by the nodes in R that are not in any tree in *Solved* and the roots of the subtrees of R that are in *Solved*. Moreover, such a set must help us *complete* mapping the trees in $\mathcal{P}$ that have subtrees in *Solved*, excluding the tree containing R. Thus it also contains their nodes, excluding those that belong to trees in *Solved* and are not their roots. The number of nodes such set a must contain may be less than *size*, and thus we map several sets of *size* nodes (we add nodes of trees in $\mathcal{P}$ that do not have subtrees in *Solved*).

In the base cases of the recursion, $size \leq 2$. Otherwise we divide our problem into two subproblems as follows. Any set of *size* nodes that we attempt to map may contain nodes of different trees in $\mathcal{P}$, and we do not know in advance how to *connect* them.[3] Thus we examine several options for dividing the set of nodes we must map into two sets to be used in the first and second subproblems. Such a division may not imply the number of nodes we should map in each subproblem (since the number of nodes we must map may be less than *size*), and thus we also examine several options for these numbers. As in AQ-Alg, we randomly divide U into two subsets. We use the first subset to solve our first subproblem. Then we use the results and the second subset to solve our second subproblem.

3.2 Preliminaries

Each definition presented in this section is preceded by an explanation of its relevance, and is illustrated in Fig. 2.

Choose $p_2 \in V(P_1)$, and add to P_1 a new node p_1 and an edge $\{p_1, p_2\}$ of weight 0. Define $\forall l \in L : \Delta(l(p_1), l) = -\infty$. Add a new node h^* to H and connect it to all the nodes in $V(H)$ by edges of weight 0.

Since we may not know a topology for a solution (we only know it is a tree), we cannot define a preorder on its nodes and use the form $T(p, n_p)$, as in AQ-Alg.

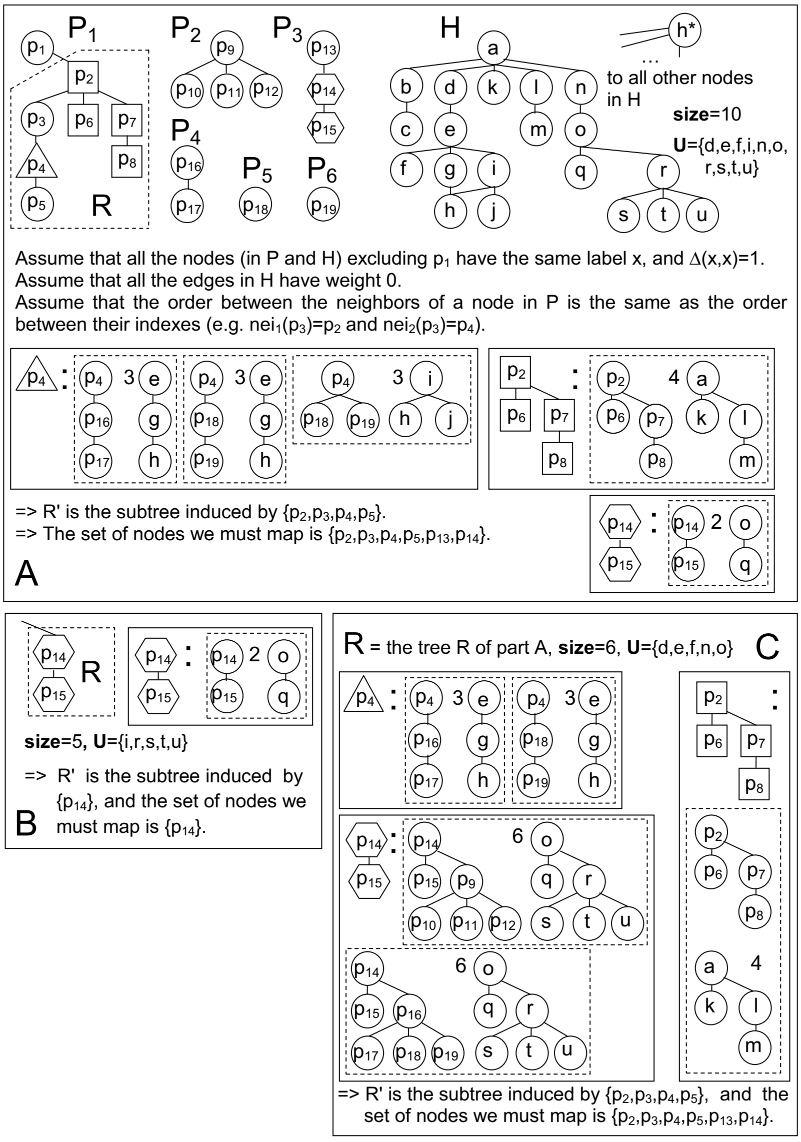
We order the neighbors of each $p \in V(\mathcal{P})$ (arbitrarily), and denote them by $nei_1(p), \ldots, nei_{|N(p)|}(p)$. Denote $N(p, nei_i(p), nei_j(p)) = \{nei_l(p) \in N(p) : i \leq l < j \vee j < i \leq l \vee l < j < i\}$, $N(p, nei_i(p), nil) = \{nei_l(p) \in N(p) : i \leq l\}$ and $N(p, nil, nil) = \{\}$. $P(p)$ denotes the tree in $\mathcal{P}$ that contains p, and $T(p, nei_i(p), nei_j(p))$ denotes the subtree induced by the nodes reachable from p in $P(p)[V(P(p)) \setminus (N(p) \setminus N(p, nei_i(p), nei_j(p)))]$. We root $T(p, nei_i(p), nei_j(p))$ at p. In PIQ-Alg, $T(p, nei_i(p), nei_j(p))$ is the form of the trees that we have at each recursive stage. Definition 5 concerns these trees.

Definition 5. *Given Solved* $\subseteq \{(p, n_p^1, n_p^2) : p \in V(\mathcal{P}), n_p^1 \in N(p) \cup \{nil\}, n_p^2 \in N(p) \cup \{nil\}$ *s.t.* $n_p^1 = nil \rightarrow n_p^2 = nil\}$, $r \in V(\mathcal{P}), n_r^1 \in N(r) \cup \{nil\}$ *and* $n_r^2 \in N(r) \cup \{nil\}$ *s.t.* $n_r^1 = nil \rightarrow n_r^2 = nil$, *we say that Solved is an* (r, n_r^1, n_r^2)-*subtree set if its trees are disjoint, those that are subtrees of* $P(r)$ *are also subtrees of* $T(r, n_r^1, n_r^2)$, *and Solved*$[(1, r), (3, n_r^2)] \neq \emptyset$.

Now we define the information of each tree in *Solved*. It is similar to Definition 2, but now each tree also has information on a set $\hat{\mathcal{P}}$ of trees in $\mathcal{P}$ that are connected to it, and each of its scores also corresponds to their mappings.

Definition 6. *Let PS* $\subseteq \{(p, n_p^1, n_p^2, \hat{\mathcal{P}}, h, s) : p \in V(\mathcal{P}), n_p^1 \in N(p) \cup \{nil\}, n_p^2 \in N(p) \cup \{nil\}$ *s.t.* $n_p^1 = nil \rightarrow n_p^2 = nil, h \in V(H), \hat{\mathcal{P}} \subseteq \mathcal{P}, s \in \mathbb{R}\}$, $r \in V(\mathcal{P}), n_r^1 \in$

[3] We need to connect these trees to get one tree that is mapped to a subtree of H



Assume that all the nodes (in P and H) excluding p1 have the same label x, and Δ(x,x)=1.
Assume that all the edges in H have weight 0.
Assume that the order between the neighbors of a node in P is the same as the order between their indexes (e.g. nei1(p3)=p2 and nei2(p3)=p4).
to all other nodes in H
size=10
U={d,e,f,i,n,o,r,s,t,u}
=> R' is the subtree induced by {p2,p3,p4,p5}.
=> The set of nodes we must map is {p2,p3,p4,p5,p13,p14}.
A
size=5, U={i,r,s,t,u}
=> R' is the subtree induced by {p14}, and the set of nodes we must map is {p14}.
B
R = the tree R of part A, size=6, U={d,e,f,n,o}
C
=> R' is the subtree induced by {p2,p3,p4,p5}, and the set of nodes we must map is {p2,p3,p4,p5,p13,p14}.

Fig. 2. An illustration for Section 3.1. Part A presents the input for PIQ-Alg and a recursive stage. *Solved* contains the squares, triangles and hexagons trees. Each tree in *Solved* has information on several isomorphisms that are represented by triples (e.g., the triples of the triangles tree: $(\{P_4\}, e, 3)$, $(\{P_5, P_6\}, e, 3)$ and $(\{P_5, P_6\}, i, 3)$). We divide the problem of Part A into the subproblems of Parts B and C. We solve Part B and use its answer (i.e., the isomorphisms of the hexagons tree in Part C) to solve Part C. Examples for the definitions of Section 3.2 (see Part A): $T(p_2, p_3, p_1) = R$, $Solved = \{(p_2, p_6, p_1), (p_4, p_3, p_3), (p_{14}, p_{15}, p_{13})\}$ is a (p_2, p_3, p_1)-subtree set, $PS = \{(p_2, p_6, p_1, \{\}, a, 4), (p_4, p_3, p_3, \{P_4\}, e, 3), (p_4, p_3, p_3, \{P_5, P_6\}, e, 3), (p_4, p_3, p_3, \{P_5, P_6\}, i, 3), (p_{14}, p_{15}, p_{13}, \{\}, o, 2)\}$ is a (p_2, p_3, p_1, U)-set, $V(PS) = \{p_2, p_3, p_4, p_5, p_{13}, p_{14}\}$.

$N(r) \cup \{nil\}, n_r^2 \in N(r) \cup \{nil\}$ s.t. $n_r^1 = nil \to n_r^2 = nil$, and $U \subseteq V(H)$. PS is an (r, n_r^1, n_r^2, U)-set if

1. *$\{(p, n_p^1, n_p^2) : PS[(1,p),(2,n_p^1),(3,n_p^2)] \neq \emptyset\}$ is an (r, n_r^1, n_r^2)-subtree set.*
2. *$\forall(p, n_p^1, n_p^2, \hat{\mathcal{P}}, h, s) \in PS$: $\forall s'[(p, n_p^1, n_p^2, \hat{\mathcal{P}}, h, s') \in PS \to s = s']$, $(p \neq r \leftrightarrow h \in U)$ and $\forall \hat{\mathcal{P}}'[PS[(4, \hat{\mathcal{P}}')] \neq \emptyset \to P(p) \notin \hat{\mathcal{P}}']$.*

Next we define the set of nodes that a given (r, n_r^1, n_r^2, U)-set PS implies we must map. This is a modification of Definition 3.

Definition 7. *Given an (r, n_r^1, n_r^2, U)-set PS, $V(PS)$ is the set of nodes $(V(T(r, n_r^1, n_r^2)) \cup (\bigcup_{p \text{ s.t. } PS[(1,p)] \neq \emptyset \wedge P(p) \neq P(r)} V(P(p)))) \setminus (\bigcup_{(p, n_p^1, n_p^2, \hat{\mathcal{P}}, h, s) \in PS} V(T(p, n_p^1, n_p^2)) \setminus \{p\})$.*

We do not use a definition similar to Definition 4 since we may not know a topology for $V(PS)$ in a solution, and thus cannot determine how each node divides it. We consider every node in $V(PS)$ as a possible divisor of our problem and examine several options for the sizes of the resulting smaller subproblems.

The next definition is used for the sake of clarity of the pseudocode. Given (r, n_r^1, n_r^2, U)-sets PS and PS', we define a calculation that uses PS' to update PS to hold the information of both sets that corresponds to the best scores.

Definition 8. *Given (r, n_r^1, n_r^2, U)-sets PS and PS', $PS \overset{+}{\Leftarrow} PS'$ is defined as $\{(p, n_p^1, n_p^2, \hat{\mathcal{P}}, h, s) \in PS \cup PS' : \forall s'[(p, n_p^1, n_p^2, \hat{\mathcal{P}}, h, s') \in PS \cup PS' \to s' \leq s]\}$.*

3.3 The Algorithm

First note that an input for PIQ-Rec is of the form $(r, n_r^1, n_r^2, U, size, PS)$, where $r \in V(\mathcal{P})$, $n_r^1 \in N(p) \cup \{nil\}$, $n_r^2 \in N(p) \cup \{nil\}$ s.t. $n_r^1 = nil \to n_r^2 = nil$, $U \subseteq V(H)$, $size \in \mathbb{N}$ and $\emptyset$ or an (r, n_r^1, n_r^2, U)-set PS s.t. $|V(PS)| \leq size$.

The output SOL is $\emptyset$ or an $(r, n_r^1, n_r^2, U, size)$-set such that $SOL[(1,r),(2,n_r^1),(3,n_r^2)] = SOL$.

```
PIQ-Alg($\mathcal{P}, H, \Delta, W$):

1. Add elements to the input as noted in Section 3.2.
2. $SOL \Leftarrow$ PIQ-Rec($p_1, p_2, nil, V(H) \setminus \{h^*\}, k+1, \{(p_1, nil, nil, h^*, 0)\}$).
3. Accept iff ($SOL \neq \emptyset \wedge \max_{(p, n_p^1, n_p^2, \hat{\mathcal{P}}, h, s) \in SOL}\{s\} \geq W$).
```

Now we present the pseudocode of PIQ-Rec.

```
PIQ-Rec($r, n_r^1, n_r^2, U, size, PS$):

1. If $PS = \emptyset \vee size = 1$: Return $PS$.
2. $SOL \Leftarrow \emptyset$.
```

Step 1 concerns two base cases that we handle as in AQ-Rec. SOL, as in AQ-Rec, will hold tuples that represent the best mappings we find.

3. If $size = 2$:
   (a) If $|V(PS)| = 2$: Denote by v the node in $V(PS)$ which is not r.
   (b) If $|V(PS)| = 1$ then ForEach $(p, n_p^1, n_p^2, \hat{\mathcal{P}}, h, s) \in PS, v \in V(\mathcal{P}) \setminus \{r\}$ s.t. $[P(v) \notin \hat{\mathcal{P}} \ \wedge \ N(v) = \emptyset], h' \in U \cap N(h)$:
      – $SOL \overset{+}{\Leftarrow} \{(r, n_r^1, n_r^2, \hat{\mathcal{P}} \cup \{P(v)\}, h, s + w(\{h, h'\}) + \Delta(l(v), l(h')))\}$.
   (c) ElsIf $PS[(1, v)] = \emptyset$ then ForEach $(p, n_p^1, n_p^2, \hat{\mathcal{P}}, h, s) \in PS, h' \in U \cap N(h)$:
      – $SOL \overset{+}{\Leftarrow} \{(r, n_r^1, n_r^2, \hat{\mathcal{P}}, h, s + w(\{h, h'\}) + \Delta(l(v), l(h')))\}$.
   (d) ElsIf $v \in V(P(r))$ then ForEach $(p, n_p^1, n_p^2, \hat{\mathcal{P}}, h, s) \in PS[(1, r)], \hat{\mathcal{P}}' \subseteq \mathcal{P} \setminus \hat{\mathcal{P}}, h' \in N(h)$, s' s.t. $\exists n_v[(v, n_v, r, \hat{P}', h', s') \in PS]$:
      – $SOL \overset{+}{\Leftarrow} \{(r, n_r^1, n_r^2, \hat{\mathcal{P}} \cup \hat{\mathcal{P}}', h, s + w(\{h, h'\}) + s')\}$.
   (e) Else ForEach $(p, n_p^1, n_p^2, \hat{\mathcal{P}}, h, s) \in PS[(1, r)], \hat{\mathcal{P}}' \subseteq \mathcal{P} \setminus \hat{\mathcal{P}}, h' \in N(h)$, s' s.t. $\exists n_v[(v, n_v, nil, \hat{\mathcal{P}}', h', s') \in PS]$:
      – $SOL \overset{+}{\Leftarrow} \{(r, n_r^1, n_r^2, \hat{\mathcal{P}} \cup \hat{\mathcal{P}}' \cup \{P(v)\}, h, s + w(\{h, h'\}) + s')\}$.
   (f) Return SOL.

If $size = 2$, we have four base cases. For each of the two base cases corresponding to whether or not $PS[(1, v)] = \emptyset$ (these are the base cases of AQ-Rec), we need two base cases that correspond to whether or not v and r belong to the same tree in $\mathcal{P}$ (note that if they do, then they are neighbors).

4. ForEach $m \in V(\mathcal{P}), n_m^1 \in N(m) \cup \{nil\}, n_m^2 \in N(m) \cup \{nil\}$ s.t. $(n_m^1 = nil \rightarrow n_m^2 = nil), size_L \in \{\max\{1, \lceil \frac{size}{3} \rceil - 1\}, \max\{2, \lceil \frac{size}{3} \rceil\}, \ldots, \lfloor \frac{2size}{3} \rfloor - 1\}$, partition (P_L, P_R) of $\{p : PS[(1, p)] \neq \emptyset]\}$ s.t. $m \notin P_R$:

We examine choices for m, n_m^1 and n_m^2 that may divide our problem of mapping a set of $size$ nodes, which is a superset of $V(PS)$, into smaller subproblems.

Since we do not know the entire set of nodes we need to map in each of the resulting subproblems, we examine several options to choose their sizes. We examine only options in which both are at most $\lfloor \frac{2size}{3} \rfloor + 1$ (if $size = 3$, then at most 2) to get the running time stated in Theorem 2. This still allows us to find a solution (see Lemma 1 for intuition). For the same reason, we examine all the partitions of the set roots of trees in PS into two sets, P_L and P_R, to be used in the first and second subproblems, respectively. We require that $m \notin P_R$ since in our second subproblem we will only be interested in the mappings of m that were found for our first subproblem (as in AQ-Rec).

4. (a) $size_R \Leftarrow size - size_L - 1, prob_L \Leftarrow \frac{size_L}{size-1}$ and $prob_R \Leftarrow 1 - prob_L$.
   (b) Repeat $\frac{1}{(1-1/e)^2 prob_L{}^{size_L} prob_R{}^{size_R}}$ times:
      i. $U_L \Leftarrow \emptyset$ and $U_R \Leftarrow U, SOL_L \Leftarrow \emptyset$ and $SOL_R \Leftarrow \emptyset$.
      ii. ForEach $h \in U$: With probability $prob_L$ move h from U_R to U_L.

As in AQ-Rec, we examine several partitions of U into U_L and U_R, and SOL_L and SOL_R hold the solutions we find for our first and second subproblems.

4. (b) iii. $PS_L \Leftarrow \{(p, n_p^1, n_p^2, \hat{\mathcal{P}}, h, s) \in PS : p \in P_L, p \neq m \leftrightarrow h \in U_L\}$.
      iv. If $m \notin P_L$: $PS_L \overset{+}{\Leftarrow} \bigcup_{h \in U_R} \{(m, n_m^2, n_m^2, \emptyset, h, \Delta(l(m), l(h)))\}$.

```
      v. ForEach $(p, n_p^1, n_p^2, \hat{\mathcal{P}}, h, s) \in PS_L$ s.t. $P(m) \in \hat{\mathcal{P}}$: $PS_L \Leftarrow PS_L \setminus \{(p, n_p^1, n_p^2, \hat{\mathcal{P}}, h, s)\}$.
      vi. If $\exists p \in P_L$ s.t. $PS_L[(1,p)] = \emptyset \vee PS_L$ is not an $(m, n_m^1, n_m^2, U_L)$-set $\vee |V(PS_L)| > size_L + 1$: Next iteration.
```

As in AQ-Rec, PS_L holds the tuples of PS that are relevant to our first subproblem, and if it does not have a tree rooted at m, we add all the options for mapping the tree that contains only m to a node in U_R. We remove from PS_L tuples that do not correspond to m and yet map its entire tree. If P_L has a node that is not a root of a tree in PS_L, or PS_L is not a (m, n_m^1, n_m^2, U)-set, or PS_L requires mapping too many nodes, we skip the iteration.

```
4. (b) vii. $SOL_L \Leftarrow$ PIQ-Rec$(m, n_m^1, n_m^2, U_L, size_L + 1, PS_L)$.
```

We solve our first subproblem.

```
4. (b)viii. $PS_R \Leftarrow SOL_L \cup \{(p, n_p^1, n_p^2, \hat{\mathcal{P}}, h, s) \in PS : p \in P_R, h \notin U_L\}$.
      ix. ForEach $(p, n_p^1, n_p^2, \hat{\mathcal{P}}, h, s) \in PS_R$ s.t. $\exists p' \in P_R \cup \{m\}$ for which $P(p') \in \hat{\mathcal{P}}$: $PS_R \Leftarrow PS_R \setminus \{(p, n_p^1, n_p^2, \hat{\mathcal{P}}, h, s)\}$.
      x. If $\exists p \in P_R \cup \{m\}$ s.t. $PS_R[(1,p)] = \emptyset \vee PS_R$ is not an $(r, n_r^1, n_r^2, U_R)$-set $\vee |V(PS_R)| > size_R + 1$: Next iteration
```

PS_R holds the solutions of the first subproblem and the tuples of PS that are relevant to our second subproblem. We remove from PS_R tuples that do not correspond to a node $p' \in P_R \cup \{m\}$ and yet map its entire tree. If the resulting PS_R is illegal (the check is similar to that in Step 4(b)vi), we skip the iteration.

```
4. (b) xi. $SOL \overset{+}{\Leftarrow}$ PIQ-Rec$(r, n_r^1, n_r^2, U_R, size_R + 1, PS_R)$.
5. Return $SOL$.
```

We solve our second subproblem and update SOL.

Theorem 2. PIQ-Alg *solves inputs for PIQ where $\mathcal{P}$ is a set of trees in $O(6.75^{k+O(\log^2 k)} 3^t |E(H)|)$ time and $O(2^t |V(H)| \log^2 k)$ space.*

References

1. Alon, N., Yuster, R., Zwick, U.: Color coding. J. Assoc. Comput. Mach. 42(4), 844–856 (1995)
2. Björklund, A., Husfeldt, T., Kaski, P., Koivisto, M.: Narrow sieves for parameterized paths and packings. CoRR abs/1007.1161 (2010)
3. Blin, G., Sikora, F., Vialette, S.: Querying graphs in protein-protein interactions networks using feedback vertex set. IEEE/ACM Trans. Comput. Biol. Bioinform. 7(4), 628–635 (2010)
4. Bruckner, S., Hüffner, F., Karp, R.M., Shamir, R., Sharan, R.: Topology-free querying of protein interaction networks. In: Batzoglou, S. (ed.) RECOMB 2009. LNCS, vol. 5541, pp. 74–89. Springer, Heidelberg (2009)

5. Chen, J., Kneis, J., Lu, S., Molle, D., Richter, S., Rossmanith, P., Sze, S., Zhang, F.: Randomized divide-and-conquer: Improved path, matching, and packing algorithms. SIAM J. on Computing 38(6), 2526–2547 (2009)
6. Dost, B., Shlomi, T., Gupta, N., Ruppin, E., Bafna, V., Sharan, R.: Qnet: a tool for querying protein interaction networks. J. Comput. Biol. 15(7), 913–925 (2008)
7. Downey, R.G., Fellows, M.R.: Fixed-parameter tractability and completeness ii: on completeness for w [1]. Theor. Comput. Sci. 141(1-2), 109–131 (1995)
8. Fionda, V., Palopoli, L.: Biological network querying techniques: Analysis and comparison. J. Comput. Biol. 18(4), 595–625 (2011)
9. Garey, M.R., Johnson, D.S.: Computers and intractability: a guide to the theory of NP-completeness. W.H. Freeman, New York (1979)
10. Guillemot, S., Sikora, F.: Finding and counting vertex-colored subtrees. In: Hliněný, P., Kučera, A. (eds.) MFCS 2010. LNCS, vol. 6281, pp. 405–416. Springer, Heidelberg (2010)
11. Hüffner, F., Wernicke, S., Zichner, T.: Algorithm engineering for color-coding with applications to signaling pathway detection. Algorithmica 52(2), 114–132 (2008)
12. Koutis, I.: Faster algebraic algorithms for path and packing problems. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part I. LNCS, vol. 5125, pp. 575–586. Springer, Heidelberg (2008)
13. Lacroix, V., Fernandes, C.G., Sagot, M.F.: Motif search in graphs: Application to metabolic networks. IEEE/ACM Trans. Comput. Biol. Bioinform. 3(4), 360–368 (2006)
14. Mano, A., Tuller, T., Beja, O., Pinter, R.Y.: Comparative classification of species and the study of pathway evolution based on the alignment of metabolic pathways. BMC Bioinform 11(S-1), S38 (2010)
15. Niedermeier, R.: Invitation to fixed-parameter algorithms. Oxford University Press (2006)
16. Pinter, R.Y., Rokhlenko, O., Tsur, D., Ziv-Ukelson, M.: Approximate labelled subtree homeomorphism. J. Discrete Algorithms 6(3), 480–496 (2008)
17. Pinter, R.Y., Rokhlenko, O., Yeger-Lotem, E., Ziv-Ukelson, M.: Alignment of metabolic pathways. Bioinformatics 21(16), 3401–3408 (2005)
18. Pinter, R.Y., Zehavi, M.: Algorithms for topology-free and alignment queries. Technion Technical Reports CS-2012-12 (2012)
19. Shlomi, T., Segal, D., Ruppin, E., Sharan, R.: Qpath: a method for querying pathways in a protein-protein interaction networks. BMC Bioinform. 7, 199 (2006)
20. Sikora, F.: An (almost complete) state of the art around the graph motif problem. Université Paris-Est Technical reports (2012)
21. Wang, H., Xiang, T., Hu, X.: Research on pattern matching with wildcards and length constraints: methods and completeness (2012), `http://www.intechopen.com/books/bioinformatics`

An Application of Completely Separating Systems to Graph Labeling

Leanne Rylands[1], Oudone Phanalasy[2,3], Joe Ryan[4], and Mirka Miller[2,5,6,7]

[1] School of Computing, Engineering and Mathematics, University of Western Sydney, Sydney, Australia
l.rylands@uws.edu.au

[2] School of Mathematical and Physical Sciences, University of Newcastle, Newcastle, Australia
oudone.phanalasy@gmail.com, mirka.miller@newcastle.edu.au

[3] Department of Mathematics, National University of Laos, Vientiane, Laos

[4] School of Electrical Engineering and Computer Science, University of Newcastle, Newcastle, Australia
joe.ryan@newcastle.edu.au

[5] Department of Mathematics, University of West Bohemia, Pilsen, Czech Republic

[6] Department of Informatics, King's College London, London, UK

[7] Department of Mathematics, ITB Bundung, Bundung, Indonesia

Abstract. In this paper a known algorithm used for the construction of completely separating systems (CSSs), Roberts' Construction, is modified and used in a variety of ways to build CSSs. The main interest is in CSSs with different block sizes. A connection between CSSs and vertex antimagic edge labeled graphs is then exploited to prove that various non-regular graphs are antimagic. An outline for an algorithm which produces some of these non-regular graphs together with a vertex antimagic edge labeling is presented.

Keywords: completely separating system, antimagic labeling, non-regular graph.

1 Introduction

In 1969, Dickson [4] introduced completely separating systems. A *completely separating system* (CSS) on $[n] = \{1, 2, \ldots, n\}$, or (n)CSS, is a collection of subsets $\mathcal{C}$ of $[n]$ in which for each pair of distinct elements $a, b \in [n]$, there exist $A, B \in \mathcal{C}$ such that $a \in A$, $b \notin A$, $a \notin B$ and $b \in B$. We say that a and b are completely separated.

The sets in the (n)CSS are called *blocks*. Let k be a positive integer and $\mathcal{C}$ an (n)CSS. If $|A| = k$ for all $A \in \mathcal{C}$, then $\mathcal{C}$ is an (n, k)CSS. A *d-element* in a collection of sets is an element which occurs in exactly d sets in the collection. We often omit the brackets and commas when writing a block, for example, the set $\{a, b, c\}$ is often written as abc. An example of a CSS is $\mathcal{C} = \{123, 145, 246, 356\}$; it is a $(6, 3)$CSS.

© Springer-Verlag Berlin Heidelberg 2013

Several variants on the theme of CSSs have been explored in [13,15,16], among others. Roberts [16] gives a method for the construction of various CSSs; a special case of this is given below as it proves useful for the study of antimagic labelings of graphs.

Roberts' Construction [16] Assume that $k \geq 2$, $n \geq \binom{k+1}{2}$ and $k|2n$, and let $R = 2n/k$. Roberts' Construction is the following algorithm:

```
1. Begin with an $R \times k$ array $M = (m_{ij})$ with each entry set to 0;
   $t := 1$.
2. Repeat
       Place $t$ in the two positions $m_{ij}$ determined by
           $\min_j \min_i \{m_{ij} \mid m_{ij} = 0\}$ and
           $\min_i \min_j \{m_{ij} \mid m_{ij} = 0\}$
       $t := t + 1$
   Until $t = n + 1$.
```

An $(R \times k)$-array M is constructed; each row of M forms a subset of $[n]$ and the R rows of M are the blocks of an $(n, k)CSS$.

Roberts' Construction was used to obtain the array of the $(12, 4)$CSS shown in Figure 1. Roberts' Construction produces an (n, k)CSS for every n and k, with $k \geq 2$, $n \geq \binom{k+1}{2}$ and $k|2n$.

Magic labeling of graphs was first introduced by Sedláček [17]. A graph G is *magic* (more specifically, *vertex magic*) if it has an edge labeling, with range the real numbers, such that the sums of the edge labels incident with a vertex are all equal to the same integer, independent of the choice of the vertex.

The idea of a magic labeling has been generalized and used as an inspiration by many authors, for example, [9,8,10,11,12,18]. For more details see [19]. Instead of labeling edges of a graph, it is possible to label vertices, or both edges and vertices. In this paper we deal with edge labeling.

An *edge labeling* is a bijection

$$l : E(G) \to \{1, 2, \ldots, |E(G)|\}.$$

The *weight* of a vertex x, $x \in V(G)$ is defined as

$$\mathrm{wt}(x) = \sum l(xy)$$

where the sum is taken over all vertices y adjacent to x.

At the other extreme in terms of magicness, is the situation whereby all the vertex weights are pairwise distinct. The notion of an antimagic labeling of a graph was introduced by Hartsfield and Ringel [6] in 1990. An *antimagic labeling* of a graph G with q edges is a bijection from the set of edges to the set of positive integers $\{1, 2, \ldots, q\}$ such that all the vertex weights are pairwise distinct. We call such a labeling a 'vertex antimagic edge labeling', or simply, 'antimagic labeling', and we say that a graph G is antimagic if there exists an antimagic labeling of G. Figure 1 shows an antimagic graph.

Note that most edge labelings of a given graph will be neither magic nor antimagic. However, there is the Hartsfield and Ringel conjecture [6], proposed in 1990.

Conjecture 1. [6] Every connected graph, except K_2, is antimagic.

While in general the Hartsfield and Ringel conjecture remains open, research has been conducted in two main directions: some investigate antimagic labelings with restrictions placed on the weights, while others stay with the idea of plain antimagicness but restrict their attention to particular classes of graphs. For example, Bodendiek and Walther [2] introduced the concept of (a, d)-*vertex antimagic* edge labeling. This is an antimagic edge labeling such that the vertex weights form an arithmetic progression starting at a and with difference d. Clearly, any (a, d)-vertex antimagic edge labeling is an antimagic labeling.

The second direction of research into antimagic labeling has been proving the antimagicness of particular families of graphs. Alon *et al.* [1] used probabilistic methods and some techniques from analytic number theory to show that the conjecture is true for all graphs having minimum degree at least $\Omega(\log |V(G)|)$. They also proved that if G is a graph with order $|V(G)| \geq 4$ and maximum degree $\Delta(G)$, $|V(G)| - 2 \leq \Delta(G) \leq |V(G)| - 1$, then G is antimagic. Alon *et al.* [1] also have shown that all complete multipartite graphs, except K_2, are antimagic. Hefetz [7] used the combinatorial nullstellensatz to prove that a graph with 3^k vertices, where k is a positive integer, and admits a K_3-factor is antimagic. Many, many papers have appeared proving that particular classes of graphs are antimagic. Gallian's dynamic survey [5] summarises much of this work.

However, to date there are not many results on the antimagicness of non-regular graphs and, especially, of trees [3,6].

In this paper we present a powerful relationship between completely separating systems and edge labeling of graphs. Based on this relationship we produce a family of regular vertex antimagic edge labeled graphs and then we extend our results to construct vertex antimagic edge labeling for regular and non-regular graphs.

2 Relationship between Completely Separating Systems and Labeling of Graphs

The following two theorems, stated without proof in [14], provide a relationship between completely separating systems and edge labeled graphs.

Theorem 1. *Let $V = \{v_1, \ldots, v_p\}$ be a collection of subsets of $[q]$. If V is a (q)CSS in which each element of $[q]$ is a 2-element and E is the set of all unordered pairs $\{v_i, v_j\}$, where $v_i \cap v_j \neq \emptyset$, then $G = (V, E)$ is a simple graph, $|V| = p$ and $|E| = q$. Also, G has an edge labeling l given by $l(v_i, v_j) = v_i \cap v_j$.*

Proof. Let $V = \{v_1, v_2, \ldots, v_p\}$ be a (q)CSS. Since V is a (q)CSS consisting of 2-elements, an element e of $[q]$ must be in exactly two sets of V. Take V to be the set of vertices of a graph. Define the edges to be the pairs of vertices $\{v_j, v_l\}$ for which there is, in the (q)CSS V, an element e common to both. This e is the label of that edge. The set of edges E can be identified with the set $[q]$. If $G = (V, E)$ is not a simple graph, then there exists either a loop or multiple edge. This means that either there exists at least one member of V containing a duplicate element e of $[q]$ or there is a pair of sets which both contain e_i and e_j, $i \neq j$. Both scenarios contradict V being a CSS.

Note that if $V = \{v_1, \ldots, v_p\}$ is a (q,k)CSS then G is a k-regular graph together with an edge labeling.

Theorem 2. *Let $G = (V, E)$ be a simple graph with $|V| = p$, $|E| = q$ with an edge labeling given by bijection $l : E \to [q]$. For $v \in V$, let S_v be the set of labels of edges incident with v. Then the collection $\{S_v \mid v \in V\}$ is a (q)CSS consisting of 2-elements.*

Proof. Let $G = (V, E)$ be a graph having an edge labeling. As each edge has a unique label from the set $[q]$, E can be identified with $[q]$. Let $V = \{v_1, v_2, \ldots, v_p\}$. Identify $v_j \in V$, with S_{v_j}. Then $v_j \subseteq [q]$, and V is a collection of subsets of $[q]$. As each edge is incident with two distinct vertices, each $e \in [q]$ appears in exactly 2 sets in V.

Assume that V is not a completely separating system on E. Then there exist at least two numbers $e_j, e_l \in [q]$ which are not separated from each other. As each element of $[q]$ is a 2-element, this means that there exist two members v_j and v_l, $j \neq l$, both containing e_j and e_l. Hence G contains multiple edges which contradicts the fact that G is a simple graph. Therefore, V is a (q)CSS.

Note that in Theorem 2, if G is a k-regular graph, that is, each vertex of G has degree k, then V is a (q,k)CSS.

1	2	3	4
1	5	6	7
2	6	8	9
3	7	10	11
4	8	10	12
5	9	11	12

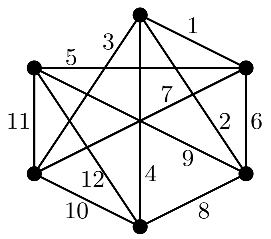


Fig. 1. A CSS obtained using Roberts' Construction and the corresponding graph with antimagic edge labeling

Based on the relationship between CSSs and edge labeled graphs we present results concerning the antimagicness of some regular and non-regular graphs obtained from modificaitons of Roberts' Construction for CSSs.

A CSS is often written as an "array" of numbers, which need not be rectangular (the array can be considered as rectangular with some places unused). This is done in Figure 1. If an array is the array of a CSS consisting of 2-elements then it defines a labeled graph. Consequently we write $G(V, E, L)$ (or simply, $G(L)$) to denote a graph $G = (V, E)$ (or G) with the array L as an edge labeling.

Theorem 3. *Let L be the array of a (q, k)CSS obtained by Roberts' Construction. Then the k-regular graph $G(V, E, L)$, where $|V| = p = 2q/k$, $|E| = q$, is antimagic.*

Proof. Use Roberts' Construction to obtain $V = \{v_1, v_2, \ldots, v_p\}$, a (q, k)CSS, where v_i is the set of elements in row i of the array L. It is clear that V consists of 2-elements. Let $e_{i,j}$ be the element in row i and column j of L. Given any pair v_i and v_l with $i < l$, for $1 \leq j \leq k$, either $e_{i,j}$ is less than $e_{l,j}$ or there exists at most one pair $e_{i,j} = e_{l,j}$. Therefore, $\mathrm{wt}(v_i) = \sum_{e \in v_i} e$ must be less than $\mathrm{wt}(v_l) = \sum_{e \in v_l} e$.

Thus for every integer $p = 2q/k$, Roberts' Construction defines an antimagic k-regular graph $G = (V, E)$ with $|V| = p$ and $|E| = q$; write $B(p, k)$ for this labeled graph. This family of graphs includes all cycles $C_n = B(n, 2)$ and all complete graphs $K_n = B(n, n-1)$, for $n \geq 3$. Figure 1 shows the antimagic graph $B(6, 4)$ (corresponding to the $(12, 4)$CSS shown).

Disjoint unions of graphs in the family $B(n, k)$ are proved to be antimagic in the following lemma.

Lemma 1. *Let $G_j(L_j)$, $1 \leq j \leq s$, where L_j is the array of a (q_j, k_j)CSS obtained using Roberts' Construction. Then the disjoint union $H = \bigcup_{j=1}^{s} G_j$ is antimagic.*

Proof. Let $G_j(L_j)$, $1 \leq j \leq s$ be a k-regular graph with p_j vertices and q_j edges. We may assume that $k_j \leq k_{j+1}$. We construct the array A of edge labels of H as follows.

1. Relabel the edge labels in the array L_j, $1 \leq j \leq s$, by adding $\sum_{t=1}^{j-1} q_t$ to each of the original edge labels;
2. Form the array A as shown below.

$$\begin{array}{c} L_1 \\ L_2 \\ \vdots \\ L_s \end{array}$$

By the construction of the array A, it is clear that the weight of each vertex (row) in the array is less than the weight of the vertex below.

3 Modification of Completely Separating Systems

In this section Roberts' Construction and Theorems 1 and 2 are exploited to provide further constructions of antimagic edge labeling of graphs.

3.1 Edge Deletion with No Isolated Vertex

Let $G = (V, E)$ be a graph and $D \subset E$. Define the edge deletion subgraph $G - D$ to be the subgraph of G obtained from G by deleting all edges in D and isolated vertices if any exist.

Theorem 4. *Let $G(V, E, L)$ be a graph, where L is the array of edge labels obtained by Roberts' Construction. Identify E with $\{1, 2, \ldots, |E|\}$, the set of edge labels. Let $D = [t] \subset E$ with $t \leq |E| - 2$. Then $G - D$ is antimagic.*

Proof. Construct the array L' of edge labels of $G - D$ by subtracting t from each edge label in L and deleting all non positive edge labels and then the vertices (rows) with no entries. It is clear that L' defines a CSS. By Theorem 1 $G(L')$ is an edge labeled graph and it easy to check that the weight of each vertex (row) in the array L' is less than the weight of the vertex (row) below; hence $G(L') = G - D$ is antimagic.

Recall the antimagic graph $G = (V, E)$ from Figure 1. Let $D = \{1, 2, 3\}$. The antimagic graph $G - D$ is shown in Figure 2. Note that the deleted edges are shown by dashed lines.

$$\begin{array}{r} 1 \\ 2\ 3\ 4 \\ 3\ 5\ 6 \\ 4\ 7\ 8 \\ 1\ 5\ 7\ 9 \\ 2\ 6\ 8\ 9 \end{array}$$

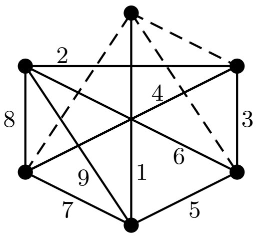


Fig. 2. The array (CSS) L' corresponding to the graph $G - D$ and an antimagic edge labeling

3.2 Edge Switching

Let $G_j(L_j)$, for $1 \leq j \leq 2$, be a labeled k_j-regular graph with q_j edges, where L_j is the CSS, or array of edge labels obtained by Roberts' Construction. Assume that $k_1 \leq k_2$. We construct the edge switching of $n_1 < k_1$ edges in G_1 with $n_2 < k_2$ edges in G_2 as follows.

1. Replace the edge labels in L_2 with the new labels obtained by adding q_1 to each of the original edge labels;
2. Switch the last largest n_1 labels in the last row of L_1 with the smallest n_2 labels in the first row of L_2.

Denote by $G_1^{n_1} \bowtie G_2^{n_2}$ the graph obtained from this edge switching construction.

To guarantee the antimagicness of $G_1^{n_1} \bowtie G_2^{n_2}$, the conditions $k_1 \leq k_2$, $1 \leq n_2 \leq \lfloor \frac{k_2}{2} \rfloor$ and $k_1 \leq k_1 - n_1 + n_2 \leq k_2 - n_2 + n_1 \leq k_2$ must hold. Note that when $k_1 = k_2$ then n_1 and n_2 must be equal.

It is easy to prove that the weights of all vertices in $G_1^{n_1} \bowtie G_2^{n_2}$ are pairwise distinct. It is clear that $G_1^{n_1} \bowtie G_2^{n_2}$ is connected.

As an example, take K_4 and its antimagic labeling obtained using Roberts' Construction as shown in Figure 3 and recall the 4-regular antimagic graph G from Figure 1. Then we have the graph $K_4^2 \bowtie G^2$ and its antimagic labeling in Figure 4.

1	2	3
1	4	5
2	4	6
3	5	6

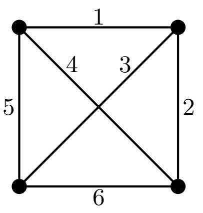


Fig. 3. A CSS L and corresponding graph $G(L) = K_4$ with antimagic edge labeling

	1	2	3
	1	4	5
	2	4	6
	3	7	8
5	6	9	10
7	11	12	13
8	12	14	15
9	13	16	17
10	14	16	18
11	15	17	18

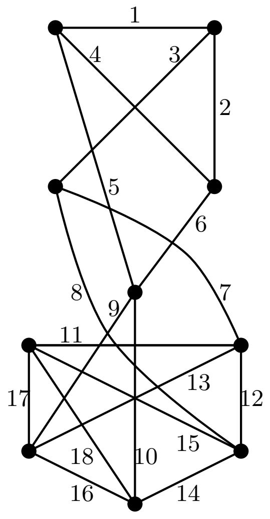


Fig. 4. An array (CSS) obtained by edge switching and corresponding graph $K_4^2 \bowtie G^2$ with antimagic edge labeling

By repeating this process, we can switch the edges of more such k_j-regular graphs G_j, $1 \leq j \leq m$, with $k_j \leq k_{j+1}$ and $n_j < k_j$, $1 \leq n_{j+1} \leq \lfloor \frac{k_{j+1}}{2} \rfloor$ and $k_j \leq k_j - n_j + n_{j+1} \leq k_{j+1} - n_{j+1} + n_j \leq k_{j+1}$, for $1 < j < m-1$, then $(((G_1^{n_1} \bowtie G_2^{n_2}) \bowtie G_3^{n_3}) \bowtie \ldots \bowtie G_{m-1}^{n_{m-1}}) \bowtie G_m^{n_m}$ is antimagic.

3.3 Splitting of Roberts' Construction

Let $G = (V, E, L)$ be a labeled graph with $|V| = p$ and $|E| = q$, where L is an array of edge labels using Roberts' Construction. We split L into two subarrays L_1 and L_2 (not rectangular) as follows.

1. Choose any integer q_1, $1 \leq q_1 < q$. Take L_1 to be the array containing edge labels from 1 to q_1 and L_2 containing edge labels from $q_1 + 1$ to q;
2. Replace each edge label e_i in L_1 with a new edge label $q_1 + 1 - e_i$;
3. Replace each edge label e_i in L_2 with a new edge label $e_i - q_1$.

The arrays L_1 and L_2 clearly define CSSs. By the construction of the array L_1, it is clear that the weight of each vertex (row) in the array is greater than the weight of the vertex (row) below. Similarly, for the array L_2, the weight of each vertex (row) in the array is less than the weight of the vertex (row) below.

Note that if L_1 or L_2 is not the array of edge labeling of K_2 (that is, $q_1 = 1$ or $q_1 = q - 1$), then we have two non-regular antimagic graphs corresponding to the edge labelings L_1 and L_2, otherwise we have only one non-regular antimagic graph.

For an example of the splitting of Roberts' Construction, recall the 4-regular antimagic graph G from Figure 1. If we choose $q_1 = 7$, then we obtain the CSS L_1 and its corresponding antimagic graph $G(L_1)$, shown in Figure 5, and L_2 and its corresponding antimagic graph $G(L_2)$, shown in Figure 6. Note that the solid dots and dark lines represent the vertices and edges of $G(L_j)$, $j = 1, 2$, respectively, while the dashed lines and circles show where edges and vertices of the original graph have been deleted.

$$\begin{array}{cccc}
7 & 6 & 5 & 4 \\
7 & 3 & 2 & 1 \\
6 & 2 & . & . \\
5 & 1 & . & . \\
4 & . & . & . \\
3 & . & . & .
\end{array}$$

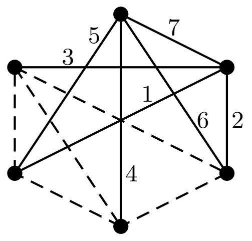


Fig. 5. The array (CSS) L_1 and graph $G(L_1)$ with antimagic edge labeling

$$\begin{array}{cccc}
. & . & . & . \\
. & . & . & . \\
. & . & 1 & 2 \\
. & . & 3 & 4 \\
. & 1 & 3 & 5 \\
. & 2 & 4 & 5
\end{array}$$

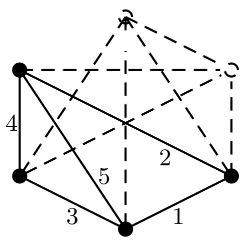


Fig. 6. The array (CSS) L_2 and graph $G(L_2)$ with antimagic edge labeling

3.4 Antimagic Graphs with Degrees k and $k - 1$

The next theorem gives a new construction for particular CSSs with two different block sizes. The corresponding graphs are antimagic.

Lemma 2. *The number of blocks in an (n)CSS with $n > 1$ is greater than the number of elements in the largest block.*

Proof. This is clear if all blocks have size 1. Otherwise, each element in a largest block B must appear again at least once to be completely separated from the other elements in B. This requires at least $|B|$ blocks in addition to block B.

Theorem 5. *Let $r \geq k + 1 \geq 3$ and let L be an $r \times k$ array with the bottom s places of the last column unused for some s with $r > s \geq 1$ and $rk - s$ even. Let $n = \frac{rk-s}{2}$.*

Entries for L can be found so that

1. *L represents an (n)CSS with $r - s$ blocks of size k and s blocks of size $k - 1$;*
2. *L consists of 2-elements;*
3. *If a and b are entries in the same column of L with a above b, then $a \geq b$.*

Proof. The proof is by induction on the number of rows r, beginning with $r = 3$ (so $k = n = 2$). There exists one such CSS, and it is the smallest satisfying the conditions.

For $r > 3$ use Roberts' Construction to fill the array L, starting at n and decreasing, but stopping when the first $r - s$ blocks have been filled. Let l be the last number to have been entered. Such an array is illustrated in Figure 7, where the shaded portion has been filled.

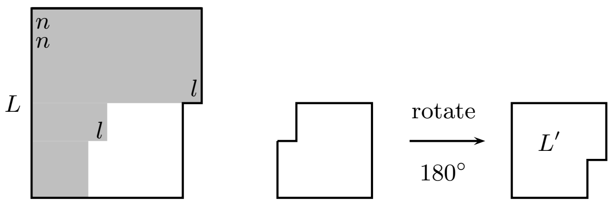


Fig. 7. A partially filled array; the grey portion contains numbers, the white portion is yet to be filled

Remove the unfilled portion and rotate it 180 degrees; call this smaller array L'. There are 3 cases:
(a) L' is a 0×0 array—there is nothing further to be done,
(b) L' is a rectangular $s \times k'$ array and so can be filled using Roberts' Construction from top to bottom with $1, 1, 2, 2, \ldots, l - 1, l - 1$,
(c) L' is an $s \times k'$ $(k' \geq 2)$ array L', which has some, but not all, of the bottom part of the last column removed. Roberts' Construction ensures that $s \neq 1$ and the previous cases cover $s = 2$. Hence $s \geq 3$. Roberts' Construction could

be continued to yield an array which would be a CSS, therefore, by Lemma 2 $s > k' + 1$. By induction L' can be filled, but work from 1 up to $l - 1$, so as to form a CSS consisting of 2-elements.

The construction ensures that if a and b are entries in L' with a above b, then $a \leq b$; when L' is rotated 180 degrees and placed in L we will have a above b implies $a \geq b$.

Each integer appears twice in L, so to show that complete separation is achieved it is only necessary to show that no two pairs of integers appear in the same two rows.

The numbers $n, n-1, \ldots, l$ are completely separated by Roberts' Construction. The numbers $l-1, \ldots, 2, 1$ are completely separated from each other by Roberts' Construction. Each $a \in \{l, l+1, \ldots, n\}$ appears at least once in the first $r - s$ rows of L, and each $b \in \{1, 2, \ldots, l-1\}$ appears twice after the first $r - s$ rows, so each pair a, b are completely separated. Hence L represents an (n)CSS.

Example 1. The recursive algorithm used in the proof of Theorem 5 is applied here to obtain a (10)CSS with two blocks of size 4 and four blocks of size 3:

$$L = \begin{array}{cccc} 10 & 9 & 8 & 7 \\ 10 & 6 & 5 & 4 \\ 9 & 5 & . & \\ 8 & 4 & . & \\ 7 & . & . & \\ 6 & . & . & \end{array} \text{. So } L' \text{ is } \begin{array}{cc} . & . \\ \cdot & \cdot \\ . & \\ . & \end{array} \rightarrow \begin{array}{cc} 1 & 2 \\ 1 & 3 \\ 2 & \\ 3 & \end{array} \text{. Hence } L \text{ becomes } \begin{array}{cccc} 10 & 9 & 8 & 7 \\ 10 & 6 & 5 & 4 \\ 9 & 5 & 3 & \\ 8 & 4 & 2 & \\ 7 & 3 & 1 & \\ 6 & 2 & 1 & \end{array} .$$

Corollary 1. *The weights of the blocks in the CSSs produced in Theorem 5 are distinct, and so the graph determined by the CSS is antimagic.*

Proof. The graph has $r - s$ vertices have degree k and s vertices have degree $k - 1$. The construction ensures that each element is no smaller than the one below, if there is one. As $k \geq 2$, and as it is only possible for one element in a row to equal the one below, each row sum is greater than the one below.

The proof of Theorem 5 gives a recursive algorithm for producing CSSs, and hence antimagic graphs:

```
1. Begin with $L$ as in Theorem 5, with all places marked as empty.
2. Repeat
       Case based on $L$
           $L$ is a $0 \times 0$ array : Halt.
           $L$ is rectangular : Use Roberts' Construction; Halt.
           Default : Fill $L$ until the large rows are full.
               $L :=$ unfilled portion rotated $180°$.
       Endcase
   Until $L$ filled.
```

The antimagic graph corresponding to the CSS in Example 1 is shown in Figure 8.

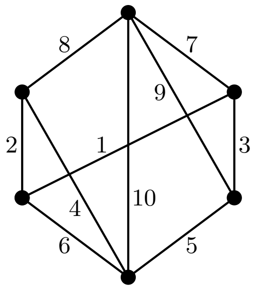


Fig. 8. The antimagic graph corresponding to the CSS in Example 1

Consider the degree sequence of a connected regular graph with r vertices: s of degree $k-1$ and $r-s$ of degree k with $r > s \geq 1$. We must have $r > k \geq 2$ and the sum of the degrees $s(k-1)+(r-s)k = rk-s$ even. By Theorem 5 there is an (r)CSS, which by Corollary 1 yields an antimagic graph with the original degree sequence. One can prove that the graph obtained is connected. Therefore, for any degree sequence corresponding to a connected non-regular graph whose vertices have degrees k and $k-1$, there exists a connected antimagic graph with that degree sequence.

Theorem 5 does not generalise to part of two or more columns being removed. For example, it is not possible to place integers in the array with three rows of size 4 and two rows of size 2 so as to form a CSS. Other methods will be needed to prove that, in general, for the degree sequence of a graph with vertices of degrees k and k' there is an antimagic graph with that degree sequence.

The work presented in this paper is a small step towards proving a weak version of the Hartsfield and Ringel conjecture:

Conjecture 2. For every degree sequence of connected graph, except for $1, 1$, there is a connected antimagic graph with that degree sequence.

References

1. Alon, A., Kaplan, G., Lev, A., Rodity, Y., Yuster, R.: Dense graphs are antimagic. J. Graph Theory 47(4), 297–309 (2004)
2. Bodendiek, R., Walther, G.: On number theoretical methods in graph labelings. Res. Exp. Math. 21, 3–25 (1995)
3. Chawathe, P.D., Krishna, V.: Antimagic labelings of complete m-ary trees. In: Number Theory and Discrete Mathematics. Trends Math., pp. 77–80. Birkhäuser, Basel (2002)
4. Dickson, T.J.: On a problem concerning separating systems of a finite set. J. Combinatorial Theory 7, 191–196 (1969)
5. Gallian, J.A.: A dynamic survey of graph labeling. Electron. J. Combin. 19(♯DS6) (2012)
6. Hartsfield, N., Ringel, G.: Pearls in Graph Theory: A Comprehensive Introduction. Academic Press Inc., Boston (1990)

7. Hefetz, D.: Anti-magic graphs via the combinatorial nullstellensatz. J. Graph Theory 50(4), 263–272 (2005)
8. Kotzig, A., Rosa, A.: Magic valuations of complete graphs. Publ. Centre de Recherches Mathématiques, Université de Montréal 175, CRM–175 (1972)
9. Kotzig, A.: On certain vertex-valuations of finite graphs. Utilitas Math. 4, 261–290 (1973)
10. Kotzig, A., Rosa, A.: Magic valuations of finite graphs. Canad. Math. Bull. 13, 451–461 (1970)
11. MacDougall, J.A., Miller, M., Slamin, Wallis, W.D.: Vertex magic total labelings of graphs. Utilitas Math. 61, 3–21 (2002)
12. MacDougall, J.A., Miller, M., Wallis, W.D.: Vertex magic total labelings of wheels and related graphs. Utilitas Math. 62, 175–183 (2002)
13. Phanalasy, O.: Covering Separating Sytems. Master's thesis, Northern Territory University, Australia (1999)
14. Phanalasy, O., Miller, M., Rylands, L., Lieby, P.: On a relationship between completely separating systems and antimagic labeling of regular graphs. In: Iliopoulos, C.S., Smyth, W.F. (eds.) IWOCA 2010. LNCS, vol. 6460, pp. 238–241. Springer, Heidelberg (2011)
15. Ramsay, C., Roberts, I.T., Ruskey, F.: Completely separating systems of k-sets. Discrete Math. 183(1–3), 265–275 (1998)
16. Roberts, I.T.: Extremal Problems and Designs on Finite Sets. Ph.D. thesis, Curtin University of Technology, Australia (1999)
17. Sedláček, J.: Problem 27. In: Theory of Graphs and its Applications (Proc. Sympos. Smolenice, 1963), pp. 163–164. Publ. House Czechoslovak Acad. Sci, Prague (1964)
18. Stewart, B.M.: Magic graphs. Canad. J. Math. 18, 1031–1056 (1966)
19. Wallis, W.D.: Magic graphs. Birkhäuser Boston Inc., Boston (2001)

Universal Cycles for Weight-Range Binary Strings

Joe Sawada[1,⋆], Aaron Williams[2,⋆⋆], and Dennis Wong[1]

[1] School of Computer Science, University of Guelph, Canada
{jsawada,cwong}@uoguelph.ca

[2] Department of Mathematics and Statistics, McGill University, Canada
haron@uvic.ca

Abstract. We present an efficient universal cycle construction for the set of binary strings of length n with weight (number of 1s) in the range $c, c+1, \ldots, d$ where $0 \leq c < d \leq n$. The construction is based on a simple lemma for gluing universal cycles together, which can be implemented to generate each character in constant amortized time using $O(n)$ space. The Gluing lemma can also be applied to construct universal cycles for other combinatorial objects including passwords and labeled graphs.

1 Introduction

Let $\mathbf{B}(n)$ denote the set of all binary strings of length n. A *universal cycle* for a set $\mathbf{S}$ is a cyclic sequence $u_1 u_2 \cdots u_{|\mathbf{S}|}$ where each substring of length n corresponds to a unique object in $\mathbf{S}$. When $\mathbf{S} = \mathbf{B}(n)$ these sequences are commonly known as *de Bruijn sequences* [6, 7, 14] and efficient constructions are well known [9, 10, 17]. For example, the cyclic sequence 0000100110101111 is a universal cycle (de Bruijn sequence) for $\mathbf{B}(4)$; the 16 unique substrings of length 4 when the sequence is considered cyclicly are:

$$0000, 0001, 0010, 0100, 1001, 0011, 0110, 1101, 1010, 0101, 1011, 0111, 1111, 1110, 1100, 1000.$$

Universal cycles have been studied for a variety of combinatorial objects including permutations, partitions, subsets, labeled graphs, various functions, and passwords [1, 3–5, 12, 13, 15, 16, 18, 21]. In this paper we focus on the set $\mathbf{B}_c^d(n)$, which denotes the subset of $\mathbf{B}(n)$ containing strings with *weight* (number of 1s) in the range $c, c+1, \ldots, d$, or in other words, the subset of $\mathbf{B}(n)$ containing strings with *weight-range* from c to d. We refer to universal cycles as *dual-weight universal cycles* when $c = d - 1$. We also say a weight-range is *even* if $|\{c, c+1, \ldots, d\}|$ is even, and we say a weight-range is *odd* if $|\{c, c+1, \ldots, d\}|$ is odd. As an example, $\mathbf{B}_2^3(4) = \{0011, 0101, 0110, 1001, 1010, 1100, 0111, 1011, 1101, 1110\}$ and a universal cycle for this set is 0011101011 which has even weight-range. Using standard techniques, it can be shown that universal cycles exist for all $\mathbf{B}_c^d(n)$ where $0 \leq c < d \leq n$ (when $c = d$, they exist only when $c \in \{0, 1, n-1, n\}$). However, finding efficient constructions remains a difficult problem.

⋆ Research supported by NSERC.

⋆⋆ Research supported in part by the NSERC Accelerator and Discovery Programmes, and a basic research grant from ONR.

© Springer-Verlag Berlin Heidelberg 2013

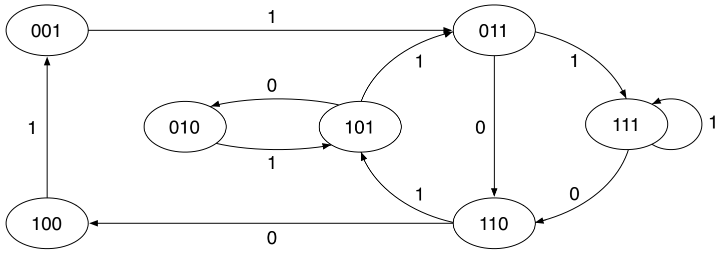


Fig. 1. The de Bruijn graph $G(\mathbf{B}_2^4(4))$.

In this paper, a universal cycle has an *efficient construction* if each successive symbol of the sequence can be generated in constant amortized time (CAT) while using a polynomial amount of space with respect to n. Some special cases on the construction of universal cycles for $\mathbf{B}_c^d(n)$ have been previously studied:

- if $c = d - 1$, then an efficient construction is known [20],
- if $c = 0$ or $d = n$, then an efficient construction is known [24],
- if the weight-range is even, then a polynomial construction is known [25, 26].

This paper provides the first efficient construction for universal cycles of $\mathbf{B}_c^d(n)$ for all $0 \leq c < d \leq n$. By applying the efficient construction known for the case when $c = d - 1$ [20], a universal cycle for $\mathbf{B}_c^d(n)$ can be constructed in constant amortized time using $O(n)$ space.

The rest of this paper is presented as follows. In Section 2 we provide a simple proof for the existence of universal cycle for $\mathbf{B}_c^d(n)$ where $0 \leq c < d \leq n$; an alternate proof is given in [2]. In Section 3, we present a generic result that considers when two universal cycles can be "glued together" to obtain a new universal cycle. We then apply this result in Section 4 and Section 5 to provide an efficient universal cycle construction for $\mathbf{B}_c^d(n)$. We conclude with Section 6, where we apply our gluing technique to universal cycles for other combinatorial objects, including passwords and labeled graphs.

2 Universal Cycle Existence for $\mathbf{B}_d^c(n)$

The *de Bruijn graph* $G(\mathbf{S})$ for a set of length n strings $\mathbf{S}$ is a directed graph whose vertex set consists of the length $n-1$ strings that are a prefix or a suffix of the strings in $\mathbf{S}$. For each string $b_1 b_2 \cdots b_n \in \mathbf{S}$ there is an edge labeled b_n that is directed from the vertex $b_1 b_2 \cdots b_{n-1}$ to the vertex $b_2 b_3 \cdots b_n$. Thus, the graph has $|\mathbf{S}|$ edges. As an example, the de Bruijn graph $G(\mathbf{B}_2^4(4))$ is illustrated in Fig. 1.

In this article, a *cycle* in a directed graph $G = (V, E)$ is a sequence $v_1, v_2, \ldots, v_j, v_1$ where $v_i \in V$ and $(v_i, v_{i+1}) \in E$. A directed graph is said to be *Eulerian* if it contains an Euler cycle, that is, a cycle that includes each edge exactly once. It is well known that $\mathbf{S}$ admits a universal cycle if and only if $G(\mathbf{S})$ is Eulerian. If $G(\mathbf{S})$ is Eulerian, then a universal cycle is produced by traversing an Euler cycle and outputting the edge labels.

However, in practice, such a method for producing a universal cycle is often impractical due to the size of the graph that must be stored in memory. For example, because the number of edges for the de Bruijn graph $G(\mathbf{B}(n))$ is equal to 2^n, the memory required to store all edges of the de Bruijn graph $G(\mathbf{B}(n))$ is $\Omega(2^n)$.

A directed graph is said to be *balanced* if the in-degree of each vertex is the same as its out-degree. It is *strongly connected* if there is a directed path between every pair of vertices. The following result is well-known, and appears in many references such as [19]:

Lemma 1. *A directed graph is Eulerian if and only if it is balanced and strongly connected.*

Theorem 1. $G(\mathbf{B}_c^d(n))$ *is Eulerian for* $0 \leq c < d \leq n$.

Proof. We prove that $G(\mathbf{B}_c^d(n))$ is Eulerian by showing that it is balanced and strongly connected.

Balanced: The vertex set of $G(\mathbf{B}_c^d(n))$ contains all strings of length $n-1$ with weight in the range $c-1, c, \ldots, d$. Each vertex with weight $c-1$ has one incoming edge and one outgoing edge, each labeled 1. Each vertex with weight d has one incoming edge and one outgoing edge, each labeled 0. All other vertices have in-degree and out-degree equal to 2.

Strongly Connected: We apply induction on the size of the weight-range $c, c+1, \ldots, d$. The base case when $c = d-1$ is proved in Theorem 2.4 of [20]. For the inductive step assume that $G(\mathbf{B}_c^{d-1}(n))$ is strongly connected for $0 \leq c < d-1$, and consider $G(\mathbf{B}_c^d(n))$. Observe:

- the vertex set of $G(\mathbf{B}_c^d(n))$ is equal to the union of the vertex sets of $G(\mathbf{B}_c^{d-1}(n))$ and $G(\mathbf{B}_{d-1}^d(n))$,
- the intersection of the vertex sets for $G(\mathbf{B}_c^{d-1}(n))$ and $G(\mathbf{B}_{d-1}^d(n))$ is non-empty,
- the edge sets of $G(\mathbf{B}_c^{d-1}(n))$ and $G(\mathbf{B}_{d-1}^d(n))$ are both subsets of the edge set for $G(\mathbf{B}_c^d(n))$.

Thus, since both $G(\mathbf{B}_c^{d-1}(n))$ and $G(\mathbf{B}_{d-1}^d(n))$ are strongly connected (inductive hypothesis and base case), there will be a directed path between any two vertices in $G(\mathbf{B}_c^d(n))$. $\square$

3 Gluing Universal Cycles

In this section we consider concatenating two universal cycles together to obtain a new universal cycle. If a directed graph is Eulerian, then an Euler cycle of the graph can be obtained by Hierholzer's algorithm [11, 19]. Hierholzer's approach is to construct an Euler cycle by exhaustively concatenating edge-disjoint cycles that share a common vertex. The algorithm repeatedly applies the following observation to produce an Euler cycle.

Observation 2. *Let* $G = (V, E)$ *and* $H = (V', E')$ *be two directed Eulerian graphs such that* $V \cap V' \neq \emptyset$ *and* $E \cap E' = \emptyset$. *Let* $C_G = u_1, u_2, \ldots, u_j, u_1$ *and* $C_H = v_1, v_2, \ldots, v_k, v_1$ *denote Euler cycles in G and H respectively such that* $u_1 = v_1$. *Then the concatenation of the two cycles* $C_{GH} = u_1, \ldots, u_j, v_1, \ldots, v_k, v_1$ *is an Euler cycle for* $G \cup H$.

As mentioned in Section 2, each universal cycle for a set $\mathbf{S}$ corresponds to an Euler cycle of its de Bruijn graph $G(\mathbf{S})$. Thus by Observation 2, universal cycles for two sets $\mathbf{S}_1$ and $\mathbf{S}_2$ can be joined together to form a new universal cycle for $\mathbf{S}_1 \cup \mathbf{S}_2$ if $G(\mathbf{S}_1)$ and $G(\mathbf{S}_2)$ are edge-disjoint and share a common vertex, or in other words, $\mathbf{S}_1$ and $\mathbf{S}_2$ are disjoint and have elements that share a length $n-1$ prefix or a length $n-1$ suffix. As an example, consider the following two universal cycles:

- universal cycle for $\mathbf{B}_1^2(5)$: $00\underline{0011}000101001$,
- universal cycle for $\mathbf{B}_3^4(5)$: $\underline{0011}11011010111$.

The de Bruijn graphs $G(\mathbf{B}_1^2(5))$ and $G(\mathbf{B}_3^4(5))$ are edge-disjoint and share a common vertex $\alpha = 0011$. Since the universal cycles are cyclic they can be re-written as $\underline{0011}00010100100$ and $\underline{0011}11011010111$ respectively. By gluing these two strings together, observe that we obtain a universal cycle for $\mathbf{B}_1^4(5) = \mathbf{B}_1^2(5) \cup \mathbf{B}_3^4(5)$:

$$\underline{0011}00010100100\underline{0011}11011010111.$$

This example illustrates the following result which follows directly from Observation 2.

Lemma 3 (The Gluing lemma). *Let U_1 and U_2 be universal cycles for the sets of length n strings $\mathbf{S_1}$ and $\mathbf{S_2}$, where $\mathbf{S_1} \cap \mathbf{S_2} = \emptyset$ and the length $n-1$ prefixes of U_1 and U_2 are the same. Then the concatenated string $U_1 \cdot U_2$ is a universal cycle for $\mathbf{S_1} \cup \mathbf{S_2}$.*

4 Universal Cycle Construction for $\mathrm{B}_c^d(n)$

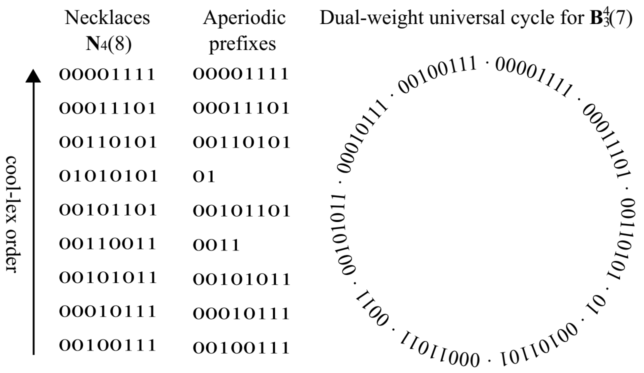
As mentioned earlier, there exists an efficient universal cycle construction for $\mathbf{B}_{d-1}^d(n)$ [20]. We use the Gluing lemma to create a universal cycle for binary strings with an even weight-range by joining these dual-weight universal cycles. To create universal cycles for binary strings with an odd weight-range, we also have to glue in individual necklaces which are defined in the following subsection.

4.1 Preliminary Definitions and Notations

A *necklace* is the lexicographically smallest string in an equivalence class of strings under rotation. The *aperiodic prefix* of a string α, denoted as $ap(\alpha)$, is its shortest prefix whose repeated concatenation yields α. That is, the aperiodic prefix of $\alpha = a_1a_2\cdots a_n$ is the shortest prefix $ap(\alpha) = a_1a_2\cdots a_p$ such that $(ap(\alpha))^{\frac{n}{p}} = \alpha$, where exponentiation denotes repeated concatenation. For example, when $\alpha = 001001001$, $ap(\alpha) = 001$. A string is a *prenecklace* if it is the prefix of some necklace. A string α is *aperiodic* if $ap(\alpha) = \alpha$. Aperiodic necklaces are also known as *Lyndon words*. Let the set of length n binary prenecklaces, necklaces and Lyndon words with weight w be denoted by $\mathbf{P}(n,w)$, $\mathbf{N}(n,w)$ and $\mathbf{L}(n,w)$ respectively. For example:

- $\mathbf{P}(6,4) = \{001111, 010111, 011011, 011101, 011110\}$,
- $\mathbf{N}(6,4) = \{001111, 010111, 011011\}$,
- $\mathbf{L}(6,4) = \{001111, 010111\}$.

Observe that the prenecklaces 011101 and 011110 are prefixes of the necklaces 01110111 and 0111101111 respectively so they are in $\mathbf{P}(6,4)$.



Necklaces $\mathbf{N}_4(8)$	Aperiodic prefixes
00001111	00001111
00011101	00011101
00110101	00110101
01010101	01
00101101	00101101
00110011	0011
00101011	00101011
00010111	00010111
00100111	00100111

Fig. 2. Concatenating the aperiodic prefixes of $\mathbf{N}_4(8)$ in reverse cool-lex order to create a dual-weight universal cycle for $\mathbf{B}_3^4(7)$.

4.2 Even Weight-Range

Suppose we want to construct a universal cycle for $\mathbf{B}_{d-3}^{d}(n)$ from universal cycles for $\mathbf{B}_{d-3}^{d-2}(n)$ and $\mathbf{B}_{d-1}^{d}(n)$. Observe that $\mathbf{B}_{d-3}^{d-2}(n)$ and $\mathbf{B}_{d-1}^{d}(n)$ are disjoint, and their universal cycles share common length $n-1$ substrings with weight $d-2$. Thus, we can apply the Gluing lemma to construct a universal cycle for $\mathbf{B}_{d-3}^{d}(n)$. We can then repeatedly apply the Gluing lemma on the resulting universal cycle and dual-weight universal cycles of lower weight-ranges to obtain a universal cycle of an even weight-range. However, the difficult task remains: How can we produce the glued universal cycle efficiently, that is, without scanning for common substrings of length $n-1$?

To find an efficient construction, we must revisit the efficient construction of universal cycles for $\mathbf{B}_{d-1}^{d}(n)$, which we will refer to as UC_{d-1}^{d} in this article:

1. List the necklaces of length $n+1$ and weight d in reverse *cool-lex order* [22],
2. Append the aperiodic prefixes of the necklaces following the order to create UC_{d-1}^{d}.

As an example, Fig. 2 demonstrates a dual-weight universal cycle for $\mathbf{B}_3^4(7)$ constructed by concatenating the aperiodic prefixes of $\mathbf{N}_4(8)$ in reverse cool-lex order. Using this construction, we can find a common length $n-1$ substring α of the universal cycles $UC_{d-3}^{d-2}(n)$ and $UC_{d-1}^{d}(n)$ by the following lemma.

Lemma 4. [20] *The first necklace in reverse cool-lex order for* $\mathbf{B}_d(n+1)$ *is* $0^{n-d+1}1^d$.

Applying this lemma, the first $n+1$ characters in $UC_{d-3}^{d-2}(n)$ and $UC_{d-1}^{d}(n)$ are $0^{n-d+3}1^{d-2}$ and $0^{n-d+1}1^d$ respectively. Thus, if we rotate $UC_{d-3}^{d-2}(n)$ to the left by 2 characters, then the first $n-1$ characters of both cycles will be $0^{n-d+1}1^{d-2}$.

Let $VC_{d-1}^{d}(n)$ denote the sequence $UC_{d-1}^{d}(n)$ with the first 2 characters removed. The following recursive formula provides a construction of the universal cycle $UE_c^d(n)$ for $\mathbf{B}_c^d(n)$ where the weight-range is even.

$$UE_c^d(n) = \begin{cases} UC_{d-1}^{d}(n) & \text{if } c = d-1; \\ UC_{c+2}^{d}(n) \cdot VC_c^{c+1}(n) \cdot 00 & \text{if } c < d-1. \end{cases}$$

We obtain the following formula by expanding the recursive function.

$$UE^d_c(n) = UC^d_{d-1}(n) \cdot VC^{d-2}_{d-3}(n) \cdot VC^{d-4}_{d-5}(n) \cdots VC^{c+1}_{c}(n) \cdot 0^{d-c-1}.$$

Theorem 2. *$UE^d_c(n)$ is a universal cycle for $\mathbf{B}^d_c(n)$ when $0 \leq c < d \leq n$ and the weight-range is even.*

The example discussed in Section 3 shows how $UE^4_1(5)$ is constructed from $UC^4_3(5)$ and $UC^2_1(5)$. Note that the universal cycle $UE^d_c(n)$ is different from those created in [25, 26].

Theorem 3. *A universal cycle $UE^d_c(n)$ for $\mathbf{B}^d_c(n)$ can be constructed in constant amortized time using $O(n)$ space when weight-range is even and* $0 \leq c < d \leq n$.

Proof. Since the universal cycle UC^d_{d-1} can be constructed in constant amortized time using $O(n)$ space by the algorithm described in [20], and VC^d_{d-1} can easily be obtained by constructing UC^d_{d-1} and removing its first 2 characters, therefore $UE^d_c(n)$ can be constructed in constant amortized time using $O(n)$ space when weight-range is even. □

This is the first known efficient construction of even weight-range universal cycles, since the previous constructions discussed in [25, 26] were not accompanied by an efficient algorithm.

4.3 Incrementing the Weight-Range (Odd Weight-Range)

In this section, we consider extending the weight-range of a universal cycle for $\mathbf{B}^{d-1}_c(n)$ into a universal cycle for $\mathbf{B}^d_c(n)$. We refer this process as *incrementing* the universal cycle's weight range. The process of incrementing the universal cycle's weight range allows us to extend an even weight-range universal cycle to an odd weight-range universal cycle.

Let $\mathbf{Neck}(\beta)$ denote the set of strings rotationally equivalent to β. We partition the strings in $\mathbf{B}_d(n)$ into their necklace equivalence classes such that $\mathbf{B}_d(n) = \mathbf{Neck}(n_1) \cup \mathbf{Neck}(n_2) \cup \cdots \cup \mathbf{Neck}(n_{|\mathbf{N}_d(n)|})$ where $n_j \in \mathbf{N}_d(n)$. For example, $\mathbf{B}_3(6)$ can be partitioned into four subsets $\mathbf{B}_3(6) = \mathbf{Neck}(000111) \cup \mathbf{Neck}(001011) \cup \mathbf{Neck}(001101) \cup \mathbf{Neck}(010101)$ with elements of each set listed as follows:

- ▷ $\mathbf{Neck}(000111) = \{000111, 001110, 011100, 111000, 110001, 100011\}$,
- ▷ $\mathbf{Neck}(001011) = \{001011, 010110, 101100, 011001, 110010, 100101\}$,
- ▷ $\mathbf{Neck}(001101) = \{001101, 011010, 110100, 101001, 010011, 100110\}$, and
- ▷ $\mathbf{Neck}(010101) = \{010101, 101010\}$.

Observe that the de Bruijn graph $G(\mathbf{Neck}(n_j))$ forms a simple cycle with concatenation of its edge labels correspond to $ap(n_j)$, which is a universal cycle for $\mathbf{Neck}(n_j)$. As an example, Fig. 3 illustrates the de Bruijn graphs for the four necklace equivalence classes that make up $\mathbf{B}_3(6)$. The concatenation of edge labels of the cycles are 000111, 001011, 001101 and 01, which correspond to the universal cycles for $\mathbf{Neck}(000111)$, $\mathbf{Neck}(001011)$, $\mathbf{Neck}(001101)$ and $\mathbf{Neck}(010101)$ respectively. Notice that there exists universal cycles for $\mathbf{Neck}(n_j)$ that have length less than n, which happens when

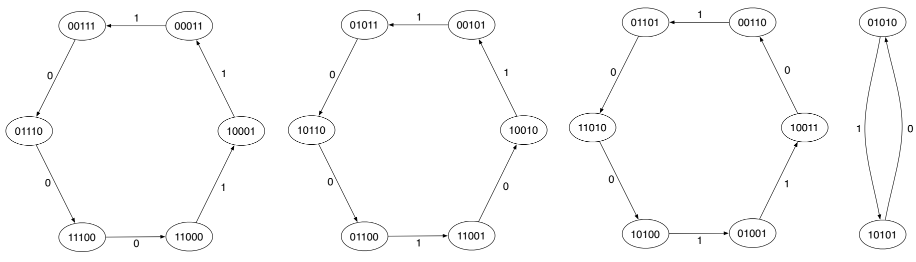


Fig. 3. Illustrating the de Bruijn graph corresponding to the universal cycles of the 4 necklace equivalence classes $\mathbf{Neck}(000111)$, $\mathbf{Neck}(001011)$, $\mathbf{Neck}(001101)$ and $\mathbf{Neck}(010101)$ that make up $\mathbf{B}_3(6)$.

n_j are periodic. The length $n-1$ prefixes of these universal cycles are the length $n-1$ prefixes of the corresponding necklace. For example, 01010 is the length 5 prefix of the universal cycle 01 when $n=6$, since the universal cycle is traversed repeatedly.

Observe that $\mathbf{B}_c^{d-1}(n)$ and $\mathbf{Neck}(n_j)$ are disjoint, and we can rotate a universal cycles for $\mathbf{B}_c^{d-1}(n)$ such that its length $n-1$ prefix is equal that of $\mathbf{Neck}(n_j)$. We can then apply Lemma 3 (the Gluing lemma) to repeatedly concatenate universal cycles for each necklace equivalence class $\mathbf{Neck}(n_j)$ with the universal cycle for $\mathbf{B}_c^{d-1}(n)$. For example, consider the universal cycles for $\mathbf{Neck}(000111)$, $\mathbf{Neck}(001011)$, $\mathbf{Neck}(001101)$, $\mathbf{Neck}(010101)$ and $\mathbf{B}_1^2(6)$:

- universal cycle for $\mathbf{Neck}(000111)$: 000111,
- universal cycle for $\mathbf{Neck}(001101)$: 001101,
- universal cycle for $\mathbf{Neck}(001011)$: 001011,
- universal cycle for $\mathbf{Neck}(010101)$: 01,
- universal cycle for $\mathbf{B}_1^2(6)$: 0001100001010001001000.

By repeatedly applying the Gluing lemma, we obtain a universal cycle for $\mathbf{B}_1^3(6)$ as follows:

1. Glue universal cycles for $\mathbf{Neck}(000111)$ and $\mathbf{B}_1^2(6)$:
   - $\underline{00011}1\underline{00011}0000101000100100$, which is equivalent to the rotation,
   - $\underline{00110}00010100010010000001110$.
2. Glue universal cycles for $\mathbf{Neck}(001101)$ and $\mathbf{B}_1^2(6) \cup \mathbf{Neck}(000111)$:
   - $\underline{00110}1\underline{00110}00010100010010000001110$, which is equivalent to the rotation,
   - $\underline{00101}0001001000000111000110100 1100$.
3. Glue universal cycles for $\mathbf{Neck}(001011)$ and $\mathbf{B}_1^2(6) \cup \mathbf{Neck}(000111) \cup \mathbf{Neck}$ (001101):
   - $\underline{00101}1\underline{00101}00010010000001110001101001100$, which is equivalent to the rotation,
   - $\underline{01010}0010010000001110001101001100001 0110$.
4. Glue universal cycles for $\mathbf{Neck}(010101)$ and $\mathbf{B}_1^2(6) \cup \mathbf{Neck}(000111) \cup \mathbf{Neck}$ $(001101) \cup \mathbf{Neck}(001011)$:
   - $\underline{0101010}0010010000001110001101001100001 0110$.

Since $\mathbf{B}_1^2(6) \cup \mathbf{Neck}(000111) \cup \mathbf{Neck}(001101) \cup \mathbf{Neck}(001011) \cup \mathbf{Neck}(010101) = \mathbf{B}_1^3(6)$, we obtain a universal cycle for $\mathbf{B}_1^3(6)$. The order of inserting the universal cycles for $\mathbf{Neck}(n_j)$ does not affect the final universal cycle.

A *linear universal string* of a universal cycle is obtained by appending the first $n-1$ characters to its end. For example, a linear universal string for the universal cycle 0000100110101111 for $\mathbf{B}(4)$ is $\underline{000}0100110101111\underline{000}$. Let $u_1 u_2 \cdots u_{|UD_c^{d-1}|+n-1}$ denote the linear universal string from UD_c^{d-1}, where UD_c^{d-1} is a universal cycle for $\mathbf{B}_c^{d-1}(n)$. The linear universal string contains each element of $\mathbf{B}_c^{d-1}$ exactly once as a substring. Observe that if $\alpha \cdot 1 \in \mathbf{N}_d(n)$, then the string $\alpha \cdot 0$ exists as a substring of UD_c^{d-1}. Thus, we can increment the weight-range of UD_c^{d-1} and output a universal cycle for $\mathbf{B}_c^d(n)$ as follows:

```
for $s$ from $1$ to $|UD_c^{d-1}|$ do
    $\alpha = u_s u_{s+1} \cdots u_{s+n-1}$
    if $\alpha \cdot 1 \in \mathbf{N}_d(n)$ then
        Print($ap(\alpha \cdot 1)$)
    Print($u_s$)
```

Let UD_c^d denote the output string that results from this construction.

Theorem 4. *UD_c^d is a universal cycle for $\mathbf{B}_c^d(n)$ when $0 \leq c < d \leq n$.*

Proof. The string $ap(\alpha \cdot 1)$ is a universal cycle for $\mathbf{Neck}(\alpha \cdot 1)$. Since $\mathbf{B}_c^{d-1}(n)$ and $\mathbf{Neck}(\alpha \cdot 1)$ are disjoint and have the same length $n-1$ prefix, that is α, by the Gluing lemma the construction exhaustively concatenates UD_c^{d-1} and universal cycles for each necklace equivalence class $\mathbf{Neck}(n_j)$, where $n_j \in \mathbf{N}_d(n)$. The resulting string UD_c^d is a universal cycle for the set $\mathbf{B}_c^{d-1}(n) \cup \mathbf{Neck}(n_1) \cup \mathbf{Neck}(n_2) \cup \cdots \cup \mathbf{Neck}(n_{|\mathbf{N}_d(n)|})$, that is $\mathbf{B}_c^d(n)$. □

5 Implementation

In this section, we efficiently increment the weight-range of a universal cycle for $\mathbf{B}_c^{d-1}(n)$ into a universal cycle for $\mathbf{B}_c^d(n)$. We assume that there is an efficient algorithm that outputs a universal cycle for $\mathbf{B}_c^{d-1}(n)$ one character at a time. We buffer this output into a sliding window, and examine it to determine if any additional characters need to be output. We first describe how this process works, and then we describe how to make the process efficient.

5.1 A Simple Algorithm: SimpleIncrement()

The construction SimpleIncrement() follows the approach in Section 4.3. The algorithm reads each character from a linear universal string $u_1 u_2 \cdots u_{|UD_c^{d-1}|+n-1}$. It examines the sliding window $\alpha = u_s u_{s+1} \cdots u_t$ of size $n-1$ and inserts $ap(\alpha \cdot 1)$ if $\alpha \cdot 1 \in \mathbf{N}_d(n)$. The weight of α is maintained by the variable w. The weight w change by at most one between successive iterations which can be maintained in constant time.

To examine if $\alpha \cdot 1 \in \mathbf{N}_d(n)$, we apply Duval's algorithm [8] which returns 0 if $\alpha \cdot 1$ is not a necklace, or otherwise returns the length of $ap(\alpha \cdot 1)$. The length of $ap(\alpha \cdot 1)$ is maintained by the variable p. The algorithm runs in $O(n)$ time per character.

```
function AperiodicNecklacePrefix($x_1 x_2 \cdots x_n$)
1: $p \leftarrow 1$
2: for $i$ from 2 to $n$ do
3:     if $x_{i-p} < x_i$ then $p \leftarrow i$
4:     else if $x_{i-p} > x_i$ then return 0
5: if $n$ mod $p = 0$ then return $p$
6: else return 0
```

```
procedure SimpleIncrement()
1: $s \leftarrow 1$
2: for $s$ from 1 to $|UD_c^{d-1}|$ do
3:     $t \leftarrow s + n - 1$
4:     $w \leftarrow$ Weight($u_s u_{s+1} \cdots u_t$)
5:     $p \leftarrow$ AperiodicNecklacePrefix($u_s u_{s+1} \cdots u_t 1$)
6:     if $p > 0$ and $w = d - 1$ then
7:         Print($u_s u_{s+1} \cdots u_{s+p-2} 1$) // Insertion of $ap(\alpha \cdot 1)$
8:     Print($u_s$)
9:     $s \leftarrow s + 1$
```

Fig. 4. Pseudocode of SimpleIncrement().

Pseudocode that produces $UD_c^d(n)$ is shown in Fig. 4. The initial call is SimpleIncrement(). The procedure calls the function AperiodicNecklacePrefix($x_1 \cdots x_n$) to examine if $\alpha \cdot 1 \in \mathbf{N}_d(n)$, which is an implementation of Duval's algorithm. Thus, each time we read a character from an input linear universal string we need $O(n)$ amount of work to examine $\alpha \cdot 1$ for its aperiodic prefix. The sliding window of size $n - 1$ can be implemented using a circular array that requires a constant amount of computation to update using $O(n)$ space.

Thus, in addition to the time and space required to produce an input linear universal string for $\mathbf{B}_c^{d-1}(n)$, the algorithm SimpleIncrement() uses an additional $O(n)$ amount of work per character and $O(n)$ space to construct a universal cycle $UD_c^d(n)$ for $\mathbf{B}_c^d(n)$. From Theorem 3, because we can construct even weight-range universal cycle in constant amortized time using $O(n)$ space, we arrive the following theorem.

Theorem 5. *A universal cycle $UD_c^d(n)$ for $\mathbf{B}_c^d(n)$ can be constructed in $O(n)$ amortized time using $O(n)$ space for any weight-range where $0 \leq c < d \leq n$.*

5.2 Extending SimpleIncrement() to CAT

The major overhead of SimpleIncrement() in runtime comes from the function AperiodicPrefix($x_1 \cdots x_n$) that examines the sliding window for its aperiodic prefix using $O(n)$ amount of work per character. To efficiently locate the position to insert the aperiodic prefixes, we instead maintain a sliding window $\beta = u_s u_{s+1} \cdots u_t$ of variable size.

Pseudocode of the efficient construction is shown in Fig. 5. The initial call is FastIncrement(). The function Update(k, w) scans the string $u_k u_{k+1} \cdots a_t$ to update s to s' such that $u_{s'} u_{s'+1} \cdots u_t$ is a prenecklace and p is the length of $ap(u_{s'} u_{s'+1} \cdots u_t)$. The algorithm is summerized into three stages as follows:

Glue Universal Cycles: We insert $ap(u_s u_{s+1} \cdots u_{t-1} \cdot 1)$ before the position s if $u_s u_{s+1} \cdots u_{t-1} \cdot 1 \in \mathbf{N}_d(n)$. The string $u_s u_{s+1} \cdots u_{t-1} \cdot 1 \in \mathbf{N}_d(n)$ if $t - s + 1 = n$, $w = d - 1$ and $u_t = 0$.

Maintain Prenecklace: We maintain the variables s and p such that β is a prenecklace and p is the length of $ap(\beta)$. There are a few possible cases here:

```
function Update(int $k$, int $w$)
1: $s' \leftarrow k; p \leftarrow 1; w' \leftarrow w$
2: for $i$ from $k+1$ to $t$ do
3:     if $u_{i-p} < u_i$ then $p \leftarrow i - s' + 1$
4:     else if $u_{i-p} > u_i$ then
5:         $s' \leftarrow s' + \lfloor \frac{i-s'}{p} \rfloor \cdot p$
6:         $p \leftarrow 1$
7:         for $j$ from $s'+1$ to $i$ do
8:             if $u_{j-p} < u_j$ then $p \leftarrow j - s' + 1$
9: // Update weight $w$
10: for $i$ from $s$ to $s'-1$ do
11:     if $u_i = 1$ then $w' \leftarrow w - 1$
12: return $(s', p, w')$
```

```
procedure FastIncrement()
1: $p \leftarrow 1; w \leftarrow 0; s \leftarrow 1$
2: for $t$ from $1$ to $|UD_c^{d-1}| + n - 1$ do
3:     $w \leftarrow w + u_t$
4:     // Glue universal cycles
5:     if $t - s + 1 = n$ and $w = d - 1$ and $u_t = 0$ then
6:         if $u_{t-p} < 1$ then Print($u_s u_{s+1} \cdots u_{t-1} 1$)
7:         else Print($u_s u_{s+1} \cdots u_{s+p-1}$)
8:     // Maintain prenecklace
9:     if $u_{t-p} < u_t$ then $p \leftarrow t - s + 1$
10:    else if $u_{t-p} > u_t$ then
11:        $(s', p, w) \leftarrow$ Update($s + \lfloor \frac{t-s}{p} \rfloor \cdot p$, $w$)
12:        Print($u_s u_{s+1} \cdots u_{s'-1}$)
13:        $s \leftarrow s'$
14:    // Maintain window-size
15:    if $t - s + 1 = n$ then
16:        if $p > n/2$ then $(s', p, w) \leftarrow$ Update($s+1$, $w$)
17:        else $(s', p, w) \leftarrow$ Update($s + p$, $w$)
18:        Print($u_s u_{s+1} \cdots u_{s'-1}$)
19:        $s \leftarrow s'$
```

Fig. 5. Pseudocode of FastIncrement().

- if $u_{t-p} < u_t$, then β is a prenecklace and $ap(\beta) = u_s u_{s+1} \cdots u_t$, thus we update $p = |\beta| = t - s + 1$;
- if $u_{t-p} = u_t$, then β is a prenecklace and the aperiodic prefix remains unchanged, that is $ap(\beta) = ap(u_s u_{s+1} \cdots u_{t-1}) = u_s u_{s+1} \cdots u_{s+p-1}$, we keep s and p unchanged;
- if $u_{t-p} > u_t$, then β is not a prenecklace; we update s to $s + \lfloor \frac{t-s}{p} \rfloor \cdot p$; we update p to be the length of $ap(u_{s+\lfloor \frac{t-s}{p} \rfloor \cdot p} \cdots u_t)$;
  - ▷ for example, consider $n = 13$, $\beta = u_1 u_2 \cdots u_{12} = 001001001000$ and $p = 3$; β is not a necklace because $u_{12-3} > u_{12}$; the variable s is thus updated to $1 + \lfloor \frac{12-1}{3} \rfloor \cdot 3 = 10$ such that the sliding window β starts with $u_{10}u_{11}u_{12} = 000$ which is lexicographically smaller than $u_1 u_2 u_3 = 001$; p is updated to the length of $ap(u_{10}u_{11}u_{12}) = |ap(000)| = 1$.

Maintain Window-size: We increment the variable s to s' when the size of β reaches n such that $u_{s'} u_{s'+1} \cdots u_t$ is a prenecklace; we update p to be the length of $ap(u_{s'} u_{s'+1} \cdots u_t)$.

Analysis: We analyze the amount of work divided by the number of character in universal cycle for $\mathbf{B}_c^d(n)$, we demonstrate that the total amount of work of FastIncrement() divided by the number of character in universal cycle for $\mathbf{B}_c^d(n)$ is bounded by a constant. The work required for each of the above stage is as follows:

Glue universal cycles: The string $u_s u_{s+1} \cdots u_{t-1} \cdot 1 \in \mathbf{N}_d(n)$ if $w = d - 1$, $u_t = 0$ and $t - s + 1 = n$, this can be verified using only a constant amount of work.

Maintain Prenecklace: If $u_{t-p} \leq u_t$, then β is a prenecklace and we update p which requires a constant amount of work; if $a_{t-p} > a_t$, we call the function Update($s + \lfloor \frac{t-s}{p} \rfloor \cdot p$, w) which requires $O(t - s - \lfloor \frac{t-s}{p} \rfloor \cdot p)$ amount of work; however, we print at least $\lfloor \frac{t-s}{p} \rfloor \cdot p$ characters. Since $t - s - \lfloor \frac{t-s}{p} \rfloor \cdot p < \lfloor \frac{t-s}{p} \rfloor \cdot p$, the amount of work is proportional to the number of character.

Maintain Window-size: The size of the sliding window reaches n. It calls the function Update($s + 1$, w) to update the variables s and p but prints only one character, thus it requires $O(n)$ amount of work per character.

Let $N(n, w)$, $L(n, w)$ and $P(n, w)$ denote the cardinality of $\mathbf{N}(n, w)$, $\mathbf{L}(n, w)$ and $\mathbf{P}(n, w)$, and let $P_0(n, w)$ and $P_1(n, w)$ denote the cardinality of the set of binary prenecklaces of length n and weight w that end with the character 0 and 1 respectively. We show that the number of length n prenecklaces is bounded by the number of elements in the universal cycle for $\mathbf{B}_c^d(n)$ over n, and thus the total amount of computation required to update all prenecklaces is proportional to the length of the universal cycle for $\mathbf{B}_c^d(n)$. The following lemma provides an upper bound of $P_1(n, w)$ in terms of $N(n, w)$ and $L(n, w)$.

Lemma 5. [23] $P_1(n, w) \leq N(n, w) + L(n, w)$.

Consider the upper bound of $P_0(n, w)$, replacing the last character of a prenecklace of weight w that ends with a 0 with the character 1 will always yield a unique necklace of weight $w + 1$, $P_0(n, w)$ is therefore bounded by the number of necklaces of weight $w + 1$.

Lemma 6. $P_0(n, w) \leq N(n, w + 1)$.

The upper bound of $N(n, w)$ and $L(n, w)$ in terms of $\binom{n}{w}$ has been discussed in [23] and are given as follows:

$$L(n, w) \leq \frac{1}{n}\binom{n}{w} \quad \text{and} \quad N(n, w) \leq 2L(n, w) \leq \frac{2}{n}\binom{n}{w}.$$

The upper bound of $P(n, w)$ in terms of $\binom{n}{w}$ is therefore as follows:

$$\begin{aligned} P(n, w) &= P_0(n, w) + P_1(n, w) \\ &\leq N(n, w + 1) + N(n, w) + L(n, w) \\ &\leq \frac{2}{n}\binom{n}{w+1} + \frac{2}{n}\binom{n}{w} + \frac{1}{n}\binom{n}{w} \\ &\leq \frac{2}{n}\binom{n}{w+1} + \frac{3}{n}\binom{n}{w}. \end{aligned}$$

Theorem 6. *Algorithm FastIncrement() is a CAT algorithm.*

Proof. Let hn be the amount of work required in stage 3 of FastIncrement() to update the prenecklace, where h is a constant. The ratio between the total amount of work required in stage 3 of FastIncrement() to the number of elements in the universal cycle for $\mathbf{B}_c^d(n)$ is as follows:

$$\begin{aligned}
\frac{\text{Total work in stage 3}}{|B_c^d|} &= \frac{(P(n,d-1)+P(n,d-2)+\cdots+P(n,c))\times hn}{\binom{n}{d}+\binom{n}{d-1}+\cdots+\binom{n}{c}} \\
&\leq \frac{(\frac{2}{n}\binom{n}{d}+\frac{5}{n}\binom{n}{d-1}+\frac{5}{n}\binom{n}{d-2}+\cdots+\frac{5}{n}\binom{n}{c+1}+\frac{3}{n}\binom{n}{c})\times hn}{\binom{n}{d}+\binom{n}{d-1}+\cdots+\binom{n}{c}} \\
&\leq \frac{(2\binom{n}{d}+5\binom{n}{d-1}+5\binom{n}{d-2}+\cdots+5\binom{n}{c+1}+3\binom{n}{c})\times h}{\binom{n}{d}+\binom{n}{d-1}+\cdots+\binom{n}{c}} \\
&< 5h.
\end{aligned}$$

Since stage 1 and 2 of FastIncrement() requires only constant amount of work per character, the algorithm FastIncrement() is a CAT algorithm. □

Theorem 7. *A universal cycle $UD_c^d(n)$ for $\mathbf{B}_c^d(n)$ can be constructed in constant amortized time using $O(n)$ space for any weight-range where $0 \leq c < d \leq n$.*

6 Other Applications of the Gluing Lemma

In this section we consider other sets of strings and their associated universal cycles and apply the Gluing lemma to produce new universal cycles.

6.1 Passwords

In [18], the set of *passwords* is defined to be the set of all strings of length n over an alphabet of size k partitioned into $q < k$ classes where each string contains at least one character from each class. For instance, 4 natural classes would be: lower case letters, upper case letters, digits, and special characters. A very secure password would contain one symbol from each class. They prove the following result:

Theorem 8. [18] *A universal cycle exists for all n-letter passwords over an alphabet of size k partitioned into $q < k$ classes, provided that $n \geq 2q$.*

We relax the definition of a *password* to be a string that contains at least one symbol from $q' \leq q$ classes. In fact, this is a common requirement of passwords where they must either contain a number or a special character. As an example, consider all passwords of length n containing characters in *at least* two classes. Such strings can be partitioned into $\binom{4}{2}$ sets of words containing exactly 2 classes, plus 4 sets of words containing exactly 3 classes, plus one set containing characters from all 4 classes. Observe that all sets are disjoint, and the sets containing strings from exactly 2 classes have many strings that have $n-1$ characters in common. For instance 'aAAAAAAA' and '1AAAAAAA' and '#AAAAAAA'. Similarly, there exist common strings of length $n-1$ between a set of exactly 2 classes and a set with one additional class. For instance 'aAAAAAAA' and 'aAAAAAA3'. Thus, the following theorem follows from Lemma 3.

Theorem 9. *Let an alphabet of size k be partitioned into $q < k$ classes. There exists a universal cycle for all strings of length n containing letters from at least $q' \leq q$ classes, provided that $n \geq 2q$.*

Observe that if $q' = 1$, then the universal cycle is a traditional de Bruijn sequence over an alphabet of size k.

6.2 Labeled Graphs

In [3], a number of universal cycle existence questions are given for various labeled graphs. Instead of strings, they consider graphs with a sliding window of size k that represent labeled graphs. In particular, they give the following result.

Theorem 10. [3] *Universal cycle exists for labeled graphs with precisely m edges (and k vertices).*

Since graphs with m edges and graphs with $m + 1$ edges are disjoint and their universal cycles have many graphs with identical $k - 1$ *windows*, we can apply Lemma 3 to obtain the following result:

Theorem 11. *Universal cycle exists for labeled graphs with between m_1 and m_2 edges (and k vertices).*

It remains an open problem to find efficient constructions for such universal cycles.

References

1. Bechel, A., LaBounty-Lay, B., Godbole, A.: Universal cycles of discrete functions. In: Proceedings of the Thirty-Ninth Southeastern International Conference on Combinatorics, Graph Theory and Computing, vol. 189, pp. 121–128. Congressus Numerantium (2008)
2. Blanca, A., Godbole, A.: On universal cycles for new classes of combinatorial structures. SIAM J. Discret. Math. 25(4), 1832–1842 (2011)
3. Brockman, G., Kay, B., Snively, E.: On universal cycles of labeled graphs. Electronic Journal of Combinatorics 17(1), 9 (2010)
4. Casteels, K., Stevens, B.: Universal cycles for $(n - 1)$-partitions of an n-set. Discrete Mathematics 309, 5332–5340 (2009)
5. Chung, F., Diaconis, P., Graham, R.: Universal cycles for combinatorial structures. Discrete Mathematics 110, 43–59 (1992)
6. de Bruijn, N.G.: A combinatorial problem. Koninklijke Nederlandse Akademie v. Wetenschappen 49, 758–764 (1946)
7. de Bruijn, N.G.: Acknowledgement of priority to C. Flye Sainte-Marie on the counting of circular arrangements of $2n$ zeros and ones that show each n-letter word exactly once. T.H. Report 75-WSK-06, p. 13 (1975)
8. Duval, J.P.: Factorizing words over an ordered alphabet. J. Algorithms 4(4), 363–381 (1983)
9. Fredericksen, H., Kessler, I.J.: An algorithm for generating necklaces of beads in two colors. Discrete Mathematics 61, 181–188 (1986)
10. Fredericksen, H., Maiorana, J.: Necklaces of beads in k colors and k-ary de Bruijn sequences. Discrete Mathematics 23, 207–210 (1978)

11. Hierholzer, C., Wiener, C.: Ueber die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren. Mathematische Annalen 6(1), 30–32 (1873)
12. Holroyd, A.E., Ruskey, F., Williams, A.: Shorthand universal cycles for permutations. Algorithmica 64(2), 215–245 (2012)
13. Hurlbert, G.: On universal cycles for k-subets of an n-element set. Siam Journal on Discrete Mathematics 7, 598–604 (1994)
14. Hurlbert, G., Jackson, B., Stevens, B. (eds.): Generalisations of de Bruijn sequences and Gray codes. Discrete Mathematics 309, 5255–5348 (2009)
15. Jackson, B.: Universal cycles of k-subsets and k-permutations. Discrete Mathematics 117, 114–150 (1993)
16. Johnson, R.: Universal cycles for permutations. Discrete Mathematics 309, 5264–5270 (2009)
17. Knuth, D.E.: Generating all tuples and permutations, fascicle 2. The Art of Computer Programming 4 (2005)
18. Leitner, A., Godbole, A.: Universal cycles of classes of restricted words. Discrete Mathematics 310, 3303–3309 (2010)
19. Rosen, K.H.: Discrete Mathematics and Its Applications, 5th edn. McGraw-Hill Higher Education (2002)
20. Ruskey, F., Sawada, J., Williams, A.: De Bruijn sequences for fixed-weight binary strings. SIAM Journal on Discrete Mathematics 26(2), 605–617 (2012)
21. Ruskey, F., Williams, A.: An explicit universal cycle for the $(n-1)$-permutations of an n-set. ACM Transactions on Algorithms 6(3), 12 (2010)
22. Ruskey, F., Williams, A., Sawada, J.: Binary bubble languages and cool-lex order. J. Comb. Theory, Ser. A 119(1), 155–169 (2012)
23. Sawada, J., Ruskey, F.: An efficient algorithm for generating necklaces with fixed density. In: Tarjan, R.E., Warnow, T. (eds.) SODA, pp. 752–758. ACM/SIAM (1999)
24. Sawada, J., Stevens, B., Williams, A.: De bruijn sequences for the binary strings with maximum density. In: Katoh, N., Kumar, A. (eds.) WALCOM 2011. LNCS, vol. 6552, pp. 182–190. Springer, Heidelberg (2011)
25. Stevens, B., Williams, A.: The coolest order of binary strings. In: Kranakis, E., Krizanc, D., Luccio, F. (eds.) FUN 2012. LNCS, vol. 7288, pp. 322–333. Springer, Heidelberg (2012)
26. Stevens, B., Williams, A.: The coolest way to generate binary strings. Theory of Computing Systems, 1–27 (2013)

Circuit Complexity of Shuffle

Michael Soltys[⋆]

McMaster University
Dept. of Computing & Software
1280 Main Street West
Hamilton, Ontario L8S 4K1, Canada
soltys@mcmaster.ca

Abstract. We show that Shuffle(x, y, w), the problem of determining whether a string w can be composed from an order preserving shuffle of strings x and y, is not in $\mathbf{AC}^0$, but it is in $\mathbf{AC}^1$. The fact that shuffle is not in $\mathbf{AC}^0$ is shown by a reduction of parity to shuffle and invoking the seminal result [FSS84], while the fact that it is in $\mathbf{AC}^1$ is implicit in the results of [Man82a]. Together, the two results provide a strong complexity bound for this combinatorial problem.

Keywords: String shuffle, circuit complexity, lower bounds.

1 Introduction

Given three strings over the binary alphabet, it is a natural question to ask whether the third string can be composed from a "shuffle" of the first two. That is, can we compose the third string by weaving together the first two, while preserving the order within each string? For example, given 000, 111, and 010101, we can obviously answer in the affirmative. [Man82a] shows that a clever dynamic programming algorithm can determine Shuffle(x, y, w) in time $O(|w|^2)$, and the same paper poses the question of determining a lower bound.

In this paper we show a strong upper and lower bound for the shuffling problem in terms of circuit complexity. We show that: (i) bounded depth circuits of polynomial size *cannot* solve shuffle, but that (ii) logarithmic depth circuits of polynomial size *can* do so. In the nomenclature of circuit complexity this can be stated as follows: Shuffle $\notin \mathbf{AC}^0$ but Shuffle $\in \mathbf{AC}^1$, which provides a good characterization of the circuit complexity of shuffle.

As a side remark, we also show a lower bound for shuffle on single-tape Turing machines; for this model of computation, we can show that a number of steps in the order of $\Omega(n^2)$ is *necessary* to solve the problem. Both lower bounds, the one in terms of bounded depth circuits, and the one in terms of single tape Turing machines, are obtained by reductions. In the former case, the reduction is from the "parity problem" and in the latter case, the reduction is from the "palindromes problem."

[⋆] Supported in part by the NSERC Discovery Grant.

© Springer-Verlag Berlin Heidelberg 2013

This paper is structured as follows: in section 2 we give the background on circuit complexity; in section 3 we give an upper bound and in section 4 we give the lower bound on the complexity of shuffle. The bounds are summarized in Theorem 1 in the conclusion, and we finish with some open problems.

Formal Definition of Shuffle. If x, y, and w are strings over an alphabet Σ, then w is a *shuffle* of x and y provided there are (possibly empty) strings x_i and y_i such that $x = x_1x_2 \cdots x_k$ and $y = y_1y_2 \cdots y_k$ and $w = x_1y_1x_2y_2 \cdots x_ky_k$. Note that $|w| = |x| + |y|$ is a necessary condition for the existence of a shuffle. Also note that [Man82a] gives a different but equivalent definition.

A shuffle is sometimes instead called a "merge" or an "interleaving". The intuition for the definition is that w can be obtained from u and v by an operation similar to shuffling two decks of cards. In this paper we assume the binary alphabet, i.e., $x, y, w \in \{0,1\}^*$.

The predicate Shuffle(x, y, w) holds iff w is a shuffle of x, y, as described in the above paragraph. We are going to present circuits and Turing machines that compute the Shuffle(x, y, w), and so they must take the three binary strings x, y, w as input. We let $\langle x, y, w\rangle$ denote the encoding of the three strings; this encoding can be just a concatenation of the strings, so that for a properly formed input, where $|x| = |y| = n$, we have $|\langle x, y, w\rangle| = 4n$, and the n can be extracted by the machine. We can also use demarcation of the strings, by encoding 0 with 00, and 1 with 01, and use 11 as separators. In that case, a well formed input where $|x| = m, |y| = n, |w| = m + n$, would be such that $|\langle x, y, w\rangle| = 2m + 1 + 2n + 1 + 2(m + n)$. The point is that it does not matter how we do it, as long as we do it "reasonably."

History. Following the presentation of the history of shuffle in [BS13], we mention that the initial work on shuffles arose out of abstract formal languages. Shuffles were later motivated by applications to modeling sequential execution of concurrent processes. The shuffle operation was first used in formal languages by Ginsburg and Spanier [GS65]. Early research with applications to concurrent processes can be found in Riddle [Rid73, Rid79] and Shaw [Sha78]. A number of authors, including [Gis81, GH09, Jan81, Jan85, Jed99, JS01, JS05, MS94, ORR78, Sho02] have subsequently studied various aspects of the complexity of the shuffle and iterated shuffle operations in conjunction with regular expression operations and other constructions from the theory of programming languages.

In the early 1980's, Mansfield [Man82b, Man83] and Warmuth and Haussler [WH84] studied the computational complexity of the shuffle operator on its own. The paper [Man82b] gave a polynomial time dynamic programming algorithm for deciding Shuffle(x, y, w). In [Man83], this was extended to give polynomial time algorithms for deciding whether a string w can be written as the shuffle of k strings $u_1, \ldots, u_k$, for a *constant* integer k. The paper [Man83] further proved that if k is allowed to vary, then the problem becomes NP-complete (via a reduction from Exact Cover with 3-Sets). Warmuth and Haussler [WH84] gave an independent proof of this last result and went on to give a rather striking improvement by showing that this problem remains NP-complete even if the k strings $u_1, \ldots, u_k$ are equal. That is to say, the question of, given strings u and w, whether w is equal to an *iterated shuffle* of u is NP-complete. Their proof used a reduction from 3-Partition.

In [BS13] we show that square shuffle, i.e., the problem of determining whether some string w is a shuffle of some x with itself, that is, $\exists x \text{Shuffle}(x, x, w)$, is NP-hard.

2 Background on Complexity

A *Boolean circuit* can be seen as a directed, acyclic, connected graph in which the input nodes are labeled with variables x_i and constants $1, 0$ (or T, F), and the internal nodes are labeled with standard Boolean connectives $\wedge, \vee, \neg$, that is, AND,OR,NOT, respectively. We often use $\bar{x}$ to denote $\neg x$, and the circuit nodes are often called *gates.*

The *fan-in* (i.e., number of incoming edges) of a $\neg$-gate is always 1, and the fan-in of $\wedge, \vee$ can be arbitrary, even though for some complexity classes (such as $\mathbf{SAC}^1$ defined below) we require that the fan-in be bounded by a constant. The *fan-out* (i.e., number of outgoing edges) of any node can also be arbitrary. Note that when the fan-out is restricted to be exactly 1, circuits become Boolean formulas. Each node in the graph can be associated with a Boolean function in the obvious way. The function associated with the output gate(s) is the function computed by the circuit. Note that a Boolean formula can be seen as a circuit in which every node has fan-out 1 (and $\wedge, \vee$ have fan-in 2, and $\neg$ has fan-in 1).

The *size* of a circuit is its number of gates, and the *depth* of a circuit is the maximum number of gates on any path from an input gate to an output gate.

A *family of circuits* is an infinite sequence $C = \{C_n\} = \{C_0, C_1, C_2, \ldots\}$ of Boolean circuits where C_n has n input variables. We say that a Boolean predicate P has polysize circuits if there exists a polynomial p and a family C such that $|C_n| \leq p(n)$, and $\forall x \in \{0,1\}^*$, $x \in P$ iff $C_{|x|}(x) = 1$. In order to make this more concrete, note that in the case of shuffle, the family C computes it if

$$\text{Shuffle}(x, y, w) = 1 \iff C_{|\langle x,y,w\rangle|}(\langle x, y, w\rangle) = 1.$$

Note that $|\langle x, y, w\rangle|$ only depends on the length of the inputs x, y, w and so the same circuit decided all the inputs of a given fixed length.

Let $\mathbf{P}$/poly be the class of all those predicates which have polysize circuits. It is a standard result in complexity that all predicates in $\mathbf{P}$ have polysize circuits; that is, if a predicate has a polytime Turing machine, it has polysize circuits. The converse of the above does not hold, unless we put a severe restriction on how the n-th circuit is generated; as it stands, there are undecidable predicates that have polysize circuits. The restriction that we place here is that there is a Turing machine that on input 1^n computes $\{C_n\}$ in space $O(\log n)$. This restriction makes a family of circuits C *uniform.* All our circuit results hold with and without the condition of uniformity.

Those predicates (or Boolean functions) that can be decided with polysize, constant fan-in, and depth $O(\log^i n)$ circuits, form the class $\mathbf{NC}^i$. The class $\mathbf{AC}^i$ is defined in the same way, except we allow unbounded fan-in. We set $\mathbf{NC} = \bigcup_i \mathbf{NC}^i$, and $\mathbf{AC} = \bigcup_i \mathbf{AC}^i$, and while it is easy to see that the uniform version of $\mathbf{NC}$ is in $\mathbf{P}$, it is an interesting open question whether they are equal.

We have the following standard result: for all i,

$$\mathbf{AC}^i \subseteq \mathbf{NC}^{i+1} \subseteq \mathbf{AC}^{i+1}.$$

Thus, $\mathbf{NC} = \mathbf{AC}$. Finally, $\mathbf{SAC}^i$ is just like $\mathbf{AC}^i$, except we restrict the $\wedge$ fan-in to be at most two. Recall that $\mathbf{NC}^1 \subseteq \mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{NC}^2$, where $\mathbf{L}$ and $\mathbf{NL}$ are deterministic and non-deterministic logarithmic space, respectively. It is not known whether any of these containments are strict. For more details see any complexity textbook; for example [Pap94, Sip06, Sol09].

3 Upper Bound

In this section we show a circuit upper bound for shuffle — that is, we show that Shuffle $\in \mathbf{SAC}^1$, which means that shuffle can be decided with a polysize family of circuits of logarithmic depth (in the size of the input), where all the $\wedge$-gates have fan-in 2. This result relies on the dynamic programming algorithm given in [Man82a] and the complexity result of [Sud78, Ven91] which shows that $\mathbf{NL} \subseteq \mathbf{SAC}^1$.

In order to show that Shuffle $\in \mathbf{NL}$, we show that shuffle can be reduced (in low complexity) to the graph reachability problem. We start with an example: consider Figure 1. On the left we have a shuffle of 000 and 111 that yields 010101, and on the right we have a shuffle of 011 and 011 that yields 001111. The left instance has a unique shuffle; there is a unique path from $(0,0)$ to $(3,3)$. On the right, there are several possible shuffles — in fact, eight of them, each corresponding to a distinct path from $(0,0)$ to $(3,3)$.

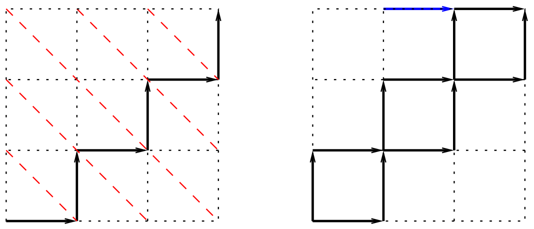


Fig. 1. On the left we have a shuffle of 000 and 111 that yields 010101, and on the right we have a shuffle of 011 and 011 that yields 001111. The dynamic programming algorithm in [Man82a] computes partial solutions along the red diagonal lines.

The number of paths is always bounded by:

$$\binom{|x|+|y|}{|x|}$$

and this bound is achieved for $\langle 1^n, 1^n, 1^{2n}\rangle$. Thus, the number of paths can be exponential in the size of the input, and so an exhaustive search is not feasible in general.

Lemma 1. Shuffle $\in$ **NL**.

Proof. The dynamic programming algorithm proposed in [Man82a] works by reducing shuffle to directed graph reachability. The graph is an $(n+1)\times(n+1)$ grid of nodes, with the lower-left corner labeled $(0,0)$, and the upper-right corner labeled (n,n). For any $i \le n$ and $j \le n$, we have edge

$$\begin{cases} ((i,j),(i+1,j)) & \text{if } x_{i+1} = w_{i+j+1} \\ ((i,j),(i,j+1)) & \text{if } y_{j+1} = w_{i+j+1}. \end{cases}$$

Note that both edges may be present, which is what introduces the element of non-determinism.

The correctness of the reduction follows from the assertion that given the edges of the grid, defined as in the paragraph above, there is a path from $(0,0)$ to (i,j) if and only if the first $i+j$ bits of w can be obtained by shuffling the first i bits of x and the first j bits of y. Thus, node (n,n) can be reached from node $(0,0)$ if and only if Shuffle(x,y,w) is true.

Thus Shuffle $\in$ **NL**. □

Corollary 1. Shuffle $\in \mathbf{SAC}^1$

Proof. Since $\mathbf{NL} \subseteq \mathbf{SAC}^1$ (see [Sud78, Ven91]) it follows directly from Lemma 1 that Shuffle $\in \mathbf{SAC}^1 \subseteq \mathbf{AC}^1$. □

Note that since the graph resulting from the shuffle reduction is planar, it follows that Shuffle $\in$ **UL**. [BTV07] shows that directed planar reachability is in the class **UL** (for "Unambiguous Logarithmic Space"; this class is just like **NL** except that we can design a non-deterministic log-space bounded machine such that "no" instances of the problem have no accepting paths, while "yes" instances have exactly one accepting path).

Note that Shuffle $\in$ **UL** does not mean that there is a unique path from $(0,0)$ to (n,n); rather, there exists a machine deciding the problem in log-space such that, while the machine is non-deterministic, at most one computational path accepts. Further, the planar graph in the proof of Lemma 1 is layered — and such graphs have been studied in [ABC$^+$09]. It would be interesting to know whether the results contained therein could improve the upper bound for shuffle. In particular, can this be used to show that Shuffle $\in \mathbf{NC}^1$, and hence shuffle can be decided with a polysize family of Boolean *formulas*? The fact that $\mathbf{NC}^1$ circuits are computationally equivalent to Boolean formulas follows from Spira's theorem (see [Sol09, Theorem 6.3]).

4 Lower Bound

In this section we show a circuit lower bound for shuffle — that is, we show that Shuffle $\notin$ **AC**^{0}, which means that shuffle cannot be decided with a polysize family of circuits of constant depth where all the $\wedge, \vee$-gates may have arbitrary fan-in. This result relies on the seminal complexity result showing that parity is not in **AC**^{0}, due to [FSS84]. A very accessible presentation of this result can be found in [SP95, Chapters 11 & 12]; many of the details of that presentation are made explicit in [Sol09, section 5.3]. We also show that shuffle requires $\Omega(n^2)$ steps on a single-tape Turing machines.

Circuit Lower Bound. We start with a definition: let $\#(x)_s$ be the number of occurrences of a symbol s in the string x. Obviously, Shuffle$(0^{\#(x)_0}, 1^{\#(x)_1}, x)$ is always true. We can use this observation in order to reduce parity to shuffle, where the reduction itself is **AC**^{0}.

Intuitively, what we claim is the following: suppose that we have a "black-box" that takes $\langle x, y, w\rangle$ as input bits and computes Shuffle(x, y, w). We could then construct a circuit for parity with the standard gates $\wedge, \vee, \neg$, plus black-boxes for computing shuffle. If the black-boxes for shuffle were computable with **AC**^{0} circuits, we would then obtain an **AC**^{0} circuit for parity, giving us a contradiction. The details are given in Lemma 2 below and in Figure 2.

Lemma 2. Parity $\in$ **AC**^{0}[Shuffle].

Proof. In order to compute the parity of x, run the following algorithm: for all odd $i \in \{0, \ldots, |x|\}$, check if Shuffle$(0^{|x|-i}, 1^i, x)$ is true; if it is the case for at least one i, then the parity of x is 1. Note that if it is true for at least one i, it is true for exactly one i. In terms of circuits, this can be expressed as follows:

$$\text{Parity}(x) = \bigvee_{\substack{0 \le i \le |x| \\ i \text{ is odd}}} \text{Shuffle}(0^{|x|-i}, 1^i, x), \tag{1}$$

which gives us an **AC**^{0} circuits with "black-boxes" for shuffle, and hence the claim follows. See Figure 2. □

Corollary 2. Shuffle $\notin$ **AC**^{0}.

Proof. Since by [FSS84] Parity $\notin$ **AC**^{0}, and by Lemma 2 we know that parity **AC**^{0}-reduces to shuffle, it follows that shuffle is not in **AC**^{0}. □

Turing Machine Lower Bound. The string x is a palindrome if it reads the same backward as forward. If x^R is the reverse of x, i.e., $x^R = x_n x_{n-1} \ldots x_1$, then x is a palindrome if and only if $x = x^R$. It is a folklore result in complexity that given a single tape Turing machine as the model of computation, testing

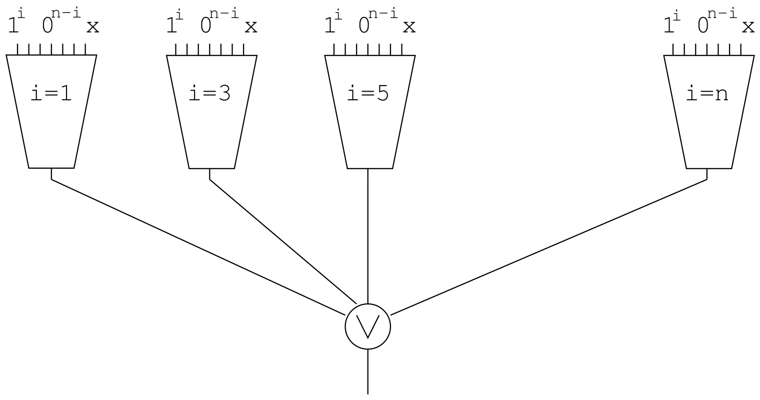


Fig. 2. Parity of x computed in terms of Shuffle; note that we assume that n is odd in this Figure. If n were even the last "black box" for shuffle would be for $i = n - 1$.

for palindromes requires $\Omega(|x|^2)$ many steps. This result uses Kolmogorov complexity and the "crossing sequences technique." The interested reader can check, for example, [SP95, Chapter 9] or [Sol09, §1.3].

We can use this lower bound for palindromes in order to show that shuffle also requires $\Omega(n^2)$ many steps on a single tape Turing machine. Let Palindrome(x) be the eponymous predicate and note that we can use shuffle to express that a string is a palindrome as follows:

$$\text{Palindrome}(x) \iff \text{Shuffle}(\varepsilon, x, x^R). \tag{2}$$

As the first string is empty, shuffle will hold iff $x_i = x_{n+1-i}$, for $i \in [n]$, which is true iff $x = x^R$.

Lemma 3. Shuffle *takes $\Omega(n^2)$ many steps on a single-tape Turing machine.*

Proof. Suppose that a single-tape Turing machine can decide, on input $\langle x, y\rangle$ whether Shuffle(ε, x, y). Then, the same machine can decide on input

$$\langle w_1 \dots w_{\lfloor \frac{n}{2} \rfloor}, w_n \dots w_{\lceil \frac{n}{2} \rceil + 1}\rangle$$

whether w, where $n = |w|$, is a palindrome. As palindromes require $\Omega(n^2)$ steps (in the worst-case), so does Shuffle(ε, x, y). As Shuffle(ε, x, y) is a special case of the general shuffle problem, the Lemma follows. □

Other Reductions to Shuffle. It is interesting that several different string predicates reduce to shuffle in a natural way. We have (1) which gives a reduction of parity to shuffle; we have that equality of strings reduces to shuffle: $x = y \iff$ Shuffle(ε, x, y); we have (2) which shows that palindromes reduce to shuffle.

We end by showing that concatenation reduces to shuffle. Let p_0, p_1 be "padding" functions on strings defined as follows:

$$\begin{aligned} p_0(x) &= p_0(x_1x_2\dots x_n) = 00x_100x_200\dots00x_n00 \\ p_1(x) &= p_1(x_1x_2\dots x_n) = 11x_111x_211\dots11x_n11 \end{aligned}$$

that is, p_b, $b \in \{0,1\}$ pads the string x with a pair of b's between any two bits of x, as well as a pair of b's before and after x. Now note that

$$w = u{\cdot}v \iff \text{Shuffle}(p_0(u), p_1(v), p_0(w_1w_2\dots w_{|u|}){\cdot}p_1(w_{|u|+1}w_{|u|+2}\dots w_{|u|+|v|})),$$

where "$\cdot$" denotes concatenation of strings. The direction "$\Rightarrow$" is easy to see; for direction "$\Leftarrow$" we use the following notation:

$$\begin{aligned} r &= p_0(u) = 00u_100\dots \\ s &= p_1(v) = 11v_111\dots \\ t &= p_0(w_1w_2\dots w_{|u|}) \cdot p_1(w_{|u|+1}w_{|u|+2}\dots w_{|u|+|v|}) = 00w_100\dots \end{aligned}$$

If t is a shuffle of r, s, i.e., Shuffle(r, s, t), then we must take the first two bits of r (00) in order to cover the first two bits of t (00). If $u_1 = w_1 = 1$, then we could ostensibly take the first bit of s (1), but the bit following w_1 is 0, and $u_1 = 1$ and the second bit of s is 1; so taking the first bit of s leads to a dead end. Thus, we must use u_1 to cover w_1. We continue showing that we must first take all of r, and then take all of s in order to cover t. This argument can be formalized with induction.

It follows that Shuffle(r, s, t) implies $t = r \cdot s$, which in turn implies $w = u \cdot v$.

5 Conclusion

Putting everything together we have the following Theorem.

Theorem 1. Shuffle $\notin$ $\mathbf{AC}^0$, *but* Shuffle $\in$ $\mathbf{SAC}^1 \subseteq \mathbf{AC}^1$. *Also, shuffle requires* $\Omega(n^2)$ *many steps on a single tape Turing machine.*

The significance of this result is that shuffle cannot be decided with bounded depth circuits of polynomial size. On the other hand, shuffle can be decided with polynomial size circuits of unbounded fan-in and logarithmic depth — which in turn implies that shuffle can be decided in the class $\mathbf{NC}^2$. In general, the classes $\mathbf{NC}^i$ capture those problems that can be solved with polynomially many processors in poly-logarithmic time, which are problems that have fast parallel algorithms. See [Coo85] for a discussion of $\mathbf{NC}^2$.

6 Open Problems

It follows from the results of [Man82a] that shuffle can be decided in $\mathbf{SAC}^1$. Can shuffle be decided in $\mathbf{NC}^1$? We know that $\mathbf{NC}^1 \subseteq \mathbf{SAC}^1 \subseteq \mathbf{NC}^2$, and $\mathbf{SAC}^1$

is almost the same as $\mathbf{NC}^1$ except that $\mathbf{SAC}^1$ allows unbounded fan-in for the $\vee$-gates (and bounded fan-in for the $\wedge$-gates), whereas $\mathbf{NC}^1$ has bounded fan-in for all gates. If shuffle were in $\mathbf{NC}^1$ it would mean that shuffle can be decided with a polysize family of Boolean formulas, which would be a very interesting result.

Acknowledgments. The author is grateful to Bill Smyth, Franya Franek and Dragan Rakas for comments on this paper, and especially to Bill Smyth for introducing me to the "shuffle problem." This paper has benefited greatly from the insights of anonymous referees.

References

[ABC$^+$09] Allender, E., Barrington, D.A.M., Chakraborty, T., Datta, S., Roy, S.: Planar and grid graph reachability problems. Theory of Computing Systems 45, 675–723 (2009)

[BS13] Buss, S.R., Soltys, M.: Unshuffling a square is NP-hard (submitted for publication, March 2013)

[BTV07] Bourke, C., Tewari, R., Vinodchandran, N.V.: Directed planar reachability is in unambiguous log-space. In: Proceedings of IEEE Conference on Computational Complexity CCC (2007)

[Coo85] Cook, S.A.: A taxonomy of problems with fast parallel algorithms. Information and Computation 64(13), 2–22 (1985)

[FSS84] Furst, M., Saxe, J.B., Sipser, M.: Parity, circuits, and the polynomial-time hierarchy. Math. Systems Theory 17, 13–27 (1984)

[GH09] Gruber, H., Holzer, M.: Tight Bounds on the Descriptional Complexity of Regular Expressions. In: Diekert, V., Nowotka, D. (eds.) DLT 2009. LNCS, vol. 5583, pp. 276–287. Springer, Heidelberg (2009)

[Gis81] Gischer, J.: Shuffle languages, petri nets, and context-sensivite grammars. Communications of the ACM 24(9), 597–605 (1981)

[GS65] Ginsburg, S., Spanier, E.: Mappings of languages by two-tape devices. Journal of the A.C.M 12(3), 423–434 (1965)

[Jan81] Jantzen, M.: The power of synchronizing operations on strings. Theoretical Computer Science 14, 127–154 (1981)

[Jan85] Jantzen, M.: Extending regular expressions with iterated shuffle. Theoretical Computer Science 38, 223–247 (1985)

[Jed99] Jedrzejowicz, J.: Structural properties of shuffle automata. Grammars 2(1), 35–51 (1999)

[JS01] Jedrzejowicz, J., Szepietowski, A.: Shuffle languages are in P. Theoretical Computer Science 250(1-2), 31–53 (2001)

[JS05] Jedrzejowicz, J., Szepietowski, A.: On the expressive power of the shuffle operator matched with intersection by regular sets. Theoretical Informatics and Applications 35, 379–388 (2005)

[Man82a] Mansfield, A.: An algorithm for a merge recognition problem. Discrete Applied Mathematics 4(3), 193–197 (1982)

[Man82b] Mansfield, A.: An algorithm for a merge recognition problem. Discrete Applied Mathematics 4(3), 193–197 (1982)

[Man83] Mansfield, A.: On the computational complexity of a merge recognition problem. Discrete Applied Mathematics 1(3), 119–122 (1983)

[MS94] Mayer, A.J., Stockmeyer, L.J.: The complexity of word problems — this time with interleaving. Information and Computation 115, 293–311 (1994)

[ORR78] Ogden, W.F., Riddle, W.E., Rounds, W.C.: Complexity of expressions allowing concurrency. In: Proc. 5th ACM Symposium on Principles of Programming Languages (POPL), pp. 185–194 (1978)

[Pap94] Papadimitriou, C.H.: Computational Complexity. Addison-Wesley (1994)

[Rid73] Riddle, W.E.: A method for the description and analysis of complex software systems. SIGPLAN Notices 8(9), 133–136 (1973)

[Rid79] Riddle, W.E.: An approach to software system modelling and analysis. Computer Languages 4(1), 49–66 (1979)

[Sha78] Shaw, A.C.: Software descriptions with flow expressions. IEEE Transactions on Software Engineering SE-4(3), 242–254 (1978)

[Sho02] Shoudai, T.: A P-complete language describable with iterated shuffle. Information Processing Letters 41(5), 233–238, 1002

[Sip06] Sipser, M.: Introduction to the Theory of Computation, 2nd edn. Thompson (2006)

[Sol09] Soltys, M.: An introduction to computational complexity. Jagiellonian University Press (2009)

[SP95] Schöning, U., Pruim, R.: Gems of Theoretical Computer Science. Springer (1995)

[Sud78] Sudborough, I.H.: On the tape complexity of deterministic context-free languages. Journal of the Association of Computing Machinery 25, 405–415 (1978)

[Ven91] Venkateswaran, H.: Properties that characterize LOGCFL. Journal of Computer and System Science 43, 380–404 (1991)

[WH84] Warmuth, M.K., Haussler, D.: On the complexity of iterated shuffle. Journal of Computer and System Sciences 28(3), 345–358 (1984)

An Optimal Algorithm for the Popular Condensation Problem

Yen-Wei Wu[1], Wei-Yin Lin[1], Hung-Lung Wang[4], and Kun-Mao Chao[1,2,3]

[1] Department of Computer Science and Information Engineering
[2] Graduate Institute of Biomedical Electronics and Bioinformatics
[3] Graduate Institute of Networking and Multimedia
National Taiwan University, Taipei, Taiwan 106
[4] Institute of Information and Decision Sciences
National Taipei College of Business, Taipei, Taiwan 100

Abstract. We consider an extension of the popular matching problem: the *popular condensation problem*. An instance of the popular matching problem consists of a set of applicants A and a set of posts P. Each applicant has a strictly ordered preference list, which is a sequence of posts in order of his/her preference. A matching M mapping from A to P is *popular* if there is no other matching M' such that more applicants prefer M' to M than prefer M to M'. Although some efficient algorithms have been proposed for finding a popular matching, a popular matching may not exist for those instances where the competition of some applicants cannot be resolved. The popular condensation problem is to find a popular matching with the minimum number of applicants whose preferences are neglected, that is, to optimally condense the instance to admit a local popular matching. We show that the problem can be solved in $O(n + m)$ time, where n is the number of applicants and posts, and m is the total length of the preference lists.

1 Introduction

Consider the problem of matching applicants to a set of posts, where every applicant provides a strictly ordered preference list, ranking a non-empty subset of posts. This problem has been extensively studied due to its plentiful applications [18]. Various optimality criteria have been proposed to measure the quality of a matching, like *Pareto optimality*, *rank maximality*, and *fairness* [1,2,11]. *Popularity* is another criterion that has drawn much attention in recent years [20].

A matching M of applicants to posts is popular if there is no other matching M' such that more applicants prefer M' to M than prefer M to M'. Abraham *et al.* [3] gave a linear-time algorithm to determine if an instance admits a popular matching, and to report a largest one if any popular matching exists. For those instances admitting no popular matching, we introduce an extension, which is called the *popular condensation problem*, to deal with the circumstance.

© Springer-Verlag Berlin Heidelberg 2013

1.1 Motivation

In the popular matching problem, a given instance may admit no popular matching when there are more applicants competing for fewer specific posts. For example, if there are three applicants and three posts, and all applicants have the same preference lists, then there exists no popular matching in such an instance. However, many applications prefer a compromising answer rather than just reporting "none exists," and thus it makes sense to find an alternative solution for all the instances.

In [15], Kavitha and Nasre looked for an alternative solution by copying the posts until the "expanded" instance admits a popular matching. Not surprisingly, a solution is to copy some applicants' favorite posts. Based on this result, the applicant who is matched with one of the copied posts can be viewed as a "lucky guy" since he/she will certainly get his/her favorite post. From this viewpoint, the problem of finding some "candidates" among the applicants (either lucky ones or unlucky ones) to be neglected until the "condensed" instance admits a popular matching arises. We define the problem as the popular condensation problem. In the popular condensation problem, we ask how many preference lists of applicants should be neglected to guarantee the existence of popular matchings. In other words, the objective is to find a popular matching with a minimum number of applicants being neglected. Similar manipulations were also investigated in voting theory [5].

1.2 Related Works

Since the popular matching problem was proposed by Gardenfors [7] and was characterized by Abraham *et al.* [3] in 2005, there have been several exciting extensions in recent years. For those instances with more than one popular matching, Kavitha and Nasre [14] and McDermid and Irving [20] investigated the notions of optimality among all popular matchings. There were also extensions such as weighted popular matchings [21], many-to-one [18] and many-to-many matchings [22], mixed matchings (with probability distributions over matchings) [13], dynamic matchings (where applicants and posts can be inserted or deleted dynamically) [4], random popular matchings [17], and popular matchings in the marriage and the roommates problems [6, 8, 9, 12].

Similar to our motivation, there were some investigations that focused on the case where no popular matching exists in the instance. Kavitha and Nasre [15] considered the problems stating that: "Suppose that there are upper bounds on the number of copies for all the items. Is there an instance induced by copying the items (respecting these upper bounds) such that it admits a popular matching?" and "Given a subset of items that can be deleted, is there any possibility to delete some items so that the resulting instance admits a popular matching?" Kavitha *et al.* [16] also considered the problem where each copy is associated with a price, and sought for a popular matching by augmenting the instance with minimum costs. Unfortunately, both extensions have been shown to be NP-hard. The concept of unpopularity [10, 19] has also been investigated.

1.3 Problem Definition

An input instance is a bipartite graph $G = (A \cup P, E)$, where A is the set of *applicants*, P is the set of *posts*, and $\{E_1, E_2, ..., E_\ell\}$ is a partition of the edge set E. We call such a graph the *instance graph* or the *instance*. The cardinalities of $A \cup P$ and E are denoted by n and m, respectively. For an instance graph G, its *rank* function $r_G : E \mapsto \mathbb{N}$ is defined as $r_G(e) = i$ if $e \in E_i$. We also denote $r_G(e)$ as $r_G(a, p)$ if e has end vertices a and p, and r_G is abbreviated as r if there is no ambiguity. We say that *a prefers p* to p' if $r_G(a, p) < r_G(a, p')$. For any applicant a, a's preference list can be viewed as the ordered sequence of posts adjacent to a. All the preference lists have to be nonempty and strictly ordered. For an instance graph $G = (A \cup P, E)$, if $p \in P$ is the first post on some applicant's preference list, we say that p is an *f-post* with respect to G and denote the set of all *f-posts* of G by F_G. Let $f_G(a)$ denote the first post on a's preference list, and let $s_G(a)$, which is called an *s-post*, denote the first non-*f-post* on a's preference list. As in [3], there is a unique last resort post appended at the end of every applicant's preference list. By appending last resort posts, every applicant has at least one choice that nobody else competes with him/her.

We say that an instance $H = (A_H \cup P, E_H)$ is a *popular condensation* of a given instance graph $G = (A \cup P, E)$ if H admits a popular matching, $A_H \subseteq A$, $E_H = \{(a, p) \in E : a \in A_H\}$, and E_H is partitioned such that $r_H(e) = r_G(e)$ for all $e \in E_H$. The set $A - A_H$ is called a *condensing set* of G with respect to H. The set of all popular condensations of G is denoted by Cond(G). Figure 1(b) is an illustration of the above definition for the input instance shown in Figure 1(a); from the examples, we can see that the s-posts may alter after some applicants are neglected. A formal problem definition is stated below.

$$\begin{array}{l}
a_1 : \underline{\underline{p_1}}\ p_2\ \underline{p_3}\ p_4\ p_5 \\
a_2 : \underline{\underline{p_1}}\ p_2\ p_4\ \underline{p_3}\ p_5 \\
a_3 : \underline{\underline{p_2}}\ p_4\ \underline{p_3}\ p_1\ p_5 \\
a_4 : \underline{\underline{p_2}}\ \underline{p_3}\ p_1\ p_4 \\
a_5 : \underline{\underline{p_4}}\ \underline{p_3}\ p_1
\end{array}
\qquad\qquad
\begin{array}{l}
a_1 : \underline{\underline{p_1}}\ p_2\ \underline{p_3}\ p_4\ p_5 \\
a_2 : \underline{\underline{p_1}}\ p_2\ \underline{p_4}\ p_3\ p_5 \\
a_3 : \underline{\underline{p_2}}\ \underline{p_4}\ p_3\ p_1\ p_5 \\
a_4 : \underline{\underline{p_2}}\ \underline{p_3}\ p_1\ p_4
\end{array}$$

(a) G (b) H_1

Fig. 1. The f-posts and s-posts of a graph and its popular condensation. The subfigures (a) and (b) are the preference lists w.r.t. G and H_1, respectively, where H_1 is a popular condensation of G with $\{a_5\}$ being a condensing set. The double-underlined posts represent f-posts and the underlined posts represent s-posts on each instance graph. We can see that $M_1 = \{(a_1, p_1), (a_2, p_4), (a_3, p_2), (a_4, p_3)\}$ is a popular matching on H_1.

Problem 1 (The Popular Condensation Problem). Given an instance graph $G = (A \cup P, E)$, the popular condensation problem asks for a popular matching of a popular condensation $H^* = (A_{H^*} \cup P, E_{H^*})$ of G satisfying

$$H^* = \arg \max_{H \in \mathrm{Cond}(G)} |A_H|.$$

For an instance graph G, the popular condensation H^* of G we seek for in Problem 1 is called an *optimal popular condensation*, and the *condensation number* $\mu(G)$ is defined as $|A| - |A_{H^*}|$.

Notice that when an instance consists of only one applicant, the instance must admit a popular matching since each preference list is nonempty. Therefore, we have the following property, which implies that the popular matching sought in Problem 1 always exists.

Property 1. For an instance graph G, $\mathrm{Cond}(G)$ is nonempty.

The rest of this paper is organized as follows. First, we define some basic notations in Section 2. Then, we give our solution to the popular condensation problem in Section 3. Finally, concluding remarks are given in Section 4.

2 Preliminaries

In this section, we define the notation and summarize some useful previous results. By the definitions of f-post and s-post, Property 2 holds immediately.

Property 2. For any instance graph G, if $r_G(a, f_G(a)) < r_G(a, p) < r_G(a, s_G(a))$, then $p \in F_G$.

The *reduced graph* $G' = (A \cup P, E')$ is a subgraph of G containing only two edges, $(a, f_G(a))$ and $(a, s_G(a))$, for each applicant a (see Figure 2). The *neighborhood* $N_G(a)$ of a vertex a in G is the set of vertices adjacent to a. For simplicity, we denote $\bigcup_{x \in X} N_G(x)$ as $N_G(X)$. We quote Lemma 1, which will be used later to evaluate whether a matching is popular. For any matching M of G', M is said to be *applicant-complete* if all applicants are matched in M.

Lemma 1 (See Theorem 2.1 in [3]). *M is a popular matching of G if and only if*

(i) every f-post with respect to G is matched in M, and
(ii) M is an applicant-complete matching of the reduced graph G'.

Lemma 2. *Let $G' = (A \cup P, E')$ be the reduced graph of an instance graph G. If there is a component $C_i = (A_i \cup P_i, E_i')$ of G' satisfying $|A_i| > |P_i|$, where $A_i \subseteq A, P_i \subseteq P$, and $E_i' = \{(a, p) \in E' : a \in A_i \text{ and } p \in P_i\}$, then G admits no popular matching.*

Proof. Since C_i contains more applicants than posts, applicant-complete matchings do not exist in G'. By Lemma 1, G admits no popular matching. □

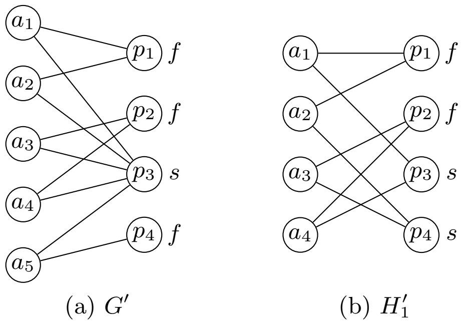


Fig. 2. The reduced graphs of G and H_1 in Figure 1, denoted by G' and H'_1, respectively. Note that $H'_1 \nsubseteq G'$

3 Popular Condensations

This section focuses on the popular condensation problem. First, for a given instance graph, we give a linear-time algorithm to find a popular matching of a popular condensation H. Then, we prove that H is an optimal popular condensation. A sketch of our algorithm is given below.

Step 1. For a given instance graph G, compute its reduced graph G'.
Step 2. Compute $\tilde{G}$ by removing a matching M^* from G'.
Step 3. Find a maximum matching $\tilde{M}$ of $\tilde{G}$.
Step 4. Modify $M^* \cup \tilde{M}$ to be the requested popular matching.

We name the algorithm as PopularCondensation, and the pseudocode is given in Algorithm 1.

Basically, line 1 computes the reduced graph of an input instance as in [3]. Lines 3 to 6 then iteratively match a post p with degree 1 to the applicant a adjacent to p, and remove both a and p (no succeeding augmenting path can involve p). After that, posts with degree 0 are removed by lines 7 to 8 (up to now, the modified graph of G' is $\tilde{G}$). Lines 9 to 16 compute a matching of $\tilde{G}$ by repeatedly taking an unmatched post p if any, and finding a path by arbitrarily walking from p till encountering a matched post. All the edges passed by are removed, and two ends of any odd edge are matched. These steps are taken for each component of $\tilde{G}$ (the outer while loop). Line 17 puts all unmatched applicants into a set D. For each f-post p w.r.t. G, if it is not matched, lines 18 to 20 *promote* an applicant a to p, i.e., rematching a to p, where $f_G(a) = p$. Finally, line 21 outputs a matching together with the set D.

Similar to [3], PopularCondensation simplifies the discussions by restricting our attention to the reduced graph. Each applicant in the reduced graph has degree equal to two. For bipartite graphs with such a restriction, the following lemma holds.

Algorithm 1. PopularCondensation($G = (A \cup P, E)$)

```
1  $G'$ := the reduced graph of $G$;
2  $M := \emptyset$;
3  while some post $p$ in $G'$ has degree 1 do
4      $a$ := the applicant which is adjacent to $p$;
5      $M := M \cup \{(a, p)\}$;
6      $G' := G' - \{a, p\}$; /* remove $a$, $p$, and related edges */
7  while some post $p$ in $G'$ has degree 0 do
8      $G' := G' - \{p\}$ ; /* remove $p$ */
9  while some post $p$ in $G'$ is not in $M$ do
10     do
11         $a$ := any applicant adjacent to $p$;
12         $M := M \cup \{(a, p)\}$;
13         $G' := G' - \{(a, p)\}$; /* remove edge (a,p) */
14         $p$ := the post adjacent to $a$;
15         $G' := G' - \{(a, p)\}$; /* remove edge (a,p) */
16     while $p$ is not in $M$;
17 $D$ := the set of applicants not in $M$;
   /* $G - D$ is the selected popular condensation */
18 foreach $f$-post $p$ w.r.t. $G$ which is not in $M$ do
19     $a$ := any applicant satisfying $r_G(a, p) = 1$;
20     promote $a$ to $p$ in $M$;
21 return $M$, $D$;
```

Lemma 3. *Let $G = (A \cup P, E)$ be a bipartite graph where all applicants have degree 2. If there is a component $C = (A_c \cup P_c, E_c)$ of G satisfying $|A_c| < |P_c|$, then either $\exists p \in P_c$ such that p has degree 1 or C contains only an isolated vertex.*

Due to Lemma 3, after executing lines 3 to 8, all the components $\tilde{C}_i = (\tilde{A}_i \cup \tilde{P}_i, \tilde{E}_i)$ of the intermediate graph $\tilde{G}$ satisfy $|\tilde{P}_i| \leqslant |\tilde{A}_i|$. Intuitively, the partite set $\tilde{A} = \bigcup_i \tilde{A}_i$ contains the applicants who compete for the same posts, and some of them might not be matched with any post. Let $\tilde{M}$ be the matching of $\tilde{G}$ produced by lines 9 to 16 of PopularCondensation.

Lemma 4. *$\tilde{M}$ is a maximum matching of $\tilde{G}$, which includes all posts in $\tilde{P}$.*

Proof. By Lemma 3, we have that $|\tilde{P}| \leqslant |\tilde{A}|$. Thus, it suffices to show that all posts in $\tilde{P}$ are matched. Suppose to the contrary that a post p is not matched. Since $p \in \tilde{P}$, p has degree at least 2. Let a_1 be the first matched neighbor of p. According to lines 9 to 16 of PopularCondensation, there must be another neighbor of p that is matched with p, which is a contradiction. $\square$

Let M^* be the matching produced by lines 3–8. Together with Lemma 4, the following lemma holds immediately.

Lemma 5. *$M^* \cup \tilde{M}$ is a maximum matching of G'.*

We prove in the following lemma that the set of unmatched applicants D is a condensing set of G, and M is a popular matching of the corresponding popular condensation.

Lemma 6. *For an instance graph* $G = (A \cup P, E)$, PopularCondensation(G) *computes a popular matching of a popular condensation of* G.

Proof. Let M and D be the output of PopularCondensation(G), and let $H = (A_H \cup P, E_H)$ be an instance graph where $A_H = A - D$, $E_H = E - \{(a,p) : a \in D\}$, and $r_H(e) = r_G(e)$. For the reduced graphs G' and H', we claim the following:

- **claim 1.** H' is a subgraph of G', and $F_H = F_G$.
  Clearly, each applicant $a \in A_H$ has the same preference list w.r.t. G and H. Denote two subsets $F_G^1 = \{f_G(a) : a \in A - A_H\}$ and $F_G^2 = \{f_G(a) : a \in A_H\}$ of F_G. For each post $p \in F_G^2$, there is always an applicant $a \in A_H$ that regards p as the first preferred post; hence, $p \in F_H$ holds intuitively. For each post $p \in F_G^1$, p must be matched in $M^* \cup \tilde{M}$, otherwise $M^* \cup \tilde{M} \cup \{(a,p) : a \in A - A_H, p = f_G(a)\}$ can be a larger matching, a contradiction. However, if an f-post p is matched to an applicant a_p in $M^* \cup \tilde{M}$, a_p must be put into A_H, and $p \in F_G^2$ holds. It follows that $F_G^1 \subseteq F_G^2 = F_G$, and $F_G = F_H$. For each applicant $a \in A_H$, a's preference list remains the same as that in G, and all f-posts w.r.t. G are still f-posts w.r.t. H; this implies that $s_H(a) = s_G(a)$ holds for each applicant $a \in A_H$. We conclude that H' is a subgraph of G'.
- **claim 2.** The output matching M is a popular matching of H.
  By Lemma 1, it suffices to show that M is an applicant-complete matching of H' and that every f-post w.r.t. H is matched in M. By the definition of D, M is applicant-complete. Together with claim 1, lines 18 to 20 promote every f-post w.r.t. H to some applicant, and therefore claim 2 holds.

□

In the proof of Lemma 6, it has been shown that D is a condensing set of G. Thus, $\mu(G) \leqslant |D|$. To prove the correctness of PopularCondensation, it remains to show that $\mu(G) = |D|$. We derive this equality by showing that

$$\alpha \leqslant \mu(G) \leqslant |D| \leqslant \alpha,$$

for some value α. Let $C_i' = (A_i' \cup P_i', E_i')$ be the components of G' satisfying $|P_i'| \leqslant |A_i'|$. We define the "trouble" subgraph of G' as $\bigcup_i C_i' = (A_T \cup P_T, E_T)$. We shall prove that $\alpha = |A_T| - |P_T|$ by showing that $|D| = |A_T| - |P_T|$ and $|A_T| - |P_T| \leqslant \mu(G)$ in Lemma 7 and Theorem 1, respectively.

Lemma 7. $|D| = |A_T| - |P_T|$.

Proof. Let $\tilde{\mathcal{C}} = \{\tilde{C}_1, \tilde{C}_2, \ldots, \tilde{C}_s\}$ be the components of $\tilde{G}$, and let $\mathcal{C}' = \{C_1', C_2', \ldots, C_t'\}$ be the components of G' satisfying $|P_i'| \leqslant |A_i'|$, where $C_i' = (A_i' \cup P_i', E_i')$. By Lemma 3, we have that $s = t$, and without loss of generality assume that $\tilde{C}_i$ is a subgraph of C_i'. Let $\tilde{C}_i = (\tilde{A}_i \cup \tilde{P}_i, \tilde{E}_i)$. According to lines 3 to 6 of

PopularCondensation, $\tilde{C}_i$ is reduced from C'_i by iteratively removing one vertex from each partite set. As a result, the following property holds:

$$|\tilde{A}_i| - |\tilde{P}_i| = |A'_i| - |P'_i|, \forall \tilde{C}_i \in \tilde{\mathcal{C}} \text{ and } C'_i \in \mathcal{C}'. \tag{1}$$

According to line 17, D contains the applicants that are not in M. By Lemma 4, we have that $|D| = |\tilde{A}| - |\tilde{P}|$. It follows that

$$\begin{aligned} |D| &= |\tilde{A}| - |\tilde{P}| = \sum_{\{i:\tilde{C}_i\in\tilde{\mathcal{C}}\}} \left(|\tilde{A}_i| - |\tilde{P}_i|\right) \\ &\underset{(1)}{=} \sum_{\{i:C'_i\in\mathcal{C}'\}} (|A'_i| - |P'_i|) \\ &= \sum_{i\in I_1} (|A'_i| - |P'_i|) + \sum_{i\in I_2} (|A'_i| - |P'_i|) \\ &= |A_T| - |P_T| + 0 = |A_T| - |P_T|, \end{aligned}$$

where $I_1 = \{i : C'_i \in \mathcal{C}' \text{ and } |P'_i| < |A'_i|\}$ and $I_2 = \{i : C'_i \in \mathcal{C}' \text{ and } |P'_i| = |A'_i|\}$. □

To deal with a lower bound on $\mu(G)$, we consider the condensing set Δ of G w.r.t. an arbitrary popular condensation H. By the definition of popular condensations, $r_H(a, p) = r_G(a, p)$ for all $a \in A_H$. It implies that a has the same preference list w.r.t. both G and H. We state this observation in Lemma 8.

Lemma 8. *Let $G = (A \cup P, E)$ be an instance graph and $H = (A_H \cup P, E_H)$ be a popular condensation of G. We have that $f_H(a) = f_G(a)$ for all applicants $a \in A_H$.*

The set Δ can be partitioned into Δ_1 and Δ_2, where

$$\Delta_1 = \{\delta : \delta \in \Delta \text{ and } \delta \in A_T\} \text{ and } \Delta_2 = \{\delta : \delta \in \Delta \text{ and } \delta \notin A_T\}.$$

As illustrated in Figure 1, the set of f-posts (also, the s-posts) may be different w.r.t. G and H. Fortunately, we show in Lemma 9 that for any applicant, if there is a "new neighbor" p in H', then p has to be $f_G(\delta)$ for some $\delta \in \Delta$.

Lemma 9. *For any applicant $a \in A_H$, if $\exists p \in N_{H'}(a) - N_{G'}(a)$, then $\exists \delta \in \Delta$ such that $p = f_G(\delta)$.*

Proof. Consider the following two cases: (i) $p = f_H(a)$ and (ii) $p = s_H(a)$. Case (i) is not possible since otherwise, by Lemma 8, $p = f_G(a) \in N_{G'}(a)$. For case (ii), by Property 2, we have that $s_H(a) = f_G(a^*)$ for some $a^* \in \Delta$. □

Based on Lemma 9, we give a lower bound on $\mu(G)$.

Theorem 1. *For an instance graph G, $\mu(G) = |A_T| - |P_T|$.*

Proof. In Lemma 6 and Lemma 7, it has been shown that $\mu(G) \leqslant |A_T| - |P_T|$. Therefore, in the following, we show that $|A_T| - |P_T| \leqslant \mu(G)$. Since H admits a popular matching, by Lemma 2, it follows that

$$|X| \leqslant |N_{H'}(X)|, \forall X \subseteq A_H. \tag{2}$$

For each $a \in A_T - \varDelta_1$, by Lemma 9, we have that $N_{H'}(a) \subseteq P_T \cup \{f_G(\delta) : \delta \in \varDelta\} = P_T \cup \{f_G(\delta) : \delta \in \varDelta_2\}$. Thus, $N_{H'}(A_T - \varDelta_1) \subseteq P_T \cup \{f_G(\delta) : \delta \in \varDelta_2\}$.

By integrating the above arguments, we have that

$$\begin{aligned} |A_T| - |\varDelta_1| &= |A_T - \varDelta_1| \\ &\underset{(2)}{\leqslant} |N_{H'}(A_T - \varDelta_1)| \\ &\leqslant |P_T| + |\{f_G(\delta) : \delta \in \varDelta_2\}| \\ &\leqslant |P_T| + |\varDelta_2|, \end{aligned}$$

which implies that $|A_T| - |P_T| \leqslant |\varDelta_1| + |\varDelta_2| = |\varDelta|$. Thus, $|A_T| - |P_T| \leqslant \mu(G)$. □

We conclude this section by proving Theorem 2.

Theorem 2. *The popular condensation problem can be solved in $O(n+m)$ time.*

Proof. Lemma 6, Lemma 7, and Theorem 1 demonstrate that algorithm PopularCondensation solves the popular condensation problem. Here, we take a look at the time complexity of the algorithm. It takes $O(n + m)$ time to construct the reduced graph in line 1. There are $O(n)$ edges in the reduced graph. By properly recording degree information, lines 3–8 can be done in $O(n)$ time. Since p takes an arbitrary neighbor a in line 11, lines 9–16 also can be done in $O(n)$ time. Lastly, line 17 and lines 18–20 take $O(n)$ time intuitively. Therefore, the popular condensation problem can be solved in $O(n + m)$ time. □

4 Concluding Remarks

In this paper, we provide a solution to find relatively good matchings in practice when the instance admits no popular matching, and we show that the popular condensation problem can be solved in linear time. For an instance $G = (A \cup P, E)$, according to the proposed algorithm, we get an upper bound for the condensation number $\mu(G)$. Furthermore, we derive that the upper bound, $|A_T| - |P_T|$, which we get for $\mu(G)$ is at the same time a lower bound for $\mu(G)$.

We note that in the problem of copying posts to get a popular matching, Kavitha and Nasre gave the exact value on the minimum number of copies needed [15]. We denote the value by $\nu(G)$, and it can be easily shown by Lemma 7 that $\nu(G) = |A_T| - |P_T|$ if the input preference lists are strictly ordered. As a result, we conclude this paper by the following theorem.

Theorem 3. *For any instance graph G, $\mu(G) = \nu(G)$.*

Acknowledgements. Yen-Wei Wu, Wei-Yin Lin, and Kun-Mao Chao were supported in part by NSC grants 100-2221-E-002-131-MY3 and NSC 101-2221-E-002-063-MY3, and Hung-Lung Wang was supported in part by the NSC grant 101-2221-E-141-002- from the National Science Council, Taiwan.

References

1. Abdulkadiroğlu, A., Sönmez, T.: Random serial dictatorship and the core from random endowments in house allocation problems. Econometrica 66(3), 689–701 (1998)
2. Abraham, D.J., Cechlárová, K., Manlove, D.F., Mehlhorn, K.: Pareto optimality in house allocation problems. In: Fleischer, R., Trippen, G. (eds.) ISAAC 2004. LNCS, vol. 3341, pp. 3–15. Springer, Heidelberg (2004)
3. Abraham, D.J., Irving, R.W., Kavitha, T., Mehlhorn, K.: Popular matchings. SIAM Journal on Computing 37(4), 1030–1045 (2007)
4. Abraham, D.J., Kavitha, T.: Dynamic Matching Markets and Voting Paths. In: Arge, L., Freivalds, R. (eds.) SWAT 2006. LNCS, vol. 4059, pp. 65–76. Springer, Heidelberg (2006)
5. Bartholdi, J.J., Tovey, C.A., Trick, M.A.: How hard is it to control an election? Mathematical and Computer Modeling 16(8/9), 27–40 (1992)
6. Biró, P., Irving, R.W., Manlove, D.F.: Popular Matchings in the Marriage and Roommates Problems. In: Calamoneri, T., Diaz, J. (eds.) CIAC 2010. LNCS, vol. 6078, pp. 97–108. Springer, Heidelberg (2010)
7. Gardenfors, P.: Match Making: assignments based on bilateral preferences. Behavioural Sciences 20, 166–173 (1975)
8. Huang, C.-C., Kavitha, T.: Near-popular matchings in the roommates problem. SIAM Journal on Discrete Mathematics 27(1), 43–62 (2013)
9. Huang, C.-C., Kavitha, T.: Popular matchings in the stable marriage problem. Information and Computation 222, 180–194 (2013)
10. Huang, C.-C., Kavitha, T., Michail, D., Nasre, M.: Bounded Unpopularity Matchings. In: Gudmundsson, J. (ed.) SWAT 2008. LNCS, vol. 5124, pp. 127–137. Springer, Heidelberg (2008)
11. Irving, R.W., Kavitha, T., Mehlhorn, K., Michail, D., Paluch, K.: Rank-maximal matchings. ACM Transactions on Algorithms 2(4), 602–610 (2006)
12. Kavitha, T.: Popularity vs maximum cardinality in the stable marriage setting. In: Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 123–134 (2012)
13. Kavitha, T., Mestre, J., Nasre, M.: Popular mixed matchings. Theoretical Computer Science 412(24), 2679–2690 (2011)
14. Kavitha, T., Nasre, M.: Optimal popular matchings. Discrete Applied Mathematics 157(14), 3181–3186 (2009)
15. Kavitha, T., Nasre, M.: Popular matchings with variable item copies. Theoretical Computer Science 412, 1263–1274 (2011)
16. Kavitha, T., Nasre, M., Nimbhorkar, P.: Popularity at Minimum Cost. In: Cheong, O., Chwa, K.-Y., Park, K. (eds.) ISAAC 2010, Part I. LNCS, vol. 6506, pp. 145–156. Springer, Heidelberg (2010)
17. Mahdian, M.: Random popular matchings. In: Proceedings of the 7th ACM Conference on Electronic Commerce, pp. 238–242 (2006)

18. Manlove, D.F., Sng, C.T.S.: Popular Matchings in the Capacitated House Allocation Problem. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 492–503. Springer, Heidelberg (2006)
19. McCutchen, R.M.: The Least-Unpopularity-Factor and Least-Unpopularity-Margin Criteria for Matching Problems with One-Sided Preferences. In: Laber, E.S., Bornstein, C., Nogueira, L.T., Faria, L. (eds.) LATIN 2008. LNCS, vol. 4957, pp. 593–604. Springer, Heidelberg (2008)
20. McDermid, E., Irving, R.W.: Popular matchings: structure and algorithms. Journal of Combinatorial Optimization 22(3), 339–358 (2011)
21. Mestre, J.: Weighted Popular Matchings. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4051, pp. 715–726. Springer, Heidelberg (2006)
22. Paluch, K.: Popular and Clan-Popular b-Matchings. In: Chao, K.-M., Hsu, T.-s., Lee, D.-T. (eds.) ISAAC 2012. LNCS, vol. 7676, pp. 116–125. Springer, Heidelberg (2012)

Maximum st-Flow in Directed Planar Graphs via Shortest Paths

Glencora Borradaile[1] and Anna Harutyunyan[2,⋆]

[1] Oregon State University
[2] Vrije Universiteit Brussel

Abstract. In this paper, we give a correspondence between maximum flows and shortest paths via duality in *directed* planar graphs with no constraints on the source and sink.

1 Introduction

The asymptotically best algorithm for max st-flow in directed planar graphs is the $O(n \log n)$-time *leftmost-paths algorithm* due to Borradaile and Klein [4], a generalization of the seminal *uppermost-paths algorithm* by Ford and Fulkerson for the st-planar case [6]. Both algorithms augment $O(n)$ paths, with each augmentation implemented in $O(\log n)$ time. However, this bound is achieved using the overly versatile dynamic-trees data structure [11] in Borradaile and Klein's algorithm, and only priority queues in the st-planar case [7]; priority queues are arguably simpler and more practical than dynamic trees [13]. The reason priority queues are sufficient for the st-planar case is due the equivalence between flow and shortest-paths problems via duality. We conjecture that an augmenting-paths algorithm for the more general problem can be implemented via $O(n)$ priority-queue operations. We make progress toward this goal by showing that maximum flow in the general case is equivalent to computing shortest paths in a covering graph, even though, algorithmically, we do not improve on the algorithm of Johnson and Venkatesan [9].

Background. For omitted proofs and full definitions and an additional shortest-path based algorithm, please see the full version of this paper [2]. For a full background on planarity and flow, see Refs. 1 and 10.

For a path P, left(P) (right(P)) is the maximal subset of the darts whose head or tail (but not both) is in P, and who enter or leave P from the left (right). We define the graph $G \not\!\!\times P$ as the graph *cut open along* P. $G \not\!\!\times P$ contains two copies of P, P_L and P_R, so that the edges in left(P) are adjacent to P_R and the edges in right(P) are adjacent to P_L. The parameter ϕ is the minimum number of faces that any s-to-t curve must pass through [10].

An *excess (deficit)* vertex is that with more flow entering (leaving) than leaving (entering). A flow is *maximum* if and only if there are no residual source-to-sink paths [6]. A pseudoflow is *maximum* if and only if there are no residual paths

⋆ Work done while at Oregon State University.

© Springer-Verlag Berlin Heidelberg 2013

from a source or excess vertex to a deficit or sink vertex [8]. The notion of *leftmost* flows is essential in understanding this paper, and is covered in Ref. 1; a flow is leftmost if it admits no clockwise residual cycles. The following lemma follows directly from definitions of leftmost and the following theorem provides structural insight into leftmost paths and flows.

Lemma 1. *A leftmost circulation can be decomposed into a set of flow-carrying clockwise simple cycles.*

Theorem 1. *Let L be the leftmost residual s-to-t-path in G w.r.t. c.w. acyclic capacities* $\mathbf{c}$. *Let* $\mathbf{f}$ *be any st-flow (of any value). Then no simple s-to-t flow path crosses L from the left to right.*

Infinite Covers. Embed G on a sphere and remove the interiors of f_t and f_s. The resulting surface is a cylinder with t and s embedded on opposite ends. The repeated drawing of G on the universal cover of this cylinder [12] defines a *covering graph*[1] $\mathcal{G}$ of G. For a subgraph X of G, we denote the subgraph of $\mathcal{G}$ whose vertices and darts map to X by $\mathcal{G}[X]$. We say $\bar{X} \subset \mathcal{G}[X]$ is a *copy* of X if $\bar{X}$ maps bijectively to X. We number the copies of a vertex u from left to right: $\mathcal{G}[u] = \{\dots u^{-1}, u^0, u^1, \dots\}$, picking u^0 arbitrarily. We say that $\bar{X}$ is an *isomorphic* copy of X if $\bar{X}$ is isomorphic to X.[2] For a simple s-to-t path P in G, we denote by P^i the isomorphic copy of P in $\mathcal{G}[P]$ that ends at t^i. $G_P^i \cup P^{i+1}$ is the finite component of subgraph of $\mathcal{G} \mathbin{✁} P^i \mathbin{✁} P^{i+1}$. Lemma 2 relates clockwise cycles in $\mathcal{G}$ and G and Lemma 3 is key to bounding the size of the finite portion of $\mathcal{G}$ required by our algorithm:

Lemma 2. *The mapping of the following into G contains a clockwise cycle: Any u^i-to-u^j path in $\mathcal{G}$ for $u^i \in G_P^i$, $u^j \in G_P^j$, and $i < j$ and any simple clockwise cycle in $\mathcal{G}$.*

Lemma 3 (Pigeonhole). *Let P be a simple path in G. Then $\bar{P}$ contains a dart of at most $\phi + 2$ copies of G in $\mathcal{G}$. If P may only use s and t as endpoints, then $\bar{P}$ contains darts in at most ϕ copies of G in $\mathcal{G}$.*

2 Maximum Flow, Shortest Paths Equivalences

Starting with c.w. acyclic capacities $\mathbf{c}$, we compute the leftmost maximum flow in a finite portion of $\mathcal{G}$ (containing k copies of G), $\mathcal{G}_k$, via embedding two additional vertices T (above) and S (below) the cover, and computing the leftmost max ST-flow $\mathbf{f}^{ST}$ via the priority-queue implementation of Ford and Fulkerson's uppermost-paths algorithm (Section 2.1). Note that shortest-paths/priority-queue based implementations always produce leftmost flows. From $\mathbf{f}^{ST}$ we show how to extract the *value* of max st-flow $|\mathbf{f}|$ (Section 2.2). Using this value we are able to modify the capacities $\mathbf{c}^{ST}$ in a way that allows to extract $\mathbf{f}$ from $\mathbf{f}^{ST}$ (Section 2.3). While this method requires a factor k additional space, we believe this can be overcome in future work (Section 2.4).

[1] This is similar to a cover used by Erickson analysis [5]; we remain in the primal.

[2] Note that an isomorphic copy need not exist, e.g., the boundary of f_s.

2.1 The Finite Cover

Let L be the leftmost residual s-to-t path in G and let $\mathbf{f}$ be the (acyclic) leftmost maximum st-flow in G. Let $\mathcal{G}_k$ be the finite component of $\mathcal{G} \not\curlywedge L^0 \not\curlywedge L^k$, made of k copies of G (plus an extra copy of L). We start by relating a max multi-source, multi-sink maximum flow in $\mathcal{G}_k$ ($\mathbf{f}_1$) to a max pseudoflow $\mathbf{f}_0$ (Lemma 4), and, in turn relate $\mathbf{f}_0$ to $\mathbf{f}$ (Lemma 5). This will illustrate that the flow in the central copy of $\mathcal{G}_k$ is exactly $\mathbf{f}$. However, $\mathbf{f}_1$ is not necessarily leftmost and so, we cannot necessarily compute it. We relate $\mathbf{f}_1$ to the leftmost max ST-flow in $\mathcal{G}_k^{ST}$ (which we can compute), in Section 2.2, from which we can compute the *value* of $\mathbf{f}$.

Let $\mathbf{f}_0$ be a flow assignment for $\mathcal{G}_k$ given by $\mathbf{f}_0[\bar{d}] = \mathbf{f}[d]$, $\forall \bar{d} \in \mathcal{G}[d]$. We overload $\mathbf{c}$ to represent capacities in both G and $\mathcal{G}_k$, where capacities in $\mathcal{G}_k$ are inherited from G in the natural way. In $\mathcal{G}_k$ and G, we use residual to mean w.r.t. $\mathbf{c}_{\mathbf{f}_0}$ and $\mathbf{c}_{\mathbf{f}}$, respectively.

Lemma 4. *For $k > \phi + 2$, $\mathbf{f}_0$ is a maximum pseudoflow with excess vertices on L^0 and deficit vertices on L^k.*

Proof. Since $\mathbf{f}_0$ is balanced for all vertices in $\mathcal{G}_k$ except those on L^0 and L^k and it follows from Theorem 1 that V^+ resp. V^- belong to L^0 resp. L^k, where V^+ resp. V^- denote the set of excess resp. deficit vertices. Since a source-to-sink path in $\mathcal{G}_k$ maps to an s-to-t path in G, it remains to show that there are no V^+-to-T, S-to-V^- or V^+-to-V^- residual paths. By the Pigeonhole Lemma and Part 2 of Lemma 2, the last case implies a clockwise residual cycle in G, contradicting $\mathbf{f}$ being leftmost. The first two cases are similar, we only prove the first case here.

Consider the flow assignment for $\mathcal{G}$: $\mathbf{f}'[\bar{d}] = \mathbf{f}[d]$, $\forall \bar{d} \in \mathcal{G}[d]$. For $v^+ \in V^+$ to be an excess vertex, there must be a v-to-t flow path Q in $\mathbf{f}$ where v is the vertex in G that v^+ maps to. There is a copy $\bar{Q}$ of Q in $\mathcal{G}$ that starts at v^+, and by Theorem 1 is left of L^0. For a contradiction, let R be a v^+-to-t^i residual path, for some $t^i \in T$. Then, $rev(Q) \circ R$ is a residual t^j-to-t^i path in $\mathcal{G}$ (w.r.t. $\mathbf{f}'$), $j \leq i$. If $j = 0$, $\bar{Q} \circ rev(L^0[v^+, t^0])$ is a clockwise cycle, which, by Part 2 of Lemma 2, implies a clockwise cycle in G; contradicting the leftmost-ness of L. If $j < i$, by Part 1 of Lemma 2, $rev(Q) \circ R$ implies a clockwise residual cycle in G, contradicting the leftmost-ness of $\mathbf{f}$. □

Lemma 5. *There is a maximum ST-flow $\mathbf{f}_1$ in $\mathcal{G}_k$ that is obtained from $\mathbf{f}_0$ by removing flow on darts in the first and last ϕ copies of G in $\mathcal{G}_k$. Further, the amount of flow into sink t^i for $i \leq k - \phi$ and the amount of flow out of source s^j for $j \geq \phi$ is the same in $\mathbf{f}_0$ and $\mathbf{f}_1$.*

Proof. Since $\mathbf{f}_0$ is an acyclic max pseudoflow, it can be converted to a max flow by removing flow from source-to-excess flow paths and deficit-to-sink flow paths [8]. Let P be such a flow path. P must map to a simple path in G. By the Pigeonhole Lemma, P must be contained within ϕ copies of G. This proves the first part of the lemma. Since P cannot start at s^j for $j \geq \phi$ without going through more than ϕ copies (and likewise, P cannot end at t^i for $i \leq k - \phi$), the second part of the lemma follows. □

2.2 Value of the Maximum Flow

In the next lemma, we prove that from $\mathbf{f}^{ST}$, the leftmost maximum ST-flow in $\mathcal{G}_k^{ST}$, we can extract $|\mathbf{f}|$, the value of the maximum st-flow in G.

Lemma 6. *For $k \geq 4\phi$, the amount of flow through $s^{2\phi}$ in $\mathbf{f}^{ST}$ is $|\mathbf{f}|$.*

Proof. We show that the amount of flow leaving $s^{2\phi}$ in $\mathbf{f}^{ST}$ is the same as in $\mathbf{f}_1$. By Lemma 5, the amount of flow leaving $s^{2\phi}$ is the same in $\mathbf{f}_1$ as $\mathbf{f}_0$ which is the same as the amount of flow leaving s in $\mathbf{f}$; this proves the lemma.

First extend $\mathbf{f}_1$ into a (max) ST-flow, $\mathbf{f}_1^{ST}$, in $\mathcal{G}_k^{ST}$ in the natural way. To convert $\mathbf{f}_1^{ST}$ into a *leftmost* flow, we must saturate the clockwise residual cycles with a c.w. circulation. By Lemma 1 and for a contradiction, there then must be c.w. simple cycle C that changes the amount of flow through $s^{2\phi}$. C must go through S, C is residual w.r.t. $\mathbf{c}_{\mathbf{f}_1^{ST}}$, and cannot visit T, therefore C must contain a s^i-to-$s^{2\phi}$ residual path P that is in $\mathcal{G}_k$. Since C is c.w. , $i < 2\phi$. Suppose P does not use a dart in the first or last ϕ copies of G in $\mathcal{G}_k$. Then P must map to a set P' of darts in G which, by Lemma 5 are residual w.r.t. $\mathbf{c}_\mathbf{f}$. By Part 1 of Lemma 2, P' contains a clockwise cycle, contradicting the leftmostness of $\mathbf{f}$. It follows that P must cross either from the ϕ^{th} copy to $s^{2\phi}$ or from $s^{2\phi}$ to the $3\phi^{th}+1$ copy. Then, by the Pigeonhole Lemma, P contains a subpath Q that goes from $\bar{v}$ to $\bar{v}'$, where $\bar{v}$ is an earlier copy of a vertex v than $\bar{v}'$, and neither are in the first or last ϕ copies. By Part 1 of Lemma 2, the map of Q contains a clockwise cycle in G. Since Q does not contain darts in the first or last ϕ copies of G, by Lemma 5, this cycle is residual w.r.t. $\mathbf{c}_\mathbf{f}$ in G, again contradicting that $\mathbf{f}$ is leftmost. □

2.3 Maximum Flow

Now, suppose we know $|\mathbf{f}|$ (as per Lemma 2.2). We change the capacities of the arcs into T and out of S in $\mathcal{G}_k^{ST}$ to $|\mathbf{f}|$, resulting in capacities $\mathbf{c}^{|\mathbf{f}|}$. Now, $\mathbf{f}_1^{ST}$, respects $\mathbf{c}^{|\mathbf{f}|}$ since, by Lemmas 4 and 5, the amount of flow leaving any source or entering any sink in $\mathbf{f}_1$ is at most $|\mathbf{f}|$. The proof of the following is similar to that of Lemma 6:

Lemma 7. *$\mathbf{f}_1^{ST}$ can be converted into a leftmost maximum ST-flow $\mathbf{f}^{|\mathbf{f}|}$ for the capacities $\mathbf{c}^{|\mathbf{f}|}$ while not changing the flow on darts in the first or last 2ϕ copies of G in $\mathcal{G}_k$.*

To summarize, Lemmas 5 and 7 guarantee that the maximum leftmost ST-flow, $\mathbf{f}^{|\mathbf{f}|}$, in $\mathcal{G}_k^{ST}$ given capacities $\mathbf{c}^{|\mathbf{f}|}$ has the same flow assignment on the darts in copy $2\phi+1$ as $\mathbf{f}$ so long as $k \geq 4\phi+1$. Starting from scratch, we can find c.w. acyclic capacities $\mathbf{c}$ via Khuller, Naor and Klein's method (one shortest path computation); we can find $|\mathbf{f}|$ (Lemma 6, a second shortest path computation) and then $\mathbf{f}$ (Lemma 7, a third shortest path computation). Therefore, finding a maximum st-flow in a directed planar graph G is equivalent to three shortest path computations: one in G and two in a covering of G that is $4\phi+1$ times larger than G.

2.4 Discussion

The linear bound on the number of augmentations required by Borradaile and Klein's leftmost augmenting-paths algorithm is given by way on an *unusability theorem* which states that an arc can be augmented, and then its reverse can be augmented, but, if this reverse-augmentation occurs, the arc cannot be augmented again. In a companion paper, we show how to implement an augmenting paths algorithm whose analysis depends on a similar unusability theorem using only priority-queue operations [3]. In this algorithm, we also use dual shortest-paths to illustrate the priority-queue implementation. We believe that combining these ideas – the unusability theorem and dual-shortest paths correspondence – could lead to a max st-flow algorithm for planar graphs that uses $O(n)$ (instead of $O(\phi n)$ as implied by our work here) priority-queue operations. Provided the constants are reasonable, this would certainly be more efficient in practice than a dynamic-trees based implementation.

Acknowledgements. The authors thank Jéremy Barbay for very helpful discussions. This material is based upon work supported by the National Science Foundation under Grant No. CCF-0963921.

References

1. Borradaile, G.: Exploiting Planarity for Network Flow and Connectivity Problems. PhD thesis, Brown University (2008)
2. Borradaile, G., Harutyunyan, A.: Maximum st-flow in directed planar graphs via shortest paths. Technical report, arXiv:1305.5823 (2013)
3. Borradaile, G., Harutyunyan, A.: Boundary-to-boundary flows in planar graphs. In: Lecroq, T., Mouchard, L. (eds.) IWOCA 2013. LNCS, vol. 8288, Springer, Heidelberg (2013)
4. Borradaile, G., Klein, P.: An $O(n \log n)$ algorithm for maximum st-flow in a directed planar graph. J. of the ACM 56(2), 1–30 (2009)
5. Erickson, J.: Maximum flows and parametric shortest paths in planar graphs. In: Proc. SODA, pp. 794–804 (2010)
6. Ford, C., Fulkerson, D.: Maximal flow through a network. Canadian J. Math. 8, 399–404 (1956)
7. Hassin, R.: Maximum flow in (s, t) planar networks. IPL 13, 107 (1981)
8. Hochbaum, D.: The pseudoflow algorithm: A new algorithm for the maximum-flow problem. Operations Research 56(4), 992–1009 (2008)
9. Johnson, D., Venkatesan, S.: Partition of planar flow networks. In: Proc. SFCS, pp. 259–264 (1983)
10. Kaplan, H., Nussbaum, Y.: Minimum st-cut in undirected planar graphs when the source and the sink are close. In: Proc. STACS, pp. 117–128 (2011)
11. Sleator, D., Tarjan, R.: A data structure for dynamic trees. JCSS 26(3), 362–391 (1983)
12. Spanier, E.: Algebraic Topology. Springer (1994)
13. Tarjan, R., Werneck, R.: Dynamic trees in practice. J. Exp. Algorithmics 14, 5:4.5–5:4.23 (2010)

Hypergraph Covering Problems Motivated by Genome Assembly Questions

Cedric Chauve[1,2], Murray Patterson[3], and Ashok Rajaraman[2,4]

[1] LaBRI, Université Bordeaux 1, Bordeaux, France
[2] Department of Mathematics, Simon Fraser University, Burnaby (BC), Canada
{cedric.chauve,arajaram}@sfu.ca
[3] Centrum Wiskunde & Informatica, Amsterdam, The Netherlands
murray.patterson@cwi.nl
[4] International Graduate Training Center in Mathematical Biology, PIMS, Canada

Abstract. We describe some genome assembly problems as a general problem of covering a hypergraph by linear and circular walks, where vertices represent sequence elements, repeated sequences are modelled by assigning a multiplicity to vertices, and edges represent co-localization information. We show that deciding if a given assembly hypergraph admits an assembly is fixed-parameter tractable, and we provide two exact polynomial time algorithms for clearing ambiguities caused by repeats.

1 Introduction

Some genome assembly problems can be seen as hypergraph covering problems, where vertices represent genomic sequences, and weighted edges encode the co-localization of sequence elements; a cover of the hypergraph with a set of linear walks (or circular walks, for genomes with circular chromosomes) corresponds to a genome assembly that respects the co-localization information encoded in the traversed edges. *Repeats*, genomic elements that appear in several locations in the genome being assembled, often confuse assembly algorithms and introduce ambiguity in assemblies. Repeats can be modelled in graph theoretical models of genome assembly by associating a *multiplicity* to each vertex, an upper bound on the number of occurrences of this vertex in linear/circular walks that cover the hypergraph. A vertex with multiplicity greater than 1 can then belong to several walks. The general assembly problem we consider is to extract a maximum weight subset of edges such that there exists a set of linear and/or circular walks on the resulting graph that contains every edge as a subwalk and respects the multiplicity of the vertices. Recent investigations describe both hardness and tractability results for related decision and optimization problems [1,6,2,5].

We formalize these problems in terms of *covering of assembly hypergraphs* by linear and circular walks, and edge-deletion problems. We show that deciding if a given assembly hypergraph admits a covering by linear (circular) walks that respects the multiplicity of all vertices is FPT. We also describe polynomial time algorithms for decision and edge-deletion problems for certain instances of the problems which consider information allowing us to clear ambiguities due to repeats. Full proofs and details for each result are available in [3].

© Springer-Verlag Berlin Heidelberg 2013

2 Preliminaries

Definition 1. *An* assembly hypergraph *is a quadruple* (H, w, c, o) *where* $H = (V, E)$ *is a hypergraph and* w, c, o *are mappings:* $w : E \to \mathbb{R}$, $c : V \to \mathbb{N}$, $o : E \to V^*$ *where* $o(\{v_1, \ldots, v_k\})$ *is either a sequence on the alphabet* $\{v_1, \ldots, v_k\}$ *where each element appears at least once, or* λ *(the empty sequence).*

We use the notation $|V| = n$, $|E| = m$, $s = \sum_{e \in E} |e|$, $\Delta = \max_{e \in E} |e|$, $\delta = \max_{v \in V} |\{e \in E \mid v \in e\}|$, $\gamma = \max_{v \in V} c(v)$. We call $c(v)$ the *multiplicity* of v. A vertex v s.t. $c(v) > 1$ is called a *repeat*; $V_R \subseteq V$ is the set of repeats and $\rho = |V_R|$. Edges s.t. $|e| = 2$ are called *adjacencies*; w.l.o.g., we assume that $o(e) = \lambda$ if e is an adjacency. Edges s.t. $|e| > 2$ (resp. $|e = 3|$) are called *intervals* (resp. *triples*). We denote the set of adjacencies (resp. weights of adjacencies) by $E_A \subseteq E$ (resp. w_A), and the set of intervals (resp. weights of intervals) by $E_I \subseteq E$ (resp. w_I). An interval is *ordered* if $o(e) \neq \lambda$; an assembly hypergraph with no ordered interval is *unordered*. Unless explicitly specified, our assembly hypergraphs are unordered. An assembly hypergraph with no intervals, i.e. $\Delta = 2$, is an *adjacency graph*. Given an assembly hypergraph $\mathcal{H} = (H = (V, E), w, c, o)$, we denote its *induced adjacency graph* by $\mathcal{H}_A = (H_A = (V, E_A), w_A, c, o_A)$.

Definition 2. *Let* $(H = (V, E), w, c, o)$ *be an assembly hypergraph and* P *(resp.* C*) be a linear (resp. circular) sequence on the alphabet* V*. An unordered interval* e *is* compatible *with* P *(resp.* C*) if there is a contiguous subsequence of* P *(resp.* C*) whose content is* e*. An ordered interval* e *is compatible with* P *(resp.* C*) if there exists a contiguous subsequence of* P *(resp.* C*) equal to* $o(e)$ *or its mirror.*

Definition 3. *An assembly hypergraph* $(H = (V, E), w, c, o)$ *admits a* linear assembly *(resp.* mixed assembly*) if there exists a set* $\mathcal{A}$ *of linear (resp. circular) sequences on* V *such that every edge* $e \in E$ *is compatible with* $\mathcal{A}$*, and every vertex* v *appears at most* $c(v)$ *times in* $\mathcal{A}$*. The weight of an assembly is* $\sum_{e \in E} w(e)$*.*

In the following, we consider two kinds of algorithmic problems, a decision problem and an edge-deletion problem.

- The *Assembly Decision Problem*: Given an assembly hypergraph $\mathcal{H} = (H, w, c, o)$ and a genome model (linear or mixed), does there exist an assembly of $\mathcal{H}$ in this model?
- The *Assembly Maximum Edge Compatibility Problem*: Given an assembly hypergraph $\mathcal{H} = (H = (V, E), w, c, o)$ and a genome model, compute a maximum weight subset E' of E such that the assembly hypergraph $\mathcal{H}' = (H' = (V, E'), \{w(e) \mid e \in E'\}, c, \{o(e) \mid e \in E'\})$ admits an assembly in this model.

Definition 4. *Let* $(H = (V, E), w, c, o)$ *be an assembly hypergraph. A* maximal repeat cluster *is a connected component of the hypergraph* $(V_R, \{e \cap V_R \mid e \in E\})$*.*

We now summarize some known results. Theorem 1 below follows from the equivalence between the Assembly Decision Problem with no repeats and the classical Consecutive Ones Property [4].

Theorem 1. *The Assembly Decision Problem can be solved in $O(n+m+s)$ time and space when $\gamma = 1$, in the linear and mixed genome models.*

Theorem 2. [6] *(1) The Assembly Decision Problem can be solved in time and space $O(n+m+s)$ for adjacency graphs in the linear and mixed genome models. (2) The Assembly Decision Problem is NP-hard in the linear and the mixed genome models if $\Delta \geq 3$ and $\gamma \geq 2$.*

Theorem 3. [2] *The Assembly Decision Problem can be solved in polynomial time and space in the linear genome model for unordered assembly hypergraphs where, for every edge e containing a repeat, either e is an adjacency, or e is an interval that contains a single repeat r and there exists an edge $e' = e \setminus \{r\}$.*

Theorem 4. [5] *(1) The Assembly Maximum Edge Compatibility Problem can be solved in polynomial time and space in the mixed genome model for adjacency graphs. (2) The Assembly Maximum Edge Compatibility Problem is NP-hard in the mixed genome model if $\Delta \geq 3$, even if $\gamma = 1$.*

3 New Results

We now describe three positive algorithmic results, together with the corresponding algorithms and proof outlines. We first show that the Assembly Decision Problem is FPT with respect to parameters Δ, δ, γ and ρ.

Theorem 5. *The Assembly Decision Problem can be solved in space $O(n+m+s+\rho\gamma)$ and time $O\left((\delta(\Delta+\rho\gamma))^{2\rho\gamma}(n+m+s+\rho\gamma)\right)$ in the linear and mixed genome models for unordered assembly hypergraphs.*

As we consider a decision problem on unordered graphs, we omit w and o from the notation. The idea is to consider a set of derived assembly hypergraphs $\mathcal{H}_f = (H_f = (V_f, E_f), c_f)$ s.t. $c_f(v) = 1$ for all $v \in V_f$ by making $c(r)$ copies of each $r \in V_R$ and considering each possible set f of choices of 2 neighbours for each of these copies. A given $\mathcal{H}_f$ can then be checked for the existence of an assembly using Theorem 1, and $\mathcal{H}$ admits an assembly if and only if there exists an $\mathcal{H}_f$ which admits an assembly for some f. Finally, if Δ, δ, γ and ρ are bounded, there is a fixed number of such sets f, which results in an FPT algorithm.

Algorithm

1. For each $r \in V_R$, make $c(r)$ distinct copies of r. Call this set $R'(r)$, and $R' = \bigcup_r R'(r)$.
2. For each $v \in R'(r)$, choose 2 neighbours from $N'(r)$, the union of the set of non-repeat neighbours of r and of $\bigcup_p R'(p)$, the union being taken over all repeat neighbours p of r. Call this choice of neighbours for r f_r, and $f = \bigcup_{r \in V_R} f_r$.

3. Construct a new assembly hypergraph $\mathcal{H}_f = (H_f = (V_f, E_f), c_f)$ with $V_f = (V \setminus V_R) \cup R'$, $c_f(v) = 1$ for all $v \in V_f$, and E_f defined as follows: (1) for each $v_r \in R'(r)$, $r \in V_R$, $f(v_r) = \{u, v\}$ for some $u, v \in N'(r)$, add $\{v_r, u\}$ and $\{v_r, v\}$ to E_f, and (2) for each $e \in E$, add an edge $e' \in E_f$ containing $\{v \mid v \in e \setminus V_R\}$.
4. For each $v \in V_f \setminus R'$ adjacent to a vertex $r_1 \in R'$, let $v.r_1. \ \dots \ .r_k.u$ be the unique path in H_f s.t. $\{r_1, \dots, r_k\} \subseteq R'$ and $u \in V_f \setminus R'$. Add all of $\{r_1, \dots, r_k\}$ to e' for each $e' \in E_f$ such that $v \in e'$.
5. Use Theorem 1 on $\mathcal{H}_f$. Output Yes and exit if $\mathcal{H}_f$ admits an assembly in the chosen genome model.
6. Iterate over all possible sets of neighbour choices f in Step 2.
7. Output No (no $\mathcal{H}_f$ admits an assembly in the chosen genome model).

We now describe an algorithm to find a maximum weight subset $S \subseteq E_I$ for an assembly hypergraph $\mathcal{H} = (H = (V, E \setminus S), w, c, o)$ to admit a mixed assembly, given that $\mathcal{H}_A$ admits a mixed assembly. We extend the notion of compatibility for an assembly hypergraph $\mathcal{H}$ as follows.

Definition 5. *An unordered interval $e \in E_I$ is said to be* compatible *with $\mathcal{H}_A$ if there exists a walk in $H_A = (V, E_A)$ whose vertex set is exactly e.*

Then, we can state the following theorem.

Theorem 6. *Let $\mathcal{H} = (H = (V, E), w, c, o)$ be an unordered weighted assembly hypergraph such that $\mathcal{H}_A$ admits a mixed genome assembly, and each interval is a triple compatible with $\mathcal{H}_A$, containing at most one repeat. Then, we can find a maximum weight subset $S \subseteq E_I$, such that $\mathcal{H}' = (H' = (V, E' = E_A \cup S), \{w(e) \mid e \in E'\}, c, \{o(e) \mid e \in E'\})$ admits a mixed assembly, in linear space and $O((n+m)^{3/2})$ time.*

The proof relies on the following ideas: (1) repeat-free triples, as well as triples whose non-repeat vertices form an adjacency, must always be included in a maximum weight compatible set of triples, and (2) the remaining triples to include can be decided using the adjacency compatibility algorithm of [5].

Algorithm

1. Initialize empty sets S, D, and $E' = E_A$.
2. Add to S every $e \in E_I$ having no repeats, and every $e = \{v_0, v_1, r\} \in E_I$ having one repeat r s.t. $\{v_0, v_1\} \in E_A$. Let $E'_I = E_I \backslash S$.
3. For every $e = \{v_0, v_1, r\} \in E'_I$ containing exactly one repeat r:
   (i) Add an adjacency $a_e = \{v_0, v_1\}$ to D. Set $w_D(a_e) = w(e)$.
   (ii) Remove $\{v_0, r\}$ and $\{v_1, r\}$ from E', if present.
4. For every adjacency $e \in E' \setminus D$, set $w'(e) = 1 + \sum_{a_e \in D} w_D(a_e)$.
5. Apply the linearization algorithm [5] (Theorem 4(1)) on $(H_D = (V, E' \cup D), w' \cup w_D, c, o_A)$.
6. For each $a_e \in D$ retained in step 5, add the triple e to S.

We finally turn to instances with larger, but ordered, intervals.

Definition 6. *Let $(H = (V, E), w, c, o)$ be an assembly hypergraph. An interval $e \in E_I$ is an* ordered repeat spanning interval *for a maximal repeat cluster R if $e = \{u, v, r_1, \ldots, r_k\}$ with $c(u) = c(v) = 1$, $\{r_1, \ldots, r_k\} \subseteq R$, and $o(e) = u.s.v$, where s is a sequence on the set $\{r_1, \ldots, r_k\}$, containing every element at least once. The subset of ordered repeat spanning intervals in E_I is denoted by E_{rs}*

Theorem 7. *Let $\mathcal{H} = (H = (V, E), w, c, o)$ be an assembly hypergraph such that every repeat $r \in V_R$ is either contained in an adjacency, or is contained in an interval $e \in E_I$ s.t. either e is an ordered repeat spanning interval, or r is the only repeat in e and there exists an edge $e' = e \setminus \{r\}$. The Assembly Decision Problem in the linear genome model can be solved for $\mathcal{H}$ in polynomial time and space.*

The basic idea of the proof is to realize the sequence $o(e)$ for every repeat spanning interval $e \in E_{rs}$ by creating unique copies of the repeats in e and decreasing the multiplicity accordingly. This leads to an assembly graph that can then be checked using Theorem 3. As we consider a decision problem, we omit w from now.

Algorithm

```
1. Let $V' = V$, $E' = E \backslash E_{rs}$, $c' = c$, $o' = \{o(e) \mid e \in E'\}$, $D = \emptyset$.
2. For every repeat spanning interval $e \in E_{rs}$:
   (a) Let $o = o(e) = u.r_1.\ldots.r_k.v$, possibly $r_i = r_j$ for $i \neq j$ (the $r_i$ are
       repeats)
   (b) For $i$ from 1 to $k$ add a vertex $t_i$ to $V'$, with multiplicity $c'(t_i) = 1$, and
       decrease $c'(r_i)$ by 1.
   (c) For $i$ from 1 to $k - 1$ add an adjacency $\{t_i, t_{i+1}\}$ to $E'$.
   (d) Add edges $\{u, t_1\}$ and $\{v, t_k\}$ to $E'$.
   (e) If the adjacencies $\{u, r_1\}$ and $\{r_k, v\}$ are present, add them to $D$.
3. Return Yes if the assembly hypergraph $\mathcal{H}' = (H' = (V', E' \backslash D), c', o')$ admits
   a linear genome assembly (checked by using Theorem 3) and if $c'(r) \geq 0$ for
   all $r \in V_R$. Else, return No.
```

References

1. Batzoglou, S., Istrail, S.: Physical mapping with repeated probes: The hypergraph superstring problem. In: Crochemore, M., Paterson, M. (eds.) CPM 1999. LNCS, vol. 1645, pp. 66–77. Springer, Heidelberg (1999)
2. Chauve, C., Maňuch, J., Patterson, M., Wittler, R.: Tractability results for the consecutive-ones property with multiplicity. In: Giancarlo, R., Manzini, G. (eds.) CPM 2011. LNCS, vol. 6661, pp. 90–103. Springer, Heidelberg (2011)
3. Chauve, C., Patterson, M., Rajaraman, A.: Hypergraph covering problems motivated by genome assembly questions. arXiv:1306.4353 [cs.DS] (2013)
4. Dom, M.: Algorithimic aspects of the consecutive-ones property. Bull. EATCS 98, 27–59 (2009)
5. Manuch, J., Patterson, M., Wittler, R., Chauve, C., Tannier, E.: Linearization of ancestral multichromosomal genomes. BMC Bioinformatics 13(suppl. 19), S11 (2012)
6. Wittler, R., Manuch, J., Patterson, M., Stoye, J.: Consistency of sequence-based gene clusters. J. Comput. Biol. 18, 1023–1039 (2011)

Cluster Editing with Locally Bounded Modifications Revisited

Peter Damaschke

Department of Computer Science and Engineering
Chalmers University, 41296 Göteborg, Sweden
ptr@chalmers.se

Abstract. For Cluster Editing where both the number of clusters and the edit degree are bounded, we speed up the kernelization by almost a factor n compared to Komusiewicz and Uhlmann (2012), at cost of a marginally worse kernel size bound. We also give sufficient conditions for a subset of vertices to be a cluster in some optimal clustering.

1 Introduction

We consider graphs $G = (V, E)$ with n vertices and m edges and use standard notation like $N(v)$ for the open neighborhood of a vertex, $N[v] := N(v) \cup \{v\}$, $N[W] := \bigcup_{v \in W} N[v]$, and $N(W) := N[W] \setminus W$. A *cluster graph* is a disjoint union of cliques. G is a cluster graph if and only if G has no *conflict triple*, i.e., three vertices inducing exactly two edges. Cluster Editing asks to turn a graph $G = (V, E)$ into a cluster graph using at most k edge *edits*, that is, insertions or deletions. An *optimal clustering* is an optimal solution to Cluster Editing. The *edit degree* of v is the number of edits v is incident to. Cluster Editing has applications in, e.g., molecular biology, and is well studied [1,2,5,9]. Cluster Editing is NP-hard [10], but is also fixed-parameter tractable (FPT) in k. The currently smallest problem kernel has $2k$ vertices [4,3]. An obstacle for the practical use of an FPT algorithm is that k may still be too large. Thus, stronger parameters have been proposed:

(d, t)-Constrained Cluster Editing [6]: Given a graph $G = (V, E)$, parameters d, t, and k, and individual constraints $\tau(v) \leq t$ for every vertex v, transform G into a cluster graph with at most d clusters, by applying at most k edits, such that every vertex v has edit degree at most $\tau(v)$.

The problem is in FPT already in the combined parameter (d, t). A kernel with at most $4dt$ vertices is computed in $O(n^3)$ time [6]. An $O(dt)$ kernel can be preferable to an $O(k)$ kernel: If every vertex actually has edit degree $\Theta(t)$, we get $k = \Theta(nt)$, whereas d is typically much smaller than n. Besides kernel sizes, the time for the kernelization is also decisive for scalability. Here we speed up the kernelization for (d, t)-Constrained Cluster Editing. We compute a kernel with at most $5dt + d$ vertices in $O((m + dt^2) \log n)$ time, and randomization yields a kernel size arbitrarily close to $4dt$ in $O(m + dt^2 \log n)$ expected time.

© Springer-Verlag Berlin Heidelberg 2013

Note that $m = O(n^2)$, and $dt < n$ can be assumed. Hence our time is within $O(n^2 \log n)$, with a sliightly worse kernel size bound. This appears to be a good deal. Moreover, $O((m+dt^2)\log n)$ is within a logarithmic factor of the size (edge number m) of the graph. Namely, assuming $dt < n$, we have $dt^2 < nt < n^2/d$, and a graph of d disjoint cliques has $\Omega(n^2/d)$ edges. Another difference to [6] is that the "old" reduction rules are only based on the (d,t)-constraints, whereas we find edit-optimal clusters, as defined below.

Definition 1. *In a graph $G = (V, E)$, a subset $C \subseteq V$ is an* edit-optimal cluster *if G has an optimal solution to* Cluster Editing *where C is one of the clusters.*

The value of this notion is the following: Once some C is recognized as edit-optimal, we can safely split off C and work on $G - C$ in order to solve Cluster Editing on G. Of course, we cannot expect a polynomial algorithm to recognize edit-optimal clusters, because iterated application would solve Cluster Editing in polynomial time. Therefore we quest for *sufficient* conditions π: If some C satisfies π, then C is edit-optimal. Ideally, π should be both efficient and not too restrictive. Here is a known example of a condition π used in a kernelizations: Let $\gamma(C)$ be the number of edges between C and $G - C$, and let $\delta(C)$ be the number of non-edges in C. If some $v \in C$ satisfies $N[v] = C$, and $2\delta(C) + \gamma(C) < |C|$, then C is edit-optimal [3]. This inequality relates the total number of edits to $|C|$. Here we will add further conditions π that relate only the edit degrees and list edge coloring numbers to $|C|$. Note that we only study the combinatorial side. Further research could deal with algorithmic implications, e.g., the complexity of recognizing such edit-optimal clusters, and perhaps smaller kernels.

Pointing out optimal clusters in parts of a large graph can be more appropriate than computing an optimal clustering of the entire graph. For instance, it has been observed [7] that large social networks tend to consist of a core that lacks a clear clustering structure, and a periphery containing clusters, which are "exotic" groups that are highly connected but have little external interaction.

2 Faster Kernelization

Definition 2. *Let $C \subset V$ be a vertex set in a graph $G = (V, E)$. The edit degree of $v \in C$ is the number of edits incident to v when we split off C as a cluster. The edit degree of C is the maximum edit degree of the vertices in C.*

Lemma 1. *Let v be any fixed vertex, and g the degree of v. Assume that some cluster $C \ni v$ with edit degree at most t exists. Let $c := |C|$, and let $m(C)$ denote the number of edges incident to C. If $g > 4t$ or $c \geq g > 3t$, then C is uniquely determined, and we can compute C in $O((m(C) + t^2)\log c)$ time.*

Proof. Define $x := |N(v) \setminus C|$ and $y := |C \setminus N(v)|$. Since $v \in C$ is incident to at most t edits, we have $x + y \leq t$. Also note that $|C \cap N(v)| = g - x = c - y$.

Every vertex $u \in C$ is incident to at most t non-edges in C, hence u has at least $g - t - x$ neighbors in $N(v)$, even in $C \cap N(v)$. Every vertex $u \notin C$ has at most $t+x$ neighbors in $N(v)$ (namely, at most t in $C \cap N(v)$, and at most x in $N(v) \setminus C$). In the case $g > 4t$ we have $t+x \leq 2t < g/2 = g-g/4-g/4 \leq g-t-x$. In the case $c \geq g > 3t$ (hence $x \leq y$ and $x \leq t/2$) we have $t+x \leq t+(x+y)/2 \leq t+t/2 = 3t/2 < g/2 \leq g-3t/2 \leq g-t-t/2 \leq g-t-x$. Thus, by comparing $|N(u) \cap N(v)|$ with $g/2$ we can decide whether $u \notin C$ or $u \in C$, in these two cases. We refer to this comparison as the *threshold test* for u. It also follows that $C \ni v$ with edit degree at most t is uniquely determined in the mentioned cases.

Next we show how to calculate C, or recognize that v is in no cluster with edit degree at most t. Since $|C \setminus N(v)| \leq t$, every vertex in C can have at most $2t$ neighbors outside $N(v)$.

Phase 1: We do the threshold test for all $u \in N(v)$ as follows. We traverse the adjacency list of every such u and check for each neighbor of u whether it also belongs to $N(v)$. Actually we can stop as soon as (i) $g/2$ neighbors in $N(v)$ or (ii) $2t+1$ neighbors outside $N(v)$ are found, or (iii) the end of u's adcacency list is reached. In case (i) and (iii) we know $u \in C$ and $u \notin C$, respectively. In case (ii), v is in no cluster with edit degree at most t. Using a dictionary for $N(v)$, membership of a vertex in $N(v)$ can be tested in $O(\log g)$ time.

Phase 2: We also find the at most t vertices of $C \setminus N(v)$ as follows. Since they all have neighbors in $C \cap N(v)$, it suffices to do the threshold test for the $O(ct)$ vertices in $N(C \cap N(v))$ only. More precisely, we traverse the adjacency lists of all vertices in $C \cap N(v)$ again and count how often every vertex $u \notin N(v)$ appears there. According to our threshold test, we have $u \in C$ if and only if u is met at least $g/2$ times. Each time a vertex u is met, u can be retrieved (for incrementing its count) in $O(\log c + \log t)$ time.

All edges in the adjacency lists of vertices $u \in C \cap N(v)$, processed in Phase 1, are incident to C. We also have to deal with vertices $u \in N(v) \setminus C$, but there are at most t of them, and in each of their lists we consider at most $(t-1)+(2t+1)$ edges which are not incident to C: those ending in $N(v) \setminus C$ again, and at most $2t+1$ edges ending outside $N(v)$. (After that we can stop, as we saw above.) All edges in the lists processed in Phase 2 are also incident to C. Hence $O(t^2)$ processed edges are not indicent to C. Since $g = O(c)$ and $t = O(c)$, each of $\log g$, $\log c$, $\log t$ is $O(\log c)$, thus every edge is processed in $O(\log c)$ time. □

Theorem 1. *We can compute a kernel for* (d,t)-Constrained Cluster Editing *with at most* $5dt+d$ *vertices in* $O((m+dt^2)\log n)$ *time.*

Proof. As long as there exists some vertex v of degree $g > 4t$ we apply Lemma 1 to find $C \ni v$ with edit degree at most t. If no such C exists, the problem instance has no solution. If C exists, it is uniquely determined due to Lemma 1. We check whether the individual degree constraints $\tau(u)$ are satisfied by all $u \in C$, remove C and all incident edges, and reduce the individual degree constraints of vertices outside C by the respective numbers of adjacent vertices in C. If the test in C fails or a constraint drops below zero, the instance has no solution. By iteration we get rid of all vertices of degree above $4t$, and every set C we have obtained must be a cluster in any valid solution. Any vertex of degree $g \leq 4t$ must belong to a cluster of size $c \leq 5t+1$, since $c \leq g+t+1$ holds in general. Since at most d clusters are allowed, there remain at most $5dt+d$ vertices, or we know that there is no solution. By Lemma 1, every cluster C is found in $O((m(C)+t^2)\log c)$ time. Since edges incident to some C are processed only once, the $m(C)$ terms sum up to at most m. The t^2 term appears at most d times. $\square$

This kernel consists of vertices of degree at most $4t$ in at most d clusters of size at most $5t+1$. Using Lemma 1 we can keep on trying vertices v with $4t \geq g > 3t$. If v belongs to a cluster of size $c \geq g$ and edit degree t, this appears in a valid solution, by the same reasoning as in Theorem 1. However, trying all $O(dt)$ vertices would require $O((m+dt^3)\log c)$ time. This situation suggests the idea of picking v randomly. The following time bound holds with high probability, since the probability not to hit a large cluster decreases exponentially over time.

Theorem 2. *We can compute a kernel for* (d,t)-Constrained Cluster Editing *with at most* $(4+\epsilon)dt$ *vertices in* $O((m+(1/\epsilon)dt^2)\log n)$ *expected time.*

3 Sufficient Conditions for Edit-Optimal Clusters

Theorem 3. *Every* C *with* $c := |C| \geq 5t$ *is an edit-optimal cluster.*

The proof replaces any clustering $C_1,\ldots,C_d$ with $C, C_1 \setminus C, \ldots, C_d \setminus C$ (splitting off C as a new cluster) and shows by bookkeeping arguments that the number of edits is not increased. – Compare Theorem 3 to condition $2\delta(C)+\gamma(C) < c$ from [3]. We get that already $t = O(c)$ is sufficient, whereas the total number of edits can be tc, and even quadratic in c. We also connect this result to Section 2. Suppose that we have computed a kernel for (d,t)-Constrained Cluster Editing, but we may also want to solve unrestricted Cluster Editing, because it might need fewer edits. The size bounds in Section 2 remain valid if we remove only those edit-optimal clusters C with $c > 5t$. This yields:

Corollary 1. *If a graph has a clustering with at most d clusters, where every vertex has edit degree at most t, then we can compute a kernel for* Cluster Editing *with at most* $5dt$ *vertices in* $O((m+dt^2)\log n)$ *time.*

Next we derive a further condition related to another graph problem:

List Edge Coloring: Given a graph where every edge carries a list of suitable colors, paint every edge such that incident edges get distinct colors.

Definition 3. *Given a vertex set $C \subset V$ in a graph $G=(V,E)$, we define an auxiliary graph $H(C)$ on vertex set $N[C]$, where the edges of $H(C)$ are the non-edges in C and the cut edges between C and $G-C$, that is, the edits incident to C when we split off C as a cluster. Let every vertex $s \in C$ represent a* color.

Theorem 4. *Consider the graph $H(C)$, colors corresponding to the vertices of C, and let $s \in C$ be a suitable color for edge uv if and only if s is not adjacent to u,v in $H(C)$. Then the following condition is sufficient for C being an edit-optimal cluster: $H(C)$ admits a list edge coloring with the extra property that, if an edge e uses a vertex of edge f as its color, then edge f may not use a vertex of edge e as its color.*

The proof is sketched as follows: When an edge uv is colored with s, then u, s, v form a conflict triple with an edge in C. Every edit in $H(C)$ is in such a conflict triple, conversely, every such conflict triple contains exactly one edit. The conditions enforce that no two of our conflict triples share two vertices. Since, in any clustering, some pair from every conflict triple must be edited, splitting off C as a cluster requires the minimum number of edits incident to C.

This result provides further possibilities to confirm edit-optimal clusters. As a little example, let C be an isolated clique K_7 from which the 6 edges of some K_4 are removed. Then $2\delta(C) + \gamma(C) = 2 \cdot 6 + 0 = 12 > 7$, also $t = 3$, hence the previous conditions do not apply. But the 6 edges of $H(C)$ can be colored by 3 suitable colors represented by vertices of C. The algorithmic use of our coloring condition needs to be further explored. There is a well-known close connection between degrees and edge colorings (see [8]), and progress on the List Edge Coloring problem could further strengthen our conditions.

Acknowledgments. This work has been supported by project grants from the Swedish Research Council (Vetenskapsrådet), 2010-4661, "Generalized and fast search strategies for parameterized problems", and the Swedish Foundation for Strategic Research (SSF), "Data-driven secure business intelligence".

References

1. Böcker, S.: A Golden Ratio Parameterized Algorithm for Cluster Editing. J. Discr. Algor. 16, 79–89 (2012)
2. Böcker, S., Briesemeister, S., Klau, G.W.: Exact Algorithms for Cluster Editing: Evaluation and Experiments. Algorithmica 60, 316–334 (2011)
3. Cao, Y., Chen, J.: Cluster Editing: Kernelization Based on Edge Cuts. Algorithmica 64, 152–169 (2012)
4. Chen, J., Meng, J.: A $2k$ Kernel for the Cluster Editing Problem. J. Comp. System Sci. 78, 211–220 (2012)
5. Fomin, F.V., Kratsch, S., Pilipczuk, M., Pilipczuk, M., Villanger, Y.: Tight Bounds for Parameterized Complexity of Cluster Editing. In: Portier, N., Wilke, T. (eds.) STACS 2013., Dagstuhl. LIPIcs, vol. 20, pp. 32–43 (2013)
6. Komusiewicz, C., Uhlmann, J.: Cluster Editing with Locally Bounded Modifications. Discr. Appl. Math. 160, 2259–2270 (2012)
7. Leskovec, J., Lang, K.J., Dasgupta, A., Mahoney, M.W.: Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. Internet Math. 6, 29–123 (2009)
8. Misra, J., Gries, D.: A Constructive Proof of Vizing's Theorem. Info. Proc. Letters 41, 131–133 (1992)
9. Rahmann, S., Wittkop, T., Baumbach, J., Martin, M., Truß, A., Böcker, S.: Exact and Heuristic Algorithms for Weighted Cluster Editing. In: Markstein, P., Xu, Y. (eds.) CSB 2007, pp. 391–401. Imperial College Press (2007)
10. Shamir, R., Sharan, R., Tsur, D.: Cluster Graph Modification Problems. Discr. Appl. Math. 144, 173–182 (2004)

New Approximation Algorithms for the Vertex Cover Problem*

François Delbot[1], Christian Laforest[2], and Raksmey Phan[2]

[1] Université Paris Ouest Nanterre / LIP6, CNRS UMR7606
4 place Jussieu, 75005 Paris, France
`francois.delbot@u-paris10.fr`
[2] Université Blaise Pascal / LIMOS, CNRS UMR 6158
Campus des Cézeaux, 24 avenue des Landais, 63173 Aubière Cedex, France
`{laforest,phan}@isima.fr`

Abstract. The vertex cover is a classical NP-complete problem that has received great attention these last decades. A conjecture states that there is no c-approximation polynomial algorithm for it with c a *constant* strictly less than 2. In this paper we propose a new algorithm with approximation ratio strictly less than 2 (but non constant). Moreover we show that our algorithm has the potential to return *any* optimal solution.

Keywords: vertex cover, approximation, lower bound.

1 Introduction

Let $G = (V, E)$ be any undirected non weighted graph where V is the set of *vertices* and E is the set of *edges*. A *vertex cover* VC is a subset of the vertices ($VC \subseteq V$) such that each edge uv has at least one extremity in VC ($u \in VC$ or $v \in VC$ or both). The associated optimization problem (i.e. the vertex cover problem) is to construct a vertex cover of minimum size. This is a classical optimization problem and many works have been devoted to this problem or to its variations.

The vertex cover problem is shown to be not approximable (in polynomial time) within a factor of 1.166 [8]. Some very simple approximation algorithms give a tight approximation ratio of 2 (see [13] p.3 and [12]). One of them, discovered independently by Gavril and Yannakakis (see [11] p.432) which considers the *maximal matching* is certainly the most famous and studied. Despite a lot of works, no polynomial algorithm whose approximation ratio is bounded by a constant less than 2 has been found and it is conjectured that there is no smaller *constant* ratio unless $P = NP$ [10]. Bar-Yehuda and Even [3] proposed algorithms with an approximation ratio of $2 - \frac{\log \log n}{2 \log n}$ and Karakostas [9] reduced this ratio to $2 - \Theta(\frac{1}{\sqrt{\log n}})$. More recently [1,2,6] describe experimental comparisons and give analytical results on the treatment of huge graphs.

* This work is supported by the French Agency for Research under the DEFIS program TODO, ANR-09-EMER-010.

© Springer-Verlag Berlin Heidelberg 2013

Notations and definitions. Given a graph G we will note OPT the size of an optimal vertex cover of G. A vertex cover VC of G is *minimal* (for inclusion) if any subset of VC is not a vertex cover of G. If $H \subseteq V$, we note $G[H]$ the subgraph of G *induced by* H in G, ie. the graph whose set of vertices is H and edges are the ones of G linking two vertices of H: $\{uv : uv \in E, u \in H$ and $v \in H\}$. $H \subseteq V$ is a *clique* of G if $G[H]$ contains all possible edges between each pair of vertices of H in G. $H \subseteq V$ is an *independent* (or stable) if $G[H]$ contains no edges. We call a *clique partition* of a graph G, a partition of the set V of n vertices into disjoint subsets $C = \{C_1, \ldots, C_k\}$ ($\cup_{i=1}^{k} C_i = V$ and if $i \neq j$, $C_i \cap C_j = \emptyset$) such that each subgraph $G[C_i]$ induced by C_i in G is a *clique* (its number of vertices will be denoted by n_i ($1 \leq n_i \leq |V|$)). A clique that contains only one vertex is called a *trivial clique*. A clique partition is *minimal* if for all $i \neq j$ the graph $G[C_i \cup C_j]$ induced by the cliques C_i and C_j is *not* a clique.

2 Lower Bound

Let $G = (V, E)$ be any graph and $C_1, \ldots, C_k$ ($n_i = |C_i|$) be any clique partition of G. Then:

$$\sum_{i=1}^{k} (n_i - 1) = n - k \leq OPT \tag{1}$$

Our bound generalizes the classical lower bound by maximal matching (see [7] for more details).

3 A New Approximation Algorithm: *CP*

3.1 Algorithm *CP*

Let $G = (V, E)$ be any graph. Let $C_1, \ldots, C_k$ be any *minimal clique partition* of G. We suppose that cliques are sorted such that all trivial cliques are at the end. Let $l \leq k$ be the number of non-trivial cliques, we have $|C_i| \geq 2$ ($i \leq l$) and $|C_{l+1}| = \ldots = |C_k| = 1$. Algorithm *CP* takes as input any graph G and a minimal clique partition and returns: $S = \displaystyle\bigcup_{i=1}^{l} C_i$ (union of the vertices of non-trivial cliques). In the following, we suppose that G contains at least one edge (otherwise: $|S| = OPT = 0$), then we can prove Theorem 1 (see [7]).

Theorem 1. *S is a vertex cover of G and:*

$$\frac{|S|}{OPT} \leq \frac{l}{\sum_{i=1}^{l}(n_i - 1)} + 1 = \frac{l}{n-k} + 1 \leq 2$$

The bound of 2 is tight.

Proof. Trivially S is a vertex cover of G (see [7]). Now we prove the approximation ratio. With Equation 1, we know that: $OPT \geq \sum_{i=1}^{k}(n_i - 1) = \sum_{i=1}^{l}(n_i - 1)$. Note that $n_i = 1$ $(i = l+1, \ldots, k)$. By construction: $|S| = \sum_{i=1}^{l} n_i$.

Thus: $\frac{|S|}{OPT} \leq \frac{\sum_{i=1}^{l} n_i}{\sum_{i=1}^{l}(n_i-1)} = \frac{l+\sum_{i=1}^{l}(n_i-1)}{\sum_{i=1}^{l}(n_i-1)} = \frac{l}{\sum_{i=1}^{l}(n_i-1)} + 1 \leq 2$

To finish, let us consider a cycle with 4 vertices a, b, c, d in this order on the cycle and the clique partition $\{\{a, b\}, \{c, d\}\}$. In this graph $OPT = 2$ but *CP* returns the 4 vertices, leading to an approximation ratio of exactly 2. □

3.2 An Algorithm to Construct Any Minimal Clique Partition: *CPGathering*

Algorithm *CP* takes as input a graph G and a minimal clique partition of G. In this part we describe an algorithm that takes any graph G as input and is able to construct any minimal clique partition of G (Lemma 1).

Let $G = (V, E)$ be any graph. At the beginning, each vertex u is considered as a clique $\{u\}$. Then two vertices linked by an edge can be merged to get a new clique. We keep this idea over all the algorithm. That is, at each step we can merge two cliques C_i and C_j in a new clique if $G[C_i \cup C_j]$ is a clique.

We give now more details on our algorithm called *CPGathering*. We firstly associate at each vertex u the trivial clique $\{u\}$. While there is an edge, we randomly chose one uv. We create a new vertex w. The clique associated to w is the union of cliques associated to u and v. Vertices u and v are deleted (and all their incident edges). We create new edges between w and all vertices that were adjacent to both u and v. Thus at the next step, w is only adjacent to vertices with which it is allowed to be merged. At the end there is no remaining edge and the final clique partition of G is composed of the cliques associated to the remaining vertices. This algorithm is polynomial.

Lemma 1. CPGathering *returns always* minimal *clique partition and can return* any *minimal clique partition of G (see [7]).*

At this step *CP* gives a 2-approximation solution S (Theorem 1) from a minimal clique partition (given by *CPGathering* for example). In Section 4 we present a new approximation algorithm that takes benefit of the current solution to improve the approximation ratio.

4 A Refinement of *CP* for the Vertex Cover

In this section we improve the solution given by *CP* (see Section 3) by applying the *ListRight* algorithm [4,5]. We show how the new algorithm reduces the solution domain by excluding non-minimal solutions. We also show that any optimal vertex cover can be constructed (Theorem 3) and that the approximation ratio of our method is *strictly* smaller than 2 (Theorem 2).

F. Delbot and Ch. Laforest [5] have proposed in 2008 *ListRight*, a *list* heuristic for the vertex cover problem. The algorithm is simple: For a given graph G and any list of all its vertices, read this list, vertex by vertex, from right to left; For each current read vertex u, if u has at least one neighbor (in G) at its right in the list, not in the current solution VC (initially empty) then u is added to VC (in all other cases u is not added). It is shown in [5] that at the end VC is a minimal vertex cover of G. However, *ListRight* does not have a constant approximation ratio. Now we describe how to combine *CP* and *ListRight*.

4.1 Vertex Cover by Clique Partition (VCCP)

Let G be any graph and C be any sorted minimal clique partition $C = \{C_1, \ldots, C_l, C_{l+1}, \ldots, C_k\}$ ($|C_i| \geq 2, \forall i \leq l$ and $|C_i| = 1, l < i \leq k$) of G. Let $S = \bigcup_{i=1}^{l} C_i$ be the solution returned by *CP*. We construct a list L of the n vertices of G from C: At the right of L we put all the vertices, in any order, from the trivial cliques $C_{l+1}, \ldots, C_k$; At the left of this part we put all the other vertices in any order. Then we apply *ListRight* on this list L. We note S' the new solution. As *CP* and *ListRight* are all polynomial then *VCCP* is polynomial. With Lemma 2 we can prove that *VCCP* has an approximation ratio *stritly* less than 2 (Theorem 2). Moreover we prove that $VCCP$ can return any optimal vertex cover for any graph (Theorem 3).

Lemma 2. *S' is a minimal vertex cover and $S' \subseteq S$ (see [7]).*

Theorem 2. *Let $G = (V, E)$ be any graph, with at least one edge. Then: $|S'| < 2OPT$.*

Proof. G contains at least one edge, thus $OPT \geq 1$ (otherwise $OPT = |S| = |S'| = 0$). As S' is a subset of S, Theorem 1 shows that *VCCP* is a 2-approximation algorithm:

$$\frac{|S'|}{OPT} \leq \frac{|S|}{OPT} \leq \frac{l}{n-k} + 1 \leq 2$$

With this inequality, we will show that the approximation ratio is strictly smaller than 2. Suppose there is a graph G with a clique partition $\{C_1, \ldots, C_l, C_{l+1}, \ldots, C_k\}$ such that $|S'| = 2OPT$. As we have $|S| \leq 2OPT$ and $|S'| \leq |S|$ then $|S| = 2OPT$ and $\frac{l}{n-k} + 1 = 2$, thus $l = n - k$. We show now that the non-trivial cliques have exactly 2 vertices.

Suppose there is at least one clique that initially has 3 vertices. If in each non-trivial clique we remove one vertex then it remains 2 vertices in this clique. Considering these non-trivial cliques we have removed $l = n - k$ vertices and it remains at least $l+1$ vertices in these cliques. Also there are $k-l$ vertices in trivial cliques. Thus the number of vertices n is at least $n \geq l+(l+1)+(k-l) = l+k+1$. With the previous equality ($l = n - k$) we have $n \geq n + 1$: contradiction. Thus we have $|C_1| = \ldots = |C_l| = 2$ and these vertices fill the solution: $S = \bigcup_{i=1}^{l} C_i$.

We note $I = C_{l+1} \cup \ldots \cup C_k$ the set composed of the union of the trivial cliques then $I = V - S$. We remind that S is an (exact) 2-approximation of size $2l$ and an optimal vertex cover VC^* ($|VC^*| = OPT = l$) must cover the l cliques/edges C_i ($i \leq l$). Then VC^* has *at least* one vertex in each C_i ($i \leq l$). Moreover these cliques/edges are independent then they need l vertices from VC^* to be covered and as the size of VC^* is l thus VC^* has exactly *one* vertex in each clique C_i ($i \leq l$). Thus $VC^* \subseteq S$ and a consequence is that $VC^* \cap I = \emptyset$.

As $|S'| = |S| = 2OPT$, all vertices in non-trivial cliques are selected by *ListRight*. Let us consider a clique C_i ($i \leq l$) of two vertices u and v ($uv \in E$). Suppose $u \in VC^*$ and $v \notin VC^*$. Vertex v has no neighbor w in I (otherwise, edge vw would not be covered by VC^* since $VC^* \cap I = \emptyset$). Then all neighbors of v are in S' and $S' - \{v\}$ is also a vertex cover. But it is in contradiction with the fact that S' is minimal (Lemma 2). □

Theorem 3. VCCP *can return any optimal vertex cover for any graph if the clique partition is constructed by* CPGathering*.*

Proof. Let VC^* be any optimal vertex cover of G. In [7] we have proved that there is a minimal clique partition of G such that all vertices of VC^* belong to its non-trivial cliques. As VC^* is also minimal, it is easy to see ([7]) that *VCCP* can return VC^* if it has as input the minimal clique partition in which non-trivial cliques containt VC^*. To finish, Lemma 1 shows that *CPGathering* can return any minimal clique partition. □

References

1. Angel, E., Campigotto, R., Laforest, C.: Analysis and comparison of three algorithms for the vertex cover problem on large graphs with low memory capacities. Algorithmic Operations Research 6(1), 56–67 (2011)
2. Angel, E., Campigotto, R., Laforest, C.: Implementation and comparison of heuristics for the vertex cover problem on huge graphs. In: Klasing, R. (ed.) SEA 2012. LNCS, vol. 7276, pp. 39–50. Springer, Heidelberg (2012)
3. Bar-Yehuda, R., Even, S.: A local ratio theorem for approximating the weighted vertex cover problem. Annals of Discrete Mathematics 25, 27–45 (1985)
4. Birmelé, E., Delbot, F., Laforest, C.: Mean analysis of an online algorithm for the vertex cover problem. Information Processing Letters 109(9), 436–439 (2009)
5. Delbot, F., Laforest, C.: A better list heuristic for vertex cover. Information Processing Letters 107(3-4), 125–127 (2008)
6. Delbot, F., Laforest, C.: Analytical and experimental comparison of six algorithms for the vertex cover problem. ACM Journal of Experimental Algorithmics 15, 1.4:1.1–1.4:1.27 (2010)
7. Delbot, F., Laforest, C., Phan, R.: New approximation algorithms for the vertex cover problem and variants. Research Report RR-13-02, LIMOS, Clermont Ferrand, France (2013)
8. Håstad, J.: Some optimal inapproximability results. In: STOC, pp. 1–10 (1997)
9. Karakostas, G.: A better approximation ratio for the vertex cover problem. ACM Transactions on Algorithms 5, 41:1–41:8 (2009)
10. Khot, S., Regev, O.: Vertex cover might be hard to approximate to within $2 - \epsilon$. Journal of Computer and System Sciences 74(3), 335–349 (2008)
11. Papadimitriou, C.H., Steiglitz, K.: Combinatorial Optimization: Algorithms and Complexity. Prentice-Hall (1982)
12. Savage, C.: Depth-first search and the vertex cover problem. Inf. Process. Lett. 14(5), 233–235 (1982)
13. Vazirani, V.V.: Approximation algorithms. Springer (2001)

Improved Approximation Algorithm for the Number of Queries Necessary to Identify a Permutation

Mourad El Ouali and Volkmar Sauerland

Department of Computer Science
Christian-Albrechts-Universität, Kiel, Germany
{meo,vsa}@informatik.uni-kiel.de

Abstract. In the past three decades, deductive games have become interesting from the algorithmic point of view. A well known deductive game is the famous Mastermind game. In this paper, we consider the so called Black-Peg variant of Mastermind. More precisely, we deal with a special version of the Black-Peg game with n holes and $k \geq n$ colors where no repetition of colors is allowed. We present a strategy that identifies the secret code in $\mathcal{O}(n \log_2 n)$ queries. Our algorithm improves the previous result of Ker-I Ko and Shia-Chung Teng (1986) by almost a factor of 2 for the case $k = n$. To our knowledge there is no previous work dealing with the case $k > n$.

1 Introduction

The original Mastermind is a two players board game invented in 1970 by Mordecai Meirowitz. It consists of a board with several rows, each containing four holes, and pegs of six different colors. The idea of the game is that a *Codemaker* chooses a secret color combination y of pegs from the possible colors and a *Codebreaker* has to identify the code by a sequence of queries and corresponding information that is provided by the *Codemaker*. All queries are also color combinations of the possible colors. Information concerning a query σ is given about the number of it's correctly positioned colors ($\text{black}(\sigma, y)$) and further correct colors in a wrong place ($\text{white}(\sigma, y)$), respectively.

1.1 Mastermind Variants and Related Works

Let us denote the code length with n and the number of possible colors by k. Clearly, variants of Mastermind are obtained by changing the values of n and k, respectively. Further popular variants are

– **Black-Peg**: only black information about the number of correctly positioned colors is given.
– **AB Game**: color repetition is forbidden for both secret code and queries.

© Springer-Verlag Berlin Heidelberg 2013

Before its market launch, Erdös and Rényi [3] (1963) already analyzed a Mastermind variant with two colors. One of the earliest analysis of the original game with 4 wholes and 6 colors was done by Knuth [6] (1977) giving a strategy that identifies the secret code in at most 5 queries. Stuckman and Zhang [8] (2006) showed that it is $\mathcal{NP}$-complete to determine if a sequence of queries and answers is satisfiable. There are many approximation results regarding different methods.

The Black-Peg game was first introduced by Chvátal [1] (1983) for the case $k = n$. He gave a deterministic adaptive strategy that uses $2n\lceil\log_2 k\rceil + 4n$ queries. Goodrich [4] (2009) improved the result of Chvátal for arbitrary n and k to $n\lceil\log_2 k\rceil + \lceil(2 - 1/k)n\rceil + k$ queries and proved that this kind of game is $\mathcal{NP}$-complete. A further improvement to $n\lceil\log_2 n\rceil + k - n + 1$ for $k > n$ and $n\lceil\log_2 n\rceil + k$ for $k \le n$ was done by Jäger and Peczarski [5] (2011). Recently, Doerr et al. [2] improved the result obtained by Chvátal to $\mathcal{O}(n \log\log n)$ and also showed that this asymptotic order even holds for up to $n^2 \log\log n$ colors, if both black and white information is allowed.

Concerning the combination of both variants, Black-Peg game and AB game, there is only one work due to Ker-I Ko and Shia-Chung Teng [7] (1986) for the case $k = n$. They presented a strategy that identifies the secret permutation in at most $2n\log_2 n + 7n$ queries and proved that the corresponding counting problem is $\#\mathcal{P}$-complete. To our knowledge there is no result for the case $k > n$, yet.

1.2 Our Contribution

We present an algorithm for the **Black-Peg game with forbidden color repetition** (also for queries). It identifies the secret code in less than $n\log_2 n + \lambda n$ queries for the case $k = n$ and in less than $n\log_2 n + k + 2n$ queries for the case $k > n$. Our performance in the case $k = n$ is an improvement of the result of Ker-I Ko and Shia-Chung Teng [7] by almost a factor of 2.

2 A Strategy for Permutation Master Mind

We consider the case $k = n$, here. Let $[n] := \{1, \dots, n\}$ be the set of colors. Our algorithm for finding the secret code includes two main phases which are based on two ideas.

2.1 Phase 1 Queries

In the first phase the codebreaker guesses an initial sequence of n queries that has a predefined structure. The first query, say σ^1, is some permutation of the n colors. For the next query, we shift the color order of the first query circularly to the right. The procedure is repeated until query n (actually the nth query can

	queries								n_1	n_2	n_3
σ^1	1	2	3	4	5	6	7	8	**0**	**0**	**0**
σ^2	8	1	2	3	4	5	6	7	**2**	**0**	**2**
σ^3	7	8	1	2	3	4	5	6	**3**	**2**	**1**
σ^4	6	7	8	1	2	3	4	5	**1**	**1**	**0**
σ^5	5	6	7	8	1	2	3	4	**0**	**0**	**0**
σ^6	4	5	6	7	8	1	2	3	**0**	**0**	**0**
σ^7	3	4	5	6	7	8	1	2	**1**	**0**	**1**
σ^8	2	3	4	5	6	7	8	1	**1**	**0**	**1**

Fig. 1. Initial queries with associated black information n_1, coincidences with a partial solution n_2 and the difference of both n_3

be saved). As an example, let us consider the case $n = k = 8$ and suppose that the secret code y is

$$7 \quad 1 \quad 4 \quad 3 \quad 2 \quad 8 \quad 5 \quad 6$$

Figure 1 shows n possible initial queries.

2.2 Phase 2 Queries

The structure of the initial guesses and the corresponding information by the codemaker enable us to identify the colors of the secret code one after another, each by using a binary search. It is required to keep record of the identified colors within a *partial solution* x that satisfies $x_i \in \{0, y_i\}$ for all $i \in \{1, \ldots, n\}$. The non-zero components of x indicate the components of y that have already been identified.

Suppose, we have already found the 3 colors given in the partial solution of Figure 2. Now, we regard two neighbored initial queries, say σ^j and σ^{j+1}, with

$$\bullet \quad \bullet \quad \bullet \quad \bullet \quad 2 \quad \bullet \quad 5 \quad 6$$

Fig. 2. A partial solution

the property that σ^j contains some correct peg, that has not yet been identified but σ^{j+1} contains no unidentified correct peg any more. We call the index j of such a pair of queries *active*. Note, that the number n_3 of unidentified correct pegs of a query is obtained by subtracting the number n_2 of pegs in which it coincides with the partial solution from its total number of correct pegs n_1. Clearly, a pair of initial queries with the desired property exists, if at least one but not all pegs of the secret code have been identified (since $\sum_{j=1}^{n} \text{black}(\sigma^j, y) = n$). For our example we can choose the highlighted queries σ^3 and σ^4 in Figure 1. We choose one of the identified colors (here 2) as a pivot color and compose new

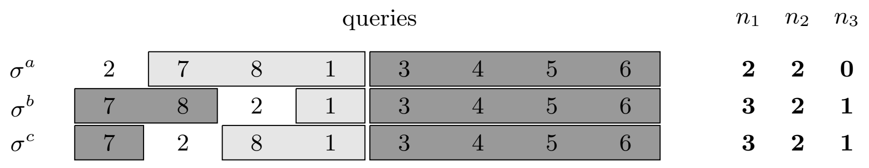

queries
n_1 n_2 n_3
σ^a 2 7 8 1 3 4 5 6 2 2 0
σ^b 7 8 2 1 3 4 5 6 3 2 1
σ^c 7 2 8 1 3 4 5 6 3 2 1

Fig. 3. Binary search queries to extend the partial solution. The highlighted subsequences correspond to subsequences of the selected initial queries.

queries from our pair of queries within a binary search that identifies the next correct peg as demonstrated in Figure 3. Since the information n_3 for query σ^a is 0, all correctly placed pegs in query σ^3 are on the left side of the pivot peg. Thus, we can apply a binary search for the left most correct peg in the first 4 places of query σ^3 using the pivot peg. The binary search is done by queries σ^b and σ^c and identifies the peg with color 7 (in general the peg that is left to the most left pivot position for which n_3 is non-zero). If the answer to query σ^a would have been greater than 0, we could have found a correct peg in query σ^3 on the right side of the pivot peg by a similar approach.

Thus, the identification of one correct peg (except for the first one) requires at most $1 + \log_2(n)$ queries. The general procedure is outlined as Algorithm 2. The first correct peg can be found in $2\log_2(n)$ queries by a similar procedure, say findFirst. Both procedures are part of the main algorithm outlined as Algorithm 1. The approach yields $(n-3)\lceil\log_2 n\rceil + \frac{5}{2}n - 1$ as an upper bound for the number of queries to break the secret code for the case $k = n$. For $k > n$ similar considerations yield an upper bound of $(n-1)\lceil\log_2 n\rceil + k + n - 2$ queries. Our algorithms were implemented in Matlab and testet for code lengths up to $n = 1000$.

Algorithm 1. Algorithm for Permutations

```
Let $y$ be the secret code and set $x := (0, 0, \dots, 0)$;
Guess the initial permutations $\sigma^i$, $i \in [n-1]$;
Initialize $v \in \{0, 1, \dots, n\}^n$ by $v_i := \text{black}(\sigma^i, y)$, $i \in [n-1]$, $v_n := n - \sum_{i=1}^{n-1} v_i$;
if $v = \mathbb{1}_n$ then
    $j := 1$;
    Find the position $m$ of the correct peg in $\sigma^1$ by at most $\frac{n}{2} + 1$ further guesses;
else
    Call findFirst for an active $j \in [n]$ to find the position of the correct peg in $\sigma^j$ by at
    most $2\lceil\log_2 n\rceil$ further guesses;
$x_m := \sigma^j_m$;
$v_j := v_j - 1$;
while $|\{i \in [n] \mid x_i = 0\}| > 2$ do
    Choose an active index $j \in [n]$;
    $m :=$ findNext$(y, x, j)$;
    $x_m := \sigma^j_m$;
    $v_j := v_j - 1$;
Make at most two more guesses to find the remaining two unidentified colors;
```

Algorithm 2. Function findNext

```
input  : Code $y$, partial solution $x \neq 0$ and an active index $j \in [n]$
output: Position $m$ of a correct open component in $\sigma^j$
if $j = n$ then $r := 1$ else $r := j + 1$;
Choose a color $c$ with identified position (a value $c$ of some non-zero component of $x$);
Let $l_j$ and $l_r$ be the positions with color $c$ in $\sigma^j$ and $\sigma^r$, respectively;
if $l_j = n$ then leftS := true else
    Guess $\sigma^{j,0} := \left(c, (\sigma_i^j)_{i=1}^{l_j-1}, (\sigma_i^j)_{i=l_j+1}^{n}\right)$;
    $s :=$ black$(\sigma^{j,0}, y, x)$;
    if $s = 0$ then leftS := true;
    else leftS := false;
if leftS then let $a := 1$ and $b := l_j$;
else let $a := l_r$ and $b := n$;
$m := n$ ;                                                  // position to be found
while $b > a$ do
    $l := \lceil \frac{a+b}{2} \rceil$ ;                                          // position for peg $c$
    if leftS then $\sigma^{j,l} := \left((\sigma_i^j)_{i=1}^{l-1}, c, (\sigma_i^j)_{i=l}^{l_j-1}, (\sigma_i^j)_{i=l_j+1}^{n}\right)$;
    else $\sigma^{j,l} := \left((\sigma_i^r)_{i=1}^{l_r-1}, (\sigma_i^r)_{i=l_r+1}^{l}, c, (\sigma_i^r)_{i=l+1}^{n}\right)$;
    Guess $\sigma^{j,l}$;
    $s :=$ black$(\sigma^{j,l}, y, x)$;
    if $s > 0$ then
        $b := l - 1$;
        if $b < m$ then let $m := b$;
    else $a := l$;
Return $m$;
```

References

1. Chvátal, V.: Mastermind. Combinatorica 3, 325–329 (1983)
2. Doerr, B., Spöhel, R., Thomas, H., Winzen, C.: Playing Mastermind with Many Colors. In: Proc. of ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 695–704. SIAM Society for Industrial and Applied Mathematics (2013)
3. Erdös, P., Rényi, C.: On Two Problems in Information Theory. Publications of the Mathematical Institute of the Hungarian Academy of Science 8, 229–242 (1963)
4. Goodrich, M.T.: On the algorithmic complexity of the Mastermind game with black-peg results. Information Processing Letters 109, 675–678 (2009)
5. Jäger, G., Peczarski, M.: The number of pessimistic guesses in generalized black-peg Mastermind. Information Processing Letters 111, 933–940 (2011)
6. Knuth, D.E.: The computer as a master mind. Journal of Recreational Mathematics 9, 1–5 (1977)
7. Ko, K., Teng, S.: On the Number of queries necessary to identify a permutation. Journal of Algorithms 7, 449–462 (1986)
8. Stuckman, J., Zhang, G.: Mastermind is $\mathcal{NP}$-complete. INFOCOMP Journal of Computer Science 5, 25–28 (2006)

Motif Matching Using Gapped Patterns

Emanuele Giaquinta[1,⋆], Kimmo Fredriksson[2], Szymon Grabowski[3], and Esko Ukkonen[1]

[1] Department of Computer Science, University of Helsinki, Finland
{emanuele.giaquinta,ukkonen}@cs.helsinki.fi

[2] School of Computing, University of Eastern Finland, P.O. Box 1627, FI-70211 Kuopio, Finland
kimmo.fredriksson@uef.fi

[3] Institute of Applied Computer Science, Lodz University of Technology, Al. Politechniki 11, 90–924 Łódź, Poland
sgrabow@kis.p.lodz.pl

1 Introduction and Basic Definitions

We consider the problem of matching a set $\mathcal{P}$ of gapped patterns against a given text T of length n, where a gapped pattern is a sequence of strings (keywords), over a finite alphabet Σ of size σ, such that there is a gap of fixed length between each two consecutive strings. We assume the RAM model, with words of size w in bits. We are interested in computing the list of matching patterns for each position in the text. This problem is a specific instance of the *Variable Length Gaps problem* [2] (VLG problem) for multiple patterns and has applications in the discovery of transcription factor (TF) binding sites in DNA sequences when using generalized versions of the Position Weight Matrix (PWM) model to represent TF binding specificities. The paper [5] describes how a motif represented as a generalized PWM can be matched as a set of gapped patterns with unit-length keywords, and presents algorithms for the restricted case of patterns with two unit-length keywords.

In the VLG problem a pattern is a concatenation of strings and of variable-length gaps. The best time bounds for this problem are: i) $O(n(k\frac{\log w}{w}+\log\sigma))$ [3], where k is the number of the strings and gaps in the pattern; ii) $O(n\log\sigma+\alpha)$ [2], where α is the total number of occurrences of the strings in the patterns within the text[1]; iii) $O(n(\log\sigma+K)+\alpha')$ [6], where K is the maximum number of suffixes of a keyword that are also keywords and α' is the number of text occurrences of pattern prefixes that end with a keyword. Recently, a variant of this algorithm based on word-level parallelism was presented in [7]. Let $\mathrm{len}(\mathcal{P})$ be the total number of alphabet symbols in the patterns. If all the keywords have unit length, the last two results are not ideal because in this case α and α' are $\Omega(n\frac{\mathrm{len}(\mathcal{P})}{\sigma})$ and $\Omega(n\frac{|\mathcal{P}|}{\sigma})$ on average, respectively, if we assume that the symbols in the patterns are sampled from Σ according to a uniform distribution. When α or α' is large, the bound of [3] may be preferable. The drawback of this algorithm is that, to our knowledge, it is not practical.

⋆ Supported by the Academy of Finland, grant 118653 (ALGODAN).

[1] Note that the number of occurrences of a keyword that occurs in r patterns and in l positions in the text is equal to $r \times l$.

© Springer-Verlag Berlin Heidelberg 2013

In this paper we present the following novel result, based on dynamic programming and word-level parallelism:

Theorem 1. *Given a set $\mathcal{P}$ of gapped patterns and a text T of length n, all the occurrences in T of the patterns in $\mathcal{P}$ can be reported in time $O(n(\log\sigma + \log^2 g_{\text{size}}(\mathcal{P})\lceil \text{k-len}(\mathcal{P})/w\rceil) + occ)$*

where $g_{\text{size}}(\mathcal{P})$ is the size of the variation range of the gap lengths, k-len$(\mathcal{P})$ is the total number of keywords in the patterns and *occ* is is the number of occurrences of the patterns in the text. Note that in the case of unit-length keywords we have k-len$(\mathcal{P}) = \text{len}(\mathcal{P})$. The proposed algorithm obtains a bound similar to the one of [3], in the restricted case of fixed-length gaps. In particular, it is a moderate improvement for $\log g_{\text{size}}(\mathcal{P}) = o(\sqrt{\log w})$. Moreover, it is also practical. For this reason, it provides an effective alternative when α or α' is large. For more details on the motivation, one additional result and an experimental evaluation of the algorithms against the state of the art see [4]. The proposed algorithms are fast in practice, and preferable if all the strings in the patterns have unit length.

Let Σ^* denote the set of all possible sequences over Σ. $|S|$ is the length of string S, $S[i], i \geq 0$, denotes its $(i+1)$-th character, and $S[i \ldots j]$ denotes its substring ranging from i to j. For any two strings S and S', we say that S' is a suffix of S (in symbols, $S' \sqsupseteq S$) if $S' = S[i \ldots |S|-1]$, for some $0 \leq i < |S|$. A gapped pattern P is of the form $S_1 \cdot j_1 \cdot S_2 \cdot \ldots \cdot j_{\ell-1} \cdot S_\ell$, where $S_i \in \Sigma^*$, $|S_i| \geq 1$, is the i-th string (keyword) and $j_i \geq 0$ is the length of the gap between keywords S_i and S_{i+1}, for $i = 1, \ldots, \ell$. We say that P occurs in a string T at ending position i if $T[i-m+1 \ldots i] = S_1 \cdot A_1 \cdot S_2 \cdot \ldots \cdot A_{\ell-1} \cdot S_\ell$, where $A_i \in \Sigma^*$, $|A_i| = j_i$, for $1 \leq i \leq \ell - 1$, and $m = \sum_{i=1}^{\ell} |S_i| + \sum_{i=1}^{\ell-1} j_i$. In this case we write $P \sqsupseteq_g T_i$. We denote by k-len$(P) = \ell$ the number of keywords in P. The gapped pattern $P_i = S_1 \cdot j_1 \cdot S_2 \cdot \ldots \cdot j_{i-1} \cdot S_i$ is the prefix of P of length $i \leq \ell$.

We use some bitwise operations following the standard C language notation: &, |, ~, ≪ for `and`, `or`, `not` and `left shift`, respectively. The position of the most significant non-zero bit of a word x is equal to $\lfloor \log_2(x) \rfloor$.

2 Online Algorithm for Matching a Set of Gapped Patterns

Let P^k be the k-th pattern in $\mathcal{P}$. We adopt the superscript notation for S_i, j_i and P_l with the same meaning. We define the set

$$D_i = \{(k, l) \mid P^k_l \sqsupseteq_g T_i\},$$

of the prefixes of the patterns that occur at position i in T, for $i = 0, \ldots, n-1$, $1 \leq k \leq |\mathcal{P}|$ and $1 \leq l \leq \text{k-len}(P^k)$. From the definition of D_i it follows that the pattern P^k occurs in T at position i if and only if $(k, \text{k-len}(P^k)) \in D_i$. Let $\mathcal{K} = \{1, \ldots, \text{k-len}(\mathcal{P})\}$ be the set of indices of the keywords in $\mathcal{P}$ and let $\bar{T}_i \subseteq \mathcal{K}$ be the set of indices of the matching keywords in T ending at position i. The sequence $\bar{T}_i$, for $0 \leq i < n$, is basically a new text with character classes over

$\mathcal{K}$. We replace each pattern $S_1 \cdot j_1 \cdot S_2 \cdot \ldots \cdot j_{\ell-1} \cdot S_\ell$ in $\mathcal{P}$ with the pattern $\bar{S}_1 \cdot \bar{j}_1 \cdot \bar{S}_2 \cdot \ldots \cdot \bar{j}_{\ell-1} \cdot \bar{S}_\ell$, with unit-length keywords over the alphabet $\mathcal{K}$, where $\bar{S}_i \in \mathcal{K}$ and $\bar{j}_i = j_i + |S_{i+1}| - 1$, for $1 \leq i < \ell$.

The sets D_i can be computed using the following lemma:

Lemma 1. *Let $\mathcal{P}$ and T be a set of gapped patterns and a text of length n, respectively. Then $(k,l) \in D_i$, for $1 \leq k \leq |\mathcal{P}|$, $1 \leq l \leq$ k-len(P^k) and $i = 0, \ldots, n-1$, if and only if*

$$(l = 1 \text{ or } (k, l-1) \in D_{i-1-\bar{j}^k_{l-1}}) \text{ and } \bar{S}^k_l \in \bar{T}_i.$$

The idea is to match the transformed patterns against the text $\bar{T}$. Let $g_{\min}(\mathcal{P})$ and $g_{\max}(\mathcal{P})$ denote the minimum and maximum gap length in the patterns, respectively. We also denote with $g_{\text{size}}(\mathcal{P}) = g_{\max}(\mathcal{P}) - g_{\min}(\mathcal{P}) + 1$ the size of the variation range of the gap lengths. For each position i in T, the main steps of the algorithm are i) compute the set $\bar{T}_i$ in $O(\log \sigma)$ time using the Aho-Corasick (AC) automaton [1] for the set of distinct keywords in $\mathcal{P}$; ii) compute the set D_i, using Lemma 1 and word-level parallelism, in time $O(g_{\text{w-span}} \lceil \text{k-len}(\mathcal{P})/w \rceil)$, where $1 \leq g_{\text{w-span}} \leq w$ is the maximum number of distinct gap lengths that span a single word in our encoding. We also describe how to obtain an equivalent set of patterns with $O(\log g_{\text{size}}(\mathcal{P}))$ distinct gap lengths at the price of $O(\log g_{\text{size}}(\mathcal{P}))$ new keywords per gap, thus achieving $O(\log^2 g_{\text{size}}(\mathcal{P}) \lceil \text{k-len}(\mathcal{P})/w \rceil)$ time.

Let Q denote the set of states of the AC automaton, *root* the initial state and *label*(q) the string which labels the path from state *root* to q, for any $q \in Q$. The transition function $\delta(q, c)$ is defined as the unique state q' such that *label*(q') is the longest suffix of *label*$(q) \cdot c$. We also store for each state q a pointer $f_o(q)$ to the state q' such that *label*(q') is the longest suffix of *label*(q) that is also a keyword, if any. Let

$$B(q) = \{(k, l) \mid S^k_l \sqsupseteq \mathit{label}(q)\},$$

be the set of all the occurrences of keywords in the patterns in $\mathcal{P}$ that are suffixes of *label*(q), for any $q \in Q$. We preprocess $B(q)$ for each state q such that *label*(q) is a keyword and compute it for any other state using $B(f_o(q))$. Let G be the set of all the distinct gap lengths in the patterns. In addition to the sets $B(q)$, we preprocess also a set $C(g)$, for each $g \in G$, defined as follows:

$$C(g) = \{(k, l) \mid \bar{j}^k_l = g\},$$

for $1 \leq k \leq |\mathcal{P}|$ and $1 \leq l <$ k-len(P^k). We encode the sets D_i, $B(q)$ and $C(g)$ as bit-vectors of k-len$(\mathcal{P})$ bits. The generic element (k, l) is mapped onto bit $\sum_{i=1}^{k-1} \text{k-len}(P^i) + \text{k-len}(P^k_{l-1})$, where k-len$(P^k_0) = 0$ for any k. We denote with D_i, $\mathsf{B}(q)$ and $\mathsf{C}(g)$ the bit-vectors representing the sets D_i, $B(q)$ and $C(g)$, respectively. We also compute two additional bit-vectors I and M, such that the bit corresponding to the element $(k, 1)$ in I and $(k, \text{k-len}(P^k))$ in M is set to 1, for $1 \leq k \leq |\mathcal{P}|$. We basically mark the first and the last bit of each pattern, respectively. Let H_i be the bit-vector equal to the bitwise `or` of the bit-vectors

$$\mathsf{D}_{i-1-g} \,\&\, \mathsf{C}(g), \tag{1}$$

```
GQ-MATCHER-preprocess ($\mathcal{P}$, $T$)
 1. $(\delta, root, \mathsf{B}, f_o) \leftarrow AC(\mathcal{P})$
 2. $G \leftarrow \emptyset$
 3. $m \leftarrow$ k-len($\mathcal{P}$)
 4. $\mathsf{I} \leftarrow 0^m, \mathsf{M} \leftarrow 0^m$
 5. for $g = 0, \ldots, g_{\max}(\mathcal{P})$ do $\mathsf{C}(g) \leftarrow 0^m$
 6. $l \leftarrow 0$
 7. for $S_1 \cdot j_1 \cdot S_2 \cdot \ldots \cdot j_{\ell-1} \cdot S_\ell \in \mathcal{P}$ do
 8.     $\mathsf{I} \leftarrow \mathsf{I} \mid 1 \ll l$
 9.     for $k = 1, \ldots, \ell$ do
10.         if $k = \ell$ then
11.             $\mathsf{M} \leftarrow \mathsf{M} \mid 1 \ll l$
12.         else $g \leftarrow j_k + |S_{k+1}| - 1$
13.             $\mathsf{C}(g) \leftarrow \mathsf{C}(g) \mid 1 \ll l$
14.             $G \leftarrow G \cup \{g\}$
15.         $l \leftarrow l + 1$
```

```
GQ-MATCHER-search ($\mathcal{P}$, $T$)
 1. $q \leftarrow root$
 2. for $i = 0, \ldots, |T| - 1$ do
 3.     $q \leftarrow \delta(q, T[i]), \mathsf{H} \leftarrow 0^m$
 4.     for $g \in G$ do
 5.         $\mathsf{H} \leftarrow \mathsf{H} \mid (\mathsf{D}_{i-1-g} \;\&\; \mathsf{C}(g))$
 6.     $\mathsf{D}_i \leftarrow ((\mathsf{H} \ll 1) \mid \mathsf{I}) \;\&\; \mathsf{B}(f_o(q))$
 7.     $\mathsf{H} \leftarrow \mathsf{D}_i \;\&\; \mathsf{M}$
 8.     REPORT($\mathsf{H}$)
```

```
REPORT($\mathsf{H}$)
 1. while $\mathsf{H} \neq 0^m$ do
 2.     $k \leftarrow \lfloor \log_2(\mathsf{H}) \rfloor$
 3.     report($k$)
 4.     $\mathsf{H} \leftarrow \mathsf{H} \;\&\; \sim(1 \ll k)$
```

Fig. 1. The GQ-MATCHER algorithm for the string matching problem with gapped patterns

for each $g \in G$. Then the corresponding set H_i is equal to

$$\bigcup_{g \in G} \{(k,l) \mid (k,l) \in D_{i-1-g} \wedge \bar{j}_l^k = g\} .$$

Let $q_{-1} = root$ and $q_i = \delta(q_{i-1}, T[i])$ be the state of the AC automaton after reading symbol $T[i]$. It is not hard to see that $B(f_o(q_i))$ encodes the set $\bar{T}_i$. The bit-vector D_i can then be computed using the following bitwise operations:

$$\mathsf{D}_i \leftarrow ((\mathsf{H}_i \ll 1) \mid \mathsf{I}) \;\&\; \mathsf{B}(f_o(q_i))$$

which correspond to the relation

$$\{(k,l) \mid (l = 1 \vee (k, l-1) \in H_i) \wedge (k,l) \in B(f_o(q_i))\} .$$

To report all the patterns that match at position i it is enough to iterate over all the bits set in D_i & M. The algorithm, named GQ-MATCHER, is given in Figure 1.

The bit-vector H_i can be constructed in time $O(g_{\text{w-span}}\lceil \text{k-len}(\mathcal{P})/w \rceil)$, $1 \leq g_{\text{w-span}} \leq w$, as follows: we compute Equation 1 for each word of the bit-vector separately, starting from the least significant one. For a given word with index j, we have to compute equation 1 only for each $g \in G$ such that the j-th word of $C(g)$ has at least one bit set. Each position in the bit-vector is spanned by exactly one gap, so the number of such g is at most w. Hence, if we maintain, for each word index j, the list G_j of all the distinct gap lengths that span the corresponding positions, we can compute H_i in time $\sum_{j=1}^{\lceil \text{k-len}(\mathcal{P})/w \rceil} |G_j|$, which yields the advertised bound by replacing $|G_j|$ with $g_{\text{w-span}} = \max_j |G_j|$.

The time complexity of the searching phase of the algorithm is then $O(n(\log \sigma + g_{\text{w-span}}\lceil \text{k-len}(\mathcal{P})/w \rceil) + occ)$, while the space complexity is $O(\text{len}(\mathcal{P}) + (g_{\max}(\mathcal{P}) + \text{k-len}(\mathcal{P}))\lceil \text{k-len}(\mathcal{P})/w \rceil)$.

The GQ-MATCHER algorithm is preferable only when $g_{\text{w-span}} \ll w$. We now show how to improve the time complexity in the worst-case by constructing an equivalent set of patterns with $O(\log g_{\text{size}}(\mathcal{P}))$ distinct gap lengths. W.l.o.g. we assume that $g_{\min}(\mathcal{P})$ and $g_{\max}(\mathcal{P})$ are a power of two (if they are not we round them down and up, respectively, to the nearest power of two). Let $lsb(n)$ be the bit position of the least significant bit set in the binary encoding of n, for $n \geq 1$. Observe that, for any positive $g \in G$, the minimum and maximum value for $lsb(g)$ are $\log g_{\min}(\mathcal{P})$ and $\log g_{\max}(\mathcal{P})$, respectively, and the number of bits set in the binary encoding of g is $O(\log g_{\text{size}}(\mathcal{P}))$. Let also $G' = \{0\} \cup \{2^i \mid \log g_{\min}(\mathcal{P}) \leq i \leq \log g_{\max}(\mathcal{P})\}$. We augment the alphabet Σ with a wildcard symbol $*$ that matches any symbol of the original alphabet and define by recursion the function

$$\phi(g) = \begin{cases} g & \text{if } g \in G' \\ (2^{lsb(g)} - 1) \cdot * \cdot \phi(g - 2^{lsb(g)}) & \text{otherwise} \end{cases}$$

that maps a gap length g onto a concatenation of l gap lengths and $l-1$ wildcard symbols, where l is the number of bits set in the binary encoding of g if g is positive or 1 otherwise. By definition, all the gaps in the resulting sequence belong to the set $G'' = G' \cup \{2^i - 1 \mid \log g_{\min}(\mathcal{P}) \leq i < \log g_{\max}(\mathcal{P})\}$. We generate a new set of patterns $\mathcal{P}'$ from $\mathcal{P}$, by transforming each pattern $\bar{S}_1 \cdot \bar{j}_1 \cdot \bar{S}_2 \cdot \ldots \cdot \bar{j}_{\ell-1} \cdot \bar{S}_\ell$ in $\mathcal{P}$ into the equivalent pattern $\bar{S}_1 \cdot \phi(\bar{j}_1) \cdot \bar{S}_2 \cdot \ldots \cdot \phi(\bar{j}_{\ell-1}) \cdot \bar{S}_\ell$. Observe that extending the algorithm presented above to handle wildcard symbols is straightforward. By definition of ϕ we have that k-len$(\mathcal{P}') < \log g_{\text{size}}(\mathcal{P})$k-len$(\mathcal{P})$, since the number of gaps that are split is at most k-len$(\mathcal{P}) - |\mathcal{P}|$ and the number of wildcard symbols that are added per gap is at most $\log g_{\text{size}}(\mathcal{P})$. The number of words needed for a bit-vector is then $< \lceil \log g_{\text{size}}(\mathcal{P})\text{k-len}(\mathcal{P})/w \rceil \leq \log g_{\text{size}}(\mathcal{P}) \lceil \text{k-len}(\mathcal{P})/w \rceil$. In this way we obtain an equivalent set of patterns such that the set G of distinct gap lengths is contained in G'' and so its cardinality is $O(\log g_{\text{size}}(\mathcal{P}))$.

References

1. Aho, A.V., Corasick, M.J.: Efficient string matching: An aid to bibliographic search. Commun. ACM 18(6), 333–340 (1975)
2. Bille, P., Gørtz, I.L., Vildhøj, H.W., Wind, D.K.: String matching with variable length gaps. Theor. Comput. Sci. 443, 25–34 (2012)
3. Bille, P., Thorup, M.: Regular expression matching with multi-strings and intervals. In: Charikar, M. (ed.) SODA, pp. 1297–1308. SIAM (2010)
4. Giaquinta, E., Fredriksson, K., Grabowski, S., Tomescu, A.I., Ukkonen, E.: Motif matching using gapped patterns. CoRR abs/1306.2483 (2013)
5. Giaquinta, E., Grabowski, S., Ukkonen, E.: Fast matching of transcription factor motifs using generalized position weight matrix models. Journal of Computational Biology 20(9), 1–10 (2013)
6. Haapasalo, T., Silvasti, P., Sippu, S., Soisalon-Soininen, E.: Online dictionary matching with variable-length gaps. In: Pardalos, P.M., Rebennack, S. (eds.) SEA 2011. LNCS, vol. 6630, pp. 76–87. Springer, Heidelberg (2011)
7. Sippu, S., Soisalon-Soininen, E.: Online matching of multiple regular patterns with gaps and character classes. In: Dediu, A.-H., Martín-Vide, C., Truthe, B. (eds.) LATA 2013. LNCS, vol. 7810, pp. 523–534. Springer, Heidelberg (2013)

Domino Graphs and the Decipherability of Directed Figure Codes

Włodzimierz Moczurad[*]

Institute of Computer Science, Faculty of Mathematics and Computer Science, Jagiellonian University; Łojasiewicza 6, 30-348 Kraków, Poland
wkm@ii.uj.edu.pl

Abstract. We consider several kinds of decipherability of directed figure codes, where directed figures are defined as labelled polyominoes with designated start and end points, equipped with catenation that may use a merging function for overlapping regions. This setting extends decipherability questions from words to 2D structures. In this paper we develop a (variant of) domino graph that will allow us to decide some of the decipherability kinds by searching the graph for specific paths. Thus the main result characterizes directed figure decipherability by graph properties.

1 Introduction

The term *unique decipherability* refers to a property of a set of words X where every message composed from these words can be uniquely decoded, *i.e.* an exact sequence of words is recovered. The set X is then called a *uniquely decipherable (UD) code*. However, in some applications it might be sufficient to decode the message with respect to a feature weaker than the exact sequence of codewords—like the multiset, the set or just the number of codewords—giving rise to three kinds of decipherability, known as *multiset* (MSD), *set* (SD) and *numeric decipherability* (ND), respectively.

In [3] we introduced directed figures defined as labelled polyominoes with designated start and end points, equipped with catenation operation that uses a merging function for overlapping regions. We proved that verification whether a given finite set of directed figures is a UD code is decidable. On the other hand, a directed figure model with no merging function, where catenation of figures is only possible when they do not overlap, has again undecidable UD testing [2]. In [4] we extended the previous results by considering not just UD codes, but also MSD, SD and ND codes over directed figures.

In the present paper we define a variant of domino graphs that allows us to decide some of the decipherability kinds by searching the graph for specific paths. Thus the main result characterizes directed figure decipherability by graph properties.

[*] Supported by National Science Centre (NCN) grant no. 2011/03/B/ST6/00418. This is an extended abstract of a paper submitted to *Schedae Informaticae.*

© Springer-Verlag Berlin Heidelberg 2013

2 Preliminaries

Let Σ be a finite, non-empty alphabet. A translation by vector $u \in \mathbb{Z}^2$ is denoted by tr_u.

Definition 1 (Directed figure, [3]). *Let $D \subseteq \mathbb{Z}^2$ be finite and non-empty, $b, e \in \mathbb{Z}^2$ and $l : D \to \Sigma$. A quadruple $f = (D, b, e, l)$ is a* directed figure *(over Σ) with* domain $\mathrm{dom}(f) = D$, start point $\mathrm{begin}(f) = b$, end point $\mathrm{end}(f) = e$, labelling function $\mathrm{label}(f) = l$. *The* translation vector *of f is defined as* $\mathrm{tran}(f) = \mathrm{end}(f) - \mathrm{begin}(f)$. *Additionally, the* empty directed figure *ε is defined as $(\emptyset, (0,0), (0,0), \{\})$, where $\{\}$ denotes a function with an empty domain. The set of all directed figures over Σ is denoted by $\Sigma^\diamond$.*

Definition 2 (Catenation, [2]). *Let $x = (D_x, b_x, e_x, l_x)$ and $y = (D_y, b_y, e_y, l_y)$ be directed figures. If $D_x \cap \mathrm{tr}_{e_x - b_y}(D_y) = \emptyset$, a* catenation *of x and y is defined as $x \circ y = (D_x \cup \mathrm{tr}_{e_x - b_y}(D_y), b_x, \mathrm{tr}_{e_x - b_y}(e_y), l)$, where*

$$l(z) = \begin{cases} l_x(z) & \text{for } z \in D_x, \\ \mathrm{tr}_{e_x - b_y}(l_y)(z) & \text{for } z \in \mathrm{tr}_{e_x - b_y}(D_y). \end{cases}$$

If $D_x \cap \mathrm{tr}_{e_x - b_y}(D_y) \neq \emptyset$, catenation of x and y is not defined.

Definition 3 (m-catenation, [3]). *Let $x = (D_x, b_x, e_x, l_x)$ and $y = (D_y, b_y, e_y, l_y)$ be directed figures. An* m-catenation *of x and y with respect to a* merging function *$m : \Sigma \times \Sigma \to \Sigma$ is defined as $x \circ_m y = (D_x \cup \mathrm{tr}_{e_x - b_y}(D_y), b_x, \mathrm{tr}_{e_x - b_y}(e_y), l)$, where*

$$l(z) = \begin{cases} l_x(z) & \text{for } z \in D_x \setminus \mathrm{tr}_{e_x - b_y}(D_y), \\ \mathrm{tr}_{e_x - b_y}(l_y)(z) & \text{for } z \in \mathrm{tr}_{e_x - b_y}(D_y) \setminus D_x, \\ m(l_x(z), \mathrm{tr}_{e_x - b_y}(l_y)(z)) & \text{for } z \in D_x \cap \mathrm{tr}_{e_x - b_y}(D_y). \end{cases}$$

3 Codes

Note that by a *code* (over Σ, with no further attributes) we mean any finite non-empty subset of $\Sigma^\diamond \setminus \{\varepsilon\}$. The symbol $\bullet$ is used instead of $\circ$ and $\circ_m$ where context makes it clear which catenation type applies.

Definition 4 (UD, MSD, SD and ND codes, [4]). *A code X over Σ is a* UD (resp. MSD, SD or ND) code, *if for any $x_1, \ldots, x_k$, $y_1, \ldots, y_l \in X$ the equality $x_1 \circ \cdots \circ x_k = y_1 \circ \cdots \circ y_l$ implies that $(x_1, \ldots, x_k)$ and $(y_1, \ldots, y_l)$ are equal as sequences (resp. $\{\!\{x_1, \ldots, x_k\}\!\}$ and $\{\!\{y_1, \ldots, y_l\}\!\}$ are equal as multisets, $\{x_1, \ldots, x_k\}$ and $\{y_1, \ldots, y_l\}$ are equal as sets or $k = l$). UD, MSD, SD and ND m-codes are defined similarly, by replacing $\circ$ with $\circ_m$.*

Definition 5 (Two-sided and one-sided codes, [4]). *Let $X = \{x_1, \ldots, x_n\}$ be a code over Σ. If there exist $\alpha_1, \ldots, \alpha_n \in \mathbb{N}$, not all equal to zero, such that $\sum_{i=1}^{n} \alpha_i \mathrm{tran}(x_i) = (0,0)$, then X is called* two-sided. *Otherwise it is called* one-sided.

Theorem 1 ([4]). *Let X be a one-sided code over Σ. It is decidable whether X is a {UD, MSD, SD or ND} {code or m-code}.*

4 Domino Graphs for Decipherability Testing

We now develop a variant of the domino graph as introduced by [1]. It will allow us to decide some of the decipherability types by searching the graph for specific paths.

Throughout this section we fix a "merging type" (*i.e.* either a merging function m, or no merging function) and use it for all catenations. Note that reduced configurations, and hence the domino graph, depend on it. We also assume that all codes are one-sided, since reduced configurations are not defined for two-sided codes. Definitions of a *configuration* and *reduced configuration* come from the proof of Theorem 1 and can be found in [4].

Let $rc(C)$ denote the reduced configuration associated with a configuration C. Given a figure $z \in X$ we define an *extension of a reduced configuration* $rc((x_1, \ldots, x_k), (y_1, \ldots, y_l))$ *by* (z, ε) as a new reduced configuration $rc((x_1, \ldots, x_k, z), (y_1, \ldots, y_l))$. It is clear that the extension is well-defined, since $rc((x_1, \ldots, x_k), (y_1, \ldots, y_l)) = rc((x'_1, \ldots, x'_{k'}), (y'_1, \ldots, y'_{l'}))$ implies $rc((x_1, \ldots, x_k, z), (y_1, \ldots, y_l)) = rc((x'_1, \ldots, x'_{k'}, z), (y'_1, \ldots, y'_{l'}))$. Extension by (ε, z) is defined similarly. Note that in the non-merging case a particular extension may be undefined.

A reduced configuration, as defined in Theorem 1, is a pair $((e_L, l_L), (e_R, l_R))$ with end points $e_L, e_R \in \mathbb{Z}^2$ and labellings l_L, l_R which are partial mappings $\mathbb{Z}^2 \to \Sigma$. Informally, the extension of $((e_L, l_L), (e_R, l_R))$ by (z, ε) is the reduced configuration $((e'_L, l'_L), (e_R, l_R))$, where $e'_L = e_L + \text{tran}(z)$ and l'_L is obtained by "catenating" l_L with z and constraining the domain appropriately.

A reduced configuration is called *final*, if it is of the form $((e, l), (e, l))$, *i.e.* its left and right components are equal. Note that $rc(C)$ is final iff $L_\bullet(C) = R_\bullet(C)$.

Let $RC(X)$ be the set of all reduced configurations over X which satisfy the RC criteria, *i.e.* $RC(X) = \{rc((x_i), (y_j)) \mid (x_i), (y_j)\}$, with (x_i) and (y_j) ranging over all finite, non-empty sequences of elements of X satisfying the RC criteria. By Theorem 1, $RC(X)$ is finite for every one-sided code X.

Definition 6 (Domino graph). *Let X be a one-sided code over Σ. A* domino graph *of X is the directed graph (V, E) with $V = RC(X) \cup \{0\}$ and $E = E_0 \cup E_1$, where*

- *E_0 contains all edges $(0, v)$ such that $v \in RC(X)$ and $v = rc((x), (y))$ for some $x, y \in X$, $x \neq y$,*
- *E_1 contains all edges (v_1, v_2) such that $v_1, v_2 \in RC(X)$, v_1 is not final and v_2 is an extension of v_1 by (z, ε) or (ε, z), for some $z \in X$.*

Additionally, we define a domino function $d : E \to \wp((X \cup \{\varepsilon\}) \times (X \cup \{\varepsilon\}))$ *that associates labels to the edges:*

$$d(0, v) = \{(x, y) \in X \times X \mid v = rc((x), (y))\}$$
$$d(v_1, v_2) = \{(x, y) \in (X \times \{\varepsilon\}) \cup (\{\varepsilon\} \times X) \mid v_2 \textit{ is an extension of } v_1 \textit{ by } (x, y)\}.$$

Observe that for an edge (v_1, v_2) with $v_1 \neq 0$, $d(v_1, v_2)$ either contains pairs of the form (z, ε), or (ε, z), but not both. Moreover, if for instance (z, ε) and

$(z', \varepsilon) \in d(v_1, v_2)$ then $\mathrm{tran}(z) = \mathrm{tran}(z') \neq (0,0)$, since X is one-sided and two reduced configurations v_1 and v_2 determine a unique translation vector required to extend v_1 to v_2. Hence, $d(0, v)$ are the only values of d that contain pairs with both figures non-empty.

The domino function can be extended to paths in a domino graph G: given a path $p = (e_1, e_2, \dots, e_n)$, where e_i's are edges in G, define

$$d(p) = d(e_1) \bullet d(e_2) \bullet \cdots \bullet d(e_n),$$

with $\bullet$ denoting the obvious extension of figure catenation to sets of figure pairs.

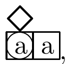
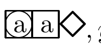
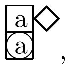
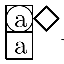
Given a path $p = (e_1, e_2, \dots, e_n)$, we also define a *realization of* p to be any sequence of figure pairs $((x_1, y_1), (x_2, y_2), \dots, (x_n, y_n))$ such that $(x_i, y_i) \in d(e_i)$. Note that $x_i, y_i \in X \cup \{\varepsilon\}$.

For a path p starting in the vertex 0, $d(p)$ describes an attempt at constructing two distinct factorizations of some figure. If p can be made to reach a final vertex, this is indeed accomplished (p is "successful") and we know that X is not a UD (m-)code. To check for other decipherability kinds, all successful paths have to be checked for specific properties, similar to the conditions in the proof of Theorem 1. This is reflected in the following theorem:

Theorem 2. *Let X be a one-sided code over Σ.*

1. *X is not a UD (m-)code iff the domino graph of X contains a path from 0 to a final vertex.*
2. *X is not an MSD (m-)code iff the domino graph of X contains a path p from 0 to a final vertex such that there exists a realization of p, $((x_1, y_1), (x_2, y_2), \dots, (x_n, y_n))$, with $\{\!\{x_1, \dots, x_n\}\!\}$ and $\{\!\{y_1, \dots, y_n\}\!\}$ being different as multisets.*
3. *X is not an SD (m-)code iff the domino graph of X contains a path p from 0 to a final vertex such that there exists a realization of p, $((x_1, y_1), (x_2, y_2), \dots, (x_n, y_n))$, with $\{x_1, \dots, x_n\}$ and $\{y_1, \dots, y_n\}$ being different as sets.*
4. *X is not an ND (m-)code iff the domino graph of X contains a path p from 0 to a final vertex such that there exists a realization of p, $((x_1, y_1), (x_2, y_2), \dots, (x_n, y_n))$, with the number of non-empty x_i's different from the number of non-empty y_i's.*

The following example shows a domino graph for a non-decipherable code. Assume $\Sigma = \{a\}$ and $m = \{(a, a) \mapsto a\}$. Edge labels denote values of the domino function d; note that $|d(e)| = 1$ for all edges. For the sake of brevity, the notation of reduced configurations omits inner parentheses and commas. Final vertices are underlined.

Example 1. Consider $X = \{w =$ , $x =$ , $y =$ , $z =$  $\}$ and set $\tau_E = (1,1), \tau_W = (-\frac{1}{2}, -\frac{1}{2}), \tau_S = (0, -1), \tau_N = (-\frac{1}{2}, 0)$. We omit pairs that can be obtained from another pair by exchanging the elements; this does not prevent us from discovering any of the properties characterized by Theorem 2.

Note that the graph contains two successful paths, $0 \rightarrow rc(w,y) \rightarrow rc(wx,y) \rightarrow rc(wx,yz)$ and $0 \rightarrow rc(w,y) \rightarrow rc(wz,y) \rightarrow rc(wz,yz) \rightarrow rc(wzz,yz)$. The former disproves UD, MSD and SD decipherability of X (but not ND); the latter disproves all four decipherability kinds.

$$\begin{tikzcd}
\bigcirc \arrow[r, "{(w,x)}"] \arrow[d, "{(w,y)}"] & \boxed{rc(w,x)} & & \\
\boxed{rc(w,y)} \arrow[d, "{(w,\varepsilon)}"] \arrow[dr, "{(x,\varepsilon)}"] \arrow[drr, "{(z,\varepsilon)}"] & & & \\
\boxed{rc(ww,y)} \arrow[d, "{(\varepsilon,z)}"] & \boxed{rc(wx,y)} \arrow[d, "{(\varepsilon,z)}"] & \boxed{rc(wz,y)} \arrow[d, "{(\varepsilon,z)}"] & \\
\boxed{rc(ww,yz)} & \boxed{\underline{rc(wx,yz)}} & \boxed{rc(wz,yz)} \arrow[d, "{(x,\varepsilon)}"] \arrow[dr, "{(z,\varepsilon)}"] & \\
 & & \boxed{rc(wzx,yz)} \arrow[d, "{(\varepsilon,x)}"] & \boxed{\underline{rc(wzz,yz)}} \\
 & & \boxed{rc(wzx,yzx)} \arrow[d, "{(x,\varepsilon)}"] & \\
 & & \boxed{rc(wzxx,yzx)} \arrow[u, bend left, "{(\varepsilon,x)}"] &
\end{tikzcd}$$

References

1. Head, T., Weber, A.: Deciding multiset decipherability. IEEE Transactions on Information Theory 41(1), 291–297 (1995)
2. Kolarz, M.: The code problem for directed figures. Theoretical Informatics and Applications RAIRO 44(4), 489–506 (2010)
3. Kolarz, M., Moczurad, W.: Directed figure codes are decidable. Discrete Mathematics and Theoretical Computer Science 11(2), 1–14 (2009)
4. Kolarz, M., Moczurad, W.: Multiset, set and numerically decipherable codes over directed figures. In: Arumugam, S., Smyth, B. (eds.) IWOCA 2012. LNCS, vol. 7643, pp. 224–235. Springer, Heidelberg (2012)

A Pretty Complete Combinatorial Algorithm for the Threshold Synthesis Problem

Christian Schilling[1], Jan-Georg Smaus[2], and Fabian Wenzelmann[1]

[1] Institut für Informatik, Universität Freiburg, Germany
[2] IRIT, Université de Toulouse, France
smaus@irit.fr

1 Introduction

A *linear pseudo-Boolean constraint* (LPB) [1,4,5] is an expression of the form $a_1\ell_1 + \ldots + a_m\ell_m \geq d$. Here each ℓ_i is a *literal* of the form x_i or $1 - x_i$. An LPB can be used to represent a Boolean function; e.g. $2x_1 + x_2 + x_3 \geq 2$ represents the same function as the propositional formula $x_1 \vee (x_2 \wedge x_3)$.

Functions that can be represented by a single LPB are called *threshold functions*. The problem of finding the LPB for a threshold function given as disjunctive normal form (DNF) is called *threshold synthesis problem*. The reference on Boolean functions [4] formulates the research challenge of recognising threshold functions through an entirely combinatorial procedure. In fact, such a procedure had been proposed in [3,2] and was later reinvented by us [7]. In this paper, we report on an implementation of this procedure for which we have run experiments for up to $m = 22$. It can solve the biggest problems in a couple of seconds.

There is another procedure solving this problem using linear programming [4], which we also implemented and compared to the combinatorial one.

2 Preliminaries

An **m-dimensional Boolean function** f is a function $Bool^m \to Bool$. A **linear pseudo-Boolean constraint** (LPB) is an inequality of the form

$$a_1\ell_1 + \ldots + a_m\ell_m \geq d \qquad a_i \in \mathbb{N}, d \in \mathbb{Z}, \ell_i \in \{x_i, 1 - x_i\}. \tag{1}$$

We call the a_i **coefficients** and d the **threshold**. A **DNF** is a formula of the form $c_1 \vee \ldots \vee c_n$ where each **clause** c_j is a conjunction of literals.

It is easy to see that an LPB can only represent *monotone* functions, i.e., functions represented by a DNF where each variable occurs in only one polarity. Without loss of generality, we assume that this polarity is positive.

3 The Combinatorial Algorithm

For space reasons, we do not give a general definition of our algorithm but rather illustrate it using a running example: $\phi \equiv$
$(x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_1 \wedge x_4) \vee (x_1 \wedge x_5) \vee (x_2 \wedge x_3) \vee (x_2 \wedge x_4) \vee (x_3 \wedge x_4 \wedge x_5)$.

© Springer-Verlag Berlin Heidelberg 2013

$(x_1 \wedge x_2) \vee (x_1 \wedge x_3)$ $\vee(x_1 \wedge x_4) \vee (x_1 \wedge x_5)$ $\vee(x_2 \wedge x_3) \vee (x_2 \wedge x_4)$ $\vee(x_3 \wedge x_4 \wedge x_5)$	$(x_2 \wedge x_3)\vee$ $(x_2 \wedge x_4)\vee$ $(x_3 \wedge x_4 \wedge x_5)$	$x_3 \wedge x_4 \wedge x_5$	*false* $x_4 \wedge x_5$	*false* *false* x_5	*false* *false* *false* *true*
		$x_3 \vee x_4$	x_4 *true*	*false* *true* *true*	
	$x_2 \vee x_3 \vee x_4 \vee x_5$	$x_3 \vee x_4 \vee x_5$ *true*	$x_4 \vee x_5$ *true* *true*	x_5 *true* *true* *true*	*false* *true* *true* *true* *true*

Fig. 1. The recursive subproblems for ϕ

Before we start, it should be noted that the basic procedure we describe here is not complete. The issue of completeness is very complicated, and [3] devote 23 pages to it! In our implementation, we have realised an extension of the basic procedure that implements some of the ideas described by [3] but still does not achieve full completeness. As it stands, for up to $m = 7$, our procedure always succeeds; up to $m = 14$, it fails on less than 1% of the threshold functions, while this rate rises up to 18.3% for $m = 22$.

For some DNFs, it is possible to establish a complete order $\succeq$ on the variables which has the following meaning: $x_i \succeq x_j$ iff starting from any given input tuple $X^* \in \mathit{Bool}^m$, setting x_i^* to true is more likely to make the DNF true than setting x_j^* to true. There is a lemma stating that $\succeq$ must be respected by any LPB (if there is one!) representing the DNF, i.e., $x_i \succeq x_j$ implies $a_i \geq a_j$. For ϕ, it is the case that $x_1 \succeq \ldots \succeq x_5$ and so if we find a solution, then $a_1 \geq \ldots \geq a_5$.

Now there is a theorem stating that the problem can be tackled using a special kind of recursion. In ϕ, we can distinguish the clauses that contain x_1 and the ones that do not. This is illustrated in Figure 1. In the leftmost column (column 0), we have ϕ. In column 1, we have two smaller DNFs: on top the clauses of ϕ that do not contain x_1, and on bottom the clauses of ϕ that contain x_1, but with those occurrences of x_1 removed. We say that we *split away* x_1 from ϕ, and we call the two formulae we obtain the *upper* and *lower* successor of ϕ. We thus have two smaller subproblems, and the theorem says that we must find solutions to these subproblems that agree on the coefficients $a_2, \ldots, a_5$ (but differ on the threshold, of course).

Similarly, we can split away x_2 from each DNF in column 1, giving the four formulae of column 2. Observe that the only clause in $x_2 \vee x_3 \vee x_4 \vee x_5$ containing x_2 is x_2, and if we remove x_2 from it, we are left with the empty conjunction which is *true*; hence we have *true* as lowermost formula in column 2.

We continue by splitting away x_3 from the DNFs in column 2. From now on, it is no more the case that the number of DNFs doubles in each step. In fact, thanks to the symmetry of the variables in $x_2 \vee x_3 \vee x_4 \vee x_5$, it happens that the lower successor of $x_3 \vee x_4 \vee x_5$ coincides with the upper successor of *true*, namely *true*. Due to this fact, Figure 1 is *not quite* a tree, as some nodes are shared.

Reducing the size of the datastructure by exploiting symmetries within the DNF is obviously good for the space complexity of our procedure, and is an advantage of [7] compared to [3,2]. In fact, [2] does consider symmetries but only at the global level: in ϕ, the variables x_3 and x_4 are symmetric, but in the subproblems, there are more symmetries.

Observe also that $x_3 \wedge x_4 \wedge x_5$ has no clause not containing x_3, and thus we get the empty DNF ($=$ *false*) as upper successor.

This process is continued until we finally obtain the "tree" in Figure 1. As leaves, it has 12 (rather than $2^5 = 32$ as a construction not exploiting any symmetries would give) occurrences of *true* or *false*.

We now generalise LPBs by recording to what extent thresholds can be shifted without changing the meaning.

Definition 1. Given an LPB $I \equiv \sum_{i=1}^{m} a_i x_i \geq d$, we call s the **minimum threshold** of I if s is the smallest number (possibly $-\infty$) such that for any $s' \in (s, d]$, the LPB $\sum_{i=1}^{m} a_i x_i \geq s'$ represents the same function as I. We call b the **maximum threshold** if b is the biggest number (possibly ∞) such that $\sum_{i=1}^{m} a_i x_i \geq b$ represents the same function as I. We denote by $\sum_{i=1}^{m} a_i x_i \geq (s, b]$ any LPB with minimum threshold s and maximum threshold b.

Now that we have constructed the "tree" containing trivial subproblems as leaves, we must work back from the right to the left: we first find LPBs for the formulae in the rightmost column, which have 0 variables and hence we must determine 0 coefficients. Next to the left, we have formulae that contain (at most) x_5, and we determine LPBs representing these, where we use the same a_5 for all formulae! Then we determine a_4, and so forth.

Instead of giving the according theorem, we stick to our example: Figure 2 is arranged in correspondence to Figure 1 and shows LPBs for all subproblems. In the top line we give the l.h.s. of the LPBs, which is the same for each LPB in a column. In the actual "tree", we list the minimum and maximum threshold of each formula. We show how to construct this "tree".

Observe first that $\sum_{i=6}^{5} a_i x_i \geq (-\infty, 0]$ and $\sum_{i=6}^{5} a_i x_i \geq (0, \infty]$ are LPB representations (with empty sum as l.h.s.) for *true* and *false*, respectively. This explains the entries in column 5.

Next observe that column 5 has three blocks separated by horizontal lines, two of which are non-empty. Consider the uppermost block consisting of four intervals, and within it, the northwest-southeast diagonals, as illustrated by the dashed shapes in the figure to the right. Each diagonal joins two numbers, and we compute the difference between the upper left and the lower right number for each diagonal, i.e., $0 - \infty$, $0 - \infty$, and $0 - 0$, which give $-\infty$, $-\infty$, and 0, respectively. Our theorem

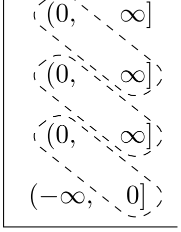


Fig. 3. A block

$4x_1 + 3x_2 +$ $2x_3 + 2x_4 +$ $x_5 \geq \ldots$	$3x_2 +$ $2x_3 + 2x_4 +$ $x_5 \geq \ldots$	$2x_3 + 2x_4 +$ $x_5 \geq \ldots$	$2x_4 +$ $x_5 \geq \ldots$	$x_5 \geq \ldots$	$\sum_{i=6}^{5} a_i x_i$ $\geq \ldots$
$(4,5]$	$(4,5]$	$(4,5]$	$(3,\infty]$ $(2,3]$	$(1,\infty]$ $(1,\infty]$ $(0,1]$	$(0,\infty]$ $(0,\infty]$ $(0,\infty]$ $(-\infty,0]$
		$(1,2]$	$(1,2]$ $(-\infty,0]$	$(1,\infty]$ $(-\infty,0]$ $(-\infty,0]$	
	$(0,1]$	$(0,1]$ $(-\infty,0]$	$(0,1]$ $(-\infty,0]$ $(-\infty,0]$	$(0,1]$ $(-\infty,0]$ $(-\infty,0]$ $(-\infty,0]$	$(0,\infty]$ $(-\infty,0]$ $(-\infty,0]$ $(-\infty,0]$ $(-\infty,0]$

Fig. 2. LPBs for ϕ and its subproblems

states that a_5 must be chosen greater than any of those numbers, and thus in particular greater than 0. The theorem also states that a_5 must be chosen less than any of the differences obtained by taking the northeast-southwest diagonals, i.e. $\infty - 0, \infty - 0, \infty - -\infty$, which however only says that $a_5 < \infty$. In the same way, constraints on a_5 can be collected from the lowermost block, in any case just stating that $a_5 > 0$. We simply choose $a_5 = 1$.

Now, each node in column 4 with upper successor $(s_u, b_u]$ and lower successor $(s_l, b_l]$, is filled by the thresholds $(\max\{s_u, s_l + a_5\}, \min\{b_u, b_l + a_5\}]$. E.g., the topmost $(1, \infty]$ is $(\max\{0, 0 + a_5\}, \min\{\infty, \infty + a_5\}]$.

In the next step, we have to choose a_4 so that

$$\begin{array}{l} \max\{1-\infty, 1-1, \quad 1-0, -\infty-0, \quad 0-0, -\infty-0, -\infty-0\} < a_4 < \\ \min\{\infty-1, \infty-0, \quad \infty - -\infty, 0 - -\infty, \quad 1 - -\infty, 0 - -\infty, 0 - -\infty\}. \end{array}$$

Choosing $a_4 = 2$ will do. Note that the bound $1 - 0 < a_4$ comes from the middle block of column 4 and thus ultimately from $x_3 \vee x_4$. Our algorithm enforces that $a_4 > a_5$, which must hold for an LPB representing $x_3 \vee x_4$.

In the next step, a_3 can also be chosen to be any number > 1 so we choose 2 again. In the next step, $2 < a_2 < 4$ must hold so we choose $a_2 = 3$. Finally, $3 < a_1 < 5$ must hold so we choose $a_1 = 4$. As result we obtain the LPB $4x_1 + 3x_2 + 2x_3 + 2x_4 + x_5 \geq (4, 5]$.

4 Experiments

Both algorithms were implemented in C++ based on a previous implementation in Java [6,8]. For evaluation we used more than 300,000 randomly generated DNFs known to be threshold functions, for $m \leq 22$.

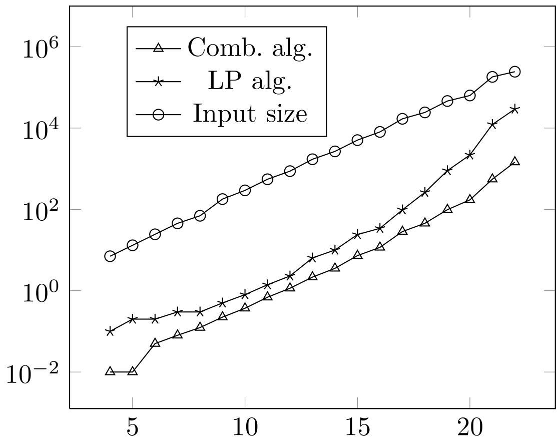


Fig. 4. Runtime

Figure 4 shows the runtime per problem for both algorithms in ms, as well as the problem size. The x-axis shows m. The y-axis is in logarithmic scale. We observe that the combinatorial algorithm could solve problems up to $m = 22$ in a couple of seconds, while the LP algorithm appears to scale worse and needs around 30 seconds for the biggest problems. Second, the runtime seems to be exponential in m. Let us now discuss the problem size. Note that the input to our procedure is a DNF. The combinatorics wants it that the size of the DNFs grows exponentially in m. The size, around 243,000 for $m = 22$, is shown in the figure. The fact that the curve is almost a perfect straight line and appears to be parallel to the curve for the runtime of the combinatorial algorithm shows that the input size increases at the same rate as that runtime, which means that the algorithm appears to run in time linear to the input, whereas the LP algorithm performs worse.

References

1. Chai, D., Kuehlmann, A.: A fast pseudo-Boolean constraint solver. In: Proceedings of the 40th Design Automation Conference, pp. 830–835. ACM (2003)
2. Coates, C.L., Kirchner, R.B., Lewis II, P.M.: A simplified procedure for the realization of linearly-separable switching functions. IRE Transactions on Electronic Computers (1962)
3. Coates, C.L., Lewis II, P.M.: Linearly-separable switching functions. Journal of Franklin Institute 272, 366–410 (1961); Also in an expanded version, GE Research Laboratory, Schenectady, N.Y., Technical Report No.61-RL-2764E
4. Crama, Y., Hammer, P.L.: Boolean Functions: Theory, Algorithms, and Applications. Encyclopedia of Mathematics and its Applications. Cambridge University Press (May 2011)
5. Dixon, H.E., Ginsberg, M.L.: Combining satisfiability techniques from AI and OR. The Knowledge Engineering Review 15, 31–45 (2000)
6. Schilling, C.: Solving the Threshold Synthesis Problem of Boolean Functions by Translation to Linear Programming. Bachelor thesis, Universität Freiburg (2011)
7. Smaus, J.-G.: On boolean functions encodable as a single linear pseudo-Boolean constraint. In: Van Hentenryck, P., Wolsey, L.A. (eds.) CPAIOR 2007. LNCS, vol. 4510, pp. 288–302. Springer, Heidelberg (2007)
8. Wenzelmann, F.: Solving the Threshold Synthesis Problem of Boolean Functions by a Combinatorial Algorithm. Bachelor thesis, Universität Freiburg (2011)

Conjunctive Hierarchical Secret Sharing Scheme Based on MDS Codes

Appala Naidu Tentu[1], Prabal Paul[2], and China Venkaiah Vadlamudi[3]

[1] CR Rao AIMSCS, University of Hyderabad Campus, Hyderabad-500046, India
[2] BITS, Pilani, Goa Campus, GOA-403726, India
[3] SCIS, University of Hyderabad, Hyderabad-500046, India
{naidunit,prabal.paul}@gmail.com, venkaiah@hotmail.com

Abstract. An ideal conjunctive hierarchical secret sharing scheme, constructed based on the Maximum Distance Separable (MDS) codes, is proposed in this paper. The scheme, what we call, is computationally perfect. By computationally perfect, we mean, an authorized set can always reconstruct the secret in polynomial time whereas for an unauthorized set this is computationally hard. Also, in our scheme, the size of the ground field is independent of the parameters of the access structure. Further, it is efficient and requires $O(n^3)$, where n is the number of participants.

Keywords: Computationally perfect, Ideal, Conjunctive hierarchical access structure, MDS code.

1 Introduction

Secret sharing is a cryptographic primitive, which is used to distribute a secret among participants in such a way that an authorized subset of participants can uniquely reconstruct the secret and an unauthorized subset can get no information about the secret in the information theoretic sense. A secret sharing scheme is called ideal if the maximal length of shares and the length of the secret are identical. Secret sharing was first proposed independently by Blakley [2] and Shamir [14]. The family of authorized subsets is known as the access structure. An access structure is said to be monotone if a set is qualified then its superset must also be qualified. Several access structures are proposed in the literature. They include the (t, n)-threshold access structure, the Generalized access structure and the Multipartite access structure. Let $\mathbb{U}$ be the set of n participants and let $2^{\mathbb{U}}$ be its power set. In multipartite access structures, the set of players $\mathbb{U}$ is partitioned into m disjoint entities $\mathbb{U}_1, \mathbb{U}_2, \cdots, \mathbb{U}_m$, called levels and all players in each level play exactly the same role inside the access structure.

Conjunctive hierarchical access structure is a multipartite access structure in which each level $\mathbb{U}_i$ is assigned with a threshold t_i for $1 \leq i \leq m$, and the secret can be reconstructed when, for every i, there are at least t_i shareholders who all belong to levels smaller than or equal to $\mathbb{U}_i$. Formally,

© Springer-Verlag Berlin Heidelberg 2013

$$\Gamma = \{\mathbb{V} \subseteq \mathbb{U} : |\mathbb{V} \cap (\bigcup_{j=1}^{i} \mathbb{U}_j)| \geq t_i, \text{ for all } i \in \{1, 2, \cdots, m\}\}.$$

A secret sharing scheme is a perfect realization of Γ if for all $A \in \Gamma$, the users in A can always reconstruct the secret and for all B not in Γ, the users in B collectively cannot learn anything about the secret, in the information theoretic sense.

The motivation for this study is to come up with an hierarchical scheme that is ideal, efficient, that does not require the ground field to be extremely large, and that offers no restrictions in assigning identities to the users. The proposed scheme is computationally perfect. By computationally perfect, we mean, an authorized set can always reconstruct the secret in polynomial time whereas for an unauthorized set this is computationally hard. This is in contrast to the majority of the schemes found in the literature, which are perfect in a probabilistic manner. A scheme is perfect in a probabilistic manner if either an authorized set may not be able to reconstruct the secret or an unauthorized set may be able to reconstruct the secret with some probability [10].

An $[n, k, d]$ block code over $\mathbb{F}_q$ is called Maximum Distance Separable (MDS) code if distance $d = n - k + 1$. Two important properties, namely,

– Any k columns of a generator matrix are linearly independent and
– Any k symbols of a codeword may be taken as message symbols, of MDS codes,

have been exploited in the construction of our scheme.

Related Work: Shamir [14] pointed out that a hierarchical variant of threshold secret sharing scheme can be introduced simply by assigning larger number of shares to higher level participants. However, such a solution can be easily seen to be not ideal. Kothari [9] introduced a scheme that is a generalization of schemes of Blakley, Shamir, Bloom, and Karnin et al. [2,14,8]. This generalized scheme is used to arrive at a hierarchical scheme, which provides different levels of shares [9]. Brickell [4] offered two schemes for the disjunctive case, both ideal but inefficient. The multilevel threshold scheme by Ghodosi et al. [6] work only for small number of shareholders [11,1].

Tassa [15] and Tassa and Dyn [16] proposed ideal secret sharing schemes, based on Birkhoff interpolation and bivariate interpolation respectively, for several families of multipartite access structures that contain the multilevel and compartmented ones. The problem of secret sharing in hierarchical (or multilevel) structures, was studied under different assumptions also in [1,5]. Linear codes have been used earlier in some constructions of threshold schemes [7,13,8,12]. Blakley and Kabatianski [3] have established that ideal perfect threshold secret sharing schemes and MDS codes are equivalent.

Our Results: In this paper, we propose an ideal secret sharing scheme for conjunctive access structure. This scheme, what we call, is computationally perfect and relies on the following hardness assumption. The construction of this scheme is based on the maximum distance separable (MDS) codes.

Assumption: Let $a \in \mathbb{F}_q$ and $f_i : \mathbb{F}_q \longrightarrow \mathbb{F}_q$, $1 \leq i \leq m$, be a set of distinct one way functions. Also, let $f_i(a) = b_i$ for $1 \leq i \leq m$. Then the computation of a from the knowledge of b_i, $i \in S$, where $S \subseteq \{1, 2, \cdots, m\}$ is computationally hard.

2 Conjunctive Hierarchical Secret Sharing Scheme

Let $U = \bigcup_{i=1}^{m} U_i$ be the set of participants partitioned into m disjoint sets $U_i, 1 \leq i \leq m$. Also, let $|U_i| = n_i$, for $i \in \{1, 2, \cdots, m\}$. Further, let t_1, t_2, $\cdots$, t_{m-1} and t_m be m positive integers such that $t_i < t_{i+1}$ for $1 \leq i \leq m-1$. Denote $\sum_{i=1}^{m} n_i + 1$ by N and $2N - t_m$ by n. Let $s \in \mathbb{F}_q$ be the secret to be shared. Also, let $f_i : \mathbb{F}_q \longrightarrow \mathbb{F}_q$, $1 \leq i \leq m$, be a set of m distinct one way functions.

Overview of the Scheme
Here the secret s to be shared is split as $s = s_1 + s_2 + \cdots + s_m \mod q$. The dealer then selects an $[n, N, n - N + 1]$ MDS code, m distinct one way functions $f_i, 1 \leq i \leq m$, and chooses m codewords of the selected MDS code. The choice of the $i^{th}, 1 \leq i \leq m$, codeword is such that the first component of the codeword is s_i, next n_1 components of the codeword are the images of the shares of the first level participants under the one way function f_i, next n_2 components of the codeword are the images of the shares of the second level participants under the same one way function f_i, and so on it goes upto the images of the shares of the i^{th} level participants under the same one way function f_i. The rest of the components of the codeword are chosen arbitrarily.

$N - t_i$ of these arbitrarily chosen components of the i^{th} codeword are made public so that if any t_i participants from the first i levels cooperate they can, with the help of the $N - t_i$ public shares, reconstruct the i^{th} codeword uniquely and hence can recover the first component, s_i, of this codeword, which is a term in the sum of the partial secrets.

Remark: Except the first component, all the components corresponding to the first i levels of the i^{th} codeword are the images of the shares under the i^{th} one-way function. Since the chosen one-way functions are all distinct, it follows that the knowledge of the components of one of the codeword does not imply the knowledge of the corresponding components in other codewords.

Setup and Distribution Phase:
Following steps constitute this phase.

1. Select an $[n, N, n - N + 1]$ MDS code over $\mathbb{F}_q$.
2. Choose arbitrarily $s_i \in \mathbb{F}_q, 1 \leq i \leq m$, such that $s = s_1 + s_2 + \cdots + s_m$.
3. Choose $v_{i,j}$, $1 \leq i \leq m$, $1 \leq j \leq n_i$, from the elements of $\mathbb{F}_q$. Compute $v_{i,j}^k = f_k(v_{i,j})$, $1 \leq i \leq m$, $1 \leq j \leq n_i$, $i \leq k \leq m$. Distribute $v_{i,j}, 1 \leq i \leq m, 1 \leq j \leq n_i$, to the j^{th} player in the i^{th} compartment.
4. Choose m codewords

$$C_i = (s_i, v_{1,1}^i, v_{1,2}^i, \cdots, v_{1,n_1}^i, v_{2,1}^i, \cdots, v_{2,n_2}^i, \cdots, v_{i,1}^i, v_{i,2}^i, \cdots, v_{i,n_i}^i,$$
$$u_{i,\sum_{j=1}^{i} n_j+2}, u_{i,\sum_{j=1}^{i} n_j+3}, \cdots, u_{i,\sum_{j=1}^{m} n_j+\sum_{j=1}^{m}(n_j)-t_m+2}), \quad 1 \leq i \leq m,$$

of the above mentioned MDS code, where $C_i = m.R_i.G$, m is the vector of length N, chosen by dealer, having first coordinate as partial secret s_i followed by images of the private shares of first i levels $v^i_{1,1}, v^i_{1,2}, \cdots, v^i_{1,n_1}, v^i_{2,1}, \cdots, v^i_{2,n_2}, \cdots, v^i_{i,1}, v^i_{i,2}, \cdots, v^i_{i,n_i}$ and rest of them are chosen arbitrarily. G is a generator matrix of the code and R_i is the matrix which when multiplied with G reduces the j^{th} column of G to $e_j, 1 \leq j \leq 1 + \sum_{k=1}^{i} n_k$, where $e_j = (0, 0, \cdots, 0, 1, 0, \cdots, 0)^T$ with 1 in the j^{th} position.

5. Publish f_i, $1 \leq i \leq m$.
6. Publish $u_{i,j}, j \in S_i$, as public shares corresponding to the codeword C_i, $1 \leq i \leq m$, where $S_i \subseteq \{\ell : \sum_{j=1}^{i} n_j + 2 \leq \ell \leq n\}$ and $\mid S_i \mid = N - t_i$.
7. Also publish the generator matrix of the MDS code.

Recovery Phase:
If at least t_i players from the first i levels cooperate they will be able to reconstruct the codeword C_i, $1 \leq i \leq m$, and hence its first component s_i, which is a term in the sum of the secret s. So, if at least t_i players participate for every i, $1 \leq i \leq m$, they will be able to recover all the terms of the sum and hence the secret. Assume that $j_r, 1 \leq r \leq m$, such that $\sum_{r=1}^{k} j_r \geq t_k$ for every $k, 1 \leq k \leq m$, players participate from the r^{th} level in the recovery phase. Also, assume that $l_{1,r}$, $l_{2,r}$, $\cdots$, $l_{j_r,r}$, be the corresponding indices of the cooperating players of the r^{th}, $1 \leq r \leq m$, level. Then the recovery phase consists of the following steps:

1. Fix i such that $1 \leq i \leq m$. Select arbitrarily $N - \sum_{k=1}^{i} j_k$ public shares to recover the codeword C_i. Let the indices of these public shares be $l_{1,m+1}, l_{2,m+1}, \cdots, l_{(N-\sum_{k=1}^{i} j_k),m+1}$.
2. Reduce, using the elementary row operations, the generator matrix to another matrix that has the following structure:
   a) $(l_{t,1} + 1)^{th}, 1 \leq t \leq j_1$, column of the generator matrix has 1 in the t^{th} row and zeros elsewhere,
   b)$(\sum_{j=1}^{k-1} n_j + l_{t,k} + 1)^{th}, 2 \leq k \leq i, 1 \leq t \leq j_k$, column of the generator matrix has 1 in the $(\sum_{r=1}^{k-1} j_r + t)^{th}$ row and zeros elsewhere,
   c) $(\sum_{j=1}^{i} n_j + l_{t,m+1} + 1)^{th}, 1 \leq t \leq \sum_{k=1}^{m} n_k + 1 - \sum_{k=1}^{i} j_k$, column of the generator matrix has 1 in the $(\sum_{r=1}^{i} j_r + t)^{th}$ row and zeros elsewhere.
3. Cooperating participant computes $f_i(v_{k,l_{tk}}) = v^i_{k,l_{tk}}$, $1 \leq k \leq i$, $1 \leq t \leq j_k$, and sends it as the participant's share in the recovery of the codeword C_i.
4. Form the message vector as
   $(v^i_{1,l_{11}}, v^i_{1,l_{21}}, \cdots, v^i_{1,l_{j_1 1}}, v^i_{2,l_{12}}, v^i_{2,l_{22}}, \cdots, v^i_{2,l_{j_2 2}}, \cdots v^i_{i,l_{1i}}, v^i_{i,l_{2i}}, \cdots, v^i_{i,l_{j_i i}},$
   $u_{i,\sum_{j=1}^{i}(n_j)+1+l_{1,m+1}}, u_{i,\sum_{j=1}^{i}(n_j)+1+l_{2,m+1}}, \cdots, u_{i,\sum_{j=1}^{i}(n_j)+1+l_{(\sum_{k=1}^{m} n_k+1-\sum_{k=1}^{i} j_k),m+1}})$.
5. Multiply the reduced generator matrix computed in step 2 by the message vector formed in step 4 to arrive at a codeword. First component of the resulting codeword is the i^{th} component (i.e.,s_i) in the sum of the secret to be recovered, which is s.
6. Do steps 1 to 5 to recover s_i, $1 \leq i \leq m$.
7. Recover the secret $s = \sum_{i=1}^{m} s_i$.

Due to space constraint, the following theorems are stated without proof. Interested reader can approach the authors for a proof.

Theorem 1. *The secret can be recovered by the recovery phase described above if and only if the set of participants recovering the secret is an authorized set and the hardness assumption stated earlier is fulfilled.*

Theorem 2. *The Proposed scheme is ideal.*

Theorem 3. *The probability that an unauthorized set being able to recover the secret is equal to that of the exhaustive search, which is* $1/q$.

Acknowledgement. V Ch Venkaiah would like to acknowledge the support received by the DST, Govt. of India, New Delhi under Project No. SR/s4/MS:516/07 dated 21st April 2008 during his tenure at CR Rao AIMSCS, Hyderabad.

References

1. Belenkiy, M.: Disjunctive multi-level secret sharing.document, `http://eprint.iacr.org/2008/018`
2. Blakley, G.R.: Safeguarding cryptographic keys. In: AFIPS Conference Proceedings, vol. 48, pp. 313–317 (1979)
3. Blakley, G.R., Kabatianski, G.A.: Ideal perfect threshold schemes and MDS codes. In: IEEE Conf. Proc., Int. Symp. Information Theory, ISIT 1995, p. 488 (1995)
4. Brickell, E.F.: Some ideal secret sharing schemes. J. Comb. Math. Comb. Comput. 9, 105–113 (1989)
5. Farras, O., Padro, C.: Ideal hierarchical secret sharing schemes. IEEE Trans. Inf. Theory (January 2012)
6. Ghodosi, H., Pieprzyk, J., Safavi-Naini, R.: Secret sharing in multilevel and compartmented groups. In: Boyd, C., Dawson, E. (eds.) ACISP 1998. LNCS, vol. 1438, pp. 367–378. Springer, Heidelberg (1998)
7. Pieprzyk, J., Zhang, X.-M.: Ideal Threshold Schemes from MDS Codes. In: Lee, P.J., Lim, C.H. (eds.) ICISC 2002. LNCS, vol. 2587, pp. 253–263. Springer, Heidelberg (2003)
8. Karnin, E.D., Greene, J.W., Hellman, M.E.: On secret sharing systems. IEEE Trans. Inf. Theory 29, 35–41 (1983)
9. Kothari, S.C.: Generalized linear threshold scheme. In: Blakely, G.R., Chaum, D. (eds.) CRYPTO 1984. LNCS, vol. 196, pp. 231–241. Springer, Heidelberg (1985)
10. Kaskaloglu, K., Ozbudak, F.: On hierarchical threshold access structures. IST panel symposium, Tallinn, Estonia, Nov.document (2010)
11. Lin, C., Harn, L.: Ideal perfect multilevel threshold secret sharing scheme. In: Proc. Fifth Intl. Conf. Inf. Assur. and Security, pp. 118–121 (2009)
12. Massey, J.L.: Minimal codewords and secret sharing. In: Proc. 6th Joint Swedish - Russian Workshop on Inform. Theory, pp. 269–279 (1993)
13. McEliece, R.J., Sarwate, D.V.: On sharing secrets and Reed Solomon codes. Comm. of ACM 24, 583–584 (1981)
14. Shamir, A.: How to share a secret. Comm. ACM 22, 612–613 (1979)
15. Tassa, T.: Hierarchical Threshold Secret Sharing. Journal of Cryptology, 237-264 (2007)
16. Tassa, T., Dyn, N.: Multipartite Secret Sharing by Bivariate Interpolation. Journal of Cryptology 22, 227–258 (2009)

An FPT Certifying Algorithm for the Vertex-Deletion Problem

Haiko Müller and Samuel Wilson[⋆]

University of Leeds, School of Computing
Leeds, LS2 9JT, United Kingdom
{scshm,scsswi}@leeds.ac.uk

Abstract. We provide a fixed-parameter certifying algorithm for the Vertex-deletion problem. An upper bound for the size of the forbidden set for $\mathcal{C}+k\mathsf{v}$ is shown demonstrating that, for all hereditary classes $\mathcal{C}$ characterised by a finite forbidden set, the class $\mathcal{C}+k\mathsf{v}$ is characterised by a finite forbidden set.

The certifying algorithm runs in time $O(f(k) \cdot n^c)$ and can be verified in $O(n^c)$ in the affirmative case and $O(f(k))$ in the negative case. This is the first known fixed-parameter certifying algorithm, as far as the authors are aware.

1 Introduction

A recognition algorithm is an algorithm that decides if some input belongs to a set where all elements of the set share a specific property. In the context of graph theory a recognition algorithm takes as input a graph and decides if the graph has the property which defines a graph class.

Here we consider the parameterized graph class $\mathcal{C}+k\mathsf{v}$ where $\mathcal{C}$ is a hereditary graph class. We present a fixed-parameter certifying algorithm for recognising the class $\mathcal{C}+k\mathsf{v}$. In the case of the algorithm returning an affirmative output the certificate consists of a set of at most k vertices, in the negative case the algorithm outputs a minimal forbidden graph. The algorithm runs in time $O(f(k) \cdot n^c)$ matching that of the best known non-certifying algorithm, the verifier runs in polynomial time.

In Section 2.3 we provide an explicit bound for the maximum order of a graph in $\mathrm{Forb}(\mathcal{C}+k\mathsf{v})$ where $\mathrm{Forb}(\mathcal{C})$ is finite. We build on this in Section 4 by presenting a fixed-parameter certifying algorithm for the vertex-deletion problem.

2 Preliminaries

All graphs we consider are finite, undirected and simple. For a graph $G = (V, E)$ let $V(G) = V$, $E(G) = E$, $n = |V(G)|$ and $m = |E(G)|$. Two graphs G and

[⋆] The research presented in this paper is work towards the Ph.D of the second author, who is funded by EPSRC Doctoral Training Grant.

© Springer-Verlag Berlin Heidelberg 2013

H are *isomorphic*, denoted as $G \simeq H$, if there is an edge-preserving bijection between $V(G)$ and $V(H)$. We do not distinguish between isomorphic graphs.

Let $\leqslant$ be a binary relation on the class $\mathcal{G}$ of all graphs that is reflexive, transitive and antisymmetric and let $<$ be the corresponding binary relation that is irreflexive, transitive and asymmetric. Both variants we call a *partial order*. Incomparability with respect to $\leqslant$ is denoted by $\|$.

A graph H is *contained* within a graph G with respect to the partial order $\leqslant$ if $H \leqslant G$. A class $\mathcal{C}$ is *closed* under $\leqslant$ if $H \leqslant G$ and $G \in \mathcal{C}$ implies $H \in \mathcal{C}$. Each such class $\mathcal{C}$ can be characterised by forbidding certain sets of graphs. The set of *minimal forbidden* graphs is $\mathrm{Forb}(\mathcal{C}) = \{H \mid \forall H, H' \notin \mathcal{C}\ (H' \not< H)\}$. That is, a graph class $\mathcal{C}$ which is closed under $\leqslant$ consists of all graphs G that do not contain a member of $\mathrm{Forb}(\mathcal{C})$.

The graph H is an *induced subgraph* of G if $V(H) \subseteq V(G)$ and $E(H) = \{uv \mid u, v \in V(H) \wedge uv \in E(G)\}$. The induced subgraph relation defines a partial order on the set of all graphs. A class $\mathcal{C}$ is hereditary if the class is closed under the induced subgraph relation. By $G - U$, or $G - u$ if $U = \{u\}$, we refer to the graph $G[V(G) \setminus U]$.

For every class $\mathcal{C}$ of graphs and every integer $k \geq 0$ we define the class $\mathcal{C}{+}k\mathsf{v}$ by

$$\mathcal{C}{+}k\mathsf{v} = \{G \mid \exists\, U \subseteq V(G)\ (|U| \leq k \wedge G - U \in \mathcal{C})\}\,.$$

2.1 Certifying Algorithms

The study of certifying algorithms is motivated by software reliability and bug-free implementations, the adoption of certifying algorithms is becoming wide spread as the knowledge required to understand the technical details of an algorithm increases past that of the end user. A certifying algorithm gives the end user confidence that the algorithm is correct and that the implementation is free of bugs. The end user is not required to understand the technical details of the algorithm, only that the certificate justifies the correctness of the algorithm and that the verifying algorithm implementation is free of bugs.

A *certifying algorithm* consists of a *prover* and a *verifier*. The prover solves the problem and computes a *certificate*, which provides justification for its output. The certificate is evidence that the output is correct for the input. The verifier takes as input

– the problem instance, which is also the input of the prover,
– the solution computed by the prover, and
– the certificate provided by the prover,

and decides if the triple is valid, that is, the certificate verifies the correctness of the output. A theory of certifying algorithms is presented in [6].

2.2 Fixed-Parameter Tractability

Fixed-parameter tractable (FPT) algorithms have an important role in providing efficient algorithms for NP-complete problems. An FPT algorithm aims to separate the intractable part of the problem and bound it by some value k such that the remainder of the algorithm runs in polynomial time. Formally a problem is FPT if there exists a parameter k such that the problem can be computed in $O(f(k) \cdot n^c)$ where c is some constant independent of k and f is a computable function depending only on k.

Here we give an FPT certifying recognition algorithm for the classes $\mathcal{C}+k\mathsf{v}$ for all $k \geq 0$ where $\mathcal{C}$ is closed under induced subgraphs and is characterised by a finite set of minimal forbidden subgraphs. The prover runs in time $O(f(k) \cdot n^c)$ where c is the order of the largest minimal forbidden graph for $\mathcal{C}$. The verifier has a running time of $O(f(k))$ in the case of non-membership and $O(n^c)$ otherwise.

By fixed-parameter tractable certifying algorithm we mean that the certifying algorithm runs in time $f(k) \cdot n^{O(1)}$ where k is the parameter and the verifying algorithm also runs in $f'(k) \cdot n^{O(1)}$ time. The concept of what constitutes a strong certificate directly translates into the context of fixed-parameter tractable certifying algorithms, a certificate is strong if the verification algorithm has running time equal to or better than the running time of the certifying algorithm on the original problem.

2.3 Outline

Certifying recognition algorithms for graph classes are desirable especially when the certificate for non-membership is a minimal forbidden graph, this is the approach used in [5,2]. We adopt the same technique for certifying non-membership of the class $\mathcal{C}+k\mathsf{v}$ by finding a minimal forbidden graph. In order to achieve this we first prove the finiteness of the minimal forbidden set via a hypergraph transversal argument. The bound obtained from the hypergraph transversal argument is then used in a modified version of the algorithm presented in [1]. The resulting algorithm returns in the case of membership a set of k vertices whose removal from the input graph yields a graph in the class $\mathcal{C}$, or in the case of non-membership finds a minimal forbidden graph. We assume that the recognition of graphs in $\mathcal{C}$ is trivial or that a certifying algorithm for this task exists.

3 Bounding Forb($\mathcal{C}+k\mathsf{v}$)

Theorem 1. *For each hereditary class $\mathcal{C}$ of graphs with a finite set* Forb($\mathcal{C}$) *and all integers $k \geq 0$ the set* Forb($\mathcal{C}+k\mathsf{v}$) *is finite. More precisely, we have for all $H \in$* Forb($\mathcal{C}+k\mathsf{v}$),

$$|H| \leq \begin{cases} k+1 & s = 1 \\ \binom{(k+1)+s-2}{s-2}(k+1) + (k+1)^{s-1} & s \geq 2 \end{cases}$$

where $s = \max\{|V(H)| \mid H \in \mathrm{Forb}(\mathcal{C})\}$.

Theorem 1 builds on the bounds published in [4,3]

4 The Algorithm

The algorithm we give in this section is a certifying algorithm for recognising the class $\mathcal{C}+k\mathsf{v}$ where $\mathcal{C}$ has a finite forbidden set and $k \geq 0$. The input is a graph G and an integer $k \geq 0$. The prover decides if $G \in \mathcal{C}+k\mathsf{v}$. In the affirmative case the prover provides a set $U \subseteq V(G)$, where $|U| \leq k$ such that $G - U \in \mathcal{C}$, as the certificate. This certificate can be verified in $O(n^c)$. For the negative case, that is, $G \notin \mathcal{C}+k\mathsf{v}$, the prover provides a minimal forbidden graph of $\mathcal{C}+k\mathsf{v}$ embedded in G. We show that this certificate can be verified in time $O(f(k))$.

4.1 Prover

Every hereditary class $\mathcal{C}$ of graphs with $|V(H)| \leq c$ for all $H \in \mathrm{Forb}(\mathcal{C})$ can be recognised in $O(n^c)$ by exhaustively checking each subset on up to c-element of the input graph. For some graph classes more efficient recognition algorithms are known which could be used in the place of the previously mentioned brute force technique. The exhaustive search finds a graph in $\mathrm{Forb}(\mathcal{C})$ in the input graph if there is one.

The recognition of a graph in the class $\mathcal{C}+k\mathsf{v}$ has been shown to be FPT by Cai [1], the approach used is a recursive algorithm which constructs a search tree. The FPT time bound is proved by establishing the maximum size of the search tree. The algorithm presented in [1] can easily be extended to return a set of k vertices such that the removal of these vertices yields a graph in $\mathcal{C}$. Additionally the algorithm can be extended to find a minimal forbidden graph in time independent of the input size. The prover generates the minimal forbidden graphs for $\mathcal{C}+k\mathsf{v}$, in the same order as the verifier would, and checks for an isomorphism with the minimal forbidden graph found in the input. The algorithm outputs the number of graphs generated before the algorithm finds a graph isomorphic to the minimal forbidden graph in the input graph and the injective function mapping the forbidden graph into the input graph.

Theorem 2. *The prover, given the input $G = (V, E)$ and an integer $k \geq 0$, returns either a set U of at most k vertices such that $(G - U) \in \mathcal{C}$ or an index of a minimal forbidden graph $H \in \mathrm{Forb}(\mathcal{C}+k\mathsf{v})$ in an ordered set and a function $f : V(H) \to V(G)$ in $O(f(k) \cdot n^c)$ time.*

4.2 Verifier

The verifier confirms that the certificate is valid for the input, this is achieved by either checking that when the set of up to k vertices are removed the input graph is a member of $\mathcal{C}$, or that the embedding of a forbidden graph into the input graph is valid. In the negative case the verifier takes as input a number x and an injective function between the vertices of the xth generated graph in the forbidden set for $\mathcal{C}+k\mathsf{v}$ and the vertices of the input graph, and checks if the injection maps the forbidden graph into the input graph. The affirmative certificate can be checked trivially in $O(n^c)$ time and the negative certificate can be checked in constant time.

The forbidden set for $\mathcal{C}+k\mathsf{v}$ can be generated in constant time for each $\mathcal{C}$ and k. From Theorem 1 we have that the order of a graph in $\mathrm{Forb}(\mathcal{C}+k\mathsf{v})$ is bounded, let N be the bound. For a fixed $\mathcal{C}$ we can generate the set of all graphs on up to N vertices in time dependent only on k. Let X be the ordered set of all graphs on up N vertices, and assume X is ordered however the generating algorithms outputs the graphs. It is possible to find a minimal forbidden graph in FPT time therefore by removing elements from X that are not minimal forbidden for $\mathcal{C}+k\mathsf{v}$ we obtain the set of minimal forbidden graphs. The overall run time of this procedure is dependent only on k, *i.e.* generating the minimal forbidden set can be achieved in $O(f(k))$ time.

To verify the negative certificate the verifier is required to generate the set of minimal forbidden graphs up to the index provided by the prover then verify that the injection is a valid mapping between a minimal forbidden graph and a subgraph of the input graph.

5 Conclusion

In this paper we have presented an upper bound on the size of the forbidden set for $\mathcal{C}+k\mathsf{v}$ where $\mathcal{C}$ is characterised by a finite forbidden set and $k \geq 0$. This result leads to a general technique for constructing FPT certifying algorithms for testing the membership of a graph in the class $\mathcal{C}+k\mathsf{v}$. This extends the result of Cai [1] into the domain of certifying algorithms. We comment that the algorithm presented here works in the most general case and improvements can be made when the class of graphs is restricted. For instance, consider split graphs: there exists a linear time recognition algorithm which would vastly improve the running time of the verifier from $O(n^5)$ obtained by the general construction to $O(n)$ [5]. Additionally the bound for the maximum size of a forbidden graph for $\mathcal{C}+k\mathsf{v}$ is very general and improvements may be possible for particular classes.

References

1. Cai, L.: Fixed-parameter tractability of graph modification problems for hereditary properties. Information Processing Letters 58(4), 171–176 (1996)
2. Chang, M.-S., Kloks, T., Kratsch, D., Liu, J., Peng, S.-L.: On the recognition of probe graphs of some self-complementary classes of perfect graphs. In: Wang, L. (ed.) COCOON 2005. LNCS, vol. 3595, pp. 808–817. Springer, Heidelberg (2005)
3. Erdős, P., Gallai, T.: On the minimal number of vertices representing the edges of a graph. Magyar Tud. Akad. Mat. Kutató Int. Közl 6, 181–203 (1961)
4. Gyárfás, A., Lehel, J., Tuza, Z.: Upper bound on the order of τ-critical hypergraphs. Journal of Combinatorial Theory, Series B 33(2), 161–165 (1982)
5. Heggernes, P., Kratsch, D.: Linear-time certifying recognition algorithms and forbidden induced subgraphs. Nordic Journal of Computing 14(1), 87–108 (2007)
6. McConnell, R.M., Mehlhorn, K., Näher, S., Schweitzer, P.: Certifying algorithms. Computer Science Review 5(2) (2011)

Author Index