Ralf Klasing (Ed.)

LNCS 7276

Experimental Algorithms

11th International Symposium, SEA 2012
Bordeaux, France, June 2012
Proceedings

Springer

Ralf Klasing (Ed.)

Experimental Algorithms

11th International Symposium, SEA 2012
Bordeaux, France, June 7-9, 2012
Proceedings

Springer

Volume Editor

Ralf Klasing
CNRS - LaBRI - Université Bordeaux 1
351 Cours de la Libération
33405 Talence cedex
France
E-mail: klasing@labri.fr

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-30849-9 e-ISBN 978-3-642-30850-5
DOI 10.1007/978-3-642-30850-5
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012938881

CR Subject Classification (1998): F.2, I.2, E.1, H.3-4, G.2, C.2, D.2

LNCS Sublibrary: SL 2 – Programming and Software Engineering

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

The 11th International Symposium on Experimental Algorithms (SEA 2012) took place during June 7–9, 2012, in Bordeaux, France.

SEA, previously known as WEA (Workshop on Experimental Algorithms), is an international forum for researchers in the area of design, analysis, and experimental evaluation and engineering of algorithms, as well as in various aspects of computational optimization and its applications. The preceding symposia were held in Riga, Monte Verita, Rio de Janeiro, Santorini, Menorca, Rome, Cape Cod, Dortmund, Ischia, and Crete.

The Program Committee of SEA 2012 received 64 submissions. Each submission was reviewed by at least three Program Committee members and some trusted external referees, and evaluated on its quality, originality, and relevance to the symposium. The Committee selected 31 papers, leading to an acceptance rate of 48%.

In addition to the accepted contributions, this volume also contains papers of the invited talks given by Marco E. Lübbecke (RWTH Aachen University), Friedhelm Meyer auf der Heide (University of Paderborn), and Peter Sanders (Karlsruhe Institute of Technology).

I would like to thank the Steering Committee and its Chair, José Rolim, for giving me the opportunity to serve as Program Chair of SEA 2012, and for the responsibilities of selecting the Program Committee, the conference program, and publications.

I would like to thank all the authors who responded to the call for papers, the invited speakers, the members of the Program Committee, the external referees, and — last but not least— the members of the Organizing Committee.

I gratefully acknowledge financial support from the following institutions: the French National Research Agency (ANR), Enseirb-Matméca, Institut Universitaire de France (IUF), LaBRI, University of Bordeaux, CNRS, Inria, Région Aquitaine, GDR IM, Communauté Urbaine de Bordeaux (CUB).

I would like to thank Springer for publishing the proceedings of SEA 2012 in their LNCS series and for their support.

Finally, I acknowledge the use of the EasyChair system for handling the submission of papers, managing the review process, and generating these proceedings.

June 2012 Ralf Klasing

Organization

Program Committee

Ioannis Caragiannis	University of Patras and CTI, Greece
Colin Cooper	King's College London, UK
David Coudert	INRIA Sophia, France
Jurek Czyzowicz	Université du Québec en Outaouais, Canada
Robert Elsässer	University of Paderborn, Germany
Thomas Erlebach	University of Leicester, UK
Sándor P. Fekete	TU Braunschweig, Germany
Paola Festa	University of Naples Federico II, Italy
Michele Flammini	University of L'Aquila, Italy
Pierre Fraigniaud	CNRS and Paris Diderot University, France
Leszek A. Gąsieniec	University of Liverpool, UK
Juraj Hromkovič	ETH Zürich, Switzerland
Christos Kaklamanis	University of Patras and CTI, Greece
Ralf Klasing (Chair)	CNRS and University of Bordeaux, France
Mirosław Korzeniowski	Wrocław University of Technology, Poland
Adrian Kosowski	INRIA Bordeaux, France
Arie M.C.A. Koster	RWTH Aachen, Germany
Dariusz R. Kowalski	University of Liverpool and IMDEA Networks, UK
Christian Laforest	Blaise Pascal University, Clermont Ferrand, France
Leo Liberti	LIX, Ecole Polytechnique, France
Andrea Lodi	University of Bologna, Italy
Alberto Marchetti-Spaccamela	University of Rome "La Sapienza", Italy
Luca Moscardelli	University of Pescara, Italy
Petra Mutzel	TU Dortmund, Germany
Alfredo Navarra	University of Perugia, Italy
Marina Papatriantafilou	Chalmers University of Technology, Göteborg, Sweden
Panos M. Pardalos	University of Florida, USA
Vangelis Th. Paschos	Paris-Dauphine University, France
Joseph G. Peters	Simon Fraser University, Canada
Guido Proietti	University of L'Aquila, Italy
Tomasz Radzik	King's College London, UK
Mauricio G.C. Resende	AT&T Labs Research, USA
Celso C. Ribeiro	University Federal Fluminense, Brazil
Nicolas Schabanel	CNRS and Paris Diderot University, France

Christian Scheideler	University of Paderborn, Germany
Leen Stougie	VU University and CWI, Amsterdam, The Netherlands
Walter Unger	RWTH Aachen, Germany
Annegret Wagler	Blaise Pascal University, Clermont Ferrand, France
Christos Zaroliagis	University of Patras and CTI, Greece

Steering Committee

Edoardo Amaldi	Politecnico di Milano, Italy
David A. Bader	Georgia Institute of Technology, USA
Josep Diaz	Technical University of Catalonia, Spain
Giuseppe F. Italiano	University of Rome “Tor Vergata”, Italy
David S. Johnson	AT&T Labs - Research, USA
Klaus Jansen	University of Kiel, Germany
Kurt Mehlhorn	MPII Saarbrücken, Germany
Ian Munro	University of Waterloo, Canada
Sotiris Nikoletseas	University of Patras and CTI, Greece
José Rolim (Chair)	University of Geneva, Switzerland
Pavlos Spirakis	University of Patras and CTI, Greece

Organizing Committee

Lionel Eyraud-Dubois	INRIA, Bordeaux, France
Florent Foucaud	University of Bordeaux, France
Ralf Klasing (Chair)	CNRS and University of Bordeaux, France
Miroslaw Korzeniowski	Wroclaw University of Technology, Poland
Adrian Kosowski	INRIA, Bordeaux, France
Nicole Lun	LaBRI, Bordeaux, France
Lebna Mizani	LaBRI, Bordeaux, France
Thomas Morsellino	University of Bordeaux, France
Dominik Pająk	INRIA, Bordeaux, France
Corentin Travers	ENSEIRB, Bordeaux, France
Petru Valicov	University of Bordeaux, France

External Reviewers

Daniel Aloise
Rafael Andrade
Giannakos Aristotelis
Luca Becchetti
Michael Bender
Davide Bilò
Hans-Joachim Böckenhauer
Nicolas Boria
Radu Ioan Bot
Valentina Cacchiani
Sonia Cafieri
Daniel Cederman
Bapi Chatterjee
Andrew Collins
Alberto Costa
Claudia D’Ambrosio
Gianlorenzo D’Angelo

Mattia D’Emidio
Cid C. De Souza
Daniel Delling
Marc Demange
Jillian Dicker
Neng Fan
Tobias Friedrich
Daniele Frigioni
Loukas Georgiadis
Pando Georgiev
Sascha Geulen
Oliver Göbel
Luciano Gualà
Stefan Heinz
Frank Hellweg
Luc Hogie
Martina Hüllmann
Nikos Karanikolas
Sebastian Kniesburges
Dennis Komm
Andreas Koutsopoulos
Scott Kristjanson
Sacha Krug
Maria Kyropoulou
Guanghui Lan
Marc Lelarge
Stephan Lemkens
Dimitrios Letsios
Giorgio Lucarelli
Ashutosh Mahajan
Russell Martin
Paulo Vieira Milreu
Tobias Mömke
Michele Monaci
Farnaz Moradi
Syed Mujahid
Ioannis Nikolakopoulos
Nicolas Nisse
Marcel Ochel
Adrian Ogierman
György Ottucsák
Dominik Pajak
Thomas Pajor
Vijay Pappu
Grigory Pastukhov
Khoa Phan
Raksmey Phan
Oleg Prokopyev
Klaus Radke
Benjamin Ries
Bernard Ries
Christiane Schmidt
Rene Sitters
Cole Smith
Stefano Smriglio
Alexey Sorokin
Andreas Sprock
Jukka Suomela
My Thai
Martin Tieves
Valentin Tudor
Przemyslaw Uznanski
Leo van Iersel
Chrysafis Vogiatzis
Prudence W.H. Wong
Petros Xanthopoulos
Katharina Zweig

Sponsors

AGENCE NATIONALE DE LA RECHERCHE
ANR

IPB
ENSEIRB
MATMECA
BORDEAUX

IUF

LaBRI

UNIVERSITÉ
BORDEAUX 1
Sciences Technologies

cnrs

Inria
INVENTORS FOR THE DIGITAL WORLD

REGION
AQUITAINE

gdr
iM

COMMUNAUTÉ
URBAINE DE BORDEAUX
LACUB

Table of Contents

Invited Papers

Contributed Papers

Automatic Decomposition and Branch-and-Price—A Status Report

Marco E. Lübbecke

RWTH Aachen University, Operations Research, Kackertstraße 7,
D-52072 Aachen, Germany
marco.luebbecke@rwth-aachen.de

Abstract. We provide an overview of our recent efforts to automatize Dantzig-Wolfe reformulation and column generation/branch-and-price for structured, large-scale integer programs. We present the need for and the benefits from a generic implementation which does not need any user input or expert knowledge. A focus is on detecting structures in integer programs which are amenable to a Dantzig-Wolfe reformulation. We give computational results and discuss future research topics.

1 Modeling with Integer Programs

Integer programming offers undeniably a powerful and versatile, yet industrially relevant approach to model and solve discrete optimization problems from virtually all areas of scientific and practical applications. To get an impression on modeling, consider a simple combinatorial optimization problem, the *bin packing problem.* We are given n items of size a_i, $i = 1, \dots, n$, which have to be packed into a minimum number of bins of capacity b each. A standard integer program for this problem is built on binary variables $x_{ij} \in \{0, 1\}$ to decide whether item i is packed in bin j or not. It is common that a *single* variable imposes relatively little structure on the overall solution. The model is as follows.

$$\min \sum_{j=1}^{n} y_j \tag{1a}$$

$$\sum_{j=1}^{n} x_{ij} = 1 \qquad i = 1, \dots, n \tag{1b}$$

$$\sum_{i=1}^{n} a_i x_{ij} \leq b \qquad j = 1, \dots, n \tag{1c}$$

$$x_{ij} \leq y_j \qquad i, j = 1, \dots, n \tag{1d}$$

$$x_{ij},\, y_j \in \{0, 1\} \qquad i, j = 1, \dots, n \tag{1e}$$

We call this the *original formulation.* Every item has to be packed because of the *set partitioning constraint* (1b); whenever a bin is used it has to be opened

R. Klasing (Ed.): SEA 2012, LNCS 7276, pp. 1–8, 2012.

via the logical implication (1d); and no bin is overpacked because of the *knapsack constraint* (1c). The objective function (1a) reflects the goal of minimizing the number of opened bins. Note that n bins always suffice. A few remarks are in order. We observe that there is a symmetry w.r.t. the bins, that is, for a given solution $(\bar{x}, \bar{y})$ and a permutation σ of the bin indices, we get essentially "the same" solution with the same objective function value by replacing $\bar{x}_{ij}$ by $\bar{x}_{i\sigma(j)}$ and $\bar{y}_j$ by $\bar{y}_{\sigma(j)}$. Also note that there are rather "local" constraints (1c), namely those concerning the packing of a *single* bin; and there are "global" constraints (1b), namely those which ensure that *for every item* we open *some* bin. In particular, there is a knapsack problem to solve for each bin (which is NP-hard, but a computationally very easy combinatorial optimization problem), and the "individual" solutions to the knapsack problems are linked by a "coordinating" constraint which enforces a global structure in the overall solution. This is typical for many practical situations in which decisions are taken in a distributed way, but which in fact need synchronization in order to achieve a global goal (this is a feature which brings optimal solutions to such decision problems way out of reach of human planners). Examples are vehicle routing, crew and machine scheduling, location problems, and many more.

A standard solver does not "see" this concept of constructing a complex solution out of easier building blocks either, as a branch-and-bound algorithm works "everywhere" on the overall solution simultaneously by construction. One way to make these partial solutions "visible" to the solver is by formulating a different model which is based on "more meaningful" variables. For bin packing, we could base a model on binary variables $\lambda_{pj} \in \{0,1\}$ which represent whether or not we pack an entire configuration or *pattern* p in bin j. A pattern is a collection of items that respects the knapsack capacity and therefore "knows" about the local constraints. All patterns for bin j are collected in a set P_j, and with the shorthand notation $i \in p$ to state that pattern p contains item i, the new model reads as follows.

$$\min \sum_{j=1}^{n} \sum_{p \in P_j} \lambda_{pj} \tag{2a}$$

$$\sum_{j=1}^{n} \sum_{p \in P_j : i \in p} \lambda_{pj} = 1 \qquad i = 1, \ldots, n \tag{2b}$$

$$\sum_{p \in P_j} \lambda_{pj} \leq 1 \qquad j = 1, \ldots, n \tag{2c}$$

$$\lambda_{pj} \in \{0,1\} \qquad j = 1, \ldots, n,\ p \in P_j \tag{2d}$$

Constraint (2b) has the same role as constraint (1b) before: we must be sure that among all patterns for all bins, every item is contained in exactly one of those selected. The *convexity constraint* (2c) is new and ensures that at most one pattern is chosen per bin (a bin may also be "empty," that is, closed). No knapsack constraint is needed any more, at the expense of the fact that, for each bin, we essentially enumerated all feasible solutions to the knapsack constraint. After all, this is what we wanted. From a solution to this model we can uniquely reconstruct a solution to the original model (1) via $x_{ij} = \sum_{p \in P_j : i \in p} \lambda_{pj}$, that is, summing over all patterns for bin j that contain an item i. The new model (2) is still symmetric in the bins as all sets of patterns are identical for all bins. This symmetry could be eliminated by noting that we only need to ensure that for each item *some* pattern is selected; exactly which bin is used is not of importance. This leads us to an *aggregated* version of model (2).

$$\min \sum_{p \in P} \nu_p \tag{3a}$$

$$\sum_{p \in P : i \in p} \nu_p = 1 \qquad i = 1, \dots, n \tag{3b}$$

$$\nu_p \in \{0, 1\} \qquad p \in P \tag{3c}$$

A binary variable ν_p represents whether we select a pattern p or not. Set P contains all feasible patterns, but no longer any information on bin indices. This is also true for the aggregated variables $\nu_p = \sum_{j=1}^{n} \lambda_{pj}$. As a consequence of the symmetry breaking, there is no unique reconstruction of an original solution to model (1) from a solution to model (3). Of course, one could have formulated a *set partitioning model* like (3) for the bin packing problem without going through the above reformulations, and this is what often happens.

2 Dantzig-Wolfe Reformulation

There is a major reason for favoring models (2) or (3) over model (1): the former is usually *stronger* in the sense that the linear relaxation, i.e., relaxing variable domains from $\{0, 1\}$ to $[0, 1]$, gives a tighter bound on the integer optimum. Intuitively, this is because the "more meaningful" variables in (3) impose more structure on the overall solution because a part of all original constraints (the "local" knapsack constraints) is already fulfilled with integrality. The theoretical reason is that model (2) is derived from (1) via a Dantzig-Wolfe reformulation.

A sketch of this reformulation is as follows (see e.g., [3] for details). Consider an *original* integer program of the form

$$\min\{c^t x : Ax \geq b,\ Dx \geq d,\ x \in \mathbb{Z}_+^n\} \ . \tag{4}$$

The polyhedron $X := \operatorname{conv}\{x \in \mathbb{Z}_+^n : Dx \geq d\}$ gives an inspiration for the "more meaningful" variables. We assume that X is bounded, but this is no restriction. We express $x \in X$ as a convex combination of the (finitely many) extreme points P of X, which leads to an equivalent *extended* formulation

$$\min\{c^t x : Ax \geq b,\, x = \sum_{p \in P} \lambda_p p,\, \sum_{p \in P} \lambda_p = 1,\, \lambda_p \geq 0,\, x \in \mathbb{Z}_+^n\} \ . \tag{5}$$

The reformulation (5) contains the so-called *master constraints* $Ax \geq b$, the convexity constraint, and the constraint linking the *original* x variables to the *extended* λ variables. In general, model (5) has an exponential number (in n) of λ variables, so its LP relaxation needs to be solved by column generation [3]. That is, one starts with a small subset of λ variables and iteratively adds more variables of negative reduced cost until no such variables can be identified. The pricing subproblem to check whether there exist variables with negative reduced cost is a minimization problem of a linear objective function over X, so it can be solved again as an integer program. The column generation process needs to be invoked in every node of the branch-and-bound tree, yielding a branch-and-price algorithm. Special care must be taken when deciding on how to branch on fractional variables [4,16,17].

In the classical setting, k disjoint sets of constraints are reformulated, namely when the matrix D has a *block-diagonal* form

$$D = \begin{pmatrix} D^1 & & & \\ & D^2 & & \\ & & \ddots & \\ & & & D^k \end{pmatrix} , \tag{6}$$

where $D^i \in \mathbb{Q}^{m_i \times n_i}$, $i = 1, \ldots, k$. In other words, $Dx \geq d$ partitions in $D^i x^i \geq d^i$, $i = 1, \ldots, k$, where $x = (x^1, x^2, \ldots, x^k)$, with an n_i-vector x^i, $i = 1, \ldots, k$. Every $D^i x^i \geq d^i$ is individually Dantzig-Wolfe reformulated. We call k the *number of blocks* of the reformulation. A matrix of the form

$$\begin{pmatrix} D^1 & & & \\ & D^2 & & \\ & & \ddots & \\ & & & D^k \\ A^1 & A^2 & \cdots & A^k \end{pmatrix} \tag{7}$$

with $A^i \in \mathbb{Q}^{m_\ell \times n_i}$, $i = 1, \ldots, k$ is called *bordered block-diagonal*, see Fig. 1(b). It is this form we would like to see in the coefficient matrix of an integer program in order to apply a Dantzig-Wolfe reformulation.

Depending on your background, the following may or may not apply to you: you regularly formulate models like (1) (but cannot solve even moderately sized instances); you have already noticed that going from model (1) to model (2) is by application of a Dantzig-Wolfe reformulation (but you don't know of what use this knowledge may be to you, practically speaking); you would like to implement your own column generation and branch-and-price code and use it to optimally solve models like (2) or (3) (but you don't know whether it is worth the time and considerable effort); you already have your branch-and-price code running (but don't want to change and adapt it every time you consider a new problem). You are any of this kind? Read on...

3 In Need for an Automatic Decomposition

In 2004, when collecting material for “the primer” [3], we learned about François Vanderbeck’s efforts to write a generic branch-and-price code BaPCod [12], and we were fascinated by the idea ever since. Vanderbeck developed important contributions [11,13,14,15], so that not only a Dantzig-Wolfe reformulation could be performed according to a user specification but also branching was done in a generic way. Further generic decomposition codes became available like DIP [10] and within the G12 project [9]. However, none of these codes could be used by someone not knowledgeable in decomposition techniques as the user needs to propose how the input integer program is to be decomposed. It is still up to the modeler which constraints $Dx \geq d$ she considers as “local,” that is, to be reformulated. Sometimes the choice may be rather obvious as in our introductory example, but sometimes there is more freedom, and thus more freedom to make mistakes, and one needs to know what one is doing.

It simply felt wrong that there are the well-understood and in many special cases successfully applied concepts of Dantzig-Wolfe decomposition, column generation, and branch-and-price, but they are not “out-of-the-box” usable e.g., to “everyday” OR practitioners, despite the availability of generic implementations. And yet those who ran their own codes often needed to start all over with every new application.

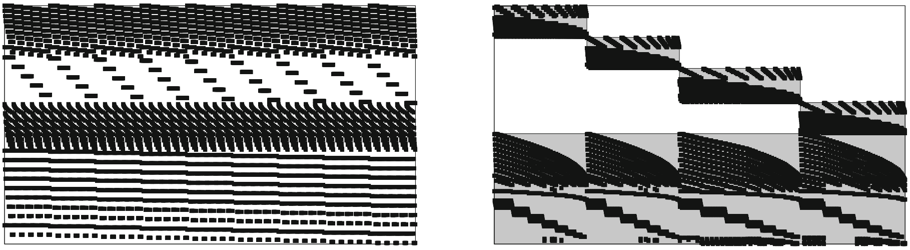
In order to close the last—but maybe most crucial—gap, we made first experiments in [2] with detecting matrix structures suited for reformulation (see Section 4). At that time, this detection and our generic branch-and-price code GCG (see Section 5) were not merged into one project, but results were encouraging. Testing on general mixed integer programs was very nice and gave a successful proof-of-concept, but diverted our attention from the true target instances of this research: those which bear structure. In this talk, we report on experiments with automatic detection of decomposable matrix structures, and generic branch-and-price on a suitable test set of “structured” instances.

4 Recognizing Matrix Structures for Decomposition

The typical matrix structure for which Dantzig-Wolfe decomposition was proposed is a bordered block-diagonal form (7). With a little experience with the technique and an automatic recognition of types of constraints (like set partitioning constraints, knapsack constraints, and the like) one can come up quite easily with a suggestion for a decomposition. This works well for standard problems, but may fail for “unknown” problems. Then, a possibility is to exploit a folklore connection between matrices and graphs [5].

Given a matrix A, construct a hypergraph $H = (V, R \cup C)$ as follows. With every $a_{ij} \neq 0$ associate a vertex $v_{ij} \in V$. For every row i introduce a hyperedge $r_i \in R$ which contains exactly all vertices $v_{r_i j} \in V$ that correspond to non-zero entries of the row; analogously introduce a hyperedge $c_j \in C$ for every column j. When H partitions into several connected components, the matrix A is a block-diagonal matrix, with a bijection between blocks in A and connected components in H.

Hypergraph H can also be used to detect a bordered block-diagonal form. Without the rows in the "border" the remaining matrix is block-diagonal. Thus, a removal of (a minimum number of) hyperedges from R such that the remaining graph partitions into connected components reveals a bordered block-diagonal form in A (with a minimum number of rows in the border). The problem is NP-hard and we experimented with heuristics to solve it. Figure 1 shows a matrix as given in the original model, and a structure detected with this *minimum hypergraph partitioning approach*.



(a) original *10teams*

(b) detected structure

Fig. 1. (a) Matrix structure directly from the LP file (*10teams*) and (b) with a bordered block-diagonal structure detected by our algorithm

The algorithm needs as input the number k of connected components we look for in H. Thus, in practice, we check for different small numbers of k. As the results sometimes look artificial, we suggest that a more tailored hypergraph partitioning algorithm should be sought, exploiting the fact that H is extremely sparse and of degree 2. In a different line of research we replace hypergraph partitioning by hypergraph clustering, which eliminates the need to specify the number of blocks (connected components, clusters) beforehand. Experimentation with these alternatives is still under way, but preliminary results look plausible.

5 Towards a Standalone Solver

Based on the discussion above, we developed GCG [6] ("generic column generation") which is based on the SCIP framework [1] which is free for academic purposes (scip.zib.de). Together with the LP file describing the integer program, GCG takes as input a second file ("the decomposition") describing which constraints belong to the master problem and which to the blocks, respectively. Alternatively, several of the matrix structure detection algorithms only sketched above are applied to the instance. GCG then performs a Dantzig-Wolfe reformulation according to a "best guess" (in addition one can specify whether the so-called convexification or discretization approach should be applied); identifies identical blocks, and aggregates them (the same as going from model (2) to model ([3])); performs column generation on the given decomposition; maintains both formulations (the *original* in the x-variables, and the *extended* in the λ-variables) which allows branching and cutting plane separation on the original variables; it has specialized branching rules like Ryan/Foster branching and generic primal heuristics [7,8]; and overall `GCG` uses `SCIP`'s rich functionality of being a state-of-the-art MIP solver (like availability of pseudo-costs, pre-solving, propagation techniques, etc.). This turns the branch-price-and-cut *framework* `SCIP` into a branch-price-and-cut *solver* [6]. A first stable version is about to be released as this abstract goes to press.

6 Discussion

What are the goals of our project? Certainly, expecting a decomposition code to beat a state-of-the-art branch-and-cut code on the average instance is not realistic. On the other hand, anything but outperforming the general-purpose solver on instances that contain a decomposable problem structure would be a failure. Thus, the art remains to tell the instances that are amenable to a Dantzig-Wolfe reformulation from those which are not. And so we are back at the most important and most interesting algorithmic challenge: to efficiently and reliably detect "structure" in an instance or conclude that "none" is contained.

This is an area which may not only produce new and improved algorithms, e.g., for partitioning/clustering the graph underlying a coefficient matrix. We also need a much better theoretical understanding of what makes a good "structure" to look for, and we believe that this will give us insights into how to set up a good integer programming model in the first place. It is this *algorithm engineering* feature which makes this project so interesting to us: to improve the design of a long-known algorithm, letting computational experiments guide our way. We hope that our work contributes to closing the gap between the algorithm on paper and its usefulness to a non-expert in practice.

Acknowledgments. I enjoyed discussions with Michael Bastubbe, Martin Bergner, Alberto Ceselli, Jacques Desrosiers, Fabio Furini, Gerald Gamrath, Marc Pfetsch, and Emiliano Traversi on automated decompositions. This project was supported by the German Science Foundation's (DFG) priority program 1307 "algorithm engineering," under contracts LU 770/4-1 and LU 770/4-2, within the project "generic decomposition algorithms for integer programs."

References

1. Achterberg, T.: SCIP: Solving constraint integer programs. Math. Programming Computation 1(1), 1–41 (2009)
2. Bergner, M., Caprara, A., Furini, F., Lübbecke, M.E., Malaguti, E., Traversi, E.: Partial Convexification of General MIPs by Dantzig-Wolfe Reformulation. In: Günlük, O., Woeginger, G.J. (eds.) IPCO 2011. LNCS, vol. 6655, pp. 39–51. Springer, Heidelberg (2011)

3. Desrosiers, J., Lübbecke, M.: A primer in column generation. In: Desaulniers, G., Desrosiers, J., Solomon, M. (eds.) Column Generation, pp. 1–32. Springer, Heidelberg (2005)
4. Desrosiers, J., Lübbecke, M.: Branch-price-and-cut algorithms. In: Cochran, J. (ed.) Encyclopedia of Operations Research and Management Science. John Wiley & Sons, Chichester (2011)
5. Ferris, M., Horn, J.: Partitioning mathematical programs for parallel solution. Math. Programming 80, 35–61 (1998)
6. Gamrath, G., Lübbecke, M.E.: Experiments with a Generic Dantzig-Wolfe Decomposition for Integer Programs. In: Festa, P. (ed.) SEA 2010. LNCS, vol. 6049, pp. 239–252. Springer, Heidelberg (2010)
7. Joncour, C., Michel, S., Sadykov, R., Sverdlov, D., Vanderbeck, F.: Column generation based primal heuristics. In: International Conference on Combinatorial Optimization (ISCO). Electronic Notes in Discrete Mathematics, vol. 36, pp. 695–702. Elsevier (2012)
8. Lübbecke, M., Puchert, C.: Primal heuristics for branch-and-price algorithms. In: Operations Research Proceedings 2011. Springer (to appear, 2012)
9. Puchinger, J., Stuckey, P., Wallace, M., Brand, S.: Dantzig-Wolfe decomposition and branch-and-price solving in G12. Constraints 16(1), 77–99 (2011)
10. Ralphs, T., Galati, M.: DIP – decomposition for integer programming (2009), https://projects.coin-or.org/Dip
11. Vanderbeck, F.: On Dantzig-Wolfe decomposition in integer programming and ways to perform branching in a branch-and-price algorithm. Oper. Res. 48(1), 111–128 (2000)
12. Vanderbeck, F.: BaPCod – a generic branch-and-price code (2005), https://wiki.bordeaux.inria.fr/realopt/pmwiki.php/Project/BaPCod
13. Vanderbeck, F.: Implementing mixed integer column generation. In: Desaulniers, G., Desrosiers, J., Solomon, M. (eds.) Column Generation, pp. 331–358. Springer (2005)
14. Vanderbeck, F.: A generic view of Dantzig-Wolfe decomposition in mixed integer programming. Oper. Res. Lett. 34(3), 296–306 (2006)
15. Vanderbeck, F.: Branching in branch-and-price: A generic scheme. Math. Programming 130(2), 249–294 (2011)
16. Vanderbeck, F., Wolsey, L.: Reformulation and decomposition of integer programs. In: Jünger, M., Liebling, T., Naddef, D., Nemhauser, G., Pulleyblank, W., Reinelt, G., Rinaldi, G., Wolsey, L. (eds.) 50 Years of Integer Programming 1958–2008. Springer, Berlin (2010)
17. Villeneuve, D., Desrosiers, J., Lübbecke, M., Soumis, F.: On compact formulations for integer programs solved by column generation. Ann. Oper. Res. 139(1), 375–388 (2005)

Continuous Local Strategies for Robotic Formation Problems

Barbara Kempkes and Friedhelm Meyer auf der Heide

Heinz Nixdorf Institute & Department of Computer Science,
University of Paderborn, 33102 Paderborn
{barbaras,fmadh}@uni-paderborn.de

Abstract. We consider a scenario with a team of autonomous mobile robots which are placed in the plane. Their goal is to move in such a way that they eventually reach a prescribed formation. Such a formation may be a straight line between two given endpoints (Robot Chain Problem), a circle or any other geometric pattern, or just one point (Gathering Problem). In this survey, we assume that there is no central control that guides the robot's decisions, thus the robots have to self-organize in order to accomplish global tasks like the above-mentioned formation problems. Moreover, we restrict them to simple local strategies: the robots are limited to "see" only robots within a bounded viewing range; their decisions where to move next are solely based on the relative positions of robots within this range. Most strategies for these type of problems assume a discrete time model, i.e., time is divided into rounds, in a round, each robot moves to some target point defined by the observations of its environment.

In this talk, we focus on a much less examined class of local strategies, namely *continuous* strategies. Here, each robot continuously observes his environment and continuously adapts its speed and direction to these observations. focus on these type of strategies and survey recent results on local strategies for short robot chains and gathering in the continuous time model. We present such strategies for the Robot Chain and the Gathering Problem and analyze them w.r.t. the distance traveled by the robots. For both problems, we survey bounds for the "price of locality", namely the ratio between the cost of our local algorithms and the optimal cost assuming global view, for each individual start configuration.

1 Introduction

We consider a scenario, in which a team of small and cheap mobile robots cooperates in order to achieve a common global goal like the exploration of an unknown environment, or evacuations from dangerous areas. An important kind of tasks for such a team is to build geometric formations out of an arbitrary configuration of initial positions. Especially, it is important to determine the sensor and actor capabilities of the robots which are needed to achieve the formation. Naturally, the goal is to require as few capabilities as possible in order to be able

R. Klasing (Ed.): SEA 2012, LNCS 7276, pp. 9–17, 2012.

to use robots which are as cheap as possible. Current research focuses on basic tasks such as building lines [1–5] or circles [6, 7], or simply gathering in a point [8–14].

All these results are based on simple models of robots and the environment they live in, so that correctness and efficiency proofs are possible. The environment is a plane without obstacles (for environments with obstacles see, e.g. [4, 15, 16]); the robots are considered as points in the plane (for robots with an extent see, e.g. [17, 18]). The main restriction we are focusing on is their bounded viewing range: robots can only "see" other robots within a fixed viewing radius around their current positions. In the sequel, this viewing radius is normalized to one. They have no compass, but can compute the exact relative positions of their neighbors within their viewing range, i.e. the distances and the angles between the rays to these neighbors (for inaccurate measurements see [19, 20]). Thus a robot has to base its decision where to move next solely on the relative positions of its neighbors within its current viewing range. We refer to such strategies as *local strategies.*

Most strategies developed until now, including all citations from above, assume a discrete time model, in which all robots act in rounds. One round consists of a Look-Operation, where the robots observes the positions of its neighbors, a Compute-Operation, where the strategy uses the observed positions to determine a target point, and a Move-Operation, during which the robot moves to the computed target point. In this model, efficiency of algorithms is typically measured in the number of rounds needed until a formation is reached.

In this talk, we focus on continuous strategies. In a *continuous strategy*, the robots continuously sense their neighborhood and directly adjust their direction and speed. We only demand a speed limit which we normalize to one, the viewing radius. We abstract from several physical limitations of real robots; the most severe one is our assumption that there is no delay between sensing the neighborhood and reacting to the gathered information: The robots can adjust their direction and speed *at the same time* as they observe their neighborhood. We are interested in the *maximum distance traveled* by the robots.

The formation problems considered in this talk are the *Gathering Problem* and the *Robot Chain Problem.*

The Robot Chain Problem is defined as follows: In addition to n mobile robots r_1, ..., r_n, a base camp r_0 and an explorer r_{n+1} are given, which are both stationary. We assume that, in the beginning, r_{i-1} and r_{i+1} are in the viewing range of r_i for $i = 1, ..., n$. Thus, the robots form a maybe winding chain connecting the base camp with the explorer. Moreover, the decisions of r_i are only based on the relative positions of its direct neighbors r_{i-1} and r_{i+1}. The goal is to let all robots move towards the straight line between the base camp and the explorer, the so-called *target line*, while each robot must stay in visibility range of its two neighbors.

The Gathering Problem is to let the n mobile robots $r_1, \ldots, r_n$ gather in one point, which is not prescribed, but must be determined by the robots. The point in which the robots eventually meet is called the *gathering point.* We assume that the visibility graph, i.e., the unit disk graph defined by the viewing radius of the robots, is connected in the beginning. The local strategies will maintain this connectivity.

The next chapters survey the state of the art for the two above-mentioned formation problems in the continuous time model.

2 The Robot Chain Problem

A local strategy for the Robot Chain Problem in the continuous setting gets as input the current positions of the robot's neighbors. It must determine a direction in which the robot should move and its speed based only on this information. A very simple and intuitive strategy for this setting is the **Move-On-Bisector** Strategy introduced in [1]. The idea is to let robot r_i move in direction of the angle bisector of the angle which is formed by r_{i-1}, r_i, r_{i+1} (see Fig. 1). It moves with the maximum speed of 1 in this direction (Phase 1). As soon as a robot reaches the line between its neighbors, it adapts its speed and movement direction in order to stay on and move with this line, keeping the ratio of distances to its neighbors constant (Phase 2). Since the neighbors are also restricted to the maximum speed of 1, this is always possible: a robot will not have to move faster than with speed 1 to stay on this line and to keep the ratio. It is not too hard to see that the distance between two neighbors never increases. Thus, the connectivity of the chain is maintained.

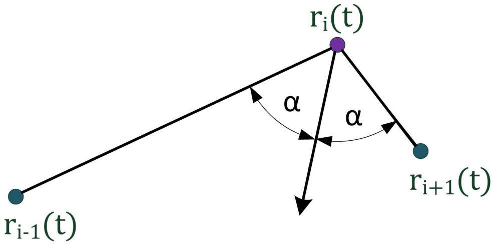
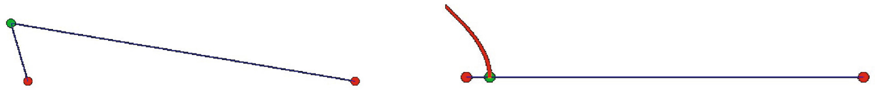


Fig. 1. The Move-On-Bisector Strategy

One property of this strategy is that the robots reach the line and stop moving as soon as the last robot reaches the line between its neighbors. Moreover, since the robots stay in Phase 2 of the algorithm when they have reached it, this last robot is always in Phase 1 and therefore it moves with speed 1 all the time. Its movement path is thus the longest path among the robots, and the length of this path is equal to the time it takes the robots to reach the target line. Two examples can be seen in Figs. 2 and 3.

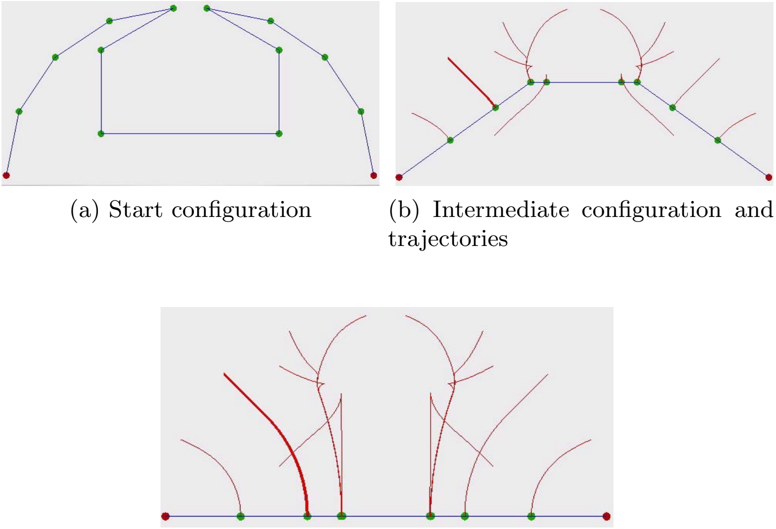
In order to analyze the time or, equivalently, the maximum traveled distance, progress measures need to be defined. In [1], two progress measures are used. The first is the *length* l of the chain, which is defined as the sum of the distances between neighboring robots. It is shown that the length decreases with constant



(a) Start configuration

(b) End configuration and trajectory

Fig. 2. Example for Move-On-Bisector with one robot



(a) Start configuration

(b) Intermediate configuration and trajectories

(c) End configuration and trajectories

Fig. 3. Example for Move-On-Bisector with several robots

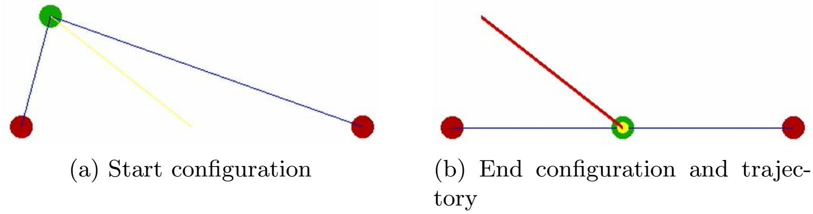
speed, if at least one angle of the chain is smaller than some constant. The second progress measure is a chain's *height* h which is the maximum distance of a robot to the target line. It can be shown that the height decreases with constant speed, if all angles are greater than some constant. From these two progress measures an upper bound of $O(h+l)$ for the maximum traveled distance follows. Now one can observe that $h \leq l \leq n$, which results in an upper bound of $O(l) \subseteq O(n)$.

This upper bound is asymptotically optimal for worst case instances. To see this, consider a configuration with each robot positioned in distance 1 from its two neighbors, such that the middle robot is in distance $n/2$ from the target line. No algorithm, even an optimal global one, can solve this configuration faster than in time $n/2$. This observation is a special case of the more general observation that no configuration with height h can be solved faster than in time h: $OPT \geq h$, if OPT indicates the time needed by a *global algorithm* , i.e., one with global view on the configuration.

Now we turn to a bound on the "price of locality", i.e., we want to compare the distance traveled by the Move-On-Bisector strategy to the one traveled by an optimal global algorithm, for each individual start configuration. It is shown in [1] that the robots reach the target line in time $O((h + d) \log l)$, d denoting the distance between the two stations. Thus, for an arbitrary instance with not too large d, the local strategy is by a factor of at most $O(\log l) \subseteq O(\log n)$ slower than an optimal global algorithm. This yields the following result about the price of locality.

Theorem 1. *When the Move-On-Bisector strategy is performed, the maximum distance traveled by a robot is* $O(\min\{n, (OPT + d) \log(n)\})$*, where* d *is the distance between two stations, and* OPT *the optimal distance traveled by an algorithm with global view on the configuration.*

Instead of moving in direction of the bisector, another intuitive strategy for the same setting is the **Go-To-The-Middle** strategy [21]. Here, each robot moves continuously with speed 1 towards the midpoint towards its neighbors (see Fig. 4). As soon as it reaches this point, it stays on it. Similar to the Move-On-Bisector strategy, the robots reach the target line in time $O(n)$. Runtime bounds depending on h are not known for this strategy. A larger example can be seen in Fig. 5.


(a) Start configuration

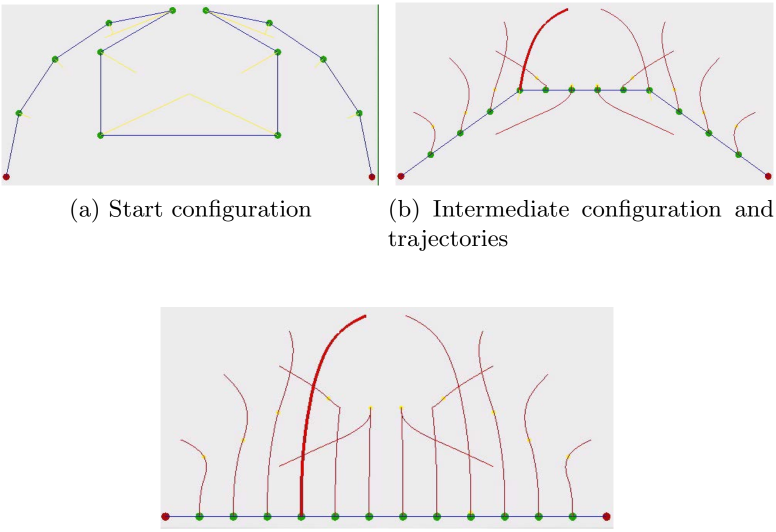
(b) End configuration and trajectory

Fig. 4. Example for Go-To-The-Middle with one robot

3 The Gathering Problem

The gathering problem has some similarities to the robot chain problem. Again, the strategy gets as input the current positions of the neighboring robots and must determine the movement direction and speed based only on this information. A major difference is the larger and dynamic neighborhood of a robot: It is no longer fixed, but can change over time. (Note: at the end all robots are neighbors of each other.)

One algorithm has so far been described and analyzed for this setting. We call it **Gathering-Move-On-Bisector** due to its similarities to the Move-On-Bisectorstrategy. It was first introduced in [14], and here it was also shown that the strategy gathers the robots in finite time. A runtime analysis was presented in [22]. The idea of the algorithm is as follows. Each robot continuously observes its neighborhood and computes the convex hull of all robot positions which it



(a) Start configuration

(b) Intermediate configuration and trajectories

(c) End configuration and movement paths

Fig. 5. Example for Go-To-The-Middle with several robots

can observe. If it is a vertex of the convex hull, it moves (similarly to Move-On-Bisector) with speed 1 on the angle bisector of the inner angle of the convex hull at its own position. If the robot is positioned on a line which forms a border of the convex hull, it moves with the line and stays on it, keeping the ratio of distances to its neighbors on the convex hull constant. This case is also similar to Move-On-Bisector. Otherwise, it does not move at all.

For the analysis of Gathering-Move-On-Bisector, again a *height* h, the maximum distance of a robot to the gathering point, is defined. Note that the gathering point, although not known to the robots, is fixed, since the algorithm is deterministic. A second progress measure is based on the unit disk graph $UDG = (V, E)$ of the robots, which has one vertex for each robot position and an edge between each pair of robots, which are mutually visible. This graph has a well-defined outer border. The *length* l is defined as the sum of the distances between neighboring robots on the outer border.

Unlike Move-On-Bisector, it can be shown for Gathering-Move-On-Bisector that the length decreases with constant speed, independent of the angles into which the robots move. This yields an upper bound of $O(l) \subseteq O(n)$, which is again asymptotically optimal for worst case start configurations. Also here, a bound for the price of locality can be achieved. First note that $h \leq OPT$ holds. A refined version of the technique used for Move-On-Bisectoryields an upper bound of $O(h \log l)$ Together with the insight that $l \leq h^2$ holds, we get an upper bound of $O(h \log l) = O(h \log h) = O(OPT \log OPT)$. Therefore, the local algorithm is only by a factor of $\log OPT$ slower than an optimal global algorithm. These considerations yield the following result on the price of locality.

Theorem 2. *When the Gathering-Move-On-Bisector strategy is performed, the maximum distance traveled by a robot is* $O(\min\{OPT \log OPT, n\})$*, where* OPT *denotes the optimal distance traveled by an algorithm with global view on the configuration.*

4 Discussion of Models and Strategies

Some questions about the described strategies remain open. For all strategies, only upper runtime bounds have been shown. It is therefore not yet known whether the described bounds are tight: the competitive factors might even be better. Of interest are therefore better upper bounds as well as lower bounds for the described strategies and for general local strategies.

A classical time model which is often used for this type of problems is a discrete and synchronous model like described in the introduction, in which all robots act in synchronous rounds according to the Look-Compute-Move model. A bridge between this discrete model and the continuous model described in this talk is given by so-called δ-*bounded* strategies, which allow the robots to move only up to a distance of $\delta \leq 1$ in one round. For $\delta = 1$, we get the classical discrete model, and for $\delta \to 0$ and a speed limit of 1, we have the continuous time model. The Go-To-The-Middle strategy for the robot chain problem has been analyzed in this setting in [21].

Our continuous model is still far away from real robotic capabilities. Bridging the gap could start with introducing a delay between the observation of the neighborhood and the reaction to the observation. This extension makes the model more realistic, but also more difficult to analyze. In this setting, situations can occur which are similar to those for discrete and synchronous settings: Robots move too far, before they observe this and move back again.

One of the physically unrealistic features is the ignorance of problems with acceleration and turning. It would be interesting to find ways to incorporate such restrictions in a model so that properties of strategies still can be formally proven.

References

1. Degener, B., Kempkes, B., Kling, P., Meyer auf der Heide, F.: A Continuous, Local Strategy for Constructing a Short Chain of Mobile Robots. In: Patt-Shamir, B., Ekim, T. (eds.) SIROCCO 2010. LNCS, vol. 6058, pp. 168–182. Springer, Heidelberg (2010)
2. Dynia, M., Kutylowski, J., Lorek, P., Meyer auf der Heide, F.: Maintaining Communication Between an Explorer and a Base Station. In: IFIP 19th World Computer Congress, TC10: 1st IFIP International Conference on Biologically Inspired Computing (BICC 2006), pp. 137–146 (2006)
3. Kutylowski, J., Meyer auf der Heide, F.: Optimal strategies for maintaining a chain of relays between an explorer and a base camp. Theoretical Computer Science 410(36), 3391–3405 (2009)

4. Dynia, M., Kutylowski, J., Meyer auf der Heide, F., Schrieb, J.: Local Strategies for Maintaining a Chain of Relay Stations between an Explorer and a Base Station. In: SPAA 2007: Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 260–269. ACM Press, New York (2007)
5. Kling, P., Meyer auf der Heide, F.: Convergence of Local Communication Chain Strategies via Linear Transformations. In: SPAA 2011: Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures, pp.159–166 (2011)
6. Défago, X., Konagaya, A.: Circle formation for oblivious anonymous mobile robots with no common sense of orientation. In: Proceedings of the 2002 Workshop on Principles of Mobile Computing, POMC 2002, pp. 97–104 (2002)
7. Chatzigiannakis, I., Markou, M., Nikoletseas, S.E.: Distributed Circle Formation for Anonymous Oblivious Robots. In: Ribeiro, C.C., Martins, S.L. (eds.) WEA 2004. LNCS, vol. 3059, pp. 159–174. Springer, Heidelberg (2004)
8. Meyer auf der Heide, F., Schneider, B.: Local Strategies for connecting stations by small robotic networks. In: IFIP 20th World Computer Congress, TC10: 2nd IFIP International Conference on Biologically Inspired Computing (BICC 2008), pp. 95–104 (2008)
9. Degener, B., Kempkes, B., Meyer auf der Heide, F.: A local O(nˆ2) gathering algorithm. In: SPAA 2010: Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures, pp. 217–223 (2010)
10. Ando, H., Suzuki, Y., Yamashita, M.: Formation and agreement problems for synchronous mobile robots with limited visibility. In: Proceedings of the 1995 IEEE International Symposium on Intelligent Control, ISIC 1995, pp. 453–460 (August 1995)
11. Flocchini, P., Prencipe, G., Santoro, N., Widmayer, P.: Gathering of asynchronous robots with limited visibility. Theoretical Computer Science 337(1-3), 147–168 (2005)
12. Ando, H., Oasa, Y., Suzuki, I., Yamashita, M.: Distributed memoryless point convergence algorithm for mobile robots with limited visibility. IEEE Transactions on Robotics and Automation 15(5), 818–828 (1999)
13. Katreniak, B.: Convergence with Limited Visibility by Asynchronous Mobile Robots. In: Kosowski, A., Yamashita, M. (eds.) SIROCCO 2011. LNCS, vol. 6796, pp. 125–137. Springer, Heidelberg (2011)
14. Gordon, N., Wagner, I.A., Bruckstein, A.M.: Gathering Multiple Robotic A(ge)nts with Limited Sensing Capabilities. In: Dorigo, M., Birattari, M., Blum, C., Gambardella, L.M., Mondada, F., Stützle, T. (eds.) ANTS 2004. LNCS, vol. 3172, pp. 142–153. Springer, Heidelberg (2004)
15. Desai, J.P., Ostrowski, J.P., Kumar, V.: Controlling Formations of Multiple Mobile Robots. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA 1998), vol. 4, pp. 2864–2869 (1998)
16. Fahimi, F., Nataraj, C., Ashrafiuon, H.: Real-time obstacle avoidance for multiple mobile robots. Robotica 27(2), 189–198 (2009)
17. Czyzowicz, J., Gasieniec, L., Pelc, A.: Gathering few fat mobile robots in the plane. Theoretical Computer Science 410(6-7), 481–499 (2009)
18. Cord-Landwehr, A., Degener, B., Fischer, M., Hüllmann, M., Kempkes, B., Klaas, A., Kling, P., Kurras, S., Märtens, M., Meyer auf der Heide, F., Raupach, C., Swierkot, K., Warner, D., Weddemann, C., Wonisch, D.: Collisionless Gathering of Robots with an Extent. In: Černá, I., Gyimóthy, T., Hromkovič, J., Jefferey, K., Královič, R., Vukolić, M., Wolf, S. (eds.) SOFSEM 2011. LNCS, vol. 6543, pp. 178–189. Springer, Heidelberg (2011)

19. Oasa, Y., Suzuki, I., Yamashita, M.: A robust distributed convergence algorithm for autonomous mobile robots. In: IEEE International Conference on Systems, Man, and Cybernetics (SMC 1997), vol. 1, pp. 287–292 (1997)
20. Cord-Landwehr, A., Degener, B., Fischer, M., Hüllmann, M., Kempkes, B., Klaas, A., Kling, P., Kurras, S., Märtens, M., Meyer auf der Heide, F., Raupach, C., Swierkot, K., Warner, D., Weddemann, C., Wonisch, D.: A New Approach for Analyzing Convergence Algorithms for Mobile Robots. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part II. LNCS, vol. 6756, pp. 650–661. Springer, Heidelberg (2011)
21. Brandes, P., Degener, B., Kempkes, B., Meyer auf der Heide, F.: Energy-Efficient Strategies for Building Short Chains of Mobile Robots Locally. In: Kosowski, A., Yamashita, M. (eds.) SIROCCO 2011. LNCS, vol. 6796, pp. 138–149. Springer, Heidelberg (2011)
22. Kempkes, B., Meyer auf der Heide, F.: Optimal and competitive runtime bounds for continuous, local gathering of mobile robots. To Appear in Proceedings of SPAA 2012 (2012)

Engineering Graph Partitioning Algorithms

Vitaly Osipov, Peter Sanders, and Christian Schulz

Karlsruhe Institute of Technology (KIT), 76128 Karlsruhe, Germany
{osipov,sanders,christian.schulz}@kit.edu

Abstract. The paper gives an overview of our recent work on balanced graph partitioning – partition the nodes of a graph into k blocks such that all blocks have approximately equal size and such that the number of cut edges is small. This problem has numerous applications for example in parallel processing. We report on a scalable parallelization and a number of improvements on the classical multilevel approach which leads to improved partitioning quality. This includes an integration of flow methods, improved local search, several improved coarsening schemes, repeated runs similar to the approaches used in multigrid solvers, and an integration into a distributed evolutionary algorithm. Overall this leads to a system that for many common benchmarks leads to both the best quality solution known and favorable tradeoffs between running time and solution quality.

1 Introduction

Graph partitioning is a common technique in computer science, engineering, and related fields. For example, good partitionings of unstructured and irregular graphs are very valuable in the area of *high performance computing*, e.g., when solving *partial differential equations*. These equations are usually discretized and then solved numerically using a parallel computer, e.g. using a CG method. To effectively balance the load we need a graph model of computation and communication. Roughly speaking, vertices in the graph represent computation units and edges denote communication. Now this graph needs to be partitioned such that there are few edges between the blocks (pieces). In particular, when we want to solve the partial differential equation in parallel on k PEs (processing elements) we want to partition the graph into k blocks of about equal size. In this paper we focus on a version of the problem that constrains the maximum block size to $(1 + \epsilon)$ times the average block size and tries to minimize the total cut size, i.e., the number of edges that run between blocks.

A successful heuristics for partitioning large graphs is the *multilevel* approach depicted in Figure 1 where the graph is recursively *contracted* to achieve a smaller graph with the same basic structure. After applying an *initial partitioning* algorithm to this small graph, the contraction is undone and, at each level, a *local refinement* method is used to improve the partitioning induced by the coarser level. Refer to [1–4] for overviews on existing methods. Although several successful multilevel partitioners have been developed in the last 14 years, we had the impression that certain aspects of the method are not well understood. We therefore have built our own graph partitioner KaPPa [5] (Karlsruhe Parallel Partitioner) with focus on scalable parallelization. Somewhat astonishingly, we also obtained improved partitioning quality through rather simple methods. This motivated us to make a fresh start putting all aspects of MGP on trial.

R. Klasing (Ed.): SEA 2012, LNCS 7276, pp. 18–26, 2012.

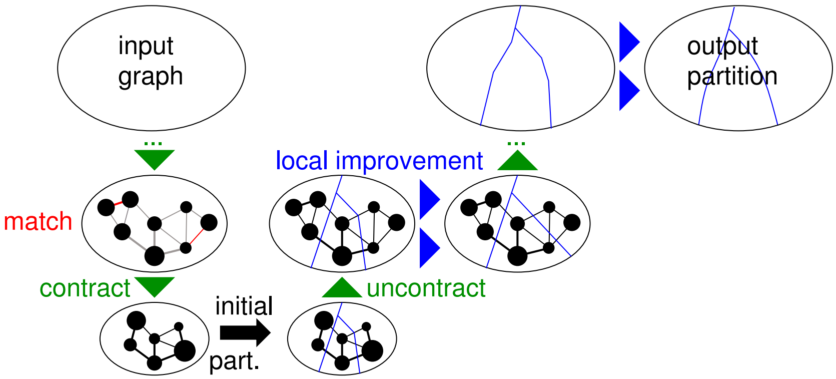


Fig. 1. Multilevel graph partitioning

This paper gives an overview of our recent work on balanced graph partitioning. We present four algorithms: KaPPa [5], KaSPar [6] (Karlsruhe Sequential Partitioner) which contracts only a single edge per level, KaFFPa [7] (Karlsruhe Fast Flow Partitioner) which uses advanced refinement techniques, and the distributed evolutionary algorithm, KaFFPa(E)volutionary [8]. We only give a short outline of the main ideas and refer to the respective papers for more details.

To give some minimalistic experimental data, we consider Walshaw's benchmark [9] which consists of 34 graphs with up to 3.3 million edges, $k \in \{2, 4, 8, 16, 32, 64\}$ blocks and imbalance $\epsilon \in \{0, 1\%, 3\%, 5\%\}$. Excluding the case $\epsilon = 0$ which our codes do not handle yet, we obtain 612 instances. For each algorithm we will report the number of instances where we are the record holder or at least as good as the record holder. On the first glance it looks more fair to exclude ties but this would complicate the figure since we would also have to differentiate between ties with our own codes and improvements that entered the archive recently. Moreover, for $k = 2$ and many of the smaller graphs no improvements have been found for a long time which indicates that the solutions might already be optimal.

2 *Ka*rlsruhe *P*arallel *Pa*rtitioner [5]

We now present our parallel approach to graph partitioning. First of all the graph is distributed among all k PEs. This is done by computing a preliminary partition of the graph. Currently we have implemented a recursive bisection algorithm for nodes with 2D coordinates that alternately splits the data by the x-coordinate and the y-coordinate [10, 11]. We can also use the initial numbering of the nodes.

Now we have to compute matchings to create coarser versions of the graph. Following the basic approach from [12] we combine a sequential matching algorithm running on each PE and a parallel matching algorithm running on the *gap graph*. The gap graph consists of those edges $\{u, v\}$ where u and v reside on different PEs and $\omega(\{u, v\})$ exceeds the weight of the edges that may have been matched by the local matching algorithms to u and v.

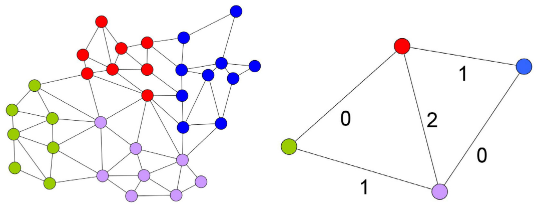


Fig. 2. A graph which is partitioned into four blocks and its corresponding quotient graph $\mathcal{Q}$. The quotient graph has an edge coloring indicated by the numbers and each edge set induced by edges with the same color form a matching $\mathcal{M}(c)$. Pairs of blocks with the same color can be refined in parallel.

In [5], expanding on an idea already present in [13], we proposed to make contraction more systematic by separating two issues: A *rating function* indicates how much sense it makes to contract an edge based on *local* information. A *matching* algorithm tries to maximize the sum of the ratings of the contracted edges looking at the *global* structure of the graph. While the rating functions allows us a flexible characterization of what a "good" contracted graph is, the simple, standard definition of the matching problem allows us to reuse previously developed algorithms for weighted matching. Matchings are contracted until the graph is "small enough". In most previous work, the edge weight $\omega(e)$ itself is used as a rating function (see [5] for more details). We have shown in [5] that the rating function $\mathrm{expansion}^{*2}(\{u,v\}) := \frac{\omega(\{u,v\})^2}{c(u)c(v)}$ works best among other edge rating functions where $c(\cdot)$ is the node weight – usually the number of input nodes contracted into a node in the current graph.

We employed the *Global Path Algorithm (GPA)* as sequential matching algorithm. It was proposed in [14] as a synthesis of the Greedy algorithm and the Path Growing Algorithm [15]. This algorithm achieves a half-approximation in the worst case, but empirically, GPA gives considerably better results than Sorted Heavy Edge Matching and Greedy (for more details look into [5]). Our implementation of the parallel matching algorithm proposed in [12] iteratively matches edges in the gap graph $\{u, v\}$ that are locally heaviest both at u and v until no more edges can be matched.

The contraction is stopped when the number of remaining nodes is small enough. We employ Scotch [16] as an initial partitioner since it empirically performs better than Metis [3]. This algorithm is then run simultaneously on all PEs, each with a different seed for the random number generator. The best solution is then broadcast to all PEs.

Recall that the refinement phase iteratively uncontracts the matchings contracted during the contraction phase. After a matching is uncontracted, local search based refinement algorithms move nodes between block boundaries in order to reduce the cut while maintaining the balancing constraint. As most other current systems, we adopt the FM-algorithm [17] which runs in linear time. The main difference of our approach to previous systems is that at any time, each PE may work on only one pair of neighboring blocks performing a local search constrained to moving nodes between these two blocks. Thus, we need parallel algorithms for deciding which processors work on which pairs of blocks. For this purpose, we use the *quotient graph* Q whose nodes are blocks of the current partition and whose edges indicate that there are edges between these blocks in the underlying graph G. Since we have the same number of PEs and blocks, each PE will work on the block assigned to it and at one of its neighbors in Q. Figure 2 gives an example. We schedule all possible pairwise refinement operations by using a distributed parallel edge coloring algorithm on the quotient graph. The edges in each color class define a matching in the quotient graph and can be handled in parallel. The PEs at both endpoints of an edge work on the same pairwise refinement operation redundantly but using different random number seeds in tie breaking operations. After the local search is finished, the better partitioning of the two blocks is adopted.

In the past, parallelizing a graph partitioner meant giving up some quality for speed. Somewhat surprisingly, this was not the case for KaPPa. In the Walshaw benchmark we obtain 189 best values, in particular for the largest graphs. Besides the improvements like edge ratings that are also easy to integrate into previous solvers, one source of improvement was the more focused local search. Our interpretation is that this makes it more likely that the local search is successful.

3 The n-Level Approach [6]

The success of focusing local search in KaPPa gave us the idea to drive this focusing idea into the extreme. This fits well the the idea of an n-level multilevel algorithm that we have previously used successfully for route planning [18] and the nearest neighbor problem [19].

The central idea behind this (sequential) approach we called KaSPar (Karlsruhe Sequential Partitioner) [20] is to make subsequent levels as similar as possible – we (un)contract only a *single* edge between two levels. We call this n-GP since we have (almost) n levels of hierarchy. Figure 1 gives a high-level recursive summary of n-GP. We use similar edge rating functions and initial partitioning as in KaPPa. However, note that no matching algorithm is needed. Rather, on each level, we choose a single edge to be contracted using a priority queue.

In order to make contraction and uncontraction efficient, we use a "semidynamic" graph data structure: When contracting an edge $\{u, v\}$, we mark both u and v as deleted, introduce a new node w, and redirect the edges incident to u and v to w. The advantage of this implementation is that edges adjacent to a node are still stored in adjacency arrays which are more efficient than linked lists needed for a full fledged dynamic graph data structure. A disadvantages of our approach is a certain space overhead. However, it is relatively easy to show that this space overhead is bounded by a logarithmic factor even if we contract edges in some random fashion (see [21]). Overall, with respect to asymptotic memory overhead, n-GP is no worse than methods with a logarithmic number of levels.

The local search strategy is similar to the FM-algorithm [17]. We now outline our variant. Initially, all nodes are unmarked and inactive. The neighbors of the recently expanded edge are activated. Active nodes reside in priority queues – one for each block they could be moved to. The key for the priority queue is the *gain*, i.e., the decrease in edge cut when the node is moved (which can also be negative). We call a queue P_B

Algorithm 1. n-GP(G, k, ϵ).

```
if $G$ is small then
    return initialPartition$(G, k, \epsilon)$
pick the edge $e = \{u, v\}$ with highest rating
contract $e$; $\mathcal{P} := n-GP(G, k, \epsilon)$; uncontract $e$
activate$(u)$;   activate$(v)$;   localSearch()
return $\mathcal{P}$
```

eligible if the highest gain node in P_B can be moved to block B without violating the balance constraint for block B. Local search repeatedly looks for the highest gain node v in any eligible priority queue P_B and moves v to block B. When this happens, node v becomes nonactive and marked, the unmarked neighbors of v get activated and the gains of the active neighbors are updated. The local search is stopped if either no eligible nonempty queues remain or some additional stopping criterion hold. After the local search stopped, it is rolled back to the lowest cut state reached during the search. Subsequently all previously marked nodes are unmarked. The local search is repeated until no improvement is achieved.

The problem with this approach is that the local search would usually spread over the whole graph which in conjunction with the linear number of levels could lead to quadratic running time. We therefore introduced additional stopping criteria. First of all, our local search does nothing if none of the uncontracted nodes is a *border node*, i.e., has a neighbor in another block. Other FM-algorithms initialize the search with all border nodes. Indeed, our experiments indicate that for large graphs and small number of partitions k, the overall local search effort may grow sublinearly with the input size. To achieve this we also need a way to abort unpromising local searches that *are* started. We do this by modelling the local search as a random walk on the cut size axis. If within this model it becomes unlikely to reach an overall improvement in a linear number of steps, we abort the search.

KaSPar achieves 238 best values for the Walshaw benchmark although it is a much simpler code than KaPPa.

4 ***Ka*****rlsruhe *F*****ast *F*****low *P*****artitioner [7]**

In order to implement further ideas for improving the quality of sequential multilevel graph partitioning we went back to the traditional variant with a logarithmic number of levels – this allows us to use simple and fast static graph representations and enables a more global view on refinement. We still exploit the advantages of highly focused local search by performing many focused searches initialized with a single border node.

Max-Flow Min-Cut Local Improvement. A more global method is based on max-flow min-cut computations between pairs of blocks, in other words, a method to improve a given bipartition. Roughly speaking, this improvement method is applied between all pairs of blocks that share a nonempty boundary. The algorithm basically constructs a flow problem by growing an area around the given boundary vertices of a pair of blocks such that each s-t cut in this area yields a feasible bipartition of the original graph/pair of blocks *within* the balance constraint. One can then apply a max-flow min-cut algorithm to obtain a min-cut in this area and therefore an improved cut between the original pair of blocks. This can be improved in multiple ways, for example, by iteratively applying the method, searching in larger areas for feasible cuts, and applying most balanced minimum cut heuristics.

Global Search. KaFFPa extends the concept of *iterated multilevel algorithms* which was introduced by [22] and can be traced back to multigrid solvers for sparse systems of linear equations. The main idea is to iterate the coarsening and uncoarsening phase. Once the graph is partitioned, edges that are between two blocks are not contracted. An F-cycle works as follows: on *each* level we perform at most *two recursive calls* using different random seeds during contraction and local search. A second recursive call is only made the second time that the algorithm reaches a particular level. Figure 3 illustrates a F-cycle. As soon as the graph is partitioned, edges that are between blocks are not contracted. This ensures nondecreasing quality of the partition since our refinement algorithms guarantee no worsening and break ties randomly. These so called *global search strategies* are more effective than plain restarts of the algorithm.

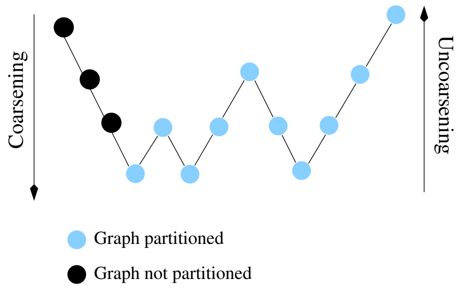


Fig. 3. An F-cycle

KaFFPa achieves 435 best values in the Walshaw benchmark.

5 KaFFPa *E*volutionary [8]

In the Walshaw benchmark, KaFFPa was beaten mostly for small graphs that combine multilevel partitioning with an evolutionary algorithm. We therefore developed an improved evolutionary algorithm that also employs coarse grained parallelism. Roughly speaking, KaFFPaE uses KaFFPa to create individuals and modifies the coarsening phase to provide new effective combine operations. We restrict ourselves to the description of the combine operator framework and the parallelization.

An EA starts with a population of individuals (in our case partitions of the graph) and evolves the population into different populations over several rounds. In each round, the EA uses a selection rule based on the fitness of the individuals (in our case the edge cut) of the population to select good individuals and combine them to obtain improved offspring.

Combine Operators. In KaFFPaE we have a general combine operator framework, i.e. a partition $\mathcal{P}$ can be combined with another partition of the population or an arbitrary clustering of the graph. This is achieved by running a modified version of KaFFPa that during coarsening will not contract edges that are cut in one of the input partitions/clusterings. As soon as the coarsening phase is stopped, we apply the partition $\mathcal{P}$ to the coarsest graph and use this as initial partitioning. This way the resulting partition is at least as good as the input partition and in addition, the refinement algorithms can effectively exchange good parts of the solution on the coarse levels by moving only a few vertices.

Parallelization. We parallelize the evolutionary algorithm on p processing elements (PEs), by distributing the population over the PEs. Basically, every PE runs a sequential evolutionary algorithm on its subpopulation. To achieve fast initialization for large p, each PE will begin with a single individual. Subsequently, using random cyclic communication patterns, each PE is equipped with a random selection of these initial individuals. After initialization, some interaction is achieved by periodically sending the best local solution to a random PE. Communication volume is limited by sending the same solution at most $\log p$ times. This has been implemented using MPI.

KaFFPaE achieves 470 best values in the Walshaw benchmark, investing two hours of time on the small to medium sized graphs and eight hours of time on the eight largest graphs of the archive, on two somewhat outdated 8-core nodes. It should also be noted that very good values are already achieved in seconds to minutes of parallel execution time so that the solutions can also be used for many applications rather than only for record hunting. The parallelization scales well to hundreds of processors even for small graphs.

6 10th DIMACS Implementation Challenge

We coorganized a DIMACS implementation challenge on graph partitioning and clustering[1]. An important outcome is collection of benchmark graphs that not only has more instances than the Walshaw collection but also more varied applications and larger graphs with up to 3 billion edges. Our partitioners also work very well on these instances achieving the best marks both with respect to quality and running time versus quality among all participants. A surprising result was obtained for a part of the challenge where the objective function was not cut size but a measure of communication volume. This objective function can be expressed as a hypergraph partitioning problem. Interestingly, KaFFPaE outperformed dedicated hypergraph partitioners by just changing the fitness function to prefer solutions with low communication volume – the multilevel algorithm still optimizes cuts. This is an indication that some of our techniques would also be useful in a multilevel algorithm for optimizing communication volume or even for a general hypergraph partitioner.

[1] http://www.cc.gatech.edu/dimacs10/

7 Conclusions and Future Work

The perspective taken in this paper is that we developed our graph partitioners KaPPa, KaSPar, KaFFPa, and KaFFPaE in a benchmark driven way achieving overall 550 out of 612 optimal entries in the Walshaw benchmark with $\epsilon > 0$. Another equally valid perspective is that we applied the methodology of algorithm engineering to all aspects of the multi-level graph partitioning approach, achieving improvements in coarsening, refinement, parallelization, global search guidance, and embedding into metaheuristics.

Both perspectives also allow an outlook on open problems: Within the Walshaw benchmark, the perfectly balanced case ($\epsilon = 0$) is an interesting case requiring new techniques to obtain good solutions fast. The DIMACS challenge indicates that for difficult instances like social networks, a lot of work remains to be done. Considerations for massively parallel computing including petascale and exascale indicate that we have to go back to scalable parallelization for big instances that do not fit into internal memory of a single node and that much larger values of k will become relevant. On the other hand, its also not clear how to employ many processors for small values of k. This is important for recursive partitioning schemes that are useful in many cases and can also more easily adapt to hierarchical computer architectures.

A feature oriented view has initial partitioning as an obvious open point where we still do not have our own solution – currently we are using Scotch [16] in all our systems. Another interesting issue are new approaches to coarsening. In [23] we are cooperating on an approach where nodes are fractionally assigned to nodes on the coarser levels. This gives us more flexibility in shaping the coarse levels and seems promising for social networks and other graphs that have problems with edge contraction approaches.

Yet another view could be the different activities in algorithm engineering. The reported work is strong on design, implementation, experimental evaluation and benchmarking but weak on other aspects. We plan to release an easy to use algorithm library with some of our codes soon. Theoretical analysis of complex metaheuristics like KaFFPaE is very difficult. However note that an astonishingly large set of "easier" graph algorithms are used "under the hood" that are theoretically better understood: weighted matching, spanning trees, edge coloring, BFS, shortest paths, diffusion, maximum flows, and strongly connected components. Perhaps this is one justification why a group coming from algorithm theory can be successful in real world graph partitioning. Yet the most interesting activity from algorithm engineering for finding future work is modeling. Applications indicate that we should also look at other objective functions (e.g., communication volume, separator size, and bottleneck variants), hypergraph partitioning, and clustering (where balance is not directly relevant and k may not be known or flexible). Some of our techniques like edge ratings, F-cycles, or n-level contraction might also be relevant for other multilevel graph algorithms, e.g., for graph drawing.

References

1. Fjallstrom, P.: Algorithms for graph partitioning: A survey. Linkoping Electronic Articles in Computer and Information Science 3(10) (1998)
2. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM Journal on Scientific Computing 20(1), 359–392 (1998)

3. Schloegel, K., Karypis, G., Kumar, V.: Graph partitioning for high performance scientific simulations. Technical Report 00-018, University of Minnesota (2000)
4. Walshaw, C., Cross, M.: JOSTLE: Parallel Multilevel Graph-Partitioning Software – An Overview. In: Magoules, F. (ed.) Mesh Partitioning Techniques and Domain Decomposition Techniques, pp. 27–58. Civil-Comp Ltd. (2007) (invited chapter)
5. Holtgrewe, M., Sanders, P., Schulz, C.: Engineering a Scalable High Quality Graph Partitioner. In: 24th IEEE International Parallal and Distributed Processing Symposium (2010)
6. Osipov, V., Sanders, P.: *n*-Level Graph Partitioning. In: de Berg, M., Meyer, U. (eds.) ESA 2010. LNCS, vol. 6346, pp. 278–289. Springer, Heidelberg (2010) see also arxiv preprint arXiv:1004.4024
7. Sanders, P., Schulz, C.: Engineering Multilevel Graph Partitioning Algorithms. In: Demetrescu, C., Halldórsson, M.M. (eds.) ESA 2011. LNCS, vol. 6942, pp. 469–480. Springer, Heidelberg (2011)
8. Sanders, P., Schulz, C.: Distributed Evolutionary Graph Partitioning. In: 12th Workshop on Algorithm Engineering and Experimentation (2011)
9. Soper, A., Walshaw, C., Cross, M.: A combined evolutionary search and multilevel optimisation approach to graph-partitioning. Journal of Global Optimization 29(2), 225–241 (2004)
10. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Commun. ACM 18(9), 509–517 (1975)
11. Berger, M.J., Bokhari, S.H.: A partitioning strategy for pdes across multiprocessors. In: ICPP, pp. 166–170 (1985)
12. Manne, F., Bisseling, R.H.: A Parallel Approximation Algorithm for the Weighted Maximum Matching Problem. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2007. LNCS, vol. 4967, pp. 708–717. Springer, Heidelberg (2008)
13. Abou-Rjeili, A., Karypis, G.: Multilevel algorithms for partitioning power-law graphs. In: International Parallel & Distributed Processing Symposium (2006)
14. Maue, J., Sanders, P.: Engineering Algorithms for Approximate Weighted Matching. In: Demetrescu, C. (ed.) WEA 2007. LNCS, vol. 4525, pp. 242–255. Springer, Heidelberg (2007)
15. Drake, D., Hougardy, S.: A simple approximation algorithm for the weighted matching problem. Information Processing Letters 85, 211–213 (2003)
16. Pellegrini, F.: Scotch home page, http://www.labri.fr/pelegrin/scotch
17. Fiduccia, C.M., Mattheyses, R.M.: A Linear-Time Heuristic for Improving Network Partitions. In: 19th Conference on Design Automation, pp. 175–181 (1982)
18. Geisberger, R., Sanders, P., Schultes, D., Vetter, C.: Exact routing in large road networks using contraction hierarchies. Transportation Science (to appear, 2012)
19. Birn, M., Holtgrewe, M., Sanders, P., Singler, J.: Simple and fast nearest neighbor search. In: 11th Workshop on Algorithm Engineering and Experiments, ALENEX (2010)
20. Osipov, V., Sanders, P.: *n*-Level Graph Partitioning. In: de Berg, M., Meyer, U. (eds.) ESA 2010. LNCS, vol. 6346, pp. 278–289. Springer, Heidelberg (2010)
21. Dementiev, R., Sanders, P., Schultes, D., Sibeyn, J.: Engineering an external memory minimum spanning tree algorithm. In: IFIP TCS, Toulouse, pp. 195–208 (2004)
22. Walshaw, C.: Multilevel refinement for combinatorial optimisation problems. Annals of Operations Research 131(1), 325–372 (2004)
23. Safro, I., Sanders, P., Schulz, C.: Advanced Coarsening Schemes for Graph Partitioning. In: Klasing, R. (ed.) SEA 2012. LNCS, vol. 7276, pp. 369–380. Springer, Heidelberg (2012)

Space Efficient Modifications to Structator— A Fast Index-Based Search Tool for RNA Sequence-Structure Patterns

Benjamin Albrecht and Volker Heun

Institut für Informatik, Ludwig-Maximilians-Universität,
Amalienstr. 17, 80333 München, Germany
{benjamin.albrecht,volker.heun}@bio.ifi.lmu.de

Abstract. This work deals with the program Structator — a fast index-based search tool for RNA sequence-structure patterns. We present two space efficient modifications which both outperform the existing methods when searching large databases. Thus, our new methods make the program accessible for a wider range of real-world applications.

1 Introduction

Over the last years the importance of non-coding RNA has been grown [11, 12]. To store information about those RNA sequences, different databases, e. g. the Rfam database of the Sanger institute [6], have been created. Nevertheless, by increasing the size of such a database, the search for a particular sequence is getting more and more complex. Thus, a fast search-tool became indispensable. For developing such a tool the properties of non-coding RNA have to be taken into account.

Non-coding RNAs are grouped into different families each fulfilling a certain function. Examples of such families are rRNAs, tRNAs, and small RNAs like miRNAs, siRNAs, or piRNAs. It is widely known that the function of a particular non-coding RNA molecule strongly depends on its structure. Hence, usually the structures within one family are even more conserved than its sequences. This fact directly implies that a search for a particular non-coding RNA sequence additionally has to take its secondary structure into account [5].

Up to now several sequence-structure alignment tools exist. A short summary is given by Meyer et al. [13]. The main problem of these tools is that the more target sequences exist the more complex a database search for a particular sequence-structure pattern gets. To overcome the complexity of this search tools, an index-based search method has been introduced [15]. This method is based on an affix array which consists of two main tables — one for each reading direction of the concatenated target-sequences of the database. Each main table itself consists of three sub-tables containing the *suffix array*, the *lcp array*, and the *affix link array* (cf. Tab. 1).

While the lcp array contains rather small values and, thus, can be stored close to 1 byte per character, the suffix array and the affix link array each have a space

R. Klasing (Ed.): SEA 2012, LNCS 7276, pp. 27–38, 2012.

Table 1. An affix array for the sequence AUAGCUGCUGCUGCA

i	$\mathrm{suf}_{\mathrm{F}}[i]$	$\mathrm{lcp}_{\mathrm{F}}[i]$	$\mathrm{aflk}_{\mathrm{F}}[i]$	$\mathcal{S}^{\mathrm{F}}_{\mathrm{suf}_{\mathrm{F}}[i]}$	$(\mathcal{S}^{\mathrm{R}}_{\mathrm{suf}_{\mathrm{R}}[i]})^{-1}$	$\mathrm{aflk}_{\mathrm{R}}[i]$	$\mathrm{lcp}_{\mathrm{R}}[i]$	$\mathrm{suf}_{\mathrm{R}}[i]$	i
0	2	0	0	AGCUGC...	...GCUGCA	0	0	0	0
1	1	1		AUAGCU...	AUA		1	12	1
2	14	1		A	A		1	14	2
3	13	0	3	CA	AUAGC	7	0	10	3
4	10	1	4	CUGCA	AUAGCUGC	8	2	7	4
5	7	4	5	CUGCUGCA	...GCUGCUGC	9	5	4	5
6	4	7		CUGCUGCU...	...UGCUGCUGC		8	1	6
7	12	0	3	GCA	AUAG	7	0	11	7
8	9	2	4	GCUGCA	AUAGCUG	8	1	8	8
9	6	5	5	GCUGCUGCA	AUAGCUGCUG	9	4	5	9
10	3	8		GCUGCUGCU...	...CUGCUGCUG		7	2	10
11	1	0	11	UAGCUG...	AU	11	0	13	11
12	11	1	4	UGCA	AUAGCU	8	1	9	11
13	8	3	5	UGCUGCA	AUAGCUGCU	9	3	6	13
14	5	6		UGCUGCUGCA	...GCUGCUGCU		6	3	14
15	15	0					0	15	15

consumption of 4 bytes per character. Providing two tables of this kind, for the affix array data structure in total 18 bytes per character (plus the size of the target sequences) are needed. Note that for each space analysis we assume that the database does not contain more than 2^{32} characters such that each index can be represented with at most 4 bytes. Of course, for small databases this factor does not play an important role. Nevertheless, the more information the database stores, the more important the space factor can get.

For example, let us consider the Rfam database [6] release 10.0. It contains around 3.2 million sequences having a size of 630 MiB. This means that the affix array needs $18 * 630$ MiB ≈ 10.13 GiB memory. Of course, there are computer systems, which are able to handle such a large amount of memory. Nowadays, however, such systems are very expensive and not available for every scientist. In this work, we present two methods that reduce the space consumption of the precomputed index from 18 to 10 or 12 bytes per character. For the given example this means, that applying one of these modifications only $10 * 630$ MiB ≈ 6.3 GiB or $12 * 630$ MiB ≈ 7.6 GiB memory instead of 10.13 GiB is needed.

Applying an index based search tool to a large database, there can arise two main problems. On the one hand, the memory allocation during the index construction can fail and, thus, no index is constructed. In this case there is no possibility for the user to perform any pattern searches on his system. On the other hand, the index files may be too large to be stored in the main memory and, thus, have to be swapped all the time. Since these swapping operations can be very exhaustive, the run time of a pattern search suffers from that. To overcome these memory problems, the user can either invest in more memory or apply an algorithm with less space consumption. Since the second solution is at no charge, it should be the most preferred one.

2 Structator—A Fast Index-Based Search Tool for RNA Sequence-Structure Patterns

This work is based on the index-based search tool Structator developed by Meyer et al. [13]. Structator makes use of affix arrays — a data structure which supports efficient unidirectional as well as bidirectional search (cf. Fig. 1). Its source code is available under the GNU General Public License Version 3 and can be downloaded from www.zbh.uni-hamburg.de/Structator.

The program Structator provides two main features. Given a FASTA file containing all target sequences, the user can construct different index files which are necessary for the search step. Given a file containing several *RNA sequence-structure patterns* (RSSPs) all corresponding to a certain RNA sequence, the user can perform a search in a precomputed index after similar RNA sequences. A RSSP always consists of three lines, providing the description, the sequence itself and the secondary structure in dot-bracket notation. For example see Fig. 5.

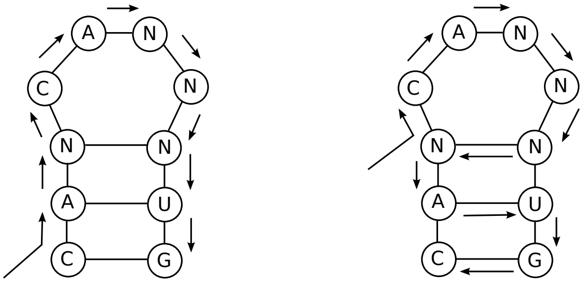


Fig. 1. (left) Outline of an unidirectional search of a hairpin loop. At the beginning, the first half of the stem region is visited, followed by the loop region and the other half of the stem region. (right) Outline of a bidirectional search step. At the beginning, the loop region is visited, followed by the base pairs consecutively from top to bottom.

The search for a particular RNA sequence in a precomputed index is performed as follows:

1. In a first step, the sequence has to be broken down into several RSSPs each corresponding to a non-branching structure, i.e. hairpin loops. Afterwards, the computed RSSPs are searched within the index, as follows:
   (a) Firstly, the non-pairing region is searched by an unidirectional search.
   (b) Secondly, if the sequence of the non-pairing region could be located somewhere in the target sequences, the existence of the pairing region, given by the structural sequence, is checked by a bidirectional search. This means in detail, that the located pattern is extended step by step by taking two new bases at each boundary into account. The two new bases are checked instantly whether they are complementary, which guarantees a fast detection of non-pairing bases in the stem region. Hence, in

practice this method is far more efficient than first searching one half of the pairing region at the beginning of the sequence and afterwards checking whether the other half of the pairing region at the end of the sequence is complementary.

2. In a second step, a chaining algorithm is looking for a valid sequence of the located non-branching structures and reports each valid sequence as a hit.

For our work only the application of the bidirectional search is of interest. Readers that are interested in a more detailed description of the whole search algorithm are referred to the original literature [13]. When applying a bidirectional search, a switch of the search direction is necessary which is done via affix links. To illustrate the application of affix links, we first have to introduce some formal definitions.

3 Formal Definitions

Note that we keep all definitions presented in this section close to the definitions of the original literature [13]. Let $\mathcal{A}$ be the alphabet of the target sequences and let $\$ \notin \mathcal{A}$ be a special character which is lexicographically greater than each character of $\mathcal{A}$. We denote the concatenated forward and backward target sequences by $\mathcal{S}^{\mathrm{F}}$ and $\mathcal{S}^{\mathrm{R}}$, respectively. In the following, the symbol $\mathrm{X} \in \{\mathrm{F}, \mathrm{R}\}$ is used for definitions which exist for $\mathcal{S}^{\mathrm{F}}$ and $\mathcal{S}^{\mathrm{R}}$. Furthermore, if $\mathrm{X} = \mathrm{F}$ holds, $\overline{\mathrm{X}}$ denotes R and vice versa.

The affix array consists of the following three different tables: $\mathrm{suf}_{\mathrm{X}}$, $\mathrm{lcp}_{\mathrm{X}}$, and $\mathrm{aflk}_{\mathrm{X}}$. Let us first have a closer look to the space consumption of each table. The first table $\mathrm{suf}_{\mathrm{X}}$ denotes the suffix array for $\mathcal{S}^{\mathrm{X}}$ storing the lexicographical order of each suffix in $\mathcal{S}^{\mathrm{X}}\$$. The second one, $\mathrm{lcp}_{\mathrm{X}}$, stores the longest common prefix between two neighboring suffixes in the suffix array. In detail, $\mathrm{lcp}_{\mathrm{X}}[0] = 0$ and $\mathrm{lcp}_{\mathrm{X}}[i]$ is the length of the longest common prefix of both suffixes starting at position $\mathrm{suf}_{\mathrm{X}}[i]$ and $\mathrm{suf}_{\mathrm{X}}[i-1]$. Both tables, $\mathrm{suf}_{\mathrm{X}}$ and $\mathrm{lcp}_{\mathrm{X}}$, can be stored in $O(n)$ time and space [7, 8, 9, 10]. Whereas $\mathrm{suf}_{\mathrm{X}}$, however, stores all values from 0 up to $|\mathcal{S}^{\mathrm{X}}|$, $\mathrm{lcp}_{\mathrm{X}}$ contains a huge number of small ($\leq$ 1 byte) values. Large entries ($>$ 1 byte), occurring rather seldom, are stored in an extra table. Hence, in practice $\mathrm{suf}_{\mathrm{X}}$ needs 4 bytes per entry and $\mathrm{lcp}_{\mathrm{X}}$ only close to 1 byte per entry [1]. Like the suffix array $\mathrm{suf}_{\mathrm{X}}$, the affix links inside of the affix array table $\mathrm{aflk}_{\mathrm{X}}$ are values from 0 up to $|\mathcal{S}^{\mathrm{X}}|$. Thus, its space consumptions equals the space consumption of the suffix array and adds up to 4 bytes per entry. Now we can summarize the total space needed for the affix array by summing up the space consumption of each table. This is $2 * (4 + 4 + 1) = 18$ bytes per entry.

An affix link is based on the following definitions [15]. An interval $[i..j]$, $0 \leq i \leq j \leq |\mathcal{S}^{\mathrm{X}}|$, in $\mathrm{suf}_{\mathrm{X}}$ is called a *ℓ-suffix-interval*, iff the following conditions hold:

- $\mathrm{lcp}_{\mathrm{X}}[i] < \ell$
- $\mathrm{lcp}_{\mathrm{X}}[j+1] < \ell$
- $\mathrm{lcp}_{\mathrm{X}}[k] \geq \ell$ for all $k \in \{i+1, \ldots, j\}$

A suffix interval $[i..j]$ is called *lcp-interval*, denoted by ℓ-$[i..j]$, iff $i < j$ and $\exists k \in \{i+1,\ldots,j\}$ with $\mathrm{lcp}_\mathrm{X}[k] = \ell$. The common prefix of an lcp-interval is defined as $\delta_\mathrm{X}(\ell\text{-}[i..j])$. An affix interval $v = \langle k, q, \mathrm{X}\rangle$ consists of three elements: an integer k designated as the context of the interval, a ℓ-suffix-interval q and the reading direction X. Each affix interval describes a set of words $\omega_\mathrm{X}(v)$ that is $\{\mathcal{S}^\mathrm{X}[\mathrm{suf}_\mathrm{X}[i]+k..\mathrm{suf}_\mathrm{X}[i]+\ell-1],\ldots,\mathcal{S}^\mathrm{X}[\mathrm{suf}_\mathrm{X}[j]+k..\mathrm{suf}_\mathrm{X}[j]+\ell-1]\}$, where $\mathcal{S}^\mathrm{X}[i..j]$ with $i < j$ denotes the substring of $\mathcal{S}^\mathrm{X}$ starting at position i and ending at position j.

With the definitions above, we can give a definition of an affix link, as follows: Given an affix interval $v = \langle k, \ell\text{-}[i..j], \mathrm{X}\rangle$ the corresponding affix link is the affix interval $v' = \langle k', \ell'\text{-}[i'..j'], \overline{\mathrm{X}}\rangle$ in $\mathrm{suf}_{\overline{\mathrm{X}}}$, for which $\omega_\mathrm{X}(v)^{-1} = \omega_{\overline{\mathrm{X}}}(v')$ holds. For example have a look at Tab. 1. Here, the affix link for $v = \langle 0, 5\text{-}[8..10], \mathrm{F}\rangle$ is $v' = \langle 0, \ell'\text{-}[4..6], \mathrm{R}\rangle$ and $\omega_\mathrm{F}(v)^{-1} = \omega_\mathrm{R}(v') = \mathrm{CGUCG}$ holds.

4 Modifications to Structator

We modified the bidirectional search step of the algorithm such that each affix link is computed instantly. This means that for our methods both affix link arrays, aflk_F and aflk_R, of size 4 bytes per entry can be omitted in the index structure what improves its space consumption.

A switch of a search direction from X to $\overline{\mathrm{X}}$ for an affix interval $v\langle k, \ell\text{-}[i..j], \mathrm{X}\rangle$ is necessary, if the interval has more than one entry, i. e. $i \neq j$, and v has an empty context, i. e. $k = 0$. For a switch of a search direction the affix interval $v' = \langle k', \ell'\text{-}[i'..j'], \overline{\mathrm{X}}\rangle$ in $\mathrm{suf}_{\overline{\mathrm{X}}}$, for which $\omega_\mathrm{X}(v)^{-1} = \omega_{\overline{\mathrm{X}}}(v')$ holds, has to be located. In an affix array this interval v' can simply be derived directly from the affix link array. Omitting aflk_X the affix interval v' has to be computed through an extra step. In the following Sect. 4.1 and 4.2, we present two such methods performing this step.

4.1 Computation of Affix Links via Binary Search

Given the affix interval $v = \langle k, \ell\text{-}[i..j], \mathrm{X}\rangle$, the affix interval $v' = \langle k', \ell'\text{-}[i'..j'], \overline{\mathrm{X}}\rangle$, for which $\omega_\mathrm{X}(v)^{-1} = \omega_{\overline{\mathrm{X}}}(v')$ holds, is computed as follows:

1. Firstly, the pattern $p = \omega_\mathrm{X}(v)^{-1}$ is computed.
2. Secondly, the *lcp-interval* ℓ-$[i..j]$ with $\delta_{\overline{\mathrm{X}}}(\ell\text{-}[i..j]) = p$ is computed via a simple binary search in $\mathrm{suf}_{\overline{\mathrm{X}}}$.

Since the run time for a binary search of a pattern of size m in a set of size n is $O(m \log n)$, the theoretical run time of this method is $O(|\omega_\mathrm{X}(v)^{-1}| \log |\mathcal{S}^\mathrm{X}|)$. The advantage of this method is that no precomputation has to be done. This means both affix link arrays, each of size 4 bytes per entry, can be omitted. Thus, the precomputed index structure only consists of the arrays suf_X and lcp_X what improves the space factor from 18 bytes per entry to 10 bytes per entry!

4.2 Computation of Affix Links via Improved Binary Search

To accelerate the pattern search for $\omega_{\mathrm{X}}(v)^{-1}$, we implemented an improved version of the binary search algorithm first published by Manber and Myers [14]. This improved version uses the lcp-values of the current interval-borders to decide whether the search direction within the suffix array $\mathrm{suf}_{\mathrm{X}}$ turns left or right. To ensure a constant time computation of all these lcp-values, an lcp-tree array, denoted by $\mathrm{lcp}_{\mathrm{X}}^{\mathrm{tree}}$, is added to the index structure. Since its values are less than or equal to the values within the lcp array $\mathrm{lcp}_{\mathrm{X}}$, its space consumption is close to 1 byte per entry. Again, as for the lcp array $\mathrm{lcp}_{\mathrm{X}}$, large values of size > 1 byte are stored in an extra array. To receive an impression of the values which have to be computed for $\mathrm{lcp}_{\mathrm{X}}^{\mathrm{tree}}$, in the following an outline of the improved binary search algorithm is given (cf. Algorithm 1).

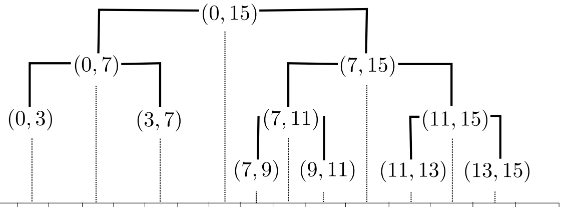
```
Data:
target sequence $t$,
suffix array $\mathrm{suf}_{\mathrm{X}}$ on $t$,
pattern $p$;

Initialization:
$L = 0, R = |t|, M = (L + R)/2$,
$l = 0, r = 0$;

if $l = r$ then                                  /* Case 1 */
    $k := l + 1$;
    while $t^{\mathrm{suf}_{\mathrm{X}}[M]+k} = p_k$ do
        $k++$;
    if $p_k > t^{\mathrm{suf}_{\mathrm{X}}[M]+k}$ then
        $L := M,\ l := k - 1$;
    else
        $R := M,\ r := k - 1$;

if $l > r$ then                                  /* Case 2 */
    if $\mathrm{lcp(L, M)} > l$ then
        $L := M$;
    if $\mathrm{lcp(L, M)} = l$ then
        $k := l + 1$;
        while $t^{\mathrm{suf}_{\mathrm{X}}[M]+k} = p_k$ do
            $k++$;
        if $p_k > t^{\mathrm{suf}_{\mathrm{X}}[M]+k}$ then
            $L := M,\ l := k - 1$;
        else
            $R := M,\ r := k - 1$;
    if $\mathrm{lcp(L, M)} < l$ then
        $R := M,\ r := \mathrm{lcp(L, M)}$;

if $l < r$ then                                  /* Case 3 */
    symmetric to Case 2;
```

Algorithm 1. The improved binary search algorithm: it is performed until pattern p is found ($l = |p| \vee r = |p|$) or not ($L > R$).

Computing $\mathrm{lcp(L, M)}$ (cf. Algorithm 1) each time by comparing both sequences, $\mathrm{suf}_{\mathrm{X}}[L]$ and $\mathrm{suf}_{\mathrm{X}}[M]$, would not improve the run time in respect of a normal binary search. However, by adding these lcp-values to the index structure, these values can be identified by a simple look up and thus, its computation only needs constant time. The idea of this precomputation step is based on the awareness that each binary search step frequents a predictable set of intervals of size $|\mathcal{S}^{\mathrm{X}}|$. For example, each search for a certain pattern p always starts with the interval $[0, |\mathcal{S}^{\mathrm{X}}|]$, followed by either $[0, |\mathcal{S}^{\mathrm{X}}|/2]$ (if $p < \mathrm{suf}_{\mathrm{X}}[|\mathcal{S}^{\mathrm{X}}|/2]$) or $[|\mathcal{S}^{\mathrm{X}}|/2, |\mathcal{S}^{\mathrm{X}}|]$ (if $p > \mathrm{suf}_{\mathrm{X}}[|\mathcal{S}^{\mathrm{X}}|/2]$) and so on. For each of those interval-borders (L, R) we can precompute its lcp-value once and store it under a particular index k in $\mathrm{lcp}_{\mathrm{X}}^{\mathrm{tree}}$, which is computed as follows:

$$k = L + \lfloor (L+R)/2 \rfloor \,.$$

In Fig. 2 an outline of $\text{lcp}_\text{F}^\text{tree}$ corresponding to the example given in Tab. 1 is depicted.



$\text{lcp}_\text{F}^\text{tree}$:	–	0	–	0	–	0	–	0	2	0	0	0	1	0	0	–
index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Fig. 2. The array $\text{lcp}_\text{F}^\text{tree}$ corresponding to the example given in Tab. 1

Now, with the help of the precomputed array $\text{lcp}_\text{X}^\text{tree}$, we can compute the lcp-value of an interval-border (L, R) in constant time. Hence, considering the run time of the first method, this second method improves the search for the pattern $\omega_\text{X}(v)^{-1}$ from $O(|\omega_\text{X}(v)^{-1}| \log |\mathcal{S}^\text{X}|)$ to $O(|\omega_\text{X}(v)^{-1}| + \log |\mathcal{S}^\text{X}|)$ [14]. However, due to the additional array $\text{lcp}_\text{X}^\text{tree}$, this second method needs 2 bytes per entry more than the first method — what is still only 2/3 of the space the corresponding affix array would need!

5 Experiments

To show the practical advantage of our two modifications, we applied our modified Structator program (version 1.01) to the Rfam database release 10.0. The Fasta format of this database has a size of 676.9 MB and contains around 3.2 million sequences. As search patterns we used 397 *RNA sequence-structure patterns* (RSSPs) for 42 highly structured Rfam 10 families available from www.zbh.uni-hamburg.de/Structator. An example of a RSSP is given in Fig. 5. All experiments presented in this section are performed on the same linux system providing two 2.4 GHz dual cores and 4 GB RAM.

5.1 Four Different Modes

Our modified program can be run in four different modes. Each mode precomputes a different index. While the first two modes are already present in version 1.01, the last two modes correspond to Sect. 4 and were implemented by us.

- **Mode *aflk+skp*** precomputes an index consisting of suf_X, lcp_X, and aflk_X. Due to a better run time, the affix link array aflk_X is computed with the help of the array skp_X [2] which is similar to the concept of a child table [3][1]. In the final index skp_X is useless and, thus, is deleted. Nevertheless, during the index computation additional space with 4 per entry for skp_X has to be allocated which makes in total 26 bytes per entry for the whole index computation.
- **Mode *aflk*** precomputes an index consisting of suf_X, lcp_X, and aflk_X. This mode omits the computation of skp_X and the affix link array aflk_X is computed via a simple binary search. Hence, only 18 Bytes per entry are necessary for the index computation.
- **Mode *bs*** precomputes an index consisting of suf_X and lcp_X. This mode corresponds to the modification presented in Sect. 4.1. Its precomputed index structure only needs 10 bytes per entry.
- **Mode *ibs*** precomputes an index consisting of suf_X, lcp_X, and $\text{lcp}_\text{X}^\text{tree}$. This mode corresponds to the modification presented in Sect. 4.2. Its precomputed index structure only needs 12 bytes per entry.

The first row of Tab. 2 shows the theoretical speedup factor of the index memory for each mode. Note that, due to the extra lcp array storing values > 1 byte, the space consumption of lcp_X is in general more than 1 byte per entry. Furthermore, Structator stores both sequences, $\mathcal{S}^\text{F}$ and $\mathcal{S}^\text{R}$, plus some additional information like the end positions and the length of the target sequences for $\mathcal{S}^\text{X}$. Hence, the listed theoretical speedup factors are upper bounds. We are aware of the fact that some of this additional information is not required in theory and, thus, could either be omitted or be replaced by smaller data structures, which would even more reduce the total space for all modes.

5.2 1st Experiment—Memory vs. Runtime

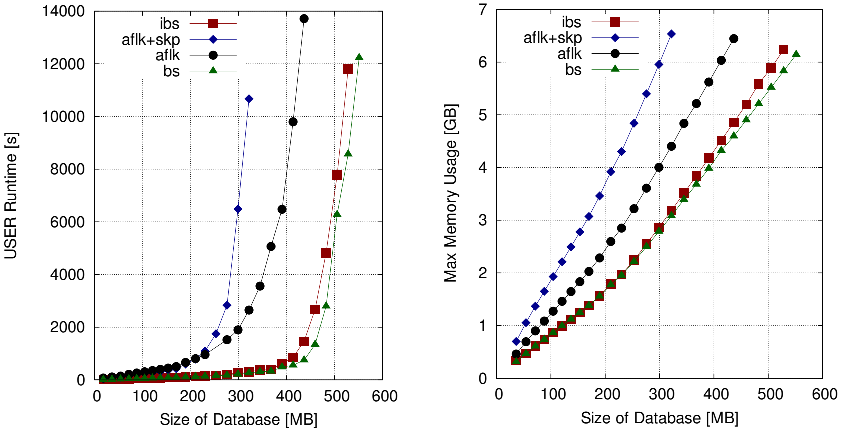
Our first experiment computes an index of an increasing part of a database and performs a sequential search of the 397 RSSPs within this index. Therefore, each part consists of the first k lines of the whole Rfam database release 10.0. To minimize the influence of disk subsystem performance, we conduct each search step ten times. Due to large caching effects, the results of the first search step are omitted whereas for each result of the other nine search steps the corresponding mean is taken. During this experiment the run time (cf. Fig. 3(a)) and maximal user memory of the index computation (cf. Fig. 3(b)) as well as the run time of the pattern search (cf. Fig. 4) is detected. Note that we limited the maximal virtual memory available for each process performing the index construction and the pattern search by 6,552,880 kB.

In Tab. 2 the detected mean speedup factors corresponding to the different database sizes for each mode are listed.

[1] A more space efficient version has been developed [4], but has not yet been integrated into Structator.

Table 2. Theoretical and detected speedup factors for each mode

Speedup Factors	aflk+skp	aflk	ibs	bs
Theoretical Index Memory	2.6	1.8	1.2	1.0
Detected User Memory	2.2098	1.4630	1.0127	1.0000
User Runtime of Index Construction	5.6010	6.9930	1.0866	1.0000
User Runtime of RSSP Search	1.0000	1.0000	2.2485	2.6089



(a) User run time of each index construction for the different databases.

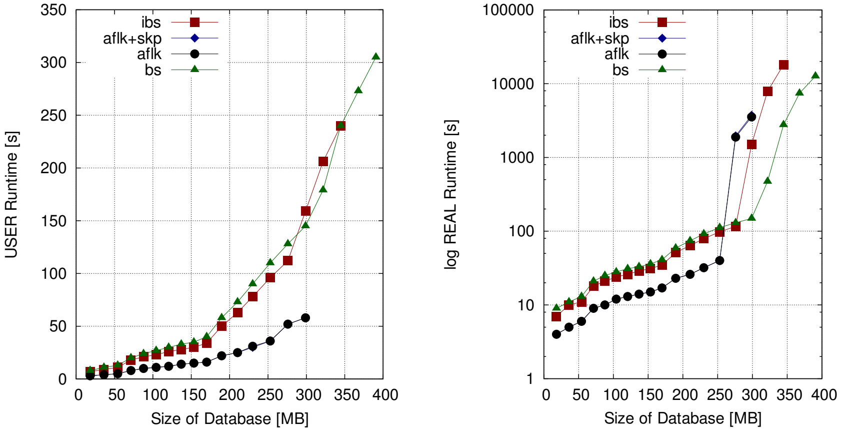
(b) Maximal user memory which is needed for the index construction.

Fig. 3. Our experimental results for the index construction

Figure 3(a) and Tab. 2 show that the index computation performed by our methods, mode *bs* and *ibs*, are a lot faster than the other two modes, *aflk+skp* and *aflk*, computing the whole affix array. Since our implemented methods omit the complex computation of aflk$_{\mathrm{X}}$, this is not surprising.

Figure 3(b) shows the maximal user memory (detected via *ps -o size PID*) that is needed for the index computation of each mode. While the computation via mode *aflk+skp* and *aflk* fails at a database size greater than 321 MB and 436 MB, respectively, our modes can compute an index for a database size up to 528 MB and 551 MB, respectively.

In Fig. 4(a) the mean *user* run time of the last nine search steps within databases of different sizes is depicted. Since the index computed by mode *aflk+skp* and *aflk* provides an affix link array, each affix link can be computed in constant time. As described in Sect. 4, our two modifications compute each affix link via a binary search, which takes some extra time. Thus, a search performed through an index constructed via mode *bs* or *ibs* is slower than the search performed through an index constructed via mode *aflk+skp* or *aflk*. However, due to the additional space of aflk$_{\mathrm{X}}$, the search performed by the two modes *aflk+skp*



(a) Mean *user* run time of the search for the 397 RSSPs.

(b) Mean *real* run time in *logscale* of the search for the 397 RSSPs.

Fig. 4. Our experimental results for the search for the 397 RSSPs

and *aflk* both fail at a database size greater than 298 MB, whereas our two modes are able to perform the search in a database up to a size of 344 MB and 390 MB respectively.

Figure 4(b) shows the *real* mean run time in log scale of the last nine search steps for the 397 RSSPs. Whereas the user run time is the number of CPU-seconds that the process used directly in user mode, the real run time is the whole time elapsed by the process. This means that some extra time produced by page faults is also taken into account. Due to such a high number of page faults, the real run time of the two modes, *aflk* and *aflk+skp*, exceeds the real run time of the other two modes, *bs* and *ibs*, at a particular database size. For our experiment this is the case for a database size of 275 MB. This means that on our system for such databases having a size greater than or equal to 275 MB mode *bs* and mode *ibs* provide a better performance in terms of time and space! Note that the index files, which are constructed by mode *aflk* or *aflk-skp* for a database size of 275 MB, have a size of ≈ 4 GB, which is the maximal memory space provided by our system. This means when performing a search for larger databases the index files are not kept in memory in total and, thus, the system has to handle a large number of page faults.

5.3 2nd Experiment—Mode IBS vs. Mode BS

Considering Fig. 4(a), there is no big difference between the run time concerning the pattern search of mode *ibs* and *bs*. To figure out the reason, we checked the total number of character comparisons of each search for an affix link. Due to the precomputed array $\mathrm{lcp}_{\mathrm{X}}^{\mathrm{tree}}$, our improved binary search algorithm (cf. Algorithm 1) in general has to perform less character comparisons than the common binary search algorithm.

When searching for all 397 RSSPs, however, we assumed that the additional number of character comparisons, which the common binary search has to perform, is too low to produce a significant difference between both run times. To figure out, if a larger difference between the number of character comparisons has an effect on both run times, we divided the whole set of RSSPs into subsets each containing different numbers of copies of one particular RSSP. For each of those copies we performed a search in a Rfam database of size 104 MB. The result confirmed our assumption — the more additional comparisons have to be performed by mode *bs* the higher the run time compared to the one of mode *ibs*. Most of the 397 RSSPs, however, produces only a few number of such additional character comparisons. Nevertheless, there is one RSSP (depicted in Fig. 5) producing an exceptional high number of such additional comparisons.

```
N C G N N G N U C N G N N N N N N N
( . ( ( . . . . . . . . . . ) ) . )
```

Fig. 5. RSSP producing a lot of additional character comparisons for mode *bs*

The number of additional comparisons obviously depends on the number of affix links that have to be computed during all bidirectional search steps. The number of computed affix links, again, depends on the number of patterns that are found during the unidirectional search for the hairpin loop region. In general you can say that the higher the number of hairpin loop regions within the database the more affix link have to be computed and, thus, the better the run time of mode *ibs* in respect to mode *bs* gets.

6 Conclusion

In Sect. 4 we present two space efficient modifications to Structator allowing the user to perform searches on large databases. Our experiments, presented in Sec. 5, show that these two modifications outperform the existing methods in time and space, if the constructed index, which is necessary for the search step, exceeds the size of the system's memory. Our modifications benefit from a smaller search index producing less page faults during each search step. Thus, we think that our implemented modifications are a necessary extension to the existing program and will make the program more valuable for scientists. Further improvements concerning the space consumption could be applied by only storing one of both sequences $\mathcal{S}^{\mathrm{F}}$ and $\mathcal{S}^{\mathrm{R}}$ as well as reducing the additional information, mentioned in Sec. 5.1, for $\mathcal{S}^{\mathrm{X}}$, at no extra costs.

Availability

The source code under the GNU General Public License Version 3 of our modified Structator version as well as a short description is available.
See http://www.bio.ifi.lmu.de/mitarbeiter/benjamin-albrecht.

Acknowledgment. We want to thank the authors of Structator for its implementation and the publishing of the source code. Special thanks goes to Fernando Meyer for the fast and detailed answering of our questions!

References

[1] Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. J. Discrete Algorithm 2, 53–86 (2004)

[2] Beckstette, M., Homann, R., Giegerich, R., Kurtz, S.: Fast index based algorithms and software for matching position specific scoring matrices. BMC Bioinformatics 7, 389 (2006)

[3] Fischer, J.: Combined Data Structure for Previous- and Next-Smaller-Values. Theor. Comput. Sci. 412(22), 2451–2456 (2011)

[4] Fischer, J., Heun, V.: Space-Efficient Preprocessing Schemes for Range Minimum Queries on Static Arrays. SIAM J. Comput. 40(2), 465–492 (2011)

[5] Gardner, P.P., Wilm, A., Washietl, S.: A benchmark of multiple sequence alignment programs upon structural RNAs. Nucleic Acids Res. 33, 2433–2439 (2005)

[6] Gardner, P.P., Daub, J., Tate, J., Moore, B.L., Osuch, I.H., Griffiths-Jones, S., Finn, R.D., Nawrocki, E.P., Kolbe, D.L., Eddy, S.R., Bateman, A.: Rfam: Wikipedia, clans and the ”decimal” release. Nucleic Acids Res. 39, D141–D145 (2011)

[7] Kärkkäinen, J., Sanders, P.: Simple Linear Work Suffix Array Construction. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 943–955. Springer, Heidelberg (2003)

[8] Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In: Amir, A., Landau, G.M. (eds.) CPM 2001. LNCS, vol. 2089, pp. 181–192. Springer, Heidelberg (2001)

[9] Kim, D.-K., Sim, J.S., Park, H.-J., Park, K.: Linear-Time Construction of Suffix Arrays. In: Baeza-Yates, R., Chávez, E., Crochemore, M. (eds.) CPM 2003. LNCS, vol. 2676, pp. 186–199. Springer, Heidelberg (2003)

[10] Ko, P., Aluru, S.: Space Efficient Linear Time Construction of Suffix Arrays. In: Baeza-Yates, R., Chávez, E., Crochemore, M. (eds.) CPM 2003. LNCS, vol. 2676, pp. 200–210. Springer, Heidelberg (2003)

[11] Mattick, J.S.: RNA regulation: a new genetics? Nat. Rev. Genet. 5, 316–323 (2004)

[12] Mattick, J.S., Taft, R.J., Faulkner, G.J.: A global view of genomic information-moving beyond the gene and the master regulator. Trends Genet. 26, 21–28 (2010)

[13] Meyer, F., Kurtz, S., Backofen, R., Will, S., Beckstette, M.: Structator: fast index-based search for RNA sequence-structure patterns. BMC Bioinformatics 12, 214 (2011)

[14] Myers, U., Manber, G.: Suffix arrays: a new method for on-line string searches. SIAM J. Comput. 22, 935–948 (1993)

[15] Strothmann, D.: The affix array data structure and its applications to RNA secondary structure analysis. Theor. Comput. Sci. 389(1-2), 278–294 (2007)

Implementation and Comparison of Heuristics for the Vertex Cover Problem on Huge Graphs*

Eric Angel[1], Romain Campigotto[2], and Christian Laforest[3]

[1] Laboratoire IBISC, EA 4526 – Université d'Évry-Val d'Essonne
IBGBI, 23 boulevard de France, 91 037 Évry Cedex, France
eric.angel@ibisc.univ-evry.fr

[2] LAMSADE, CNRS UMR 7243 – Université Paris-Dauphine
Place du Maréchal de Lattre de Tassigny, 75 775 Paris Cedex 16, France
romain.campigotto@lamsade.dauphine.fr

[3] LIMOS, CNRS UMR 6158 – Université Blaise Pascal, Clermont-Ferrand
Campus des Cézeaux, 24 avenue des Landais, 63 173 Aubière Cedex, France
christian.laforest@isima.fr

Abstract. We present in this paper an experimental study of six heuristics for a well-studied **NP**-complete graph problem: the VERTEX COVER. These algorithms are adapted to process huge graphs. Indeed, executed on a current laptop computer, they offer *reasonable* CPU running times (between twenty seconds and eight hours) on graphs for which sizes are between $200 \cdot 10^6$ and $100 \cdot 10^9$ vertices and edges.

We have run algorithms on specific graph families (we propose generators) and also on random power law graphs. Some of these heuristics can produce good solutions. We give here a comparison and an analysis of results obtained on several instances, in terms of quality of solutions and complexity, including running times.

Keywords: implementation of algorithms, experimental analysis, huge graphs, low memory, vertex cover.

1 Introduction

The VERTEX COVER problem [14] is a well-known classical **NP**-complete optimization graph problem that has received a particular attention these last decades. In particular, it occurs in many concrete situations [17], in fields such as biology, meteorology, finance, etc. where amount of data is more and more important. This leads to the problem of designing algorithms well suited to cope with such large instances.

Notations. Graphs $G = (V, E)$ considered throughout this paper are undirected, simple, unweighted and represent the *instance* to be treated here. We denote by n the number of vertices ($n = |V|$) and by m the number of edges ($m = |E|$). For any vertex $u \in V$, we denote by $N(u) = \{v \mid uv \in E\}$ the set of *neighbors* of u and we call *degree* the number of neighbors of vertex u.

* Work partially supported by the French Agency for Research under the DEFIS program *TODO*, ANR-09-EMER-010.

R. Klasing (Ed.): SEA 2012, LNCS 7276, pp. 39–50, 2012.

Definition of the Vertex Cover Problem. A *cover* C of G is a subset of vertices such that every edge contains (or *is covered by*) at least one vertex of C, that is $C \subseteq V$ and $\forall e = uv \in E$, one has $u \in C$ or $v \in C$ (or both). The VERTEX COVER problem is to find a cover of minimum size. We denote by OPT the size of an optimal cover for a given graph.

Related Work. Several studies focused on massive data sets these last decades [3]. In particular, for the MAX CLIQUE problem [9], experiments on graphs with $53 \cdot 10^6$ vertices and $170 \cdot 10^6$ edges have been performed [2]. But no such study has been done for the VERTEX COVER problem. However, it has been extensively studied theoretically: many exact (exponential) algorithms, approximation algorithms, online algorithms, etc. have been proposed (due to space limitations, we do not give references about these works: some of them can be found in the introductions of [8] and [13]). Several experimental studies have already been made, often to compare the quality of several algorithms [12,15] or validate specific methods [5,7]. Nevertheless, no one achieved the huge graph sizes we consider in this paper (in these studies, the largest graphs has 10,000 vertices).

Our General Model of Treatment. To the intrinsic **NP**-completeness is added the difficulty to manipulate graphs and run algorithms with severe constraints. Indeed, with respect to their huge sizes, the processing unit (we consider a standard computer) cannot load them entirely in its memory. Moreover, the graph, which is stored on an external disk, must not be modified, since it often comes from experimentations and can be used by different users for different goals. Specifically, the important cost of graph creation forces us to preserve its *integrity*, in order to be able to run several algorithms on it.

Organization of the Paper. We give in the next section a general description of our experiments. We present, analyze and compare in Sect. 3 results obtained by executing the six algorithms on several instances. Finally, in Sect. 4, we conclude and give some perspectives.

2 General Description

In this section, we describe elements used for experiments. Programs (executables, with source code) are available at [1].

Technical Characteristics. The "Processing Unit" is a laptop computer with a Dual Core processor running at 2.8 GHz, 6 Mb cache memory and 4 Gb RAM. Graphs and Covers are stored on the same external hard drive, which is a USB 2.0 hard disk of 2 Tb, running at 7200 revolutions/minute and equipped with 8 Mb cache memory. Programs are written in language C, C99 standard, in order to use specific data type `unsigned long long` and associated functions to read and write binary files.

Storage and Reading of Graphs. There exist many ways to store a graph: with an adjacency matrix, an adjacency list, etc. We use the method described in [4] (for more details, report to the Sect. 3.2 page 20). More precisely, our graphs are stored with two files:

`.list` file which contains $2m + 1$ values: the number of vertices in the graph (which is needed by several algorithms to create an n bits array) and the list of the neighbors of vertices in the graph;

`.deg` file which contains $n+1$ values, which are needed to access to the neighbors of a vertex and compute its degree.

The n vertices are labeled from 0 to $n-1$. The `.list` file contains first the value n, then vertices of set $N(0)$, then vertices of set $N(1)$, etc. (however, neighbors of each vertex can be stored in any order, not necessarily following the order of labels). The `.deg` file contains, for each vertex (and by increasing order of labels), the place of its first neighbor (in the `.list` file). It contains $n+1$ values, in order to compute the degree of the last vertex (the last value of the `.deg` file points to the end of the `.list` file). Indeed, to compute degree of vertex i, we subtract the i^{th} value from the $(i+1)^{\text{th}}$.

Figure 1 shows an example on a small graph.

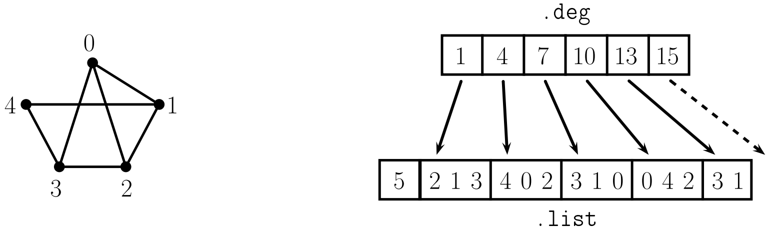


Fig. 1. Storage of a graph with 5 vertices and 7 edges

Algorithms scan graphs by reading the two files described previously. First, the `.list` file is read to know the number of vertices. Then, the `.deg` file is read to know the place of the first neighbor of the first vertex and the place of the first neighbor of the second vertex. Thereby, there are two pointers (the first one is followed by the second one) which delimit the set of neighbors of a vertex. Once these places are known, the reading process continues in the `.list` file, where neighbors of the first vertex are retrieved. If the treatment unit does not need to get all of them, it can "step over" the remaining neighbors and go immediatly to the neighbors of the second vertex. It proceeds in the same way for the following vertices.

If algorithms need to know degrees of neighbors, they read independently the `.deg` file with another playhead, which can be moved at a precise place to get the two successive values needed to compute the degree.

All the algorithms read the `.list` file in a sequential way, but they can "step over" values that are not needed. Algorithms which do not need to compute degrees of neighbors read the `.deg` file also in a sequential way.

Algorithms Implemented. We have implemented six algorithms adapted to the treatment of huge graphs: LR, ED, S-Pitt, LL, SLL and ASLL.

LR has been proposed in [11]. ED is the 2-approximative algorithm which returns vertices of a maximal matching. S-Pitt is a probabilistic algorithm inspired by the algorithm presented in [18]: it has an expected approximation ratio equal to 2. The authors have done a theoretical study of LL, SLL and ASLL in [6].

We present now a basic description of the six algorithms, by giving conditions to put vertices of the input graph into the solution.

Let $G = (V, E)$ be a graph. Let C be the cover under construction. For each vertex $u \in V$, we have

LR: if $u \notin C$, $\{v \mid uv \in E \wedge v \notin C\}$ is put in C;

ED: if $u \notin C$ and if u has a neighbor $v \notin C$, u and v are put in C;

S-Pitt: if $u \notin C$ and if u has a neighbor $v \notin C$, either u or v is put in C with equiprobability assumption;

LL: u is put in C if it has at least one neighbor v such that $v > u$ (their labels are compared);

SLL: u is put in C if it has at least one neighbor v such that $d(v) < d(u)$ or $d(v) = d(u)$ and $v > u$;

ASLL: u is put in C if it has at least one neighbor v such that $d(v) > d(u)$ or $d(v) = d(u)$ and $v < u$.

Now, we can describe how we have implemented these algorithms, in relation with the way that the graphs are stored on the external hard disk.

As described above, the algorithms scan graphs vertex by vertex and, for each current vertex u, scan its neighbors one by one. If an algorithm decides that u belongs to the solution (applying the conditions given in the descriptions of the algorithms above), u is put immediately and definitively into the cover. Then, the algorithm steps over its remaining neighbors and goes to the next vertex. Otherwise, it gets the next neighbor of u (and, at the end, requires the next vertex like in the previous case). Also, when an algorithm scans a vertex u which is already in the cover, it goes immediately to the next vertex, without scanning its neighbors.

It is worth noticing that LR, ED and S-Pitt need to allocate an n bits array to mark vertices sent to the solution (reading on the external hard drive during the execution would take too long); SLL and ASLL need to compute degrees of neighbors.

Writing the Covers on the Disk. A cover is written as a list of vertex labels into a file, which is built piece by piece: once an algorithm decides to put a vertex into the solution, it writes it into the cover file. A vertex cannot appear twice, because algorithms have been designed to produce no duplicates.

Example of Execution of Algorithm LR. We consider the graph given in Fig. 2. The execution of LR on it works as follows.

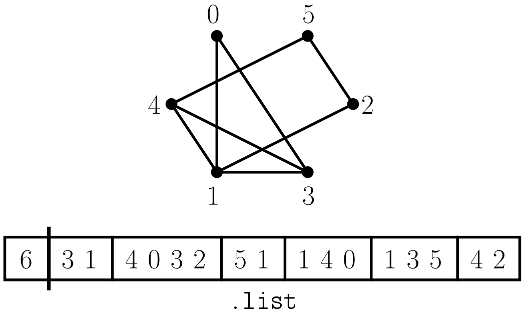


Fig. 2. Graph with 6 vertices and 8 edges

At the beginning, the cover C (which is materialized by an 6 bits array in the internal memory of the computer) is empty (i.e. all the binary flags are lowered).

1. $C = \emptyset$. We consider the vertex 0. We get its neighbor, 3: we put it in C (it is written on the disk and the corresponding flag in the memory is raised). Then, we get its neighbor, 1: we put it in C.
2. $C = \{1, 3\}$. The vertex 1 is not treated since it is already in C.
3. $C = \{1, 3\}$. We treat the vertex 2. We get its neighbor, 5: it is put in C. Then, we get its neighbor, 1: nothing is done since it is already in C.
4. $C = \{1, 3, 5\}$. The vertex 3 is not treated since it is already in C.
5. $C = \{1, 3, 5\}$. We treat the vertex 4. Its neighbors (1, 3 and 5) are retrieved but not considered, because they are already in C.
6. $C = \{1, 3, 5\}$. The vertex 5 is not treated since it is already in C.

Hence, we have just scanned seven vertices in the `.list` file (which contains sixteen labels) and the cover produced by LR contains three vertices: 1, 3 and 5.

Graph Families Used. We have executed our algorithms on different graphs:

- on sparse graphs (where $m \in \mathcal{O}(n)$): *ButterFly* graphs [21], *de Bruijn* graphs [10] and grid graphs;
- on dense graphs (where $m \in \Theta(n^2)$): hypercubes, complete bipartite graphs and *complete split graphs*[1].

[1] A *complete split graph* is a complete bipartite graph in which the vertices subset of lowest size is changed to a *clique*.

We have chosen these graphs because they can be easily generated (we can produce huge size graphs with a standard computer). In that sense, we have designed generators to construct these specific graphs. For example, CPU running times for the instances generation of size $100 \cdot 10^9$ are between 5 hours than 7 hours. Also, the size of their optimal solutions is known (we can give exact sizes, excepted for *de Bruijn* graphs where we can only give lower bounds). Thus, it is possible to present results on the quality of algorithms.

We have also chosen to execute algorithms on *random power law graphs*, where degree sequences follow a power law. We have used generator described in [19], which is based on the *Molloy and Reed* model [16].

This generator is able to produce random power law graphs, but it cannot create them in an online way: a memory space linear to the graph size is needed. However, on our computer, we can still create graphs with more than $10 \cdot 10^6$ vertices and edges, with CPU running times less than 3 hours.

Graph Sizes. The size of a graph is given by its number of vertices and edges. Hence, we denote by *graph size* the value $n + m$.

The *huge size* notion is relative: it depends on context considered (e.g. the size limits for algorithms with exponential complexity are lower than for algorithms with linear complexity). So, we have defined several levels for graph sizes.

1st level. The graphs size is about $200 \cdot 10^6$ (several Gb on disk). This is the largest random power law graphs size that can be generated on our computer.

2nd level. The graphs size is about $30 \cdot 10^9$ (more than 100 Gb on disk). The algorithms SLL and ASLL begin to reach their limits in terms of running times on our computer.

3rd level. The graphs size is about $100 \cdot 10^9$ (around 1.5 Tb on disk). With our computer, we cannot allocate an n bits array if the number of vertices is bigger than $30 \cdot 10^9$.

Experimentations. For the first level, we have executed algorithms S-Pitt, LL, SLL and ASLL five times on each instance (algorithms LR and ED are deterministics). From second level, we simulated a user having limited resources (time and disk space). So, for each graph, we have executed the six algorithms once. Based on results obtained and resources already spent, we have executed again several algorithms (often one time).

A total of thirty-four executions was made: thirteen in the first level, six in the second level and two in the third level. We have also executed our algorithms on thirteen instances for which sizes are between 10^6 and $4 \cdot 10^6$. But, due to space limitations, we do not give details on results obtained for these instances. We can however indicate that they are broadly similar to the results obtained in the first level.

3 Results and Observations

Evaluated Criteria. We have focused on quality of solutions produced by algorithms and complexities, expressed by the *number of requests* made to the instance, i.e. the number of neighbors read in the `.list` file. We have also considered running times. For that, we have used the UNIX command `/usr/bin/time`, which gives the time used by the processor during a program execution.

Results Presentation. We give one table per criterion. For algorithms that have been executed more than once, we give values corresponding to the best solution (in terms of quality).

For each instance, the best value (among the set of values presented for the six algorithms) is in bold font. Conversely, when values are bad (e.g. an algorithm which returns almost all the vertices or performs more than m requests), numbers are in italics font.

Table 1 (resp. 2) gives quality of solutions (resp. complexity in number of requests) obtained in the first level on graphs created by our generators and on random graphs. For random power law graphs, the last digit given in the instance name (starting by `rg`) denotes the minimum degree of the graph.

Table 1. Quality of solutions obtained in the first level, expressed in percentage of n (sizes of optimal covers for random power law graphs cannot be estimated)

Instance	n	*OPT*	LR	ED	S-Pitt	LL	SLL	ASLL
`butterfly-21`	46,137,344	50	**50**	*100*	78.16	90.91	85.06	90.91
`debruijn-25`	33,554,432	> 50	**66.67**	88.89	77.56	66.78	72.71	82.39
`grid-6000.9000`	54,000,000	50	**50**	*99.99*	81.41	*99.99*	*99.99*	*99.99*
`hypercube-23`	8,388,608	50	**50**	*99.97*	99.26	*99.99*	*99.99*	*99.99*
`compbip-7000.15000`	22,000	31.82	68.18	63.64	62.98	**31.82**	**31.82**	68.18
`split-7500.12000`	19,500	38.46	*99.99*	61.67	61.39	**38.46**	**38.46**	*99.99*
`rg-20m_1`	20,000,000	–	**9.94**	19.65	13.41	49.30	10.42	*99.99*
`rg-20m_2`	20,000,000	–	**34.12**	62.89	45.60	49.30	36.89	*99.88*
`rg-25m_1`	25,000,000	–	**14.19**	28.09	18.91	41	14.89	*99.95*
`rg-25m_2`	25,000,000	–	**38.76**	69.99	51.29	48.13	42.12	*99.65*
`rg-30m_1`	30,000,000	–	**43.48**	76.61	57.17	59.02	47.44	*97.57*
`rg-30m_2`	30,000,000	–	**15.68**	31.05	21.10	30.98	16.46	*99.93*
`rg-35m_2`	35,000,000	–	**43.12**	76.16	56.79	53.70	46.98	*99.12*

For the first level, expected running times of each algorithm are between twenty seconds and two minutes.

Tables 3 and 4 give respectively quality of solutions and number of requests obtained for the second and third levels. Table 5 gives CPU running times. For the third level, we only give values for algorithms that have been executed until the end.

Table 2. Number of requests performed in the first level (in percentage of m)

Instance	m	LR	ED	S-Pitt	LL	SLL	ASLL
butterfly-21	88,080,384	*100*	**60.51**	91.94	*103.30*	*104.76*	80.16
debruijn-25	67,108,861	66.67	**61.11**	87.03	*113.87*	*106.84*	*100.48*
grid-6000.9000	107,985,000	*100*	**49.93**	86.24	91.66	91.66	75.03
hypercube-23	96,468,992	*100*	**11.83**	23.57	17.54	17.58	17.51
compbip-7000.15000	105,000,000	*100*	**53.34**	54.31	*100.01*	*100.01*	*100.02*
split-7500.12000	118,121,250	**0.02**	47.47	47.82	76.20	76.20	0.17
rg-20m_1	59,624,494	49.12	47.99	55.20	**34.29**	59.22	57.73
rg-20m_2	90,808,193	40.23	36.92	48.73	38.20	56.57	**34.27**
rg-25m_1	70,911,180	45.44	44.57	53.13	**36.23**	58.19	47.08
rg-25m_2	87,837,432	45	**40.93**	55.45	51.81	65.50	41.84
rg-30m_1	82,356,722	50.09	**45.04**	62.78	57.89	75.78	52.29
rg-30m_2	81,819,916	44.86	44.02	52.84	**39.76**	58.28	45.23
rg-35m_2	96,555,269	50.14	**45.11**	62.70	62.52	75.62	53.31

Table 3. Quality of solutions obtained in the second and third levels, expressed in percentage of n

Instance	n	*OPT*	LR	ED	S-Pitt	LL	SLL	ASLL
butterfly-28	7,784,628,224	48.28	**48.28**	96.55	78.76	93.24	93.10	96.55
debruijn-33	8,589,934,592	> 50	**66.67**	88.89	77.56	96.55	*99.99*	*99.99*
grid-75000.90000	6,750,000,000	50	**50**	*99.99*	81.41	*99.99*	*99.99*	*99.99*
hypercube-30	1,073,741,824	50	**50**	*99.99*	99.78	*99.99*	*99.99*	*99.99*
compbip-35000.500000	535,000	6.54	93.46	13.08	13.11	**6.54**	**6.54**	93.46
split-70000.180000	250,000	28	*99.99*	48.19	48.01	**28**	**28**	*99.99*
butterfly-30	33,285,996,544	48.28	–	–	–	98.26	–	–
compbip-250000.380000	630,000	39.68	**60.32**	79.37	79.46	84.39	–	–

Table 4. Number of requests performed in the second and third levels, expressed in percentage of m

Instance	m	LR	ED	S-Pitt	LL	SLL	ASLL
butterfly-28	15,032,385,540	*99.99*	**61.91**	91.59	*103.50*	*101.79*	72.62
debruijn-33	17,179,869,183	66.67	**61.11**	87.03	*108.91*	*108.33*	91.67
grid-75000.90000	13,499,835,000	*100*	**49.98**	86.24	91.67	91.67	75
hypercube-30	16,106,127,360	*100*	**9.10**	18.22	13.42	13.33	13.33
compbip-35000.500000	17,500,000,000	*100*	93	**92.97**	*100.01*	*100.01*	*100.02*
split-70000.180000	15,049,965,000	**0.002**	60.24	60.45	83.72	83.72	0.02
butterfly-30	64,424,509,440	–	–	–	*102.18*	–	–
compbip-250000.380000	95,000,000,000	*100*	34.21	34.05	**25.88**	–	–

Table 5. CPU running times obtained in the second and third levels (number of executions are given in parenthesis: we give the average time here)

Instance	LR	ED	S-Pitt	LL	SLL	ASLL
`butterfly-28`	**1:11:55**	1:15:53	1:18:41 *(2)*	1:20:35 *(2)*	*5:10:47*	*3:38:43*
`debruijn-33`	**1:20:37**	1:24:10	1:26:14 *(2)*	1:29:35 *(2)*	*7:43:53*	*5:08:37*
`grid-75000.90000`	**1:02:47**	1:08:57	1:09:58 *(2)*	1:11:35 *(2)*	*3:24:52*	*3:01:32*
`hypercube-30`	0:41:06	0:33:21	0:36:02 *(3)*	**0:33:19** *(3)*	1:07:46 *(2)*	1:07:38 *(2)*
`compbip-35000.500000`	0:23:15	0:22:23	0:22:28 *(4)*	**0:22:13** *(4)*	*6:11:51*	*6:12:03*
`split-70000.180000`	**0:00:17**	0:15:21	0:15:36 *(5)*	0:16:11 *(5)*	*4:27:39*	0:00:29 *(8)*
`butterfly-30`	–	–	–	5:47:43	–	–
`compbip-250000.380000`	2:02:16	1:01:19	1:01:21	**0:32:17**	–	–

Observations on Quality of Solutions. As we can see on Tables. 1 and 3, the algorithm LR is almost always the best. Moreover, it often returns the optimal solution. However, it can be very bad on complete bipartite and split graphs.

In general, SLL offers good performance, especially on random power law graphs (its performance is close to LR). Nevertheless, it is less efficient on regular graphs[2].

The global performance of algorithms S-Pitt and LL is intermediate but, for LL, it fluctuates more than S-Pitt. Indeed, on one instance, LL can be the best or the worst, that is not the case for S-Pitt.

Finally, ED and ASLL are overall the worst algorithms (and ED reaches often its approximation ratio of 2). For ED, these results confirm observations made by *F. Delbot et al.* [12].

Observations on the Number of Requests. As we can see on Tables. 2 and 4, the algorithm ED is almost always the best. Furthermore, it always performs less than m requests.

The algorithm LR often reaches m requests (it cannot perform worse), except on instances on which it returns a bad solution (it is better on them).

The performance of S-Pitt is close to LR: it is often better on specific graphs (except on complete split graphs) but it is worse on random graphs.

Algorithms LL, SLL and ASLL can perform more than m requests. This explains the fact that LR is generally the second algorithm in terms of complexity (even if its upper bound of m requests is often reached). ASLL is better, especially on random power law graphs and complete split graphs.

Analysis of Running Times. We focus on values presented for the second level in Tab. 5 (CPU running times obtained in first level are too similar to be exploited). To obtain an estimation of real running times (observed on our computer), one can multiply by 3.2 (resp. 1.6) CPU running times given in Tab. 5 for algorithms LR, ED, S-Pitt and LL (resp. SLL and ASLL).

[2] A *regular graph* is a graph where all the vertices have the same degree.

Algorithms SLL and ASLL are different because they have to use another playhead on `.deg` file to calculate degrees of neighbors. Therefore, their CPU running times are bigger. For this reason, we focus primarily on values observed for algorithms LR, ED, S-Pitt and LL.

On sparse graphs (where n and m are similar), CPU running times are close. They depend on number of requests performed by algorithms and size of covers constructed. Indeed, the size of solutions can be as huge as n, and writing on a disk is longer than reading. Moreover, there is often a trade off between the number of requests performed and the quality of solutions constructed: algorithms which produce the best solutions often perform the biggest number of requests (in any case, on one instance, an algorithm is never both the best in terms of quality of solution and complexity).

On dense graphs (where n is negligible compared to m), the analysis is less intricate because the size of covers written is tiny compared to the number of requets performed. Thus, CPU running times are mainly influenced by the number of requests done.

But these two criteria are not sufficient to explain CPU running times observed. Another technical aspects, linked to operating systems, are involved. Indeed, the access to the hard drive is indirect: the processing unit uses buffers. Also, the atomic unit of access depends on the size of disk sectors. If we read only one vertex on `.list` file, the system loads more vertices into its buffers. Hence, the number of physical access is lower than the number of requests performed. One overtakes in this regard practical considerations highlighted in the *I/O-efficient* model (see [20] for a survey).

Limits Encountered on our Machine. In the third level, we have generated two instances: a complete bipartite graph with 630,000 vertices and a *ButterFly* graph of dimension 30. On the complete bipartite graph, we were able to run algorithms LR, ED, S-Pitt and LL: their real running times do not exceed (on our computer) eight hours (executions of SLL and ASLL were stopped after twenty hours). On the *ButterFly* graph, we can only use LL (its real running time is about fifteen hours) because, on our computer, we cannot allocate an array of $33 \cdot 10^9$ bits (and executions of SLL and ASLL would take too long).

Therefore, LL is the only algorithm that can be run with our computer on all instances.

4 General Synthesis

We have implemented six algorithms for the VERTEX COVER problem on huge graphs. We were able to run these algorithms with a standard laptop computer on instances of sizes up to $30 \cdot 10^9$ vertices and edges (about 300 Gb on disk). The CPU running times we obtained do not exceed eight hours (and corresponding real running times are lower than ten hours).

We have observed that SLL and ASLL are almost always the slowest, since they have to compute degrees of neighbors. In this direction, they are "less adapted". However, SLL is still interesting, because it can give good solutions.

To test limits of algorithms, we have generated two instances of sizes about $100 \cdot 10^9$ vertices and edges (at least 1 Tb on disk): a complete bipartite graph with 630,000 vertices and $95 \cdot 10^9$ edges (a dense graph), and a *ButterFly* graph of dimension 30, with $33 \cdot 10^9$ vertices and $64 \cdot 10^9$ edges (a sparse graph).

– On the complete bipartite graph, we were able to run LR, ED, S-Pitt and LL (executions of SLL and ASLL were stopped before the end). For the slowest (LR), its CPU running time barely reaches two hours (and its corresponding real running time is about seven hours).
– On the *ButterFly* graph, we were only able to execute the algorithm LL: executions of LR, ED and S-Pitt failed because we could not allocate an array of $33 \cdot 10^9$ bits (and, as for the complete bipartite graph above, SLL and ASLL were stopped before the end).

General Observations. By summarizing the set of results presented for instances we used, among the six algorithms, LR is the one which gives the best solutions. It is closely followed by SLL, while ED and ASLL gives the worst solutions. However, ED performs the smallest number of requests. Based on that, choosing an algorithm which satisfies both quality of solutions and complexity in number of requests is difficult. On sparse graphs, where running times are often similar, we should promote the quality of solutions. Therefore, LR is a good candidate. Unfortunately, it needs to allocate an n bits array to be run, that is not always possible. On dense graphs, running times can change significantly, this makes choice trickier, since the most efficient algorithms are often the slowest.

Perspectives. We could extend our work by designing efficient algorithms on large instances for other problems. Then, we could compare our treatment method with the existing ones, e.g. with the semi-external greedy randomized adaptive search procedure presented in [2] for the MAX CLIQUE problem.

Acknowledgements. We would like to thank the anonymous referees for their insightful comments and suggestions, which have helped to improve the presentation of this paper.

References

1. http://todo.lamsade.dauphine.fr/spip.php?article39
2. Abello, J., Pardalos, P.M., Resende, M.G.C.: On Maximum Clique Problems in Very Large Graphs. In: External Memory Algorithms. DIMACS, vol. 50, pp. 119–130. American Mathematical Society (1999)
3. Abello, J., Pardalos, P.M., Resende, M.G.C.(eds.): Handbook of Massive Data Sets. Massive Computing, vol. 4. Springer (2002)
4. Ajwani, D.: Design, Implementation and Experimental Study of External Memory BFS Algorithms. Master's thesis, Max-Planck-Institut für Informatik, Saarbrücken, Germany (2005)

5. Alber, J., Dorn, F., Niedermeier, R.: Experimental Evaluation of a Tree Decomposition-Based Algorithm for Vertex Cover on Planar Graphs. Discrete Applied Mathematics 145, 219–231 (2004)
6. 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)
7. Asgeirsson, E., Stein, C.: Vertex Cover Approximations on Random Graphs. In: Demetrescu, C. (ed.) WEA 2007. LNCS, vol. 4525, pp. 285–296. Springer, Heidelberg (2007)
8. Bar-Yehuda, R., Hermelin, D., Rawitz, D.: Minimum Vertex Cover in Rectangle Graphs. In: 18th Annual European Conference on Algorithms, pp. 255–266 (2010)
9. Bomze, I.M., Budinich, M., Pardalos, P.M., Pedillo, M.: The Maximum Clique Problem. In: Handbook of Combinatorial Optimization, pp. 1–74. Kluwer Academic Publishers (1999)
10. de Bruijn, N.G.: A Combinatorial Problem. Koninklijke Nederlandse Akademie v. Wetenschappen 49, 758–764 (1946)
11. Delbot, F., Laforest, C.: A Better List Heuristic for Vertex Cover. Information Processing Letters 107, 125–127 (2008)
12. Delbot, F., Laforest, C.: Analytical and Experimental Comparison of Six Algorithms for the Vertex Cover. ACM Journal of Experimental Algorithmics 15 (2010)
13. Escoffier, B., Gourvès, L., Monnot, J.: Complexity and Approximation Results for the Connected Vertex Cover Problem in Graphs and Hypergraphs. Journal of Discrete Algorithms 8, 36–49 (2010)
14. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York (1979)
15. Gilmour, S., Dras, M.: Kernelization as Heuristic Structure for the Vertex Cover Problem. In: Dorigo, M., Gambardella, L.M., Birattari, M., Martinoli, A., Poli, R., Stützle, T. (eds.) ANTS 2006. LNCS, vol. 4150, pp. 452–459. Springer, Heidelberg (2006)
16. Molloy, M., Reed, B.: A Critical Point for Random Graphs With a Given Degree Sequence. In: Random Structures and Algorithms, pp. 161–179 (1995)
17. Pirzada, S., Dharwadker, A.: Applications of Graph Theory. Journal of The Korean Society for Industrial and Applied Mathematics (KSIAM) 11(4), 19–38 (2007)
18. Pitt, L.: A Simple Probabilistic Approximation Algorithm for Vertex Cover. Tech. Rep. 404, Yale University, Department of Computer Science (1985)
19. Vigier, F., Latapy, M.: Random Generation of Large Connected Simple Graphs with Prescribed Degree Distribution. In: 11th International Conference on Computing and Combinatorics, Kunming, Yunnan, Chine (2005)
20. Vitter, J.S.: Algorithms and Data Structures for External Memory, Boston, Delft. Foundations and Trends in Theoretical Computer Science, vol. 2 (2009)
21. Weisstein, E.W.: Butterfly graph, from MathWorld – A Wolfram Web Ressource, http://mathworld.wolfram.com/ButterflyGraph.html

How to Attack the NP-Complete Dag Realization Problem in Practice*

Annabell Berger and Matthias Müller-Hannemann

Dept. of Computer Science, Martin-Luther-Universität Halle-Wittenberg
{berger,muellerh}@informatik.uni-halle.de

Abstract. We study the following fundamental realization problem of directed acyclic graphs (dags). Given a sequence $S := \binom{a_1}{b_1}, \ldots, \binom{a_n}{b_n}$ with $a_i, b_i \in \mathbb{Z}_0^+$, does there exist a dag (no parallel arcs allowed) with labeled vertex set $V := \{v_1, \ldots, v_n\}$ such that for all $v_i \in V$ indegree and outdegree of v_i match exactly the given numbers a_i and b_i, respectively? Recently this decision problem has been shown to be NP-complete by Nichterlein [1]. However, we can show that several important classes of sequences are efficiently solvable. In previous work [2], we have proved that yes-instances always have a special kind of topological order which allows us to reduce the number of possible topological orderings in most cases drastically. This leads to an exact exponential-time algorithm which significantly improves upon a straightforward approach. Moreover, a combination of this exponential-time algorithm with a special strategy gives a linear-time algorithm. Interestingly, in systematic experiments we observed that we could solve a huge majority of all instances by the linear-time heuristic. This motivates us to develop characteristics like dag density and "distance to provably easy sequences" which can give us an indicator how easy or difficult a given sequence can be realized.

Furthermore, we propose a randomized algorithm which exploits our structural insight on topological sortings and uses a number of reduction rules. We compare this algorithm with other straightforward randomized algorithms and observe that it clearly outperforms all other variants. Another striking observation is that our simple linear-time algorithm solves a set of real-world instances from different domains, namely ordered binary decision diagrams (OBDDs), train and flight schedules, as well as instances derived from food-web networks without any exception.

1 The Dag Realization Problem

Dag Realization Problem: Given is a finite sequence $S := \binom{a_1}{b_1}, \ldots, \binom{a_n}{b_n}$ with $a_i, b_i \in \mathbb{Z}_0^+$. Does there exist an acyclic digraph (without parallel arcs) $G = (V, A)$ with the labeled vertex set $V := \{v_1, \ldots, v_n\}$ such that we have indegree $d_G^-(v_i) = a_i$ and outdegree $d_G^+(v_i) = b_i$ for all $v_i \in V$?

* This work was supported by the DFG Focus Program Algorithm Engineering, grant MU 1482/4-2.

R. Klasing (Ed.): SEA 2012, LNCS 7276, pp. 51–62, 2012.

If the answer is "yes", we call sequence S *dag sequence* and the acyclic digraph G (a so-called "dag") a *dag realization*. A relaxation of this problem – not demanding the acyclicity of digraph G – is called *digraph realization problem*. In this case, we call G *digraph realization* and S *digraph sequence*. The digraph realization problem can be solved in linear-time using an algorithm by Wang and Kleitman [3]. Unless explicitly stated, we assume that a sequence does not contain any *zero tuples* $\binom{0}{0}$. Moreover, we will tacitly assume that $\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} b_i$, as this is obviously a necessary condition for any realization to exist, since the number of ingoing arcs must equal the number of outgoing arcs. Furthermore, we denote tuples $\binom{a_i}{b_i}$ with $a_i > 0$ and $b_i = 0$ as *sink tuples*, those with $a_i = 0$ and $b_i > 0$ as *source tuples*, and the remaining ones with $a_i > 0$ and $b_i > 0$ as *stream tuples*. We call a sequence only consisting of source and sink tuples, *source-sink-sequence*. A sequence $S = \binom{a_1}{b_1}, \ldots, \binom{a_n}{b_n}$ with q source tuples and s sink tuples is denoted as *canonically sorted*, if and only if the first q tuples in this labeling are decreasingly sorted source tuples (with respect to the b_i) and the last s tuples are increasingly sorted sink tuples (with respect to the a_i).

Hardness and Efficiently Solvable Special Cases. Nichterlein very recently showed that the dag realization problem is NP-complete [1]. On the other hand, there are several classes of sequences for which the problem is not hard. One of these sequences are source-sink-sequences, for which one only has to find a digraph realization. The latter is already a dag realization, since no vertex has incoming as well as outgoing arcs. Furthermore, sparse sequences with $\sum_{i=1}^{n} a_i \leq n-1$ are polynomial-time solvable as we will show below. We denote such sequences by *forest sequences*. The main difficulty for the dag realization problem is to find out a "topological ordering of the sequence". In the case where we have one, our problem is nothing else but a directed f-factor problem on a complete dag. The labeled vertices of this complete dag are ordered in the given topological order. This problem can be reduced to a bipartite undirected f-factor problem which can be solved in polynomial time via a further famous reduction by Tutte [4] to a bipartite perfect matching problem. In a previous paper [2], we proved that a certain ordering of a special class of sequences –*opposed sequences*– always leads to a topological ordering of the tuples for at least one dag realization of a given dag sequence. On the other hand, it is not necessary to apply the reduction via Tutte if we possess one possible topological ordering of a dag sequence. The solution is much easier. Next, we describe our approach.

Realization with a Prescribed Topological Order. We denote a dag sequence $S := \binom{a_1}{b_1}, \ldots, \binom{a_n}{b_n}$ which possesses a dag realization with a topological numbering corresponding to the increasing numbering of its tuples by *dag sequence for a given topological order* and analogously the digraph $G = (V, A)$ by *dag realization for a given topological order*. Without loss of generality, we may assume that the source tuples come first in the prescribed numbering and are ordered decreasingly with respect to their b_i values. A realization algorithm works as follows. Consider the first tuple $\binom{a_{q+1}}{b_{q+1}}$ from the prescribed topological order which is not a source tuple. Then there must exist a_{q+1} source tuples with a smaller number in the given dag sequence. Reduce the a_{q+1} first (i.e. with largest b_i) source tuples by one and set the indegree of tuple $\binom{a_{q+1}}{b_{q+1}}$ to 0. That means, we reduce sequence $S := \binom{a_1}{b_1}, \ldots, \binom{a_{q+1}}{b_{q+1}}, \ldots, \binom{a_n}{b_n}$ to sequence $S' := \binom{a_1}{b_1-1}, \ldots, \binom{a_{a_{q+1}}}{b_{a_{q+1}}-1}, \ldots, \binom{a_q}{b_q}, \binom{0}{b_{q+1}}, \ldots, \binom{a_n}{b_n}$. If we get zero tuples in S', then we delete them and denote the new sequence for simplicity also by S'. Furthermore, we label this sequence with a new numbering starting from one to its length and consider this sorting as the given topological ordering for S'. We repeat this process until we get an empty sequence (corresponding to the realizability of S) or get stuck (corresponding to the non-realizability of S). The correctness of our algorithm is proven in Lemma 1.

Lemma 1. *S is a dag sequence for a given topological order $\Leftrightarrow$ S' is a dag sequence for its corresponding topological order.*

Discussion of Our Main Theorem and its Corresponding Algorithm. We do not know how to determine a feasible topological ordering (i.e., one corresponding to a realization) for an arbitrary dag sequence. However, we are able to restrict the types of possible permutations of the tuples. For that, we need the following order relation $\leq_{opp} \subset \mathbb{Z}^2 \times \mathbb{Z}^2$, introduced in [2].

Definition 1 (opposed relation). *Given are $c_1 := \binom{a_1}{b_1} \in \mathbb{Z}^2$ and $c_2 := \binom{a_2}{b_2} \in \mathbb{Z}^2$. We define: $c_1 \leq_{opp} c_2 \Leftrightarrow (a_1 \leq a_2 \wedge b_1 \geq b_2)$.*

Note, that a pair c_1 equals c_2 with respect to the opposed relation if and only if $a_1 = a_2$ and $b_1 = b_2$. The opposed relation is reflexive, transitive and antisymmetric and therefore a partial, but not a total order. Our following theorem leads to a recursive algorithm with exponential running time and results in Corollary 1 which proves the existence of a special type of possible topological sortings provided that sequence S is a dag sequence.

Theorem 1 (main theorem [2]). *Let S be a canonically sorted sequence containing $k > 0$ source tuples. Furthermore, we assume that S is not a source-sink-sequence. We define the set*

$$V_{min} := \left\{ \begin{pmatrix} a_i \\ b_i \end{pmatrix} \Big| \begin{pmatrix} a_i \\ b_i \end{pmatrix} \begin{array}{l} \text{is stream tuple}, a_i \leq k, \\ \text{and there is no stream tuple} \end{array} \begin{pmatrix} a_j \\ b_j \end{pmatrix} <_{opp} \begin{pmatrix} a_i \\ b_i \end{pmatrix} \right\}.$$

S is a dag sequence if and only if $V_{min} \neq \emptyset$ and there exists an element $\binom{a_\ell}{b_\ell} \in V_{min}$ such that $S' :=$

$$\begin{pmatrix} 0 \\ b_1 - 1 \end{pmatrix}, \ldots, \begin{pmatrix} 0 \\ b_{a_\ell} - 1 \end{pmatrix}, \begin{pmatrix} 0 \\ b_{a_\ell + 1} \end{pmatrix}, \ldots, \begin{pmatrix} 0 \\ b_k \end{pmatrix}, \ldots, \begin{pmatrix} a_{\ell-1} \\ b_{\ell-1} \end{pmatrix}, \begin{pmatrix} 0 \\ b_\ell \end{pmatrix}, \begin{pmatrix} a_{\ell+1} \\ b_{\ell+1} \end{pmatrix}, \ldots, \begin{pmatrix} a_n \\ b_n \end{pmatrix}$$

is a dag sequence.

Algorithm 1. DagRealization(sequence S)

```
   Input  : A canonically sorted sequence $S$.
   Output: A Boolean flag indicating whether $S$ is realizable.
1  if $S$ is not a source-sink-sequence then
2      count the number of sources in $S$ and determine set $V'_{min}$;
3      for all $\binom{a_j}{b_j} \in V'_{min}$ do
4          create a working copy $S'$ of $S$ with tuples $\binom{a'_i}{b'_i} = \binom{a_i}{b_i}$;
5          set $b'_i \leftarrow b'_i - 1$ for $a'_j$ largest sources $\binom{0}{b'_i}$;
6          set $a'_j \leftarrow 0$;
7          delete $\binom{0}{0}$-tuples;
8          if DagRealization($S'$) then return TRUE;
9      return FALSE;
10 else                                // Realization of a source-sink-sequence
11     while the set of source tuples in $S$ is not empty do
12         choose a largest source tuple $\binom{0}{b_j}$;
13         if number of sinks in $S$ is smaller than $b_j$ then return FALSE;
14         set $a_i \leftarrow a_i - 1$ for $b_j$ largest sinks $\binom{a_i}{0}$;
15         delete $\binom{0}{0}$-tuples;
16     return TRUE;
```

Sequence S' may contain zero tuples. If this is the case, we delete them and call the new sequence for simplicity also S'. Theorem 1 ensures the possibility for reducing a dag sequence into a source-sink-sequence. The latter can be realized by using the algorithm for realizing digraph sequences [3]. The whole algorithm is summarized in Algorithm 1, where we consider the maximum subset V'_{min} of V_{min} only containing pairwise disjoint stream tuples. The bottleneck of this approach is the size of set V'_{min}. Our pseudocode does not specify the order in which we process the elements of V'_{min} in line 3. Several strategies are possible which have a significant influence on the overall performance. The most promising deterministic strategy (as we will learn in the next sections) is to use the lexicographic order, starting with the lexicographic maximum element within V'_{min}. In [2] we introduced a special class of dag sequences – *opposed sequences* – where we have $|V'_{min}| = 1$, if sequence S is not a source-sink-sequence. We call a sequence S *opposed sequence*, if it is possible to sort its stream tuples in such a way, that $a_i \leq a_{i+1}$ and $b_i \geq b_{i+1}$ is valid for stream tuples with indices i and $i+1$. In this case, we have the property $\binom{a_i}{b_i} \leq_{opp} \binom{a_{i+1}}{b_{i+1}}$ for all stream tuples. At the beginning of the sequence we insert all source tuples such that the b_i build a decreasing sequence and at the end of sequence S we put all sink tuples in increasing ordering with respect to the corresponding a_i. The notion *opposed sequence* describes a sequence, where it is possible to compare all stream tuples among each other and to put them in a "chain". Indeed, this is not always possible because the opposed order is not a total order. However, for opposed sequences line (3) to line (9) in Algorithm 1 are executed at most once in each recursive call, because we have always $|V'_{min}| \leq 1$. Overall, we obtain a linear-time algorithm for opposed sequences. However, there are many sequences which are not opposed, but Theorem 1 still yields a polynomial decision time. Consider for example dag sequence $S := \binom{0}{3}, \binom{0}{3}, \binom{2}{2}, \binom{3}{3}, \binom{1}{0}, \binom{2}{0}, \binom{3}{0}$ which is not an opposed sequence, because stream tuples $\binom{2}{2}$ and $\binom{3}{3}$ are not comparable with respect to the opposed ordering. However, we have $|V'_{min}| = |\{\binom{2}{2}\}| = 1$ and so we reduce S to $S' = \binom{0}{2}, \binom{0}{2}, \binom{0}{2}, \binom{3}{3}, \binom{1}{0}, \binom{2}{0}, \binom{3}{0}$, leading to the realizable source-sink-sequence $\binom{0}{1}, \binom{0}{1}, \binom{0}{1}, \binom{0}{3}, \binom{1}{0}, \binom{2}{0}, \binom{3}{0}$. Theorem 1 leads to further interesting insights. We can prove the existence of special topological sortings.

Corollary 1 ([2]). *For every dag sequence S, there exists a dag realization $G = (V, A)$ with a topological ordering $v_{l_1}, \ldots, v_{l_{n_s}}$ of all n_s vertices corresponding to stream tuples, such that we cannot find $\binom{a_{l_j}}{b_{l_j}} <_{opp} \binom{a_{l_i}}{b_{l_i}}$ for $l_i < l_j$.*

We call a topological ordering of a dag sequence obeying the conditions in Corollary 1 an *opposed topological sorting*. At the beginning of our work (when the complexity of the dag realization problem was still open), we conjectured that the choice of the lexicographical largest tuple from V'_{min} in line (3) would solve our problem in polynomial time. We call this approach *lexmax strategy* and a dag sequence which is realizable with this strategy *lexmax sequence*, otherwise we call it *non-lexmax sequence*. Hence, we conjectured the following.

Conjecture 1 (lexmax conjecture). Each dag sequence is a lexmax sequence.

We soon disproved our own conjecture by a counter-example (Example 1, described in the following section). In systematic experiments we found out that a large fraction of sequences can be solved by this strategy in polynomial time. We tell this story in the next Section 2. Moreover, we use the structural insights from our main theorem to develop a randomized algorithm which performs well in practice (Section 3). Proofs and further supporting material can be found in the extended version, see arXiv:1203.3636v1 and [5].

2 Lessons from Experiments with the Lexmax Strategy

Why We Became Curious. To see whether our lexmax Conjecture 1 might be true, we generated a set of dag sequences, called *randomly generated sequences* in the sequel, by the following principle: Starting with a complete acyclic digraph, delete k of its arcs uniformly at random. We take the degree sequence from the resulting graph. Note that we only sample uniformly with respect to random dags but not uniformly degree sequences since degree sequences have different numbers of corresponding dag realizations. In a first experiment we created with the described process one million dag sequences with 20 tuples each, and $m = \sum_{i=1}^{20} a_i = 114$. Likewise, we built up another million dag sequences with 25 tuples and $\sum_{i=1}^{25} a_i = 180$. The fact that the lexmax strategy realized all these test instances without a single failure was quite encouraging.

The lexmax conjecture 1 seemed to be true, only a correctness proof was missing. But quite soon, in an attempt to prove the conjecture, we artificially constructed a first counter-example, a dag sequence which is definitely no lexmax sequence, as can easily be verified:

Example 1. $S := \binom{0}{3}, \binom{0}{1}, \binom{1}{2}, \binom{2}{3}, \binom{4}{4}, \binom{1}{1}, \binom{1}{0}, \binom{2}{0}, \binom{3}{0}$.

Even worse: we also found an example showing that no fixed strategy which chooses an element from V'_{min} in Algorithm 1 and does not consider the corresponding set of sinks, will fail in general.

These observations give rise to several immediate questions: Why did we construct by our sampling method (for $n = 20$ and $n = 25$) only dag sequences which are lexmax sequences? How many dag sequences are not lexmax sequences? Therefore, we started with systematic experiments. For small instances with $n \in \{7, 8, 9\}$ tuples we generated systematically the set of all dag sequences with all possible $\sum_{i=1}^{m} a_i =: m$, see for an example the case $n = 9$ in Figure 1. More precisely, we considered only *non-trivial sequences*, i.e. we eliminated all source-sink sequences and all sequences with only one stream tuple. We denote this set by *systematically generated sequences*. Note that the number of sequences grows so fast in n that a systematic construction of all sequences with a larger size is impossible. We observed the following:

1. The fraction of lexmax sequences among the systematically generated sequences is quite high. For all m it is above 96.5%, see Figure 1 (blue squares).
2. The fraction of lexmax sequences strongly depends on m. It is largest for sparse and dense dags.
3. Lexmax sequences are overrepresented among one million randomly generated sequences (for each m), we observe more than 99% for all densities of dags, see Figure 1 (red triangles).

This leads to the following questions: Given a sequence for which we seek a dag realization. How should we proceed in practice? As we have seen, the huge majority of dag sequences are lexmax sequences. Is it possible to find characteristic properties for lexmax sequences or non-lexmax sequences, respectively?

Distance to Opposed Sequences. Let us exploit our characterization that opposed sequences are efficiently solvable. We propose the *distance to opposed* $d(S)$ for each dag sequence S. Consider for that the topological order of a dag realization G given by Algorithm 1, if in line (3) elements are chosen in decreasing lexicographical order. This ordering corresponds to exactly one path of the recursion tree. Thus, we obtain one unique dag realization G for S, if existing. Now, we renumber dag sequence S such that it follows the topological order induced by the execution by this algorithm, i.e. by the sequence of choices of elements from V'_{min}. Then the distance to opposed is defined as the number of pairwise incomparable stream tuples with respect to this order, more precisely,

$$d(S) := \left| \left\{ \left(\begin{pmatrix} a_i \\ b_i \end{pmatrix}, \begin{pmatrix} a_j \\ b_j \end{pmatrix} \right) \mid \begin{pmatrix} a_i \\ b_i \end{pmatrix}, \begin{pmatrix} a_j \\ b_j \end{pmatrix} \begin{array}{l} \text{incomparable stream tuples} \\ \text{w.r.t. } \leq_{opp} \text{ and } i < j \end{array} \right\} \right| .$$

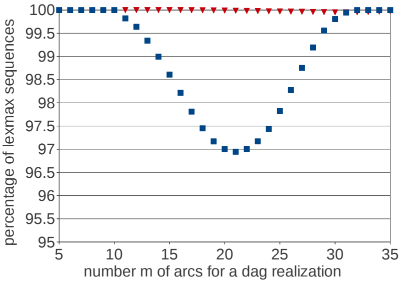


Fig. 1. Percentage of (non-trivial) lexmax sequences for systematically generated (squares) and randomly generated sequences (triangles) with 9 tuples and $m \in \{5, \ldots, 35\}$ arcs

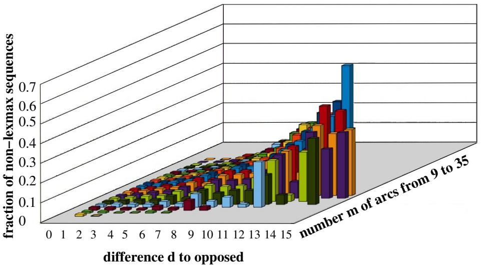


Fig. 2. Fraction of systematic non-lexmax sequences with 9 tuples, $m \in \{9, \ldots, 35\}$, and varying difference to opposed $d(S)$

Question 1: Do randomly generated sequences possess a preference to a "small" distance to opposed in comparison with systematically generated sequences? In Figure 3 (left), we show the distribution of systematically generated sequences (in %) with their distance to opposed, depending on $m := \sum_{i=1}^{n} a_i$. We compare this scenario with the same setting for randomly generated sequences, shown in Figure 3 (right).

Observations: Systematically generated sequences have a slightly larger range of the "distance to opposed" than randomly generated sequences. Moreover, when we generate dag sequences systematically, we obtain a significantly larger fraction of instances with a larger distance to opposed than for randomly generated sequences, and this phenomenon can be observed for all m.

Question 2: Do non-lexmax sequences possess a preference for large opposed distances? Since opposed sequences are easily solvable [2], we conjecture that sequences with a small distance to opposed might be easier solvable by the lexmax strategy than those with a large distance to opposed. If this conjecture were true, it would give us together with our findings from Question 1 one possible explanation for the observation that the randomly generated sequences have a larger fraction of efficiently solvable sequences by the lexmax strategy.

Observations: A separate analysis of non-lexmax sequences (that is, the subset of unsolved instances by the lexmax strategy), displayed in Figure 2, gives a clear picture: yes! For systematically generated sequences with $n = 9$, we observe in particular for instances with a middle density that the fraction of non-lexmax sequences becomes maximal for a relatively large distance to opposed.

Question 3: Can we solve real-world instances by the lexmax strategy?

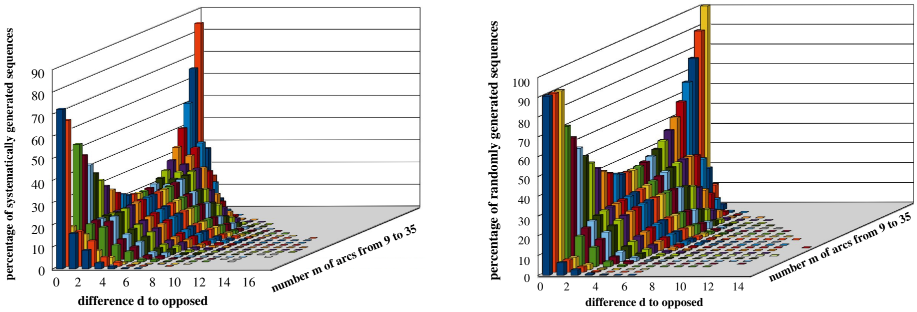


Fig. 3. Percentage of systematically generated sequences S (left) and randomized generated sequences (right) with their difference $d(S)$ to opposed for $n = 9$ tuples and $m \in \{9, \ldots, 35\}$ arcs

We consider real-world instances from different domains.

a): Ordered binary decision diagrams (OBDDs): In such networks the outdegree is two, that is constant. This immediately implies that the corresponding sequences are opposed sequences, and hence can provably be solved by the lexmax strategy.

b): Food Webs: Such networks are almost hierarchical and therefore have a strong tendency to be acyclic ("larger animals eat smaller animals"). In our experiments we analyzed food webs from the Pajek network library [6].

c): Train timetable network: We use timetable data of German Railways from 2011 and form a time-expanded network. Its vertices correspond to departure and arrival events of trains, a departure vertex is connected by an arc with the arrival event corresponding to the very next train stop. Moreover, arrival and departure events at the same station are connected whenever a transfer between trains is possible or if the two events correspond to the very same train.

d): Flight timetable network: We use the European flight schedule of 2010 and form a time-expanded network as in c).

The characteristics of our real-world networks b) - d) are summarized in Table 1. The *dag density* ρ of a network is defined as $\rho = m/\binom{n}{2}$. To compare the distance to opposed for instances of different sizes, we normalize this value by the theoretical maximum $\binom{b}{2}$, where b denotes the number of stream tuples, and so obtain a *normalized distance to opposed.* Without any exception, all real-world instances have been realized by the lexmax strategy.

Back to Theory. Inspired by our observations in the systematic experiments, we reconsidered forest sequences. We can show that an arbitrary choice of a tuple in V'_{min} in line 3 of Algorithm 1 solves the problem for forest sequences.

Table 1. Characteristics of our real-world test instances

name and kind of network	n	m	b	dag density ρ	norm. dist. to opposed
burgess shale (b)	142	770	101	0.08	0.40
chengjiang shale (b)	85	559	54	0.16	0.50
florida bay dry (b)	128	2137	125	0.26	0.32
cyprus dry (b)	71	640	68	0.26	0.43
maspalomas (b)	24	82	21	0.30	0.30
rhode river (b)	20	53	17	0.28	0.42
train schedule 2011 (c)	19359	77201	18907	0.0004	0.00
flight schedule 2010 (d)	37800	1324556	32905	0.0019	0.00

Corollary 2 (arbitrary tuple choice in V_{min} for forest sequences). *Let $S := \binom{a_1}{b_1}, \ldots, \binom{a_n}{b_n}$ with $\sum_{i=1}^{n} a_i \leq n-1$ be a canonically sorted sequence containing $k > 0$ source tuples. Furthermore, let S' be defined as in Theorem 1 where $\binom{a_{i_\ell}}{b_{i_\ell}}$ is an arbitrary tuple in V_{min}.*

S is a dag sequence if and only if S' is a dag sequence.

3 Randomized Algorithms

3.1 Four Versions of Randomized Algorithms

The main idea for developing a randomized algorithm is the following. In each trial use a randomly chosen topological sorting (a random permutation of the tuples) for a given sequence and then apply the linear-time realization algorithm as described in Section 1 and justified by Lemma 1. Clearly, it is not necessary to permute all tuples in a sequence. Instead we use a canonically sorted sequence and permute only the stream tuples. We denote this first naive version of a randomized algorithm by *stream tuple permutation algorithm* (Rand I). A random permutation of a sequence of length n can be chosen in $O(n)$ time, see for example [7]. Hence, one trial of the stream tuple permutation algorithm requires $O(m+n)$ time. This algorithm performs poorly since there are sequences with only a single realization among $(n-2)!$ many permutations of $n-2$ stream tuples. On the other hand, it is possible to restrict the number of possible topological sortings by the following lemma.

Lemma 2 (necessary criterion for the realizability of dag sequences). *Let S be a dag sequence. Denote the number of source tuples in S by q and the number of sink tuples by s. Then it follows $a_i \leq \min\{n-s, i-1\}$ and $b_i \leq \min\{n-q, n-i\}$ for all $i \in \mathbb{N}_n$ for each labeling of S corresponding to a topological order.*

Hence, a stream tuple $\binom{a_i}{b_i}$ can only be at position j in a topological ordering if $a_j \leq \min\{n-s, i-1\}$ and $b_j \leq \min\{n-q, n-i\}$ is fulfilled. We define a bipartite *bounding graph* $B_S = (V_S \cup W_S, E_S)$ for a given canonically sorted sequence as follows. We define $|S| - q - s$ vertices $v_i \in V_S$ with $i \in \{q+1, \ldots, n-s\}$ where each vertex v_i corresponds to an "upper bound tuple" $\binom{\min\{n-s,i-1\}}{\min\{n-q,n-i\}}$ for a stream tuple in S. Furthermore, we define $|S|-q-s$ vertices w_i with $i \in \{q+1, \ldots, n-s\}$ each corresponding to a stream tuple $\binom{a_i}{b_i}$. The edge set E_S is built as follows. Two vertices v_i and w_j are adjacent if and only if we find for $\binom{a_j}{b_j}$ that $a_j \leq \min\{n-s, i-1\}$ and $b_j \leq \min\{n-q, n-i\}$.

A perfect matching in this bounding graph gives us a possible topological sorting with respect to Lemma 2. This means, we assign to each stream tuple $\binom{a_j}{b_j}$ in S the number i if and only if (v_i, w_j) is a matching edge in the chosen perfect matching. Clearly, there does not exist a dag realization of sequence S if B_S does not contain a perfect matching. Unfortunately, the computation of the number of perfect matchings in a bipartite graph is known to be $\sharp P$-hard [8]. On the other hand, there exists a polynomial-time algorithm for the problem of uniform sampling a perfect matching within a bipartite graph by Jerrum, Sinclair and Vigoda [9]. They use a Markov chain based algorithm. The number of necessary steps in this algorithm is measured by the so-called *mixing time* τ_ϵ, where ϵ denotes the variation distance to the uniform distribution. They proved a worst case mixing time of $O(n^8(n \log n + \log \frac{1}{\epsilon}) \log \frac{1}{\epsilon})$. Up to know, we do not know if we really need a uniform distribution, but we do not want to eliminate certain topological orderings. Our second version of a randomized algorithm – the *bounding permutation algorithm* (Rand II) – chooses in each trial a topological sorting by uniform sampling a perfect matching in B_S and then applies the realization algorithm for a given topological order (Lemma 1). For our experiments with very small instances, we sampled uniformly by enumerating all permutations of stream tuples.

Our third randomized algorithm – the *opposed permutation algorithm* (Rand III) – exploits the non-trivial result in Corollary 1 about opposed topological sortings. It uses for one trial, Algorithm 1 with a change in line 3. We replace line 3 by: "**Sample a** $v_j \in V'_{min}$ **uniformly at random.**" If possible, we restrict the set of V'_{min} before line 3, i.e., we check for the largest $v_i \in V'_{min}$ whether the bounds of Lemma 2 are respected for later positions. Let k denote the number of recursive calls up to the current one. Expressed in terms of the original sequence, we have to choose the $(q+k)$–th tuple in the topological sorting in the current iteration. If $b_i = n - (q+k)$ for the lexicographical largest tuple $\binom{a_i}{b_i} \in V'_{min}$, then we set $V'_{min} := \{\binom{a_i}{b_i}\}$. The reason is that a larger position is not possible at all for this tuple, because the upper bound for b_i decreases strictly, as shown in Lemma 2. At first glance it is not clear whether the restriction to a subset of permutations within the randomized algorithm really increases the chance to draw a realizable topological sorting. This version of the algorithm only constructs dag realizations which possess an opposed topological sorting. Hence, we also exclude possible topological sortings which are not opposed topological sortings. However, empirically this idea pays off.

Our fourth randomized version combines the opposed permutation algorithm with several *reduction rules* which exploit the symmetric roles of in- and outdegrees

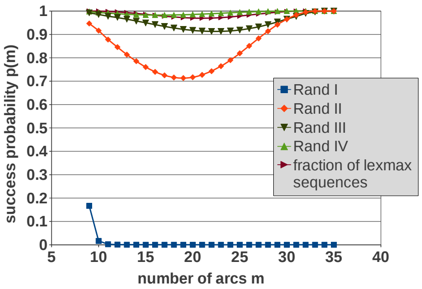


Fig. 4. Success probability $p(m)$ for all non-trivial sequences with 9 tuples with four versions of randomized algorithms and the fraction of lexmax sequences

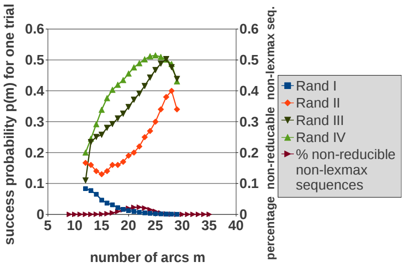


Fig. 5. Success probability $p(m)$ for all non-reducible non-lexmax sequences of 9 tuples with four versions of randomized algorithms and the percentage of non-reducible non-lexmax sequences in the set of all non-trivial sequences

and degree dominance of tuples. We additionally apply these rules whenever applicable and call the randomized algorithm *opposed permutation algorithm with reduction rules* (Rand IV).

3.2 Experimental Comparison of Randomized Algorithms

Experiment 1: Which randomized algorithm possesses the best success probability for one trial? We define the *success probability* $p(m)$ as the probability that a given sequence $S := \binom{a_1}{b_1}, \ldots, \binom{a_n}{b_n}$ with $m := \sum_{i=1}^{n} a_i$ can be realized by a specified randomized algorithm in one single trial. In this experiment we test the four versions of our randomized algorithms with all non-trivial sequences (as defined in Section 2) of 9 tuples, see Figure 4. Moreover, we display the fraction of lexmax sequences to compare the deterministic lexmax strategy with our randomized strategy.

Observations: Randomized version 4 (opposed permutation algorithm with reduction rules) clearly outperforms all other strategies. We also observe that the success probability p depends on the density m of the dag realizations. Sparse and dense dags have the best success probability. The deterministic lexmax strategy has almost the same success probability as our best randomized version. Of course, we can repeat a randomized algorithm and thereby boost the success rate which is not possible for the deterministic variant. Nevertheless the good performance of the simple lexmax strategy is quite remarkable, it clearly outperforms an arbitrary strategy to choose in line 3 of Algorithm 1 an element from V'_{min} (realized in randomized version 3).

Experiment 2: We consider the success probability for all randomized algorithms in the case of non-lexmax sequences which are not reducible by our reduction rules. Noting that an impressively large fraction of sequences is efficiently solvable by the deterministic lexmax strategy combined with our reduction rules, we should ask: How well do our randomized algorithms perform for the remaining difficult cases, that is for *non-reducible non-lexmax sequences*? Actually, this is indeed the most interesting question, because the best approach for realizing a given sequence S would be: first to test, whether S is a reducible lexmax sequence. Only if this is not the case, one would take a randomized algorithm. Hence, we now determine the success probability $p(m)$ for all non-reducible non-lexmax sequences, see Figure 5.

Observations: As in the previous experiment, randomized version 4 has the overall best success probability p, but in sharp contrast we observe a completely different dependence on m. One possible explanation could be that for high densities our reduction rules have been applied more often. Note that the overall percentage of non-reducible non-lexmax sequences in the set of all non-trivial sequences with 9 tuples is so tiny (see the brown curve in Figure 5) — in particular for low densities — that we can realize after two or three trials almost all sequences.

4 Conclusion

In this paper we have studied the performance of a simple linear-time heuristic to solve the NP-complete dag realization problem and several randomized variants. The surprisingly broad success of the lexmax strategy suggests that there might be further subclasses of instances where it runs provably correct. In future work we would like to characterize the class of instances for which the lexmax strategy works provably correct.

References

1. Nichterlein, A.: Realizing degree sequences for directed acyclic graphs is hard. CoRR abs/1110.1510v1 (2011)
2. Berger, A., Müller-Hannemann, M.: Dag Realizations of Directed Degree Sequences. In: Owe, O., Steffen, M., Telle, J.A. (eds.) FCT 2011. LNCS, vol. 6914, pp. 264–275. Springer, Heidelberg (2011); full version available as Technical Report 2011/5, Martin-Luther-Universität Halle-Wittenberg, Department of Computer Science
3. Kleitman, D.J., Wang, D.L.: Algorithms for constructing graphs and digraphs with given valences and factors. Discrete Mathematics 6, 79–88 (1973)
4. Tutte, W.: The factors of graphs. Canadian J. of Mathematics 4, 314–328 (1952)
5. Berger, A.: Directed degree sequences. PhD thesis, Department of Computer Science, Martin-Luther-Universität Halle-Wittenberg, urn:nbn:de:gbv:3:4-6768 (2011)
6. Batagelj, V.: Pajek datasets: Food webs (2004), http://vlado.fmf.uni-lj.si/pub/networks/data/bio/foodweb/foodweb.htm
7. Durstenfeld, R.: Algorithm 235: Random permutation. Commun. ACM 7, 420 (1964)
8. Valiant, L.G.: The complexity of computing the permanent. Theoretical Computer Science 8, 189–201 (1979)
9. Jerrum, M., Sinclair, A., Vigoda, E.: A polynomial-time approximation algorithm for the permanent of a matrix with nonnegative entries. Journal of the ACM 51, 671–697 (2004)

New Results about Multi-band Uncertainty in Robust Optimization*

Christina Büsing[1] and Fabio D'Andreagiovanni[2]

[1] Institut für Mathematik, Technische Universität Berlin
Strasse des 17 Juni 136, 10623 Berlin, Germany
cbuesing@math.tu-berlin.de
[2] Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB)
Takustr. 7, D-14195 Berlin, Germany
d.andreagiovanni@zib.de

Abstract. "The Price of Robustness" by Bertsimas and Sim [4] represented a breakthrough in the development of a tractable robust counterpart of Linear Programming Problems. However, the central modeling assumption that the deviation band of each uncertain parameter is single may be too limitative in practice: experience indeed suggests that the deviations distribute also internally to the single band, so that getting a higher resolution by partitioning the band into multiple sub-bands seems advisable.

In this work, we study the robust counterpart of a Linear Programming Problem with uncertain coefficient matrix, when a multi-band uncertainty set is considered. We first show that the robust counterpart corresponds to a compact LP formulation. Then we investigate the problem of separating cuts imposing robustness and we show that the separation can be efficiently operated by solving a min-cost flow problem. Finally, we test the performance of our new approach to Robust Optimization on realistic instances of a Wireless Network Design Problem subject to uncertainty.

Keywords: Robust Optimization, Multi-band Uncertainty, Compact Robust Counterpart, Cutting Planes, Network Design.

1 Introduction

A fundamental assumption in classical optimization is that all data are exact. However, many real-world problems involve data that are uncertain or not known with precision, because of erroneous measurements or adoptions of approximated numerical representations. If such uncertainty is neglected, optimal solutions

* This work was partially supported by the *German Federal Ministry of Education and Research* (BMBF), project *ROBUKOM*, grant 03MS616E, and by the DFG Research Center MATHEON - *Mathematics for key technologies*, project B23 - *Robust optimization for network applications*.

R. Klasing (Ed.): SEA 2012, LNCS 7276, pp. 63–74, 2012.

computed for nominal data values may become costly or infeasible. As a consequence, including uncertainty in an optimization model is a critical issue when dealing with real-world problems.

During the last years, Robust Optimization (RO) has become a valid methodology to deal with optimization problems subject to uncertainty. A key concept of RO is to model uncertainty as hard constraints, that are added to the original formulation of the problem. This restricts the set of feasible solutions to robust solutions, i.e. solutions that are protected from deviations of the data. Such a robust approach is crucial when dealing with high risk events, such as aircraft scheduling [12], or sensor placement in contaminant warning systems for water distribution networks [15]. In such settings, standard approaches like deterministic optimization or Stochastic Programming fail to protect against severe deviations, leading to unpredictable consequences. For an exhaustive introduction to theory and applications of RO, we refer the reader to the book by Ben-Tal et al. [2] and to the recent survey by Bertsimas et al. [3].

An approach to model uncertain data that has attracted a lot of attention is the so called Γ-scenario set, introduced by Betsimas and Sim (BS) [4] and then adapted to several applications. The uncertainty model for a Linear Program (LP) considered in BS assumes that, for each coefficient a we are given a nominal value $\bar{a}$ and a maximum deviation d and that the actual value lies in the interval $[\bar{a} - d, \bar{a} + d]$. Moreover, a parameter Γ is introduced to represent the maximum number of coefficients that deviate from their nominal value. Hence, Γ controls the conservativeness of the robust model and its introduction comes from the natural observation that it is unlikely that all coefficients deviate from their nominal value at the same time. A central result presented in BS is that, under the previous characterization of the uncertainty set, the robust counterpart of an LP corresponds to a linear formulation. This counterpart has the desirable properties of being *purely linear* and, above all, *compact*, i.e. the number of variables and constraints is polynomial in the size of the input of the deterministic problem.

The use of a single deviation band may greatly limit the power of modeling uncertainty. This is particularly evident when the probability of deviation sensibly varies within the band: in this case, neglecting the inner-band behaviour and just considering the extreme values like in BS may lead to a rough estimate of the deviations and thus to unrealistic uncertainty set, which either overestimate or underestimate the overall deviation. Having a higher modeling resolution would therefore be very desirable. This can be accomplished by breaking the single band into multiple and narrower bands, each with its own Γ. Such model is particularly attractive when historical data about the deviations are available, a very common case in real-world problems. Thus, a multi-band uncertainty set can effectively approximate the shape of the distribution of deviations built on past observations, guaranteeing a much higher modeling power than BS.

This observation was first captured by Bienstock and taken into account to develop an RO framework for the special case of Portfolio Optimization [5]. Yet, no definition and intensive theoretical study of a more general multi-band model applicable in other contexts have been done. The main goal of this paper is to close such gap.

Contributions and Outline. In this work, we study the robust counterpart of an LP with uncertain coefficient matrix, when a *multi-band uncertainty set* is considered. The main original contributions are:

- a compact formulation for the robust counterpart of an LP;
- an efficient method for the separation of robustness cuts (i.e., cuts that impose robustness), based on solving a min-cost flow instance;
- computational experiments comparing the performance of solving the compact formulation versus a cutting plane approach on realistic wireless network design instances.

In Section 2, we show that the robust counterpart of an LP under multi-band uncertainty corresponds to a compact Linear Programming formulation. We then proceed to study the separation problem of robustness cuts in Section 3. Finally, in Section 4, we test the performance of our new model and solution methods to Robust Optimization, to tackle the uncertainty affecting signal propagation in a set of realistic DVB-T instances of a wireless network design problem.

1.1 Model and Notation

We study the robust counterpart of Linear Programming Problems whose coefficient matrix is subject to uncertainty and the uncertainty set is modeled through multiple deviation bands. The deterministic Linear Program is of the form:

$$\begin{aligned} \max \; & \sum_{j \in J} c_j \, x_j && && (LPP) \\ & \sum_{j \in J} a_{ij} \, x_j \leq b_i && i \in I \\ & x_j \geq 0 && j \in J \end{aligned}$$

where $I = \{1, \ldots, m\}$ and $J = \{1, \ldots, n\}$ denote the set of constraint and variable indices, respectively. We assume that the value of each coefficient a_{ij} is uncertain and that such uncertainties are modeled through a set of scenarios $\mathcal{S}$. Each scenario $S \in \mathcal{S}$ defines a different coefficient matrix A^S. The robust counterpart of (LPP) thus corresponds to the following problem:

$$\begin{aligned} \max \; & \sum_{j \in J} c_j \, x_j \\ & \sum_{j \in J} a_{ij}^S \, x_j \leq b_i && i \in I, S \in \mathcal{S} \\ & x_j \geq 0 && j \in J. \end{aligned}$$

We note that uncertainty on the cost c and on the r.h.s. b can be included in a very straightforward way in the coefficient matrix, as explained in [3].

One of the purpose of this paper is to characterize the robust counterpart of (LPP) when the set of scenarios corresponds to what we call a *multi-band uncertainty set.* This set is denoted by $\mathcal{S}_M$ and generalizes the Bertsimas-Sim uncertainty model. Specifically, we assume that, for each coefficient a_{ij}, we are given a *nominal value* $\bar{a}_{ij}$ and maximum negative and positive deviations $d_{ij}^{K^-}, d_{ij}^{K^+}$ from $\bar{a}_{ij}$, such that the *actual value* a_{ij}^S lies in the interval $[\bar{a}_{ij} + d_{ij}^{K^-}, \bar{a}_{ij} + d_{ij}^{K^+}]$ for each scenario $S \in \mathcal{S}_M$. Moreover, we define a *system of deviation bands* by partitioning the single deviation band $[d_{ij}^{K^-}, d_{ij}^{K^+}]$ into K bands, defined on the basis of $K+1$ deviation values:

$$-\infty < d_{ij}^{K^-} < \cdots < d_{ij}^{-2} < d_{ij}^{-1} \ < \ d_{ij}^{0} = 0 \ < \ d_{ij}^{1} < d_{ij}^{2} < \cdots < d_{ij}^{K^+} < +\infty.$$

Through these deviation values, we define: 1) the zero-deviation band corresponding to the single value $d_{ij}^0 = 0$; 2) a set of positive deviation bands, such that each band $k \in \{1, \dots, K^+\}$ corresponds to the range $(d_{ij}^{k-1}, d_{ij}^{k}]$; 3) a set of negative deviation bands, such that each band $k \in \{K^-, \dots, -1\}$ corresponds to the range $[d_{ij}^{k}, d_{ij}^{k-1})$ (the interval of each band is thus closed on the endpoint with the higher absolute value). With a slight abuse of notation, in what follows we indicate a generic deviation band by $k \in K = \{K^-, \dots, -1, 0, 1, \dots, K^+\}$.

Additionally, for each band $k \in K$, we define a lower bound l_k and an upper bound u_k on the number of deviations that may fall in k, with l_k, u_k satisfying $0 \leq l_k \leq u_k \leq n$. In the case of band 0, we assume that $u_0 = n$, i.e. we do not limit the number of coefficients that take their nominal value. Furthermore, we assume that $\sum_{k \in K} l_k \leq n$ so that there always exists a feasible realization of the coefficient matrix. On the basis of these parameters, we formalize the set of scenarios $\mathcal{S}_M$: a scenario $S \in \mathcal{S}_M$ is feasible if and only if $a_{ij}^S \in [\bar{a}_{ij} + d_{ij}^{K^-}, \bar{a}_{ij} + d_{ij}^{K^+}]$ and $l_k \leq |\{j \in J \mid a_{ij}^S \text{ lies in band } k\}| \leq u_k$ for every $k \in K$, $i \in I$. In other words, we require that the deviations satisfy the system of multi-band uncertainty and thus the number of deviations falling in each band must satisfy the corresponding bounds.

We remark that, in order to avoid an overload of the notation, we assume that the number of bands K and the bounds l_k, u_k are the same for each constraint $i \in I$. Anyway, it is straightforward to modify all presented results to take into account different values of those parameters for each constraint. We now proceed to study the robust counterpart of (LPP) under multi-band uncertainty.

2 A Compact Robust LP Counterpart

The robust counterpart of an (LPP) under a multi-band uncertainty set defined by $\mathcal{S}_M$ can be equivalently written as:

$$\begin{aligned} \max \ & \sum_{j \in J} c_j \ x_j \\ & \sum_{j \in J} \bar{a}_{ij} \ x_j + DEV_i(x, d) \leq b_i && i \in I \\ & x_j \geq 0 && j \in J \end{aligned}$$

where $DEV_i(x,d)$ is the maximum overall deviation allowed by a system of deviation bands d for a feasible solution x when constraint i is considered. Note that we replace the actual value of a coefficient a_{ij} with the summation of the nominal value $\bar{a}_{ij}$ and a deviation d_{ij} falling in exactly one of the K bands. The computation of $DEV_i(x,d)$ corresponds to the optimal value of the following pure 0-1 Linear Program (note that in this case index i is fixed):

$$DEV_i(x,d) = \max \sum_{j \in J} \sum_{k \in K} d_{ij}^k \; x_j \; y_{ij}^k \qquad (DEV01)$$

$$l_k \;\leq\; \sum_{j \in J} y_{ij}^k \;\leq\; u_k \qquad k \in K \qquad (1)$$

$$\sum_{k \in K} y_{ij}^k \leq 1 \qquad j \in J \qquad (2)$$

$$y_{ij}^k \in \{0,1\} \qquad j \in J, k \in K. \qquad (3)$$

The binary variables y_{ij}^k indicate if the deviation of a coefficient a_{ij} lies in band k. Constraints (2) ensure that each coefficient deviates in at most one band (actually these should be equality constraints, but, for assumption $u_0 = n$ made in Section 1.1, we can consider inequalities). Finally, constraints (1) impose the upper and lower bounds on the number of deviations falling in each band k. Thus, the optimal solution of (DEV01) defines a distribution of the coefficients among the bands that maximizes the deviation w.r.t. the nominal values, while respecting the bounds on the number of deviations of each band.

We now show that the polytope associated with the linear relaxation of (DEV01) is integral. The linear relaxation of (DEV01) is:

$$\max \sum_{j \in J} \sum_{k \in K} d_{ij}^k \; x_j \; y_{ij}^k \qquad \text{(DEV01-RELAX)}$$

$$l_k \leq \sum_{j \in J} y_{ij}^k \leq u_k \qquad k \in K \qquad (4)$$

$$\sum_{k \in K} y_{ij}^k \leq 1 \qquad j \in J \qquad (5)$$

$$y_{ij}^k \geq 0 \qquad j \in J, k \in K \qquad (6)$$

where we dropped constraints $y_{ij}^k \leq 1$ since they are dominated by constraints (5).

Theorem 1. *The polytope described by the constraints of (DEV01-RELAX) is integral.*

Proof. We start by rewriting all the constraints of (DEV01-RELAX) into the form $\alpha^T y \leq \beta$ obtaining the following matrix form:

$$D_i\; y_i \;=\; \left(\begin{array}{c|c|c|c} -I & -I & \cdots & -I \\ \hline I & I & \cdots & I \\ \hline 1\cdots 1 & & & \\ & 1\cdots 1 & & \\ & & \ddots & \\ & & & 1\cdots 1 \end{array}\right) \left(\begin{array}{c} y_{i1}^{K^-} \\ \vdots \\ y_{i1}^{K^+} \\ \hline \vdots \\ y_{ij}^{k} \\ \vdots \\ \hline y_{in}^{K^-} \\ \vdots \\ y_{in}^{K^+} \end{array}\right) \;\le\; \left(\begin{array}{c} \vdots \\ -l_k \\ \vdots \\ \hline \vdots \\ u_k \\ \vdots \\ \hline \vdots \\ 1 \\ \vdots \end{array}\right) \;=\; g_i.$$

Consider now the submatrix $\tilde{D}_i$ obtained from D_i by eliminating the top layer of blocks $(-I| - I| \cdots | - I)$. It is easy to verify that $\tilde{D}_i$ is the incidence matrix of a bipartite graph: the elements of the two disjoint set of nodes of the graph are in correspondence with the rows of the two distinct layers of blocks in $\tilde{D}_i$. Moreover, every column has exactly two elements that are not equal to zero, one in the upper layer and one in the lower layer. Being the incidence matrix of a bipartite graph, $\tilde{D}_i$ is a totally unimodular matrix [13].

In order to show that also the original matrix D_i is totally unimodular, we first need to recall the equivalence of the following three statements [13]: 1) A is a totally unimodular matrix; 2) a matrix obtained by duplicating rows of A is totally unimodular; 3) a matrix obtained by multiplying a row of A by -1 is totally unimodular. Since D_i can be obtained from $\tilde{D}_i$ by duplicating each row of the upper block, and multiplying each row of the duplicated block by -1, D_i is totally unimodular.

As D_i is totally unimodular and the vector g_i is integral, it is well-known that the polytope defined by $D_i y_i \le g_i$ and $y_i \ge 0$ is integral, thus completing the proof. □

Since the polytope associated with (DEV01-RELAX) is integral, by strong duality we can use the dual problem of (DEV01-RELAX) to replace $\mathrm{DEV}_i(x, d)$ in the robust counterpart of (LPP). The dual problem of (DEV01-RELAX) is:

$$\begin{array}{lll} \min & \displaystyle\sum_{k \in K} -l_k\, v_i^k + \sum_{k \in K} u_k\, w_i^k + \sum_{j \in J} z_i^j & \text{(DEV01-RELAX-DUAL)} \\ & -v_i^k + w_i^k + z_i^j \ge d_{ij}^k\, x_j & j \in J, k \in K \\ & v_i^k, \;\; w_i^k \ge 0 & k \in K \\ & z_i^j \ge 0 & j \in J \end{array}$$

where the dual variables v_i^k, w_i^k, z_i^j are respectively associated with the primal constraints (4, 5, 6) of (DEV01-RELAX) defined for constraint i. Replacing $\mathrm{DEV}_i(x, d)$ by its dual yields the following compact linear robust counterpart of the original problem (LPP):

$$
\begin{aligned}
\max\ & \sum_{j \in J} c_j\ x_j && \text{(RLP)}\\
& \sum_{j \in J} \bar{a}_{ij}\ x_j - \sum_{k \in K} l_k\ v_i^k + \sum_{k \in K} u_k\ w_i^k + \sum_{j \in J} z_i^j \leq b_i && i \in I\\
& -v_i^k + w_i^k + z_i^j \geq d_{ij}^k\ x_j && i \in I, j \in J, k \in K\\
& v_i^k,\ w_i^k \geq 0 && i \in I, k \in K\\
& z_i^j \geq 0 && i \in I, j \in J\\
& x_j \geq 0 && j \in J.
\end{aligned}
$$

In comparison to (LPP), this compact formulation uses $2 \cdot K \cdot m + n \cdot m$ additional variables and includes $K \cdot n \cdot m$ additional constraints.

3 Separation of Robustness Cuts

In this section, we consider the problem of testing whether a solution $x^* \in \mathbb{R}^n$ is robust feasible, i.e. $a_i^S x^* \leq b_i$ for every scenario $S \in \mathcal{S}_M$ and $i \in I$. This problems becomes important for adopting a cutting plane approach instead of directly solving the compact robust counterpart (RLP). This approach works as follows: start by solving the nominal problem (LPP) and then check if the optimal solution is robust. If not, generate a cut that imposes robustness (*robustness cut*) and add it to the problem. This initial step is then iterated as in a typical cutting plane method [13].

In the case of the Bertsimas-Sim model, the problem of separating a robustness cut is very simple [7]: given a solution x^*, for each constraint $i \in I$, the problem consists of sorting the deviations $d_{ij}^{K+} x_j^*$ in non-increasing order and choose the highest Γ_i deviations. If for some i the sum of these deviations exceeds $b_i - \sum_{j \in J} \bar{a}_{ij} x_j$ then we found a robustness cut to be added. Otherwise, x^* is robust.

In the case of multi-band uncertainty, this simple approach does not guarantee robustness of a computed solution. However, we prove that for a given solution $x^* \in \mathbb{R}^n$ and a constraint $i \in I$, checking the robust feasibility of x^* corresponds to solving a *min-cost flow problem* [1], whose instance is denoted by $(G, c)_x^i$ and defined as follows. G is a directed graph whose set of vertices V contains one vertex v_j for each variable index $j \in J$, one vertex w_k for each band $k \in K$ and two vertices s, t that are the source and the sink of the flow, i.e. $V = \bigcup_{j \in J} \{v_j\} \cup \bigcup_{k \in K} \{w_k\} \cup \{s, t\}$. The set of arcs A is the union of three sets A_1, A_2, A_3. A_1 contains one arc from s to every variable vertex v_j, i.e. $A_1 = \{(s, v_j) \mid j \in J\}$. A_2 contains one arc from every variable vertex v_j to every band vertex w_k, i.e. $A_2 = \{(v_j, w_k) \mid j \in J, k \in K\}$. Finally, A_3 contains one arc from every band vertex w_k to the sink t, i.e. $A_3 = \{(w_k, t) \mid k \in K\}$. By construction, $G(V, A)$ is bipartite and acyclic. Each arc $a \in A$ is associated to a triple (l_a, u_a, c_a), where l_a, u_a are lower and upper bounds on the flow that can be sent on a and c_a is the cost of sending one unit of flow on a. The values of the triples (l_a, u_a, c_a) are set in the following way: $(0,1,0)$ when $a \in A_1$; $(0,1,-d_{ij}^k x_j^*)$ when $a=(v_j,w_k) \in A_2$; $(l_k,u_k,0)$ when $a=(w_k,t) \in A_3$. Finally, the amount of flow that must be sent trough the network from s to t is equal to n. The value of an (s,t)-flow is defined by $C(f)=\sum_{a \in A} c_a f_a$. An integral min-cost flow can be computed in polynomial time, using for example the successive shortest path algorithm [1].

We now prove that by solving the min-cost flow instance defined above, we obtain the maximum deviation for constraint i and solution x^*.

Lemma 1. *A solution $x^* \in \mathbb{R}^n$ is robust w.r.t. a multi-band scenario set $\mathcal{S}_M$ if and only if*

$$\bar{a}_i' x^* - C(f) \leq b_i$$

for every $i \in I$ and min-cost flow f of the instance $(G,c)_{x^*}^i$.

Proof. We show that for any flow f there exists a scenario $S_f \in \mathcal{S}_M$ with $(a_i^{S_f})'x^* = \overline{a}_i' x^* - C(f)$ and for every scenario $S \in \mathcal{S}_M$ there exists a flow f^S with $C(f^S) = \overline{a}_i' x^* - (a_i^S)'x^*$. Let $f : A \to \{0,1\}$ be a feasible flow in $(G,c)_{x^*}^i$. Then we obtain a feasible scenario $S \in \mathcal{S}_M$ by setting $a_{ij}^S = \bar{a}_{ij} + \sum_{k \in K} d_{ij}^k f_{jk}$, where f_{jk} denotes the flow on arc (v_j,w_k), i.e. $f_{jk} = f((v_j,w_k))$. Due to the flow conservation in every vertex v_j, there exists exactly one variable $f_{jk}=1$, $j \in J$, $k \in K$. Furthermore, the amount of variables whose coefficients are in band $k \in K$ is at least l_k and at most u_k due to the upper and lower bounds on the arc (w_k,t). Hence, S_f is a feasible scenario and

$$\begin{aligned}\sum_{j \in J} a_{ij}^{S_f} x_j^* &= \sum_{j \in J} \bar{a}_{ij} x_j^* + \sum_{j \in J} \sum_{k \in K} d_{ij}^k f_{ik} x_j^* \\ &= \sum_{j \in J} \bar{a}_{ij} x_j^* - C(f).\end{aligned}$$

On the other hand, let $S \in \mathcal{S}_M$ be a feasible scenario. We set $f_{jk}^S = 1$ if and only if a_{ij}^S is in band $k \in K$. The flow on the other arcs is set in such a way that we preserve flow conservation in every vertex besides s and t. Then f^S is a feasible flow, since the lower and upper capacity bounds are satisfied due to the feasibility of S, and n units of flow are sent through the network. Furthermore,

$$\begin{aligned}C(f^S) &= -\sum_{j \in J} \sum_{k \in K} d_{ij}^k x_j^* f_{jk} \\ &= \sum_{j \in J} \bar{a}_{ij} x_j^* - \sum_{j \in J} \sum_{k \in K} d_{ij}^k x_j^* f_{ik} - \sum_{j \in J} \bar{a}_{ij} x_j^* \\ &= (a_i^S)' x^* - \bar{a}_i' x_j^*.\end{aligned}$$

This concludes the proof. □

According to this lemma, we can test the robustness of a solution $x^* \in \mathbb{R}^n$ by computing a min-cost flow f^i in $(G,c)_{x^*}^i$ for every $i \in I$. If $\bar{a}' x^* - C(f^i) \leq b_i$ for every i, then x^* is a robust solution. If x^* is not robust, there exists an index i such that $\bar{a}'x^* - C(f^i) > b_i$ and thus

$$\sum_{j \in J} \overline{a}_{ij} x_{ij} + \sum_{j \in J} \sum_{k \in K} d_{ij}^k f_{jk}^i x_{ij} \leq b_i \tag{7}$$

is valid for the polytope of the robust solutions and cuts off the solution x^*.

4 Computational Study

In this section, we test our new modeling and solution approaches to Robust Optimization on a set of realistic instances of the *Power Assignment Problem*, a problem arising in the design of wireless networks. In particular, we compare the efficiency of solving directly the compact formulation (RLP) with that of a cutting plane method based on the robustness cuts presented in Section 3. In the case of the Bertsimas-Sim model, such comparison led to contrasting conclusions (e.g., [7,8]).

The Power Assignment Problem. The *Power Assignment Problem* (PAP) is the problem of dimensioning the power emission of each transmitter in a wireless network, in order to provide service coverage to a number of user, while minimizing the overall power emission. The PAP is particularly important in the (re)optimization of networks that are updated to new generation digital transmission technologies. For a detailed introduction to the PAP and the general problem of designing wireless networks, we refer the reader to [11,6,10].

A classical LP formulation for the PAP can be defined by introducing the following elements: 1) a vector of non-negative continuous variables p that represent the power emissions of the transmitters; 2) a vector $P^{\max}$ of upper bounds on p that represent technology constraints on the maximum power emissions; 3) a matrix A of the coefficients that represent signal attenuation (*fading coefficients*) for each transmitter-user couple; 4) a vector of r.h.s. δ (signal-to-interference thresholds) that represent the minimum power values that guarantee service coverage. Under the objective of minimizing the overall power emission, the PAP can be written in the following matrix form:

$$\min\ \mathbf{1}'p \quad \text{s.t.}\ \ Ap \geq \delta, \ \ 0 \leq p \leq P^{\max} \qquad\qquad (PAP)$$

where exactly one constraint $a_i'p \geq \delta_i$ is introduced for each user i to represent the corresponding service coverage condition.

Each entry of matrix A is classically computed by a propagation model and takes into account many factors (e.g., distance between transmitter and receiver, terrain features). However, the exact propagation behavior of a signal cannot be evaluated and thus each fading coefficient is naturally subject to uncertainty. Neglecting such uncertainty may provide unpleasant surprises in the final coverage plan, where devices may turn out to be uncovered for bad deviations affecting the fading coefficients (this is particularly true in hard propagation scenarios, such as dense urban fabric). For a detailed presentation of the technical aspects of propagation, we refer the reader to [14].

Following the ITU recommendations (e.g., [9]), we assume that the fading coefficients are mutually independent random variables and that each variable is log-normally distributed. The adoption of the Bertsimas-Sim model would provide only a rough representation of the deviations associated with such distribution. We thus adopt the multi-band uncertainty model to obtain a more refined representation of the fading coefficient deviations. In what follows, we denote the Bertsimas-Sim and the multi-band uncertainty model by (BS) and (MB), respectively.

Computational Results. In this computational study, we consider realistic instances corresponding to region-wide networks that implement the Terrestrial Digital Video Broadcasting technology (DVB-T) [9] and were taken as reference for the design of the new Italian DVB-T national network. The uncertainty set is built taking into account the ITU recommendations [9] and discussions with our industrial partners in past projects about wireless network design. Specifically, we assume that each fading coefficient follows a log-normal distribution with mean provided by the propagation model and standard deviation equal to 5.5 dB [9]. In our test-bed, the (MB) uncertainty set of a generic fading coefficient a_{ij} is constituted by 3 negative and 3 positive deviations bands (i.e., $K = 6$). Each band has a width equal to the 5% of the nominal fading value $\bar{a}_{ij}$. Thus the maximum allowed deviation is +/- $0.15 \cdot \bar{a}_{ij}$. For each constraint i, the bounds l_k, u_k on the number of deviations are defined considering the cumulative distribution function of a log-normal random variable with standard deviation 5.5 dB. The (BS) uncertainty set of each constraint considers the same maximum deviation of (MB) and the maximum number of deviating coefficients is $\Gamma = \lceil 0.8 \cdot u^{\max} \rceil$, where $u^{\max} = \max\{u_k : k \in K \setminus \{0\}\}$. This technically reasonable assumption on Γ ensures that (BS) does not dominate (MB) a priori.

The computational results are reported in Table 1. The tests were performed on a Windows machine with 1.80 GHz Intel Core 2 Duo processor and 2 GB RAM. All the formulations are implemented in C++ and solved by IBM ILOG Cplex 12.1, invoked by ILOG Concert Technology 2.9. We considered 15 instances of increasing size corresponding to realistic DVB-T networks. The first column of Table 1 indicates the ID of the instances. Columns $|I|, |J|$ indicate the number of variables and constraints of the problem, corresponding to the number of user devices and transmitters of the network, respectively. We remark that the coefficient matrices tend to be sparse, as only a (small) fraction of the transmitters is able to reach a user device with its signals. Columns $|I^+|, |J^+|$ indicate the number of additional variables and constraints needed in the compact robust counterpart (RLP). Columns PoR% report *the Price of Robustness* (PoR), i.e. the deterioration of the optimal value required to guarantee robustness. In particular, we consider the percentage increase of the robust optimal value w.r.t. the optimal value of the nominal problem, in the multi-band case (PoR% (MB)) and in the Bertsimas-Sim case (PoR% (BS)). Column $\Delta t\%$ reports the percentage increase of the time required to compute the robust optimal solution under (MB) by using the cutting plane method presented in Section 3 w.r.t. the time needed to solve the compact formulation (RLP). Finally, column Protect% is a measure of the protection offered by the robust optimal solution and is computed in the following way: for each instance, we generate 1000 realizations of the uncertain coefficient matrix and we then compute the percentage of realizations in which the robust optimal solution is feasible. This is done for both (MB) and (BS).

Looking at Table 1, the first evident thing is that the dimension of the compact robust counterpart under (MB) is much larger than that of the nominal problem. However, this is not an issue for Cplex, as all instances are solved within one hour and in most of the cases the direct solution of (RLP) takes less time than the cutting plane approach ($\Delta t\% < 0$). Anyway, for the instances of greater dimension the cutting plane approach becomes competitive and may even take less time ($\Delta t\% > 0$). Concerning the PoR, we note that under (MB) imposing robustness leads to a sensible increase in the overall power emission, that is anyway lower than that of (BS) in all but two cases. On the other hand, such increase of (MB) is compensated by a very good 90% protection on average. In the case of the PAP, (MB) thus seems convenient to model the log-normal uncertainty of fading coefficients, guaranteeing good protection at a reasonable price. Moreover, though (BS) offers higher protection for most instances, it is interesting to note that the increase of Protect% of (BS) w.r.t (MB) is lower than the corresponding increase of PoR% of (BS) w.r.t (MB).

Table 1. Overview of the computational results

ID	$\|I\|$	$\|J\|$	$\|I^+\|$	$\|J^+\|$	PoR% (MB)	PoR% (BS)	Δt%	Protect% (MB)	Protect% (BS)
D1	95	153	3519	10098	8.3	10.1	-18.7	88.20	92.53
D2	103	197	4728	14184	7.2	9.4	-19	91.35	92.47
D3	105	322	7406	21252	6.8	8.8	-16.9	93.12	96.40
D4	105	473	10406	28380	7.4	7.2	-15.1	92.08	91.42
D5	108	569	13087	37554	9.2	11.4	-13.6	89.23	90.29
D6	157	1088	27200	84864	6.6	9.1	-6.2	85.46	87.55
D7	165	1203	31278	101052	7.1	9.5	-4.9	87.91	89.16
D8	171	1262	32812	106008	8.7	10.8	-4.1	89.40	93.08
D9	178	1375	35750	115500	9.6	10.2	-2.8	90.11	91.90
D10	180	1448	39096	130320	7.9	9.6	-1.7	91.54	95.32
D11	180	1661	46058	159456	7.2	9.5	0.6	94.77	96.70
D12	181	1779	49812	170784	7.5	10.1	1.8	88.22	90.16
D13	183	1853	53737	189006	8.1	10.3	3.3	91.34	92.21
D14	183	1940	56260	197880	10.3	9.7	3.1	86.50	85.18
D15	185	2183	63307	222666	8.4	10.8	4.1	91.09	92.70

5 Conclusions and Future Work

In this work, we presented new theoretical results about multi-band uncertainty in Robust Optimization. Surprisingly, this natural extension of the classical single band model by Bertsimas and Sim has attracted very little attention and we have thus started to fill this theoretical gap. We showed that, under multi-band uncertainty, the robust counterpart of an LP is linear and compact and that the problem of separating a robustness cut can be formulated as a min-cost flow problem and thus be solved efficiently. Tests on realistic network design instances showed that our new approach performs very well, thus encouraging further investigations. Future research will focus on refining the cutting plane method and enlarging the computational experience to other relevant real-world problems.

References

1. Ahuja, R.K., Magnanti, T., Orlin, J.B.: Network flows: theory, algorithms, and applications. Prentice Hall, Upper Saddle River (1993)
2. Ben-Tal, A., El Ghaoui, L., Nemirovski, A.: Robust Optimization. Springer, Heidelberg (2009)
3. Bertsimas, D., Brown, D., Caramanis, C.: Theory and Applications of Robust Optimization. SIAM Review 53(3), 464–501 (2011)
4. Bertsimas, D., Sim, M.: The Price of Robustness. Oper. Res. 52(1), 35–53 (2004)
5. Bienstock, D.: Histogram models for robust portfolio optimization. J. Computational Finance 11, 1–64 (2007)
6. D'Andreagiovanni, F.: Pure 0-1 Programming Approaches to Wireless Network Design. Ph.D. Thesis, Sapienza Università di Roma, Roma, Italy (2010)
7. Fischetti, M., Monaci, M.: Robustness by cutting planes and the Uncertain Set Covering Problem. ARRIVAL Project Tech. Rep. 0162, Università di Padova, Padova, Italy (2008)
8. Koster, A.M.C.A., Kutschka, M., Raack, C.: Robust Network Design: Formulations, Valid Inequalities, and Computations. ZIB Tech. Rep. 11-34, Zuse-Institut Berlin, Berlin, Germany (2011)
9. International Telecommunication Union (ITU): DSB Handbook - Terrestrial and satellite digital sound broadcasting to vehicular, portable and fixed receivers in the VHF/UHF bands (2002)
10. Mannino, C., Rossi, F., Smriglio, S.: The Network Packing Problem in Terrestrial Broadcasting. Oper. Res. 54(6), 611–626 (2006)
11. Mannino, C., Rossi, F., Smriglio, S.: A Unified View in Planning Broadcasting Networks. DIS Tech. Rep. 08-07, Sapienza Università di Roma, Roma, Italy (2007)
12. Mulvey, J.M., Vanderbei, R.J., Zenios, S.A.: Robust Optimization of Large-Scale Systems. Oper. Res. 43, 264–281 (1995)
13. Nemhauser, G., Wolsey, L.: Integer and Combinatorial Optimization. John Wiley & Sons, Hoboken (1988)
14. Rappaport, T.S.: Wireless Communications: Principles and Practice, 2nd edn. Prentice Hall, Upper Saddle River (2001)
15. Watson, J.-P., Hart, W.E., Murray, R.: Formulation and optimization of robust sensor placement problems for contaminant warning systems. In: Buchberger, S.G., et al. (eds.) Proc. of WDSA 2006, ASCE, Cincinnati, USA (2006)

Compact Relaxations for Polynomial Programming Problems*

Sonia Cafieri[1], Pierre Hansen[2,4], Lucas Létocart[3],
Leo Liberti[4], and Frédéric Messine[5]

[1] Laboratoire MAIAA, Ecole Nationale de l'Aviation Civile, 7 av. E. Belin,
31055 Toulouse, France
sonia.cafieri@enac.fr

[2] GERAD, HEC Montreal, Canada
pierre.hansen@gerad.ca

[3] LIPN, Univ. de Paris Nord, Avenue J.B. Clément, 93430 Villetaneuse, France
lucas.letocart@lipn.univ-paris13.fr

[4] LIX, École Polytechnique, 91128 Palaiseau, France
liberti@lix.polytechnique.fr

[5] ENSEEIHT-IRIT, 2 rue Charles Camichel, BP 7122, F-31 071 Toulouse, France
frederic.messine@n7.fr

Abstract. Reduced RLT constraints are a special class of Reformulation-Linearization Technique (RLT) constraints. They apply to nonconvex (both continuous and mixed-integer) quadratic programming problems subject to systems of linear equality constraints. We present an extension to the general case of polynomial programming problems and discuss the derived convex relaxation. We then show how to perform rRLT constraint generation so as to reduce the number of inequality constraints in the relaxation, thereby making it more compact and faster to solve. We present some computational results validating our approach.

Keywords: polynomial, nonconvex, MINLP, sBB, reformulation, convex relaxation, RLT.

1 Introduction

We target Mixed-Integer Nonlinear Programming (MINLP) problems of the form:

$$\left.\begin{array}{rr}\min_x & f(x) \\ & g(x) \leq 0 \\ & Ax = b \\ & x^L \leq x \leq x^U \\ \forall\, i \in Z & x_i \in \mathbb{Z},\end{array}\right\} \qquad (1)$$

where $x, x^L, x^U \in \mathbb{R}^n$, $Z \subseteq \mathcal{N} = \{1, \ldots, n\}$, A is a full rank $m \times n$ matrix, $b \in \mathbb{R}^m$, $f : \mathbb{R}^n \to \mathbb{R}$ and $g : \mathbb{R}^n \to \mathbb{R}^{m'}$ are polynomial functions of x.

* We are grateful to Dr. Tatjana Davidović and Christoph Dürr for useful discussions. Financial support by grants: Digiteo Chair 2009-14D "RMNCCO", Digiteo Emergence 2009-55D "ARM" is gratefully acknowledged.

R. Klasing (Ed.): SEA 2012, LNCS 7276, pp. 75–86, 2012.

We describe an extension to polynomial programming of an existing automatic reformulation technique [1,2,3] called reduced Reformulation-Linearization Technique (rRLT). This technique was originally defined only for quadratic problems subject to linear equality constraints. It replaces some of the quadratic terms with suitable linear constraints, called rRLT constraints. These turn out to be a subset of the RLT constraints for quadratic programming [4]. The original RLT linearizes all quadratic terms in the problem and generates valid linear equation and inequality cuts by considering multiplications of bound factors (terms like $x_i - x_i^L$ and $x_i^U - x_i$) by linear constraint factors (the left hand side of a constraint such as $\sum_{j=1}^n a_j x_j - b \geq 0$ or $\sum_{j=1}^n a_j x_j - b = 0$). Since bound and constraint factors are always non-negative, so are their products: this way one can generate sets of valid problem constraints. An extension of the RLT to polynomial programming is described in [5], and to more general factorable programming problems in [6]. These results find their practical limitations in the extremely large number of adjoined constraints. Some heuristic techniques [4,7] were proposed to help filter out RLT constraints which are redundant. In the rRLT the presence of linear equality constraints in the original problem allows the generation of only those linear RLT constraints that are guaranteed to replace a set of quadratic terms.

We aim to improve performance of spatial Branch-and-Bound (sBB) algorithms targeted at nonconvex NLPs and MINLPs; in particular, the rRLT tightens the lower bound computed by solving a convex relaxation of (1) at each sBB node. We make two original contributions. First, we extend rRLT theory from quadratic to polynomial programs. Second, as rRLT constraint generation depends on an arbitrary choice (the basis of a certain matrix) we show how to choose this basis in such a way as to yield a more compact (i.e., fewer constraints) convex relaxation, denoted by rRLT-C; the rRLT-C relaxation may be weaker than the rRLT one, but experiments show that the loss in tightness is greatly offset by the gain in CPU time taken to solve it. We assume our polynomial programs to be dense up to Sect. 3.2 for simplicity, and deal with sparsity in Sect. 4.

Notationwise, we deal sometimes with indexed symbols which are scalars and indexed symbols which are vectors; in order to avoid ambiguities, we denote with boldface all indexed symbols indicating vectors. For example, w_{ij} is a scalar but $\mathbf{w}_{ij}$ is the vector $(w_{ij1}, \ldots, w_{ijn})$; in line with current optimization literature, we do not use boldface to indicate the name of a whole array, so w might be either a scalar or an array depending on the context.

The rest of this paper is organized as follows: Sect. 2 extends rRLT to polynomial programming. In Sect. 3 we discuss how to construct the rRLT-C compact relaxation. In Sect. 4 we address sparse polynomial programs. Sect. 5 discusses some computational experiments on randomly generated instances.

2 rRLT for Polynomial Programming

The results presented herein extend [1] to the general polynomial case. Let $Q = \{2, \ldots, q\}$. For each monomial $x_{j_1} \cdots x_{j_p}$ appearing in the original problem (1) where $p \in Q$, we define a finite sequence $J = (j_1, \ldots, j_p)$ and, consistent with the notation introduced by Sherali [5], consider defining constraints of the following form:

$$w_J = \prod_{\ell \le |J|} x_{j_\ell} \tag{2}$$

(for $|J| = 1$, i.e. $J = (j)$, we also define $w_J = x_j$). For all $p \in Q$, $J \in \mathcal{N}^p$ and any permutation π in the symmetric group S_p we have that $w_J = w_{\pi J}$ by commutativity. We therefore define an equivalence relation $\sim$ on $\mathcal{N}^p$ stating that for $J, K \in \mathcal{N}^p$, $J \sim K$ only if $\exists \pi \in S_p$ such that $J = \pi K$. We then consider the set of equivalence classes $\bar{\mathcal{N}}^p = \mathcal{N}^p/\!\sim$ to quantify over when indexing added variables w_J. In practice, we choose an equivalence class representative for each $J \in \bar{\mathcal{N}}^p$ which we also denote by J. With a slight abuse of notation, if $J' \in \bar{\mathcal{N}}^{p'}$ and $J'' \in \bar{\mathcal{N}}^{p''}$ such that $p' + p'' = p$ and (J', J'') is in the equivalence class represented by $J \in \bar{\mathcal{N}}^p$, we write $(J', J'') = J$. We also define, for all $p \in Q$, $\mathcal{M}_p = \bigcup\limits_{1<p'\le p} \bar{\mathcal{N}}^{p'}$ and $\mathcal{M}_p^1 = \bigcup\limits_{p' \le p-1} \bar{\mathcal{N}}^{p'}$.

We multiply the original linear constraints $Ax = b$ by all monomials $\prod\limits_{\ell \le p-1} x_{j_\ell}$ and replace them by the corresponding added variables $w_{(J',j)}$, where $J' \in \bar{\mathcal{N}}^{p-1}$. This yields the following rRLTS:

$$\forall p \in Q, J' \in \bar{\mathcal{N}}^{p-1} \quad A\,\mathbf{w}_{J'} = b w_{J'}, \tag{3}$$

where $\mathbf{w}_{J'} = (w_{(J',1)}, \ldots, w_{(J',n)})$. We then consider the companion system:

$$\forall p \in Q, J' \in \bar{\mathcal{N}}^{p-1} \quad A\,\mathbf{z}_{J'} = 0. \tag{4}$$

Since (4) is a linear homogeneous system, there is a matrix M such that the companion system is equivalent to $Mz = 0$, the columns of which are indexed by sequences in $\mathcal{M}_p$. We let $B \subseteq \mathcal{M}_p$ and $N \subseteq \mathcal{M}_p$ be index sets for basic and nonbasic columns of M. We define the following sets:

$$C = \{(x, w) \mid Ax = b \wedge \forall p \in Q, J \in \bar{\mathcal{N}}^p (w_J = \prod_{\ell \le |J|} x_{j_\ell})\} \tag{5}$$

$$\begin{aligned} R_N = \{(x, w) \mid Ax = b \wedge \forall p \in Q, J' \in \bar{\mathcal{N}}^{p-1} (A\,\mathbf{w}_{J'} = b w_{J'}) \wedge \\ \forall J \in N (w_J = \prod_{\ell \le |J|} x_{j_\ell})\}. \end{aligned} \tag{6}$$

Theorem 2.1. *For each partition B, N into basic and nonbasic column indices for the companion system $Mz = 0$, we have $C = R_N$.*

We remark that for this proof to hold, all possible nonlinear monomials must be present in the problem, which is generally not the case. We address this problem in Sect. 4. A different treatment of the essentially the same concepts, which only employs a bases of A instead of the (larger) companion system, was given in [8].

Replacing C with R_N for some nonbasis N effectively replaces some nonlinear monomial terms with linear constraints, and therefore contributes to simplify the problem. A convex relaxation for the reformulated problem is readily obtained by applying monomial convexification methods in the literature .

3 Compact Convex Relaxation

First, we remark that virtually no practical polynomial problem exhibits *all* possible nonlinear monomials. Let $\mathcal{M} = \mathcal{M}_n$ and $\mathcal{M}^1 = \mathcal{M}^1_n$. We introduce two sets: $\beta \subseteq \mathcal{M}$ indexing all nonlinear monomials appearing in the original problem (1) and $\beta' \subseteq \mathcal{M}$ indexing all nonlinear monomials appearing in (1) *and* the rRLTS (3). Reduced RLT constraints are likely to give rise to compact yet tight convex relaxations if $N \subsetneq \beta$, in view of the fact that, by Thm. 2.1, only monomials indexed by N need appear in the formulation — so the lower and upper relaxations to monomials outside N can be dropped. Furthermore, the proof of Thm. 2.1 also implies that the number of monomials that can be replaced is equal to the rank ρ of the rRLTS. If the original problem has few monomials, N might not be a proper subset of β, and in practice this occurrence is not rare. Limited to quadratic polynomials, we address this problem in [2]; in Sect. 4 we propose a technique to deal with sparse polynomial programs.

In this section we discuss a choice of N whereby the monomial relaxations that are dropped define "large volumes", and are therefore more likely to be dominated by the relaxations of monomials in N. Intuitively, this should yield a compact relaxation whose bound is not far from the normal rRLT relaxation.

3.1 Convexity Gap

Definition 3.1. *Consider a function $f : X \subseteq \mathbb{R}^n \to \mathbb{R}$. Let $\underline{f}(x)$ be a convex lower bounding function for f and $\bar{f}(x)$ be a concave upper bounding function for f. Then the set $\bar{S} = \{(x,w) \mid \underline{f}(x) \le w \le \bar{f}(x)\}$ is a convex relaxation of the set $S = \{(x,w) \mid w = f(x)\}$. We define the convexity gap $V(S)$ between S and $\bar{S}$ to be the volume of the set $\bar{S}$; namely,*

$$V(S) = \int_{x \in X} (\bar{f}(x) - \underline{f}(x))dx. \tag{7}$$

We denote the convexity gap for a quadratic term $x_i x_j$ with V_{ij}.

Convexity Gap for a Quadratic Term x_i^2. The convex envelope of the set $\xi = \{(x_i, w_{ii}) \mid w_{ii} = x_i^2, x_i^L \le x_i \le x_i^U\}$ (where $i \le n$) consists of the area between the function x_i^2 and the chord. The convexity gap of ξ is:

$$V_{ii} = \int_{x_i^L}^{x_i^U} \left((x_i^L)^2 + \frac{(x_i^U)^2 - (x_i^L)^2}{x_i^U - x_i^L}(x_i - x_i^L) - x_i^2 \right) dx = \frac{1}{6}(x_i^U - x_i^L)^3. \tag{8}$$

Convexity Gap for a Bilinear Term $x_i x_j$. For all $i < j \le n$, the convex envelope of the set $\{(x_i, x_j, w_{ij}) \mid w_{ij} = x_i x_j, x_i^L \le x_i \le x_i^U, x_j^L \le x_j \le x_j^U\}$ is a tetrahedron $\varDelta$ in $\mathbb{R}^3$ given by the McCormick inequalities [9,10]. The vertices of $\varDelta$ are: $(x_i^U, x_j^U, x_i^U x_j^U)$, $(x_i^U, x_j^L, x_i^U x_j^L)$, $(x_i^L, x_j^U, x_i^L x_j^U)$, $(x_i^L, x_j^L, x_i^L x_j^L)$. Let

$$\begin{aligned}
\mu &= x_i^L - x_i^U \\
\nu &= x_j^L - x_j^U \\
\xi &= x_i^L x_j^L - x_i^U x_j^U \\
\zeta &= x_i^L x_j^U - x_i^U x_j^L \\
\hat{D} &= \begin{vmatrix}
0 & 1 & 1 & 1 & 1 \\
1 & 0 & \nu^2(1+a^2) & \mu^2(1+c^2) & \mu^2+\nu^2+\xi^2 \\
1 & \nu^2(1+a^2) & 0 & \mu^2+\nu^2+\zeta^2 & \mu^2(1+d^2) \\
1 & \mu^2(1+c^2) & \mu^2+\nu^2+\zeta^2 & 0 & \nu^2(1+b^2) \\
1 & \mu^2+\nu^2+\xi^2 & \mu^2(1+d^2) & \nu^2(1+b^2) & 0
\end{vmatrix}.
\end{aligned}$$

The volume V_{ij} of $\varDelta$ can be computed using the Cayley-Menger formula in 3 dimensions [11], i.e. $V_{ij} = (x_i^L, x_i^U, x_j^L, x_j^U) = \frac{\sqrt{2\hat{D}}}{24}$.

Convexity Gap for a Multilinear Monomial. By [12], the convex under- and over-approximating envelopes of a multilinear monomial $x_1 \cdots x_p$ of degree p, are polyhedral. Therefore, the facet defining inequalities of the enveloping polytope can be computed using the 2^p polytope vertices $v^\ell = (x_1^{\ell_1}, \ldots, x_p^{\ell_p}, \prod_{j \le p} x_j^{\ell_j})$ in $\mathbb{R}^{n+1}$, for every p-sequence $\ell \in \{L, U\}^p$. We carry out this computation using the Porta software [13]: although its worst-case complexity is exponential in p, it is practically efficient for low values of p. Since our computational experiments only address problems up to degree 4, this methodological choice is appropriate. Standard methods to compute volumes of polytopes exist [14]. Our preliminary implementation uses the volume of the smallest bounding box — this will be changed later.

If J is the (ordered) sequence of p variable indices appearing in a multilinear monomial $\mu(x)$, we let V_J denote the convexity gap for $\mu(x)$.

Convexity Gap for a General Monomial. By associativity, for any $p \in Q$ and a given sequence $J \in \bar{\mathcal{N}}^p$ it is always possible to express the monomial $\mu(x) = \prod\limits_{\ell \le p} x_{j_\ell}$ as a product of multilinear factors $(x_{j_1} \cdots x_{j_2}) \cdots (x_{j_3} \cdots x_{j_4})$. After replacement by the appropriate added variable, the monomial is reduced to $w_{J_1} \cdots w_{J_2}$, where $J_1 = \{j_1, \ldots, j_2\}$ and $J_2 = \{j_3, \ldots, j_4\}$. Associativity can then be re-applied recursively. This allows us to use the results of the preceding sections to derive a convexity gap for $\mu(x)$. This approach, albeit simple, is similar, up to commutativity, to the standard reformulation exploited by sBB implementations [15,16,17,18,19] in view to obtain the convex relaxation of general monomials, and more specifically to the approach followed in [20] for quadrilinear monomials. Explicit monomial convex envelopes are also known for trilinear terms [21,22] and univariate terms of odd degree [23].

Since we already treated the multilinear case separately, we assume that $\mu(x)$ is *not* multilinear. In such cases, the recursive techniques described above yield *nonlinear* convex relaxations. Because our computational results only refer to *linear* relaxations, however, we take a simpler approach and use interval arithmetic [24] to compute an interval range $[\mu^L, \mu^U]$ such that $\mu^L \leq \mu(x) \leq \mu^U$ for all $x \in [x^L, x^U]$. If J is the (ordered) sequence of variable indices appearing in the monomial $\mu(x)$, we compute V_J as the volume of the bounding box $[x^L, x^U] \times [\mu^L, \mu^U]$.

3.2 Choosing a Good Basis for the Companion System

Let B, N be the basic/nonbasic sets of column indices of the companion system (4), which we write in this section as $Mz = 0$, or, equivalently, as $M_B z_B + M_N z_N = 0$. As shown in Sect. 2, the elements of B, N are sequences $J \in \mathcal{M}$. For $S \subseteq \mathcal{M}$ and $p \in Q$ we define $V^{S,p} = \sum\limits_{\substack{J \in S \\ |J|=p}} V_J$ and $V^S = \sum\limits_{p \in Q} V^{S,p}$. If, for all $p \in Q$, $V^{N,p} < V^{\beta,p}$ then the total convexity gap of R_N is smaller than that of C. Thus, we aim to find N such that $V^{N,p}$ is minimized, or equivalently, to find B such that $V^{B,p}$ is maximized for all $p \in Q$. This yields the multi-objective problem:

$$\left.\begin{array}{c} \forall p \in Q \quad \max V^{B,p} \\ M_B \text{ is a basis of (4)} \end{array}\right\} \tag{9}$$

Next, we show that (9) is equivalent to a single-objective problem.

Consider a block diagonal $\bar{m} \times \bar{n}$ matrix $\bar{A}$ with r blocks A_s (each a rectangular, full-rank $m_s \times n_s$ matrix for each $s \leq r$) having a basis indexed by the set $\bar{B} \subseteq \{1, \ldots, \bar{n}\}$. For all $s \leq r$ let α_s be the set of column indices corresponding to the submatrix A_s of $\bar{A}$. For a matrix T let span(T) be the space spanned by the columns of T.

Lemma 3.2. *Let $i, j \leq \bar{n}$ be such that $i \in \bar{B}, j \notin \bar{B}$ and π be the swap (i, j). If $i \in \alpha_q$ and $j \in \alpha_t$ with $q \neq t$ then the columns of $\bar{A}$ indexed by $\pi\bar{B}$ do not form a basis.*

We remark that M is a block-diagonal matrix. Instead of showing a formal proof of this fact, which would be long and tedious, we exhibit an example for the case of polynomials of degree at most 3. Example 3.3 does not exploit any specific property of the given matrix, and therefore appropriately illustrates what happens in the general case.

Example 3.3. The companion system (4) $Mz = 0$ derived from the system $Ax = b$ with $A = \begin{pmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \end{pmatrix}$, $x \in \mathbb{R}^3$ with $Q = \{2, 3\}$ is:

$$\left(\begin{array}{cccccc|cccccccccc}
a_1 & a_2 & a_3 \\
a_4 & a_5 & a_6 \\
 & a_1 & & a_2 & a_3 \\
 & a_4 & & a_5 & a_6 \\
 & & a_1 & & a_2 & a_3 \\
 & & a_4 & & a_5 & a_6 \\ \hline
 & & & & & & a_1 & a_2 & a_3 \\
 & & & & & & a_4 & a_5 & a_6 \\
 & & & & & & & a_1 & & a_2 & a_3 \\
 & & & & & & & a_4 & & a_5 & a_6 \\
 & & & & & & & & a_1 & & a_2 & a_3 \\
 & & & & & & & & a_4 & & a_5 & a_6 \\
 & & & & & & & & & a_1 & & & a_2 & a_3 \\
 & & & & & & & & & a_4 & & & a_5 & a_6 \\
 & & & & & & & & & & a_1 & & & a_2 & a_3 \\
 & & & & & & & & & & a_4 & & & a_5 & a_6 \\
 & & & & & & & & & & & a_1 & & & a_2 & a_3 \\
 & & & & & & & & & & & a_4 & & & a_5 & a_6
\end{array}\right)
\begin{pmatrix} z_{11} \\ z_{12} \\ z_{13} \\ z_{22} \\ z_{23} \\ z_{33} \\ z_{111} \\ z_{112} \\ z_{113} \\ z_{122} \\ z_{123} \\ z_{133} \\ z_{222} \\ z_{223} \\ z_{233} \\ z_{333} \end{pmatrix} = 0$$

The number of monomials of n variables of degree exactly p is given in [25] as $\binom{p+n-1}{p}$. From this, it follows that M has $m \sum_{p \in Q} \binom{p+n-2}{p-1}$ rows and $\sum_{p \in Q} \binom{p+n-1}{p}$ columns.

Theorem 3.4. *Any solution B of (9) maximizing V^B also maximizes $V^{B,p}$ for all $p \in Q$.*

The single-objective problem $\max\{v^B \mid M_B \text{ a basis of } (4)\}$ has a matroidal structure and can therefore be solved using a greedy algorithm.

Evidently, the technique based on compact rRLT constraints is not significant whenever the bounds are the same across all decision variables, because in this case all the V_J's are equal. This, however, is rarely true if bound tightening techniques [26,18,19,27] are used as a preprocessing step; and it is *never* true during sBB with rectangular partitioning schemes, as the variable ranges are partitioned at each node. We also emphasize that the remark given in [8] p. 11 is also valid in our setting: for polynomial degrees ≥ 3, the proposed convex relaxation might not be monotonically increasing w.r.t. following branches of the sBB tree down from the root, thus preventing the sBB from converging. This can be fixed at each node by adjoining those bound factor inequalities (derived from the multiplication of different bound factors) that involve the branching variable.

4 Dealing with Sparsity

Polynomial problems are rarely dense; this might prevent the set N, introduced in Sect. 2 to index the added variables corresponding to nonbasic columns of the companion system (4), from being a subset of β, the set indexing the added variables corresponding to all monomials appearing in (1). We deal with this possibility by looking for a subset ρ of rows of $Ax = b$ to be multiplied by a subset σ of added variables indexed by $\mathcal{M}^1$. The rRLTS (3) and its companion system (4) are derived in the same way as for dense polynomial problems.

Theoretically, we should require $N \subsetneq \beta$: as remarked in Sect. 3, the rRLTS is then likely to give rise to compact yet tight convex relaxations. In practice, in sparse problems, this requirement will often force ρ or σ to be empty, which means that the reformulation does not take place. Following the principle that every equation corresponds to one fewer degree of freedom, we aim to find ρ, σ such that the size of the sparse rRLTS obtained by multiplying rows in ρ by added variables in σ exceeds the number of new monomials (i.e. monomials not in β) generated by these multiplications. This flexibility in the choice of ρ, σ has a trade-off in terms of relaxation compactness. In order for Thm. 2.1 to hold, the quantifications $J \in \bar{\mathcal{N}}^p$ in (5) and $J' \in \bar{\mathcal{N}}^{p-1}$ in (6) should be replaced by $J \in \tau$, $J' \in \sigma$ respectively, where $\tau = \{(J', j) \mid J' \in \sigma \ (|J'| = p) \wedge j \in \mathcal{N}\}$. Since $\sigma \subseteq \mathcal{M}^1$, fewer monomials are replaced by the rRLTS, yielding a relaxation which might not be much more compact than the standard RLT relaxation.

We formalize the problem of finding suitable ρ, σ by considering a bipartite graph that represents the incidence of monomials indexed by $\mathcal{M}$ in products of rows $a_i x = b_i$ of $Ax = b$ multiplied by monomials indexed by $\mathcal{M}^1$. Let $U = \{(i, J') \mid i \leq m \wedge J' \in \mathcal{M}^1\}$ and define a set E such that, for $(i, J') \in U$ and $J \in \mathcal{M} \smallsetminus \beta$, $\{(i, J'), J\} \in E$ if there is $j \in \mathcal{N}$ such that $a_{ij} \neq 0$ and $(J', j) = J$. Consider the bipartite graph $G = (U, \mathcal{M} \smallsetminus \beta, E)$: we want to find an induced subgraph $G' = (U', V', E')$ of G, with $U' \subseteq U$ and $V' \subseteq \mathcal{M} \smallsetminus \beta$, such that: $|U'|$ is maximum, $|U'| > |V'|$ and $V' = N_E(U') = \{v \in \mathcal{M} \smallsetminus \beta \mid \exists u \in U' \ (\{u, v\} \in E)\}$. We define this problem using a Mathematical Programming (MP) formulation: we employ binary variables $u_{i,J'}$ for all $(i, J') \in U$ and v_J for all $J \in \mathcal{M} \smallsetminus \beta$:

$$\left.\begin{array}{rrl} & \max \displaystyle\sum_{(i,J')\in U} u_{i,J'} & \\ & \displaystyle\sum_{(i,J')\in U} u_{i,J'} & \geq \displaystyle\sum_{J\notin\beta} v_J + 1 \\ \forall \{(i,J'),J\} \in E & v_J & \geq u_{i,J'} \\ & u & \in \{0,1\}^{|U|} \\ & v & \in \{0,1\}^{|\mathcal{M}\smallsetminus\beta|}. \end{array}\right\} \qquad (10)$$

We can then define $\rho = \{i \leq m \mid \exists J' \in \mathcal{M}^1 \ (u_{i,J'} = 1)\}$ and $\sigma = \{J' \in \mathcal{M}^1 \mid \exists i \leq m \ (u_{i,J'} = 1)\}$. This is a Binary Linear Program (BLP), so it cannot be solved in polynomial time with standard MILP technology (i.e. using a Branch-and-Bound algorithm).

4.1 Bipartite Matching Based Algorithm for (10)

We propose an efficient algorithm based on bipartite matching for solving (10). With respect to a matching M of G, a vertex of G is *exposed* if it is incident to an edge which is not in M. For a subset of edges $F \subseteq E$, let $\mathsf{U}(F) = \{u \in U \mid \exists e \in$

$F\ (u \in e)\}$ and $\mathsf{V}(F) = \{v \in \mathcal{M} \smallsetminus \beta \mid \exists e \in F\ (v \in e)\}$. If M is a maximum matching of G such that there exists an exposed vertex $u \in U \smallsetminus \mathsf{U}(M)$, the search for an augmenting path from u which is alternating with respect to M will fail by maximality of M [28]. A *dilation* rooted in u with respect to a maximum matching M is a maximal simple alternating path p_u (seen as a sequence of edges in E) in G, from u to a vertex $u' \in \mathsf{U}(M)$, whose even-indexed edges are in M and odd-indexed edges are in $E \smallsetminus M$. Dilations are the certificates used in cardinality bipartite matching algorithms to prove that the current matching is optimal; informally, they certify the failure to find an augmenting alternating path to increase the cardinality of the current matching in the classical bipartite matching algorithm (see Fig. 10-3 in [29]).

Lemma 4.1. *For a dilation p_u from u in G w.r.t. a maximum matching M, we have $|\mathsf{U}(p_u)| = |\mathsf{V}(p_u)| + 1$.*

Lemma 4.2. *If p, p' are different dilations in E, then $|\mathsf{U}(p \cup p')| > |\mathsf{V}(p \cup p')|$.*

In order to deal with the case of isolated vertices, if $u \in U$ is isolated, then it is exposed w.r.t. the empty matching, and any empty path p_u rooted at u is a dilation; in this case, with a slight abuse of notation, we define $\mathsf{U}(p_u) = \{u\}$. A *dilation set* is the set of edges in all dilations rooted at u with respect to M; dilation sets can be found in polynomial time using breadth first search (bfs) from u (see [29], Sect. 10.2). By Lemma 4.2 above, dilation sets P are such that $|\mathsf{U}(P)| > |\mathsf{V}(P)|$.

Algorithm 1. Matching-based algorithm for solving (10)

```
Require: A bipartite graph $G = (U, \mathcal{M} \smallsetminus \beta, E)$
Ensure: A nontrivial subgraph $G' = (U', V', E)$ solving (10), or $\varnothing$ if none exists
Let $G' = \varnothing$
while $|U| > 0$ do
    Let $M \subseteq E$ be a maximum matching in $G$
    if $|\mathsf{U}(M)| < |U|$ then
        Find $u \in U$ exposed and a corresponding dilation set $P_u$
        Let $H = (\mathsf{U}(P_u), \mathsf{V}(P_u), P_u)$
        Update $G' \leftarrow G' \cup H$
        Update $G \leftarrow G \smallsetminus H$
    else
        break
    end if
end while
return $G'$
```

Proposition 4.3. *In polynomial time, Alg. 1 finds a subgraph $G' = (U', V', E')$ of G such that $|U'|$ is maximum, $|U'| > |V'|$ and $V' = N_E(U')$, or determines that no such subgraph exists.*

5 Computational Results

The best practical indication of the importance of rRLT techniques for linear equality constrained polynomial programming is best observed by the computational results given in [2], where the presence or absence of rRLT yields differences of up to five orders of magnitude on a class of pooling problems (these are sparse polynomial problems of degree 2) from the oil industry. Those results are limited to quadratic programs and employ a restricted version of the results for sparsity given in Sect. 4. We are currently in the process of extending our code to deal with polynomial programming, so in this paper we only show empirically that the compact rRLT-C formulation generally takes less time to solve and yields bounds that are not much worse than those given by the rRLT formulation.

We generate two sets of random polynomial programming instances of degree 2,3 and 4 with varying numbers of variables and linear equality constraints (all the monomials are in the objective function, weighted by random scalars). Set 1 consists of 10 instances of degrees 2 and 3 with random variable ranges whose widths follow a Gaussian distribution. Set 2 consists of 8 instances of degrees 3 and 4 with random variable ranges whose widths follow a superposition of two Gaussian distributions ($n/2$ ranges have width of order 1, the other $n/2$ have width of order 10); set 2 is designed to simulate the typical sBB node after a few levels of branching, where some of the variable ranges have become small whereas others are still at their original bounds.

For all these instances we construct and solve a linear relaxation with no rRLT constraints (column labeled “simple” in Table 1), the rRLT linear relaxation (column labeled “rRLT”) and the rRLT-C linear relaxation (column labeled “rRLT-C”). We recall that rRLT-C is like rRLT without the constraints relaxing monomial terms corresponding of basic columns of the companion system (4). We record bound value and CPU time. Notice some of the generated instances are infeasible: this is consistent with the fact that in a typical sBB search tree some of the nodes represent infeasible subproblems. Since the infeasibility is determined by the linear relaxation, performance on infeasible LPs is also an important factor. These results were obtained using CPLEX 11 [30] on a Pentium Xeon 2.4GHz CPU with 8 GB of RAM running Linux. The results in Table 1 show that rRLT brings considerable benefits to bound tightness within polynomial optimization, and that the (significant) CPU time reduction yielded by rRLT-C is not offset by an excessive loss in bound quality with respect to rRLT: **the cumulative bound worsening is 0.07% against a time improvement of nearly 40%**. The CPU time taken by the simple relaxation is of course much lower than those of the rRLT relaxations, but already for the quadratic case it was shown in [2] that this time difference is not sufficient to offset the benefits of the bound improvement — hence the corresponding values do not appear in Table 1. We remark that rRLT-C does not always yield better CPU time results. This is simply because the relationship between CPU time and number of constraints in solving an LP is neither regular nor monotonic.

Table 1. Comparing bound strength and CPU time for linear relaxations

instance					simple	rRLT		rRLT-C	
name	*p*	*n*	*m*	*q*	*bound*	*bound*	*time*	*bound*	*time*
1	1	20	10	2	2864.8	**2982.63**	0.20	**2982.63**	**0.17**
2	1	30	10	2	5286.23	**5517.85**	2.56	**5517.85**	**1.29**
3	1	10	3	3	48.6115	**478.184**	**0.25**	**478.184**	0.30
4	1	10	3	3	infeas	infeas	0.23	infeas	**0.22**
5	1	20	3	3	infeas	infeas	14.81	infeas	**12.92**
6	1	10	5	3	infeas	infeas	0.11	infeas	**0.10**
7	1	10	5	3	infeas	infeas	0.16	infeas	0.16
8	1	20	5	3	infeas	infeas	**29.01**	infeas	31.41
9	1	10	7	3	infeas	infeas	0.19	infeas	**0.15**
10	1	20	7	3	infeas	infeas	65.22	infeas	**46.98**
11	2	10	3	3	130.693	**1546.4**	0.20	1542.39	**0.13**
12	2	10	3	3	18.8459	**772.417**	0.09	**772.417**	0.09
13	2	15	5	3	17.3797	**701.723**	**10.13**	701.588	11.03
14	2	16	8	3	infeas	infeas	488.05	infeas	**287.38**
15	2	7	2	4	infeas	infeas	0.17	infeas	0.17
16	2	8	3	4	47.4445	**3468.56**	1.26	3458.4	**0.85**
17	2	10	3	4	26.2698	**4038.69**	250.88	**4038.68**	**131.95**
18	2	12	3	4	56.9232	**13127.5**	166.62	13118.6	**109.95**

References

1. Liberti, L.: Linearity embedded in nonconvex programs. Journal of Global Optimization 33(2), 157–196 (2005)
2. Liberti, L., Pantelides, C.: An exact reformulation algorithm for large nonconvex NLPs involving bilinear terms. Journal of Global Optimization 36, 161–189 (2006)
3. Liberti, L.: Compact linearization of binary quadratic problems. 4OR 5(3), 231–245 (2007)
4. Sherali, H., Alameddine, A.: A new reformulation-linearization technique for bilinear programming problems. Journal of Global Optimization 2, 379–410 (1992)
5. Sherali, H., Tuncbilek, C.: A global optimization algorithm for polynomial programming problems using a reformulation-linearization technique. Journal of Global Optimization 2, 101–112 (1991)
6. Sherali, H., Wang, H.: Global optimization of nonconvex factorable programming problems. Mathematical Programming 89, 459–478 (2001)
7. Sherali, H., Tuncbilek, C.: New reformulation linearization/convexification relaxations for univariate and multivariate polynomial programming problems. Operations Research Letters 21, 1–9 (1997)
8. Sherali, H., Dalkiran, E., Liberti, L.: Reduced RLT representations for nonconvex polynomial programming problems. Journal of Global Optimization 52, 447–469 (2012)
9. McCormick, G.: Computability of global solutions to factorable nonconvex programs: Part I — Convex underestimating problems. Mathematical Programming 10, 146–175 (1976)

10. Al-Khayyal, F., Falk, J.: Jointly constrained biconvex programming. Mathematics of Operations Research 8(2), 273–286 (1983)
11. Jiao, Y., Stillinger, F., Torquato, S.: Geometrical ambiguity of pair statistics I. point configurations. Technical Report 0908.1366v1, arXiv (2009)
12. Rikun, A.: A convex envelope formula for multilinear functions. Journal of Global Optimization 10(4), 425–437 (1997)
13. Christof, T., Löbel, A.: The porta manual page. Technical Report v. 1.4.0, ZIB, Berlin (1997)
14. Avis, D.: User's Guide for lrs (2009)
15. Smith, E., Pantelides, C.: A symbolic reformulation/spatial branch-and-bound algorithm for the global optimisation of nonconvex MINLPs. Computers & Chemical Engineering 23, 457–478 (1999)
16. Adjiman, C., Dallwig, S., Floudas, C., Neumaier, A.: A global optimization method, αBB, for general twice-differentiable constrained NLPs: I. Theoretical advances. Computers & Chemical Engineering 22(9), 1137–1158 (1998)
17. Sahinidis, N., Tawarmalani, M.: BARON 7.2.5: Global Optimization of Mixed-Integer Nonlinear Programs, User's Manual (2005)
18. Liberti, L.: Writing global optimization software. In: Liberti, L., Maculan, N. (eds.) Global Optimization: from Theory to Implementation, pp. 211–262. Springer, Heidelberg (2006)
19. Belotti, P., Lee, J., Liberti, L., Margot, F., Wächter, A.: Branching and bounds tightening techniques for non-convex MINLP. Optimization Methods and Software 24(4), 597–634 (2009)
20. Cafieri, S., Lee, J., Liberti, L.: On convex relaxations of quadrilinear terms. Journal of Global Optimization 47, 661–685 (2010)
21. Meyer, C., Floudas, C.: Trilinear monomials with positive or negative domains: Facet s of the convex and concave envelopes. In: Floudas, C., Pardalos, P. (eds.) Frontiers in Global Optimization, pp. 327–352. Kluwer Academic Publishers, Amsterdam (2003)
22. Meyer, C., Floudas, C.: Trilinear monomials with mixed sign domains: Facets of the convex and concave envelopes. Journal of Global Optimization 29, 125–155 (2004)
23. Liberti, L., Pantelides, C.: Convex envelopes of monomials of odd degree. Journal of Global Optimization 25, 157–168 (2003)
24. Moore, R., Kearfott, R., Cloud, M.: Introduction to Interval Analysis. SIAM, Philadelphia (2009)
25. Katzman, M.: Counting monomials. Journal of Algebraic Combinatorics 22, 331–341 (2005)
26. Messine, F., Lagouanelle, J.: Enclosure methods for multivariate differentiable functions and application to global optimization. Journal of Universal Computer Science 4(6), 589–603 (1998)
27. Belotti, P., Cafieri, S., Lee, J., Liberti, L.: Feasibility-Based Bounds Tightening via Fixed Points. In: Wu, W., Daescu, O. (eds.) COCOA 2010, Part I. LNCS, vol. 6508, pp. 65–76. Springer, Heidelberg (2010)
28. Berge, C.: Two theorems in graph theory. Proceedings of the National Academy of Science of the U.S. 43, 842–844 (1957)
29. Papadimitriou, C., Steiglitz, K.: Combinatorial Optimization: Algorithms and Complexity. Dover, New York (1998)
30. ILOG: ILOG CPLEX 11.0 User's Manual. ILOG S.A., Gentilly, France (2008)

Relaxations of Multilinear Convex Envelopes: Dual Is Better Than Primal

Alberto Costa and Leo Liberti

LIX, École Polytechnique, 91128 Palaiseau, France
{costa,liberti}@lix.polytechnique.fr

Abstract. Bilinear, trilinear, quadrilinear and general multilinear terms arise naturally in several important applications and yield nonconvex mathematical programs, which are customarily solved using the spatial Branch-and-Bound algorithm. This requires a convex relaxation of the original problem, obtained by replacing each multilinear term by appropriately tight convex relaxations. Convex envelopes are known explicitly for the bilinear case, the trilinear case, and some instances of the quadrilinear case. We show that the natural relaxation obtained using duality performs more efficiently than the traditional method.

Keywords: Global optimization, MINLP, mathematical programming.

1 Introduction

The general multilinear term is given by:

$$w(x) = x_1 \cdots x_k \tag{1}$$

for some $k \in \mathbb{N}$, and is possibly the most common nonlinear term occurring naturally in Mathematical Programming (MP) applications. As the need arises, we might also write (1) as $w(x) = x_{j_1} \cdots x_{j_k}$ with $J = \{j_1, \ldots, j_k\}$, and let $W_J = \{(x, w_J) \mid w_J = \prod_{j \in J} x_j \wedge x \in [x^L, x^U]\}$. The bilinear case is shown in Fig. 1. We let P be the set of vertices of the hyperrectangle $[x^L, x^U]$ and P_W be the lifting of P in the space spanned by (x, w_J), where, for each point $\bar{x} \in P$, the corresponding point in P_W is obtained by setting $w_J = w(\bar{x})$.

Convex envelopes for multilinear terms are available explicitly in function of x^L, x^U for $k = 2, 3$ and partly $k = 4$. Such envelopes consist of sets of constraints to be adjoined to the MP formulation. We argue in this paper that formulations obtained this way (called *primal relaxations*) are larger and less accurate than those obtained using a dual representation of such envelopes (called *dual relaxations*), i.e. the convex combination of points in P_W. One further advantage of these dual envelopes is that they are the same for all k and need no special case-by-case treatment.

1.1 Contributions

The relaxations for multilinear MPs proposed in this paper, which are based on duality, are a simple application of ideas which have been present in LP and

R. Klasing (Ed.): SEA 2012, LNCS 7276, pp. 87–98, 2012.

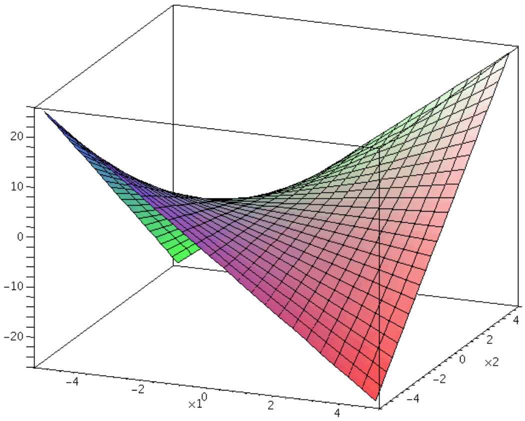


Fig. 1. The bilinear surface $w(x_1, x_2) = x_1 x_2$

MILP theory for a long time. They constitute an original contribution insofar as they have never been systematically tested computationally in the context of multilinear terms. This contribution would be negligible, had we not found empirically that dual relaxations provide a far superior way of relaxing multilinear MPs than "traditional (primal) relaxations". The fundamental purpose of this paper is to convey an important message: *it is possible that, until now, multilinear terms have been relaxed in the wrong way.* On the other hand, we think that the compact and elegant formulation of dual relaxations might provide a successful tool for future theoretical research: the primal relaxation involves remarkably different formulations for each value of k and it is difficult to see how it can be exploited in a uniform way.

We remark that duality has been used in the context of multilinear relaxations in [6]. The authors exploited in several ways the same dual relaxations we propose here. The term-wise computational comparison we perform, however, which we feel is so important to convey the message above simply, clearly and unequivocally, is absent from their treatment.

1.2 Applications

Several applications involve bilinear products between binary and continuous variables that model situations where a continuous variable takes different values depending on whether a certain boolean variable is 0 or 1 [35]. In pooling and blending problems [1,4,11,14,17,29], bilinear products ($k = 2$) occur whenever x_1 represents a (dimensionless) percentage and x_2 an oil flow in a pipe. The Hartree-Fock Problem [24] minimizes a quartic energy expression (involving quadrilinear terms) subject to some orthogonality constraints (involving bilinear terms). The Molecular Distance Geometry Problem [23] involves bilinear or quadrilinear terms depending on which formulation is used. General multilinear terms involving continuous variables occur in multilinear least-squares problems [30]. In general, such products occur over bounded variables: most applications require the variables $x = (x_1, \ldots, x_k)$ to be bounded to the hyperrectangle $[x^L, x^U]$, where $x^L = (x_1^L, \ldots, x_k^L)$ and $x^U = (x_1^U, \ldots, x_k^U)$. We remark, however, that there exists an application from code debugging [15,25] exhibiting bilinear terms $x_1 x_2$ where $x_1 \in \{0, 1\}$ and x_2 must be unbounded for the model to be correct (such variables are used to ensure that loops terminate whenever no upper bound is explicitly known for the loop counter).

1.3 Exact Linearizations

It was observed in [13,16] that if $k = 2$ and $x_1, x_2 \in \{0, 1\}$, then $w(x)$ can be replaced by an added variable $w_{12} \in [0, 1]$ whilst the *Fortet inequalities* are adjoined to the model:

$$w_{12} \leq x_1, \quad w_{12} \leq x_2, \quad w_{12} \geq x_1 + x_2 - 1. \tag{2}$$

It is easy to show that this reformulation is an exact linearization [21,22] of the original bilinear program.

Whenever $x \in [x^L, x^U]$ and at least $k-1$ variables out of k are constrained to be integer, the corresponding multilinear term can be linearized exactly. Each general integer variable is replaced by an aggregation of binary variables (for example choosing the value taken by the original integer variable), and the original multilinear term $w(x)$ is replaced by a sum of multilinear terms with at least $k-1$ binary variables. A sequence of $k-1$ Fortet's linearizations will then yield a Mixed-Integer Linear Programming (MILP) formulation of the original multilinear term.

1.4 Products of Continuous Variables

Whenever at least 2 variables in a multilinear term are continuous, exact linearizations are in general no longer possible, and one must resort to solution techniques for nonconvex programs, such as the spatial Branch-and-Bound (sBB) algorithm [2,8,12,20,32,33]. This involves repeatedly solving the original problem and a convex relaxation thereof over appropriate sets of ranges $[x^L, x^U]$. The relaxation is obtained by replacing each multilinear term with an added variable w_J and adjoining some constraints to the formulation which define a convex relaxation of W_J. In general, the tighter these relaxations are, the more efficient the sBB will be. This has spawned a growing interest in finding constraints which define the convex and concave *envelopes* $\hat{w}(x)$ and $\breve{w}(x)$ of multilinear terms. By definition, the set

$$\breve{W}_J = \{(x, w_J) \mid w_J \geq \hat{w}(x) \wedge w_J \leq \breve{w}(x) \wedge x \in [x^L, x^U]\} \tag{3}$$

is the *convex hull* of the set W_J. With a slight abuse of notation, the constraints on w_J appearing in the definition of $\breve{W}_J$ are also called *convex envelopes* of the multilinear terms.

2 Convex Envelopes of Multilinear Terms

It was shown in [31] that the convex envelopes of multilinear terms are *vertex polyhedral* [34], i.e. $\breve{W}_J$ is a polyhedron having P_W as vertex set. This makes it possible to write the convex envelopes of multilinear terms by means of linear constraints.

2.1 McCormick's Inequalities

Figure 2 shows the lower convex (left) and upper concave envelopes for the bilinear term $x_1 x_2$, each consisting of two linear constraints. The corresponding constraints:

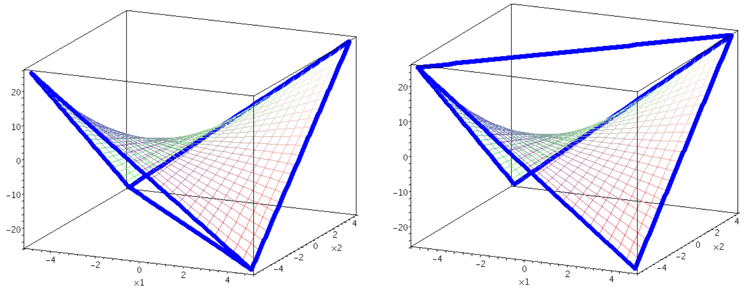


Fig. 2. Lower convex (left) and upper concave (right) envelopes for the bilinear term

$$
\begin{aligned}
w_{12} &\geq x_1^L x_2 + x_2^L x_1 - x_1^L x_2^L && (4)\\
w_{12} &\geq x_1^U x_2 + x_2^U x_1 - x_1^U x_2^U && (5)\\
w_{12} &\leq x_1^L x_2 + x_2^U x_1 - x_1^L x_2^U && (6)\\
w_{12} &\leq x_1^U x_2 + x_2^L x_1 - x_1^U x_2^L, && (7)
\end{aligned}
$$

called *McCormick inequalities*, were first described in [26] and later shown to be envelopes in [3].

The McCormick inequalities are expressed explicitly in terms of x^L, x^U, and are therefore referred to as *explicit envelopes*. By contrast, there exists software, such as PORTA [10] (which implements the Fourier-Motzkin algorithm), which, given specific values for x^L, x^U, is able to write the corresponding constraints for the convex envelopes of the points in P_W. Finding the explicit envelopes of the multilinear term for each k is of practical interest because calling PORTA to relax each multilinear term would be inefficient if there are several of them; and ever since McCormick's seminal paper, it has been a long-standing open question. The matter is settled in general for the case where $[x^L, x^U] = [0, 1]$ [31]; but since the use of such envelopes in the sBB algorithm implies that the bounds change at each node, this result may at best be useful only at the root node.

2.2 Meyer-Floudas Inequalities

Significant progress was made with Meyer and Floudas' work [27,28], who were able to write the explicit envelopes for the trilinear term $w(x) = x_1x_2x_3$. Their exact form depends on the relative sign of the variable bounds x^L, x^U. The paper [27] discusses 6 cases where the bound signs are equal (each case giving rise to 12 inequalities), whereas the other 9 cases are discussed in [28]. Several of these cases also involve checking nontrivial bound relations. Although Meyer and Floudas' results are conceptually simple to apply (it suffices to establish which is the case at hand, and adjoin the corresponding inequalities to the MP), the inequalities themselves are way more involved than McCormick's, and it is very easy to make mistakes when integrating them in a computer program.

Worst of all, however, is the fact that some coefficients appearing in Meyer-Floudas inequalities involve nontrivial floating point operations. For example, the coefficient of x_1 in [28, Case 3.5, p. 133] is $\frac{x_1^U x_2^U x_3^L - x_1^L x_2^L x_3^L - x_1^U x_2^U x_3^U + x_1^U x_2^L x_3^U}{x_1^U - x_1^L}$. As is well known, floating point additions and subtractions are error-prone [19, 4.2.1]. This will yield an inaccurate constraint representation of $\breve{W}_J$; to make things worse, the simplex method will identify optimal solutions at the vertices of the polyhedron rather than at the interior, which implies that this inaccuracy will impact the optimal solution. In particular, if variables are constrained to be integer, a feasible integer solution on or near the vertex of the polyhedron might be deemed infeasible. However, this can be avoided when using PORTA, which uses exact rational arithmetic.

By contrast, each coefficient of the the McCormick inequalities ($k = 2$) only involves floating point multiplication, which is a much safer operation.

2.3 Quadrilinear Terms

One of us (LL) has often heard Prof. C. Floudas state, at various Global Optimization conferences, that "the explicit envelopes of the quadrilinear terms haven't been found yet" to entice research in that direction. Accordingly, we undertook some effort in that direction in the past few years; although we failed to settle the question for $k = 4$, we managed to show how to choose the associative expression for $x_1x_2x_3x_4$ yielding the tightest convex relaxation [7,9], and we extended this result to associative expressions of general sequences of functions.

Very recently, Ms. S. Balram of the National University of Singapore (supervised by Prof. Karimi) continued the "race" towards multilinear envelopes for higher k: her M.Sc. thesis [5] includes 44 inequalities for the simplest of the quadrilinear cases (all bounds in the nonnegative orthant). That thesis does not mention how many cases there will be in total for $k = 4$, but several coefficients of this simplest case involve even more floating point additions and subtractions than the Meyer-Floudas' inequalities, and are therefore expected to yield inaccurate formulations. As for the trilinear case, when integer variables are involved, some feasible solutions might be incorrectly deemed infeasible.

2.4 Critique

From the cases $k = 3$ and $k = 4$ it appears clear that the explicit form of the inequalities describing $\breve{W}_J$, in function of x^L, x^U, considerably increases in complexity (from the point of view of floating point additions and subtractions) as k increases, thereby causing numerical instability. But this is not all: the number of such inequalities, even when they are found explicitly with PORTA, also increases, thereby yielding ever more sizable formulations. While it is known that this number increases as $O(2^k)$, the first column of Table 1 suggests that the increase is more like $O(k2^k)$. Lastly, finding explicit envelopes of multilinear terms for each separate value of k lacks elegance. The Meyer-Floudas inequalities required two papers and 15 separate cases, each with its own proof. However, one redeeming feature is that they only involve the primal variables of the original formulation.

In the remainder of this paper, we shall propose *dual envelopes*: these are derived in a natural way using well-known duality theory, they hold for each k, and yield more compact, accurate and numerically stable formulations. We shall henceforth refer to the convex envelopes presented in this section as *primal envelopes.*

3 Dual Envelopes

The fact that the envelopes of multilinear terms are vertex polyhedral immediately suggests the following dual approach: express a point in $\breve{W}_J$ as the convex combination of the set P_W of extreme points of $\breve{W}_J$. We look for a vector λ of 2^k nonnegative Lagrange multipliers such that:

$$x = \sum_{i\le 2^k} \lambda_i p_i \quad \wedge \quad \sum_{i\le 2^k} \lambda_i = 1,$$

where $P_W = \{p_1, \dots, p_{2^k}\} \subseteq R^{k+1}$. Now all that remains to do, in order to make (3) explicit envelopes, is to express the p_i's in function of x^L, x^U. To this aim, we define two parameter sequences $d \in \{0,1\}^{k2^k}$ and $b : \{0,1\}^k \to P_W$. Each d_{ij} is either 0 or 1 according as to whether the j-th component of p_i is a lower or upper bound, and $b_j(d_{ij})$ returns the correct component:

$$\forall i \le 2^k \qquad d_i = (d_{ij} | j \le k) = \left(\left\lfloor \frac{i-1}{2^{k-j}} \right\rfloor \bmod 2 \mid j \le k \right) \tag{8}$$

$$\forall j \le k \qquad b_j(0) = x_j^L \ \wedge \ b_j(1) = x_j^U. \tag{9}$$

We relax the k-linear term $w(x) = x_1 \cdots x_k$ as follows. We add 2^k new nonnegatively constrained variables $\lambda_i \ge 0$ (for $i \le 2^k$) and $k + 1$ new constraints:

$$\forall j \le k \quad x_j = \sum_{i \le 2^k} \lambda_i b_j(d_{ij}) \tag{10}$$

$$w = \sum_{i \le 2^k} \lambda_i \prod_{j \le k} b_j(d_{ij}) \tag{11}$$

$$\sum_{i \le 2^k} \lambda_i = 1, \tag{12}$$

where (11) is obtained by (10) and the fact that $w = \prod_{j<k} x_j$. Let $\bar{W}_J = \{(x, w, \lambda) \mid (10) - (12) \wedge \lambda \ge 0\}$. It is well known that the projection of $\bar{W}_J$ on the (x, w) variables is precisely $\breve{W}_J$.

The dual envelope adds exactly 2^k new nonnegative variables and $k + 1$ new constraints to the formulation. Table 1 reports the size increases for the cases $k \in \{2, 3, 4, 5\}$. Cases $k \in \{2, 3, 4\}$ refer to the McCormick, Meyer-Floudas and Balram [5] inequalities. The statistics for $k = 5$ are taken from [5], but devised computationally using a method similar to PORTA.

Table 1. Per-multilinear-term size increase (new constraints and variables) for primal and dual envelopes

k	*Primal*	*Dual*
2	4	7
3	12	12
4	44	21
5	130	38

3.1 Relaxations

Given a multilinear MP P, a relaxation can be obtained by replacing each multilinear term with its corresponding primal or dual envelope. This term-wise fashion of construction relaxations was initially proposed in [26], refined and exploited in a sBB in [33], and further improved in [8]. As stated earlier, we shall call relaxations constructed with primal envelopes primal relaxations and those constructed with dual envelopes dual relaxations.

4 Computational Results

Our tests, carried out on an Intel Xeon CPU at 2.66GHz with 24GB RAM, show that dual relaxations can be solved faster (as the formulation size increases) than primal relaxations, and are also more stable. We measure *speed* by simply solving the primal and dual relaxations for the same original problem using the CPLEX 12.2 [18] simplex solver, and comparing CPU times. We define a method *stable* when its CPU time increase looks empirically proportional to the increase in formulation size. Firstly we consider NLP problems, and we solve the corresponding *dual LP relaxation* and *primal LP relaxation*. Then we measure stability by enforcing integrality constraints on some of the problem variables, obtaining MINLPs: this yields a *dual MILP relaxation* and a *primal MILP relaxation*. Both are solved with the CPLEX 12.2 MILP solver, and the CPU times are recorded and compared. This is meant to simulate the behaviour of these relaxations in a Branch-and-Bound setting. It turns out that the running times of the MILP solver on the dual MILP relaxation is proportional to the relaxation size, whereas it varies wildly for the primal MILP relaxation.

We generated 2500 random multilinear nonseparable NLPs, involving linear, bilinear and trilinear terms. For each such NLP P, we generated the primal LP relaxation R_P and the dual LP relaxation Λ_P. Then we set some variables of the previously generated NLPs to be integer, thus obtaining MINLPs, and for each MINLP P, we generated the primal MILP relaxation R'_P and the dual MILP relaxation Λ'_P. We let n (the number of original variables) vary in $\{10, 20\}$. For $n = 10$ we let the number of bilinear terms β vary in $\{0, 10, 13, 17, 21, 25, 29, 33\}$ and of trilinear terms τ in $\{0, 10, 22, 34, 36, 58, 71, 83\}$. For $n = 20$, we let β vary in $\{0, 20, 38, 57, 76, 95, 114, 133\}$ and τ in $\{0, 20, 144, 268, 393, 517, 642, 766\}$. Note that the total number of combinations of (n, β, τ), given n, is 63, because the case $\beta = \tau = 0$ is excluded. For each combination of the triplet (n, β, τ) we generated 16 random instances. The variable bounds, chosen at random, were all of magnitude ±1 × 10^6.

The CPU time results (in seconds) comparing R_P, Λ_P are given in Fig. 3-4. The horizontal axis is marked by the instance ID. Each recognizable "block" corresponds to a fixed value of β. Since bilinear terms give rise to fewer relaxation

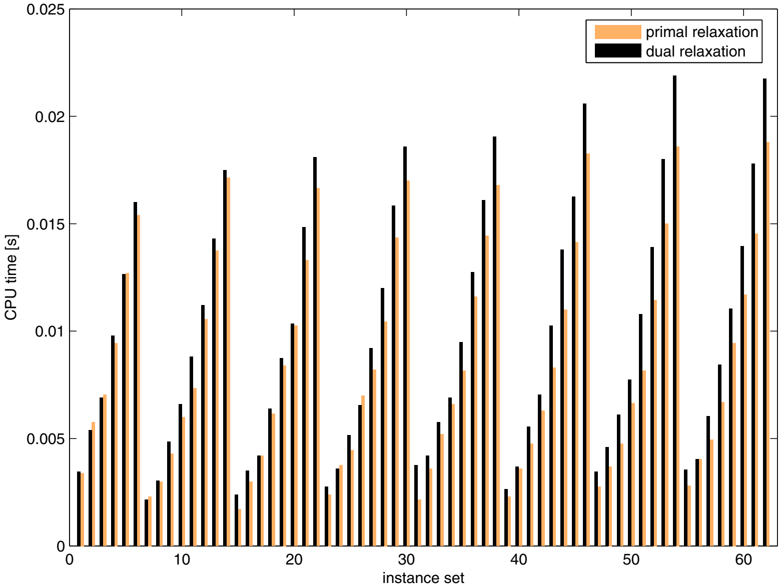


Fig. 3. CPU time averages (in seconds) over each 16-instance set with given (n, β, τ) with $n = 10$ for the LP relaxations

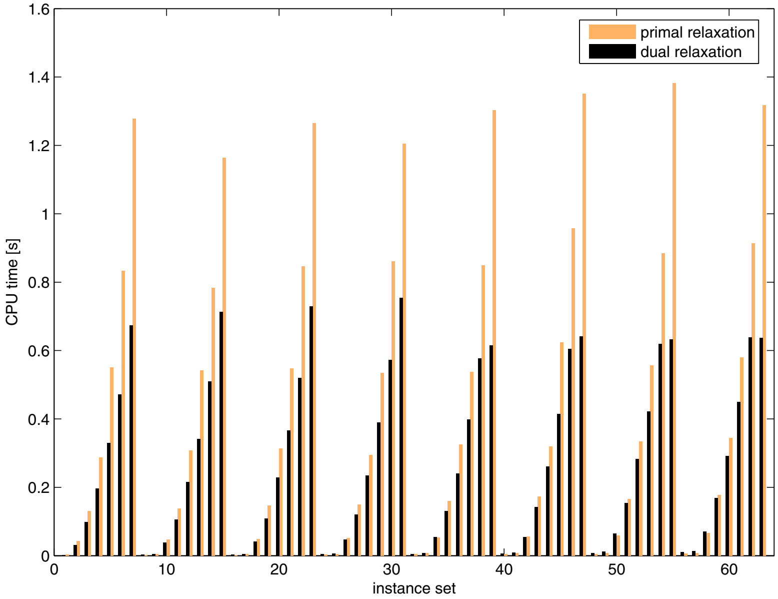


Fig. 4. CPU time (in seconds) averages over each 16-instance set with given (n, β, τ) with $n = 20$ for the LP relaxations

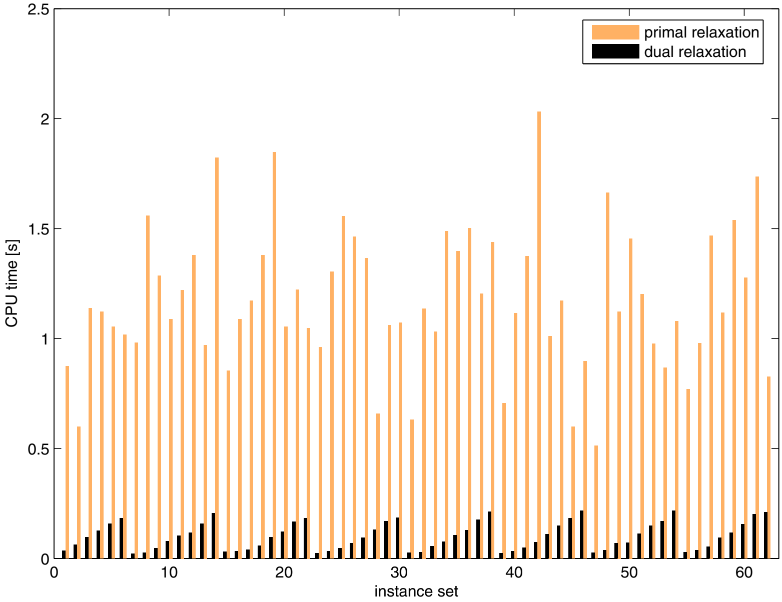


Fig. 5. CPU time averages (in seconds) over each 16-instance set with given (n, β, τ) with $n = 10$ for the MILP relaxations

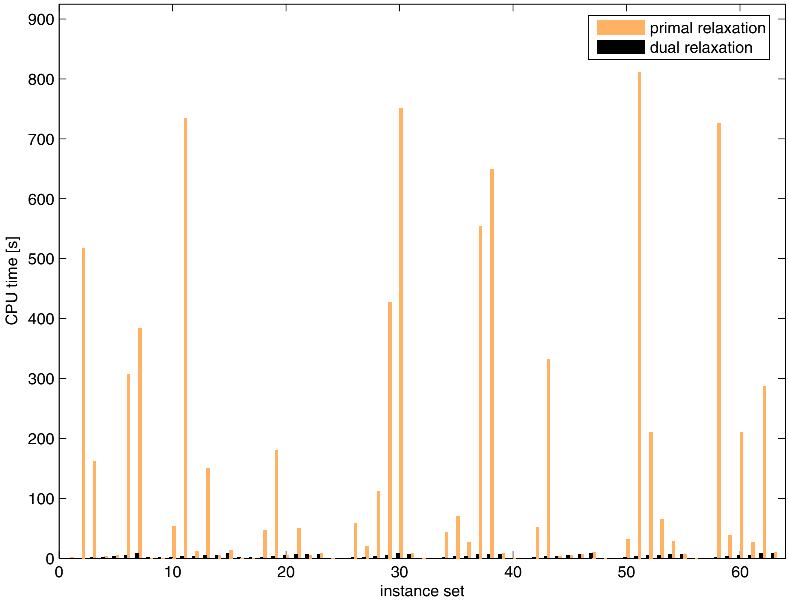


Fig. 6. CPU time averages (in seconds) over each 16-instance set with given (n, β, τ) with $n = 20$ for the MILP relaxations

variables/constraints than trilinear ones, the formulation size is strongly proportional to τ and weakly proportional to β. Although for $n = 10$ (Fig. 3) the CPU time is very slightly in favour of the primal relaxation, the situation changes visibly for $n = 20$ (Fig. 4). Although the CPU times differ, we cannot infer much on the comparative stability of the two methods.

The CPU time results (in seconds) comparing R'_P, Λ'_P are given in Fig. 5-6. The CPU differences are decidedly striking in the case $n = 10$ and even excessively so for the case $n = 20$. The CPU time taken to solve primal relaxations is far from proportional to formulation size, whereas the stability associated to the dual relaxation is remarkable.

References

1. Adhya, N., Tawarmalani, M., Sahinidis, N.V.: A Lagrangian approach to the pooling problem. Industrial and Engineering Chemistry Research 38, 1956–1972 (1999)
2. Adjiman, C.S., Dallwig, S., Floudas, C.A., Neumaier, A.: A global optimization method, αBB, for general twice-differentiable constrained NLPs: I. Theoretical advances. Computers & Chemical Engineering 22(9), 1137–1158 (1998)
3. Al-Khayyal, F.A., Falk, J.E.: Jointly constrained biconvex programming. Mathematics of Operations Research 8(2), 273–286 (1983)
4. Audet, C., Brimberg, J., Hansen, P., Le Digabel, S., Mladenović, N.: Pooling problem: Alternate formulations and solution methods. Management Science 50(6), 761–776 (2004)

5. Balram, S.: Crude transshipment via floating, production, storage and offloading platforms. Master's thesis, Dept. of Chemical and Biomolecular Engineering, National University of Singapore (2010)
6. Bao, X., Sahinidis, N.V., Tawarmalani, M.: Multiterm polyhedral relaxations for nonconvex, quadratically constrained quadratic programs. Optimization Methods and Software 24(4-5), 485–504 (2009)
7. Belotti, P., Cafieri, S., Lee, J., Liberti, L., Miller, A.: On the composition of convex envelopes for quadrilinear terms. In: Pardalos, P., et al. (eds.) Optimization and Optimal Control. Springer, New York (submitted)
8. Belotti, P., Lee, J., Liberti, L., Margot, F., Wächter, A.: Branching and bounds tightening techniques for non-convex MINLP. Optimization Methods and Software 24(4), 597–634 (2009)
9. Cafieri, S., Lee, J., Liberti, L.: On convex relaxations of quadrilinear terms. Journal of Global Optimization 47, 661–685 (2010)
10. Christof, T., Löbel, A.: The PORTA manual page. Technical Report v. 1.4.0, ZIB, Berlin (1997)
11. D'Ambrosio, C., Linderoth, J., Luedtke, J.: Valid Inequalities for the Pooling Problem with Binary Variables. In: Günlük, O., Woeginger, G.J. (eds.) IPCO 2011. LNCS, vol. 6655, pp. 117–129. Springer, Heidelberg (2011)
12. Falk, J.E., Soland, R.M.: An algorithm for separable nonconvex programming problems. Management Science 15, 550–569 (1969)
13. Fortet, R.: Applications de l'algèbre de Boole en recherche opérationelle. Revue Française de Recherche Opérationelle 4, 17–26 (1960)
14. Foulds, L.R., Haughland, D., Jornsten, K.: A bilinear approach to the pooling problem. Optimization 24, 165–180 (1992)
15. Goubault, E., Le Roux, S., Leconte, J., Liberti, L., Marinelli, F.: Static analysis by abstract interpretation: a mathematical programming approach. In: Miné, A., Rodriguez-Carbonell, E. (eds.) Proceedings of the 2nd International Workshop on Numerical and Symbolic Abstract Domains. Electronic Notes in Theoretical Computer Science, vol. 267(1), pp. 73–87. Elsevier (2010)
16. Hammer, P.L., Rudeanu, S.: Boolean Methods in Operations Research and Related Areas. Springer, Berlin (1968)
17. Haverly, C.A.: Studies of the behaviour of recursion for the pooling problem. ACM SIGMAP Bulletin 25, 19–28 (1978)
18. IBM. ILOG CPLEX 12.2 User's Manual. IBM (2010)
19. Knuth, D.E.: The Art of Computer Programming, Part II: Seminumerical Algorithms. Addison-Wesley, Reading (1981)
20. Liberti, L.: Writing global optimization software. In: Liberti, L., Maculan, N. (eds.) Global Optimization: from Theory to Implementation, pp. 211–262. Springer, Berlin (2006)
21. Liberti, L.: Reformulations in mathematical programming: Definitions and systematics. RAIRO-RO 43(1), 55–86 (2009)
22. Liberti, L., Cafieri, S., Tarissan, F.: Reformulations in Mathematical Programming: A Computational Approach. In: Abraham, A., Hassanien, A.-E., Siarry, P., Engelbrecht, A. (eds.) Foundations of Computational Intelligence Volume 3. SCI, vol. 203, pp. 153–234. Springer, Heidelberg (2009)
23. Liberti, L., Lavor, C., Mucherino, A., Maculan, N.: Molecular distance geometry methods: from continuous to discrete. International Transactions in Operational Research 18, 33–51 (2010)

24. Liberti, L., Lavor, C., Chaer Nascimento, M.A., Maculan, N.: Reformulation in mathematical programming: an application to quantum chemistry. Discrete Applied Mathematics 157, 1309–1318 (2009)
25. Liberti, L., Le Roux, S., Leconte, J., Marinelli, F.: Mathematical programming based debugging. In: Mahjoub, R. (ed.) Proceedings of the International Symposium on Combinatorial Optimization. Electronic Notes in Discrete Mathematics, vol. 36, pp. 1311–1318. Elsevier, Amsterdam (2010)
26. McCormick, G.P.: Computability of global solutions to factorable nonconvex programs: Part I — Convex underestimating problems. Mathematical Programming 10, 146–175 (1976)
27. Meyer, C.A., Floudas, C.A.: Trilinear monomials with positive or negative domains: Facets of the convex and concave envelopes. In: Floudas, C.A., Pardalos, P.M. (eds.) Frontiers in Global Optimization, pp. 327–352. Kluwer Academic Publishers, Amsterdam (2003)
28. Meyer, C.A., Floudas, C.A.: Trilinear monomials with mixed sign domains: Facets of the convex and concave envelopes. Journal of Global Optimization 29, 125–155 (2004)
29. Misener, R., Floudas, C.A.: Global optimization of large-scale generalized pooling problems: quadratically constrained MINLP models. Industrial Engineering and Chemical Research 49, 5424–5438 (2010)
30. Paatero, P.: The multilinear engine: A table-driven, least squares program for solving multilinear problems, including the n-way parallel factor analysis model. Journal of Computational and Graphical Statistics 8(4), 854–888 (1999)
31. Rikun, A.: A convex envelope formula for multilinear functions. Journal of Global Optimization 10(4), 425–437 (1997)
32. Ryoo, H.S., Sahinidis, N.V.: Global optimization of nonconvex NLPs and MINLPs with applications in process design. Computers & Chemical Engineering 19(5), 551–566 (1995)
33. Smith, E., Pantelides, C.: A symbolic reformulation/spatial branch-and-bound algorithm for the global optimisation of nonconvex MINLPs. Computers & Chemical Engineering 23, 457–478 (1999)
34. Tardella, F.: Existence and sum decomposition of vertex polyhedral convex envelopes. Technical report, Facoltà di Economia e Commercio, Università di Roma "La Sapienza" (2007)
35. Williams, H.P.: Model Building in Mathematical Programming, 4th edn. Wiley, Chichester (1999)

On Computing the Diameter of Real-World Directed (Weighted) Graphs

Pierluigi Crescenzi[1], Roberto Grossi[2], Leonardo Lanzi[1], and Andrea Marino[1]

[1] Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze, Italy
[2] Dipartimento di Informatica, Università degli Studi di Pisa, Italy

Abstract. In this paper we propose a new algorithm for computing the diameter of directed unweighted graphs. Even though, in the worst case, this algorithm has complexity $O(nm)$, where n is the number of nodes and m is the number of edges of the graph, we experimentally show that in practice our method works in $O(m)$ time. Moreover, we show how to extend our algorithm to the case of directed weighted graphs and, even in this case, we present some preliminary very positive experimental results.

1 Introduction

The analysis of real-world networks such as biological, collaboration, communication, road, social, and web networks has attracted a lot of attention in the last two decades, and many properties of these networks have been studied (see, for example, [21,5,12]). Since the size of real-world networks has been increasing rapidly, in order to study these properties, we need algorithms that can handle huge amount of data. In this paper we will focus our attention on a very basic property of real-world networks, that is, their diameter. Given a directed graph $G = (V, E)$, the diameter of G is the minimum D such that, for any pair of nodes $u, v \in V$, the *distance* $d(u, v)$ between them is at most D, where $d(u, v)$ is the length of the shortest path from u to v (whenever the graph includes a pair of nodes u, v such that $d(u, v) = \infty$, we will study the diameter of its largest strongly connected component).

The diameter is a relevant measure whose meaning depends on the semantics of the real-world network. In the case of social networks, in which every node is an individual and the edges represent their social relationships, the diameter can indicate how quickly information reaches every individual in the worst case, and it has been studied for several social networks (see, for example, [26,20,18]). In the case of web networks, in which every node corresponds to a web page and the edges correspond to hyper-links, the diameter indicates how quickly (in terms of mouse clicks) any page can be reached in the worst case: for several web networks, the diameter has been considered, for example, in [6,18,15]. In the case of communication networks, in which every node is a device and the edges represent communication links, the diameter indicates, for example, the completion time of broadcast protocols based on network flooding. In the case of biological networks, finally, the cellular metabolism is represented by a network

R. Klasing (Ed.): SEA 2012, LNCS 7276, pp. 99–110, 2012.

of metabolites linked by biochemical reactions, and the diameter indicates how many reactions have to be performed, in the worst case, in order to produce any metabolite from any other metabolite [13]: for several such biological networks, the diameter has been studied, for example, in [2,18].

Because of the huge size of real-world networks, in almost all the above cases, the diameter of the strongly connected components has been only estimated. Indeed, most algorithms for finding the exact diameter solve the *all pair shortest path* problem, that is the problem of finding the shortest path between all pairs of nodes of the graph: this can be done either by applying *text-book algorithms* (such as breadth-first searches) for solving, for any node, the *single source shortest path* problem, or by applying fast matrix multiplication algorithms with subcubic complexity (see, for example, [20]). However in the context of real-world networks, these approaches are not practical and usually just estimations or bounds can be provided.

To this aim, some algorithms have been proposed in order to estimate the cumulative distribution of the shortest path lengths of a graph and to thus obtain an estimation of the diameter with a small additive error: this is the case, for example, of the algorithms proposed in [22,4,14]. In other works lower bounds for the diameter have been provided by using a *sample* of the nodes and returning their maximum eccentricity, where the eccentricity of a node u is defined as $\max_{v \in V} d(u,v)$ (see, for example, [17]). In the case of *undirected* graphs a lower bound can also be provided by using the so called *double sweep* algorithm, in short 2-Sweep: pick the farthest node from a random node and return its eccentricity. This idea can be iterated by picking at each step the farthest node from the previous one and maintaining the highest found eccentricity (see, for example, [18]). In real-world networks, this lower bound is very good and, in order to prove its effectiveness, several works, like [16,10], propose strategies to find a matching or close upper bound. Recent advances [9,24] have experimentally shown that in real-world networks a matching between a lower and upper bound for the diameter can be found by applying a very small number of computations of breadth-first searches. The most striking result, along this line of research, has been obtained in [1], where the algorithm proposed in [9] has been applied in order to compute the diameter of the biggest connected component of the Facebook network (approximately 721.1M of nodes and 68.7G of edges). In the case of *directed* graphs, in order to obtain a lower bound for the diameter, the idea of the *double sweep* has been adapted by [6]: pick the farthest node from a random node and return its backward eccentricity, i.e. its eccentricity in the graph with reversed arcs (we will make use of this adaptation in this paper).

In this paper we generalize the idea of the algorithm proposed in [9], by presenting the directed *i*FUB (in short, D*i*FUB) algorithm, in order to calculate the diameter of the strongly connected components of directed graphs. As far as we know, D*i*FUB is the first algorithm which is able to compute exactly the diameter of the strongly connected components of huge real-world directed graphs. The D*i*FUB algorithm can also return a pair of nodes whose distance is exactly equal to the diameter, and a natural adaptation of it works also for weighted graphs.

The algorithm uses very basic operations that are provided basically by all graph software libraries. Indeed, the main operation is the breadth-first search (in short, BFS): since BFS has a good external-memory implementation [19] and works on graphs stored in compressed format [3], we have been able to analyze very large graphs. As a matter of fact, we show the effectiveness of the DiFUB algorithm with a wide set of experiments, by using real-world directed networks which have been chosen in order to cover a large set of network typologies, and have been already used to validate other popular tools, such as [22,14,3,4]. As we already said, in the case of several of these networks, the exact value of the diameter of the largest strongly connected component was still unknown: in almost all the graphs with more than 10000 nodes, the number of BFSes executed by the algorithm is less than 0.01% of the total number of nodes in the component.

Structure of the Paper. In Section 2 we present and analyze the DiFUB algorithm in the case of directed unweighted strongly connected graphs, while in Section 3 we present our dataset and we show the results of our experiments. In Section 4 we extend the DiFUB algorithm to the case of weighted graphs and we briefly describe the corresponding experimental results. We conclude in Section 5 by proposing some interesting directions for future research.

2 The DiFUB Algorithm

Let $G = (V, E)$ be a directed strongly connected graph and let u be any node in V. We denote by T_u^F (respectively, T_u^B) a forward (respectively, backward) breadth-first search tree rooted at node u, and by $\mathrm{ecc}_F(u)$ (respectively, $\mathrm{ecc}_B(u)$) its height. Let $F_i^F(u)$ be the *forward fringe* of u, that is, the set of nodes x such that $d(u, x) = i$. Similarly, let $F_i^B(u)$ be the *backward fringe*, that is, the set of nodes x such that $d(x, u) = i$. In other words, $F_i^F(u)$ (respectively, $F_i^B(u)$) includes all nodes at level i of T_u^F (respectively, T_u^B).

Remark 1. For any two integers i, j with $1 \leq i \leq ecc_B(u)$ and $1 \leq j \leq ecc_F(u)$, for any two nodes x, y such that $x \in F_i^B(u)$ and $y \in F_j^F(u)$, $d(x, y) \leq i + j \leq 2\max\{i, j\}$.

Theorem 1. *For any integer i with $1 < i \leq \mathrm{ecc}_B(u)$, for any integer k with $1 \leq k < i$, and for any node $x \in F_{i-k}^B(u)$ such that $\mathrm{ecc}_F(x) > 2(i - 1)$, there exists $y \in F_j^F(u)$, for some $j \geq i$, such that $d(x, y) = \mathrm{ecc}_F(x)$.*

Proof. Since $\mathrm{ecc}_F(x) > 2(i - 1)$, then there exists y such that $d(x, y) > 2(i - 1)$. If y was in $F_j^F(u)$ with $j < i$, then from Remark 1 it would follow that $d(x, y) \leq 2\max\{i - k, j\} \leq 2\max\{i - k, i - 1\} = 2(i - 1)$, which is a contradiction. Hence, y must be in $F_j^F(u)$ with $j \geq i$.

Similarly to the proof of Theorem 1, we can also prove the following symmetrical result.

Theorem 2. *For any integer i with $1 < i \leq \mathrm{ecc}_F(u)$, for any integer k with $1 \leq k < i$, and for any node $x \in F^F_{i-k}(u)$ such that $\mathrm{ecc}_B(x) > 2(i-1)$, there exists $y \in F^B_j(u)$, for some $j \geq i$, such that $d(y,x) = \mathrm{ecc}_B(x)$.*

In order to describe the DiFUB algorithm, we also need the following definitions. Let

$$B^F_j(u) = \begin{cases} \max_{x \in F^F_j(u)} \mathrm{ecc}_B(x) & \text{if } j \leq \mathrm{ecc}_F(u), \\ 0 & \text{otherwise} \end{cases}$$

and

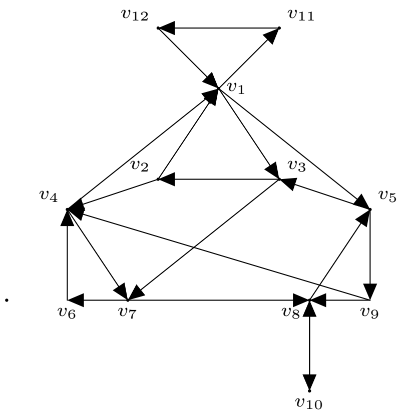
$$B^B_j(u) = \begin{cases} \max_{x \in F^B_j(u)} \mathrm{ecc}_F(x) & \text{if } j \leq \mathrm{ecc}_B(u), \\ 0 & \text{otherwise.} \end{cases}$$

By using these two definitions, we are now ready to introduce the DiFUB algorithm, which is shown in Pseudocode 1. Intuitively, Theorems 1 and 2 suggest to perform a forward and a backward BFS from a node u, and to visit T^F_u and T^B_u in a bottom-up fashion, starting from the nodes in the last fringes. For each level i, we compute the eccentricities of all the nodes in the corresponding fringes: if the maximum eccentricity is greater than $2(i-1)$ then we can discard visiting the remaining levels, since the eccentricities of all their nodes cannot be greater.

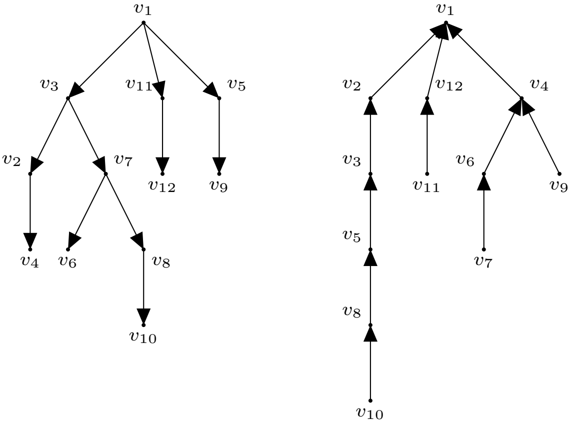
Pseudocode 1. DiFUB

```
Input: A strongly connected di-graph $G$, a node $u$, a lower bound $l$ for the diameter
Output: The diameter $D$
$i \leftarrow \max\{\mathrm{ecc}_F(u), ecc_B(u)\}$; $lb \leftarrow \max\{\mathrm{ecc}_F(u), ecc_B(u), l\}$; $ub \leftarrow 2i$;
while $ub - lb > 0$ do
    if $\max\{lb, B^B_i(u), B^F_i(u)\} > 2(i-1)$ then
        return $\max\{lb, B^B_i(u), B^F_i(u)\}$;
    else
        $lb \leftarrow \max\{lb, B^B_i(u), B^F_i(u)\}$; $ub \leftarrow 2(i-1)$;
    end
    $i \leftarrow i - 1$;
end
return $lb$;
```

An Example. Let us consider the graph shown in the top left part of Figure 1. All pairwise distances, and the forward and backward eccentricities of all its nodes are shown in the top right part of the figure. If we choose $u = v_1$, the corresponding two breadth-first search trees T^F_u and T^B_u are shown in the bottom left part of the figure. From these two trees we can easily derive the forward and backward fringe sets, which are shown in the bottom right part of the figure. If we choose $i = 2$, $j = 3$, $x = v_6$, and $y = v_8$, then it is easy to verify, by inspecting the two BFSes trees, that we can go from v_6 to v_8 by first going up in $T^B_{v_1}$ (by means of two edges) and then by going down in $T^F_{v_1}$ (by means of three edges). Hence,



	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}	ecc_F
v_1	0	2	1	3	1	3	2	3	2	4	1	2	4
v_2	1	0	2	1	2	3	2	3	3	4	2	3	4
v_3	2	1	0	2	3	2	1	2	4	3	3	4	4
v_4	1	3	2	0	2	2	1	2	3	3	2	3	3
v_5	3	2	1	2	0	3	2	2	1	3	4	5	5
v_6	2	4	3	1	3	0	2	3	4	4	3	4	4
v_7	3	4	3	2	2	1	0	1	3	2	4	5	5
v_8	4	3	2	3	1	4	3	0	2	1	5	6	6
v_9	2	4	3	1	2	3	2	1	0	2	3	4	4
v_{10}	5	4	3	4	2	5	4	1	3	0	6	7	7
v_{11}	2	4	3	5	3	5	4	5	4	6	0	1	6
v_{12}	1	3	2	4	2	4	3	4	3	5	2	0	5
ecc_B	5	4	3	5	3	5	4	5	4	6	6	7	



i	$F_i^F(v_1)$	$F_i^B(v_1)$
1	v_3, v_5, v_{11}	v_2, v_4, v_{12}
2	v_2, v_7, v_9, v_{12}	v_3, v_6, v_9, v_{11}
3	v_4, v_6, v_8	v_5, v_7
4	v_{10}	v_8
5		v_{10}

Fig. 1. A strongly connected di-graph with the corresponding all pairwise distances, forward and backward eccentricities and BFSes trees rooted at v_1, and fringe sets

as observed in Remark 1, $d(v_6, v_8) \leq 5$: indeed, $d(v_6, v_8) = 3$ (passing through v_4 and v_7). Moreover, if we choose $i = 2$, $k = 1$, and $x = v_4 \in F_1^B(v_1)$, then we have that $\mathrm{ecc}_F(v_4) = 3 > 2 = 2(i-1)$: Theorem 1 is in this case witnessed by node $y = v_2 \in F_2^F(v_1)$ (indeed, $d(v_4, v_2) = 3$). If we choose $j = 1$, we have that $B_1^F(u) = \max\{\mathrm{ecc}_B(v_3), \mathrm{ecc}_B(v_5), \mathrm{ecc}_B(v_{11})\} = \max\{3, 6\} = 6$. On the other hand, $B_1^B(u) = \max\{\mathrm{ecc}_F(v_2), \mathrm{ecc}_F(v_4), \mathrm{ecc}_F(v_{12})\} = \max\{3, 4, 5\} = 5$. Finally, suppose we invoke the algorithm shown in Pseudocode 1 with $u = v_1$ and $l = 0$. Before the execution of the **while** loop starts, the two variables i and lb are both set equal to $\max\{\mathrm{ecc}_F(v_1), \mathrm{ecc}_B(v_1)\} = \max\{4, 5\} = 5$, while variable ub is set equal to $2i = 10$. Since $ub - lb = 5 > 0$, the algorithm enters the **while** loop with $i = 5$. Since $5 > \mathrm{ecc}_F(u)$, $B_5^F(u) = 0$. On the other hand, $B_5^B(u) = \mathrm{ecc}_F(v_{10}) = 7$: since, $7 < 8 = 2(i-1)$, the algorithm enters the **else** branch and set lb equal to 7 and ub equal to 8. Once again, $ub - lb = 1 > 0$

and the algorithm continues the execution of the **while** loop with $i = 4$. This time we have that $B_4^F(u) = \mathrm{ecc}_B(v_{10}) = 6$ and $B_4^B(u) = \mathrm{ecc}_F(v_8) = 6$: hence, $\max\{lb, B_4^B(u), B_4^F(u)\} = 7 > 6 = 2(i-1)$. The algorithm thus enters the **if** branch and returns the value 7 which is the correct diameter value. In other words, the diameter has been computed by exploring only three nodes (apart from v_1): note that we did not really need to compute $B_4^B(u)$ and $B_4^F(u)$, since l was already greater than $2(i-1)$.

Theorem 3. DiFUB *Algorithm correctly computes the value of the diameter of* G.

Proof (Sketch). Let us prove that if, at the iteration corresponding to a given value i, $\max\{lb,\ B_i^B(u),\ B_i^F(u)\} > 2(i-1)$ then the diameter of G is equal to $\max\{lb,\ B_i^B(u),\ B_i^F(u)\}$. By contradiction, assume that the diameter is greater than $\max\{lb,\ B_i^B(u),\ B_i^F(u)\}$ (note that the diameter cannot be smaller than $\max\{lb,\ B_i^B(u),\ B_i^F(u)\}$ since this value is the length of a shortest path). This implies that there exists $x \in F_{i-k}^B(u) \cup F_{i-k}^F(u)$ such that $\max\{\mathrm{ecc}_F(x),\ \mathrm{ecc}_B(x)\} = D > 2(i-1)$. From the previous two theorems, it follows that there exists $y \in F_j^F(u) \cup F_j^B(u)$ such that $\max\{d(x,y), d(y,x)\} = \max\{\mathrm{ecc}_F(x), \mathrm{ecc}_B(x)\} = D$ with $j \geq i$, thus contradicting the fact that $D > \max\{lb,\ B_i^B(u),\ B_i^F(u)\}$ (note that $lb \geq \max\{B_j^B(u),\ B_j^F(u)\}$ for any $j > i$).

The time complexity of DiFUB can be in the worst case $O(nm)$ where n denotes the number of nodes and m denotes the number of edges. However, the practical performance of the algorithm depends on the chosen node u. Indeed, observe that, at each iteration of the **while** loop, $ub - lb$ decreases at least by 2: this implies that the algorithm executes at most $\max\{\lceil \mathrm{ecc}_B(u)/2 \rceil, \lceil \mathrm{ecc}_F(u)/2 \rceil\}$ iterations (note that we have that the number of iterations is bounded by $D/2$). A hopefully good starting point u (and a corresponding lower bound l) can be obtained by applying the following heuristics, called 2-*d*Sweep, which is a natural extension to directed graphs of the 2-Sweep method (in the following, the *middle* node between two nodes s and t is defined as the node belonging to the shortest path from s to t, whose distance from s is $\lceil d(s,t)/2 \rceil$).

1. Run a forward BFS from a random node r: let a_1 be the farthest node.
2. Run a backward BFS from a_1: let b_1 be the farthest node.
3. Run a backward BFS from r: let a_2 be the farthest node.
4. Run a forward BFS from a_2: let b_2 be the farthest node.
5. If $\mathrm{ecc}_B(a_1) > \mathrm{ecc}_F(a_2)$, then set u equal to the middle node between a_1 and b_1 and l equal to $\mathrm{ecc}_B(a_1)$. Otherwise, set u equal to the middle node between a_2 and b_2 and l equal to $\mathrm{ecc}_F(a_2)$.

3 Experiments

We collected several real-world directed graphs, which have been chosen in order to cover the largest possible set of network typologies. In particular, we

Table 1. For any graph, a summary of 10 executions of 2-*d*SWEEP (3rd and 4th columns) and D*i*FUB (7th and 8th columns)

Network name	D	Numb. of runs (out of 10) in which $l = D$	Worst l found	n	m	Avg. Numb. of Visits	Visits in the worst run
Wiki-Vote	9	10	9	1300	39456	17	17
p2p-Gnutella08	19	9	18	2068	9313	45.9	64
p2p-Gnutella09	19	9	18	2624	10776	202.1	230
p2p-Gnutella06	19	10	19	3226	13589	236.6	279
p2p-Gnutella05	22	9	21	3234	13453	60.4	94
p2p-Gnutella04	25	7	22	4317	18742	36.7	38
p2p-Gnutella25	21	8	20	5153	17695	85.1	161
p2p-Gnutella24	28	10	28	6352	22928	13	13
p2p-Gnutella30	23	2	22	8490	31706	255.4	516
p2p-Gnutella31	30	9	29	14149	50916	208.7	255
s.s.Slashdot081106	15	10	15	26996	337351	22.3	25
s.s.Slashdot090216	15	10	15	27222	342747	21.5	26
s.s.Slashdot090221	15	10	15	27382	346652	22.8	26
soc-Epinions1	16	9	15	32223	443506	6.1	7
Email-EuAll	10	10	10	34203	151930	6	6
soc-sign-epinions	16	10	16	41441	693737	6	6
web-NotreDame	93	10	93	53968	304685	7	7
Slashdot0811	12	10	12	70355	888662	40	40
Slashdot0902	13	3	12	71307	912381	32.9	40
WikiTalk	10	9	9	111881	1477893	13.6	19
web-Stanford	210	10	210	150532	1576314	6	6
web-BerkStan	679	10	679	334857	4523232	7	7
web-Google	51	10	51	434818	3419124	9.4	10
wordassociation-2011	10	9	9	4845	61567	412.5	423
enron	10	10	10	8271	147353	19	22
uk-2007-05@100000	7	10	7	53856	1683102	14	14
cnr-2000	81	10	81	112023	1646332	17	17
uk-2007-05@1000000	40	10	40	480913	22057738	6	6
in-2004	56	10	56	593687	7827263	14	14
amazon-2008	47	10	47	627646	4706251	136.3	598
eu-2005	82	10	82	752725	17933415	6	6
indochina-2004	235	10	235	3806327	98815195	8	8
uk-2002	218	10	218	12090163	232137936	6	6
arabic-2005	133	10	133	15177163	473619298	58	58
uk-2005	166	10	166	25711307	704151756	170	170
it-2004	873	10	873	29855421	938694394	87	87

used web graphs, communication networks, product co-purchasing networks, autonomous systems graphs, Internet peer-to-peer networks, social networks, road networks, and words adjacency networks (see Table 1). All these networks have been downloaded either from [27] or from [23]. As it can be seen in the fifth and sixth columns of the table, an important feature is that almost all graphs in our dataset are sparse (that is, $m = O(n)$). Finally, note that, in the case of several of these graphs, the diameter value was still unknown.

Our computing platform is a machine with a Pentium Dual-Core CPU (Intel(tm) E5200 @ 2.50GHz), with a 8GB shared memory. The operating system is a Debian GNU/Linux 6.0, with a Linux kernel version 2.6.32 and gcc version 4.4.5. We have performed 10 experiments on the biggest strongly connected component of each of the 36 networks, for a total of 360 experiments. The code and the data set are available at `http://piluc.dsi.unifi.it/lasagne/`.

3.1 Obtaining a Tight Lower Bound via 2-dSWEEP

For each of the analyzed graphs, we executed the 2-*d*SWEEP algorithm ten times.[1] A summary of the results obtained by ten executions of the 2-*d*SWEEP algorithm for the [23] dataset (upper part) and the [27] dataset (lower part) is shown in the middle part of Table 1. For each graph, the diameter (D), the number of runs in which the lower bound l returned by the 2-*d*SWEEP algorithm is equal to the diameter D, and the worst lower bound returned among the ten experiments are shown. It is worth noting that, in the case of web graphs and communication networks, the obtained lower bound is tight for any experiment, while in the case of the other networks, apart from `p2p-Gnutella04`, the absolute error, in all the ten experiments, is at most 1.

3.2 Obtaining the Diameter via DiFUB

For each of the analyzed graphs, we executed the D*i*FUB algorithm ten times: the results are summarized in the rightmost part of Table 1. In particular, we report the average number of visits performed by D*i*FUB in order to obtain the diameter, and the largest number of visits performed by D*i*FUB among the ten experiments. Observe that given a graph with n nodes, the number of BFSes may range between 6 (that is, the case in which the algorithm shown in Pseudocode 1 returns the exact value of the diameter without entering the **while** loop), and $O(n)$ (that is, the case in which the algorithm degenerates in the text-book algorithm). The ratio between the number of BFSes performed by the algorithm and the number of nodes gives us the fraction of visits performed by our algorithm with respect to the worst case. This *performance ratio* in almost any graph with more than 10000 nodes is less than 0.01%. Moreover it is worth observing that this ratio seems to decrease with respect to n, as shown in Figure 2: we argue that the number of visits performed is asymptotically constant. In particular, in the figure the ratio between the average number of visits and the number of nodes is reported in log scale: from the figure is clear that our *gain*, that is the ratio between the number of nodes and the average number of visits, increases exponentially with the size of the graph.

4 The DiFUB Algorithm for Weighted Graphs

Theorem 1 and 2 can be easily extended to the case of directed weighted graphs. Indeed, let T_u^F (respectively, T_u^B) denote the forward (respectively, backward) lightest path tree rooted at node u, computed, for instance, by means of the Dijkstra algorithm [11] in G (respectively, in G by reversing the orientation of the arcs). Moreover, let $\mathrm{ecc}^F(u)$ (respectively, $\mathrm{ecc}^B(u)$) denote the weighted forward (respectively, backward) eccentricity of u, that is the weight of the longest

[1] No significant variance was observed also on more experiments because central nodes are easily detected by the 2-*d*SWEEP algorithm: they usually are the same in all the experiments, even if we perform random choices.

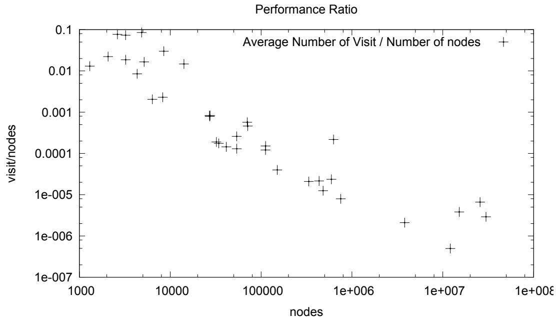


Fig. 2. For any number of nodes n, on the x-axis, the average number of visits performed by DiFUB normalized with respect to n, is reported on the y-axis in log scale (each point corresponds to a graph in Table 1)

path from (respectively, to) u to (respectively, from) one of the leaves of T_u^F (respectively, T_u^B). Finally, let $F_d^F(u)$ (respectively, $F_d^B(u)$) denote the set of nodes whose weighted distance from (respectively, to) u is equal to d: hence, $F_d^F(u) \neq \emptyset$ if and only if there exists at least one node x in T_u^F such that the weight of the path from u to x is equal to d, and $F_d^B(u) \neq \emptyset$ if and only if there exists at least one node x in T_u^B such that the weight of the path from x to u is equal to d.

Let $d_1, d_2, \ldots, d_h$ be the sequence of distinct values d such that $F_d^F(u) \neq \emptyset$ or $F_d^B(u) \neq \emptyset$ ordered in increasing order, that is, $d_1 < d_2 < \cdots < d_h$: note that $d_h = \max\{\mathrm{ecc}_F(u), \mathrm{ecc}_B(u)\}$. We then have the following two results, whose proofs are similar to the proofs of Theorems 1 and 2, respectively.

Theorem 4. *For any integer i with $1 < i \leq h$, for any integer k with $1 \leq k < i$, and for any node $x \in F_{d_{i-k}}^B(u)$ such that $\mathrm{ecc}_F(x) > 2d_{i-1}$, there exists $y \in F_{d_j}^F(u)$, for some $d_j \geq d_i$, such that $d(x, y) = \mathrm{ecc}_F(x)$.*

Theorem 5. *For any integer i with $1 < i \leq h$, for any integer k with $1 \leq k < i$, and for any node $x \in F_{d_{i-k}}^F(u)$ such that $\mathrm{ecc}_B(x) > 2d_{i-1}$, there exists $y \in F_{d_j}^B(u)$, for some $d_j \geq d_i$, such that $d(y, x) = \mathrm{ecc}_B(x)$.*

We can then appropriately modify the DiFUB algorithm in order to deal with directed weighted graphs. To this aim, we define

$$B_{d_i}^F(u) = \begin{cases} \max_{x \in F_{d_i}^F(u)} \mathrm{ecc}_B(x) & \text{if } F_{d_i}^F(u) \neq \emptyset \text{ and } d_i \leq \mathrm{ecc}_F(u), \\ 0 & \text{otherwise} \end{cases}$$

and

$$B_{d_j}^B(u) = \begin{cases} \max_{x \in F_{d_j}^B(u)} \mathrm{ecc}_F(x) & \text{if } F_{d_j}^B(u) \neq \emptyset \text{ and } d_j \leq \mathrm{ecc}_B(u), \\ 0 & \text{otherwise.} \end{cases}$$

The DiFUB algorithm for directed weighted graphs is then described in Pseudocode 2: observe that, in order to start the execution of the algorithm, we

Pseudocode 2. DiFUB for weighted directed graphs

```
Input: A weighted directed strongly connected graph $G$, a node $u$, a lower bound for
       the diameter $l$
Output: The diameter $D$
Let $d_1 < d_2 < \ldots < d_h$ be the sequence of values $d$ such that $F^{\mathrm{F}}_d(u) \neq \emptyset$ or $F^{\mathrm{B}}_d(u) \neq \emptyset$
$i \leftarrow h$; $lb \leftarrow \max\{\mathrm{ecc}_F(u), ecc_B(u), l\}$; $ub \leftarrow 2d_i$;
while $ub - lb > 0$ do
    if $\max\{lb, B^B_{d_i}(u), B^F_{d_i}(u)\} > 2d_{i-1}$ then
        return $\max\{lb, B^B_{d_i}(u), B^F_{d_i}(u)\}$;
    else
        $lb \leftarrow \max\{lb, B^B_{d_i}(u), B^F_{d_i}(u)\}$; $ub \leftarrow 2d_{i-1}$;
    end
    $i \leftarrow i - 1$;
end
return $lb$;
```

can also modify the 2-dSweep algorithm by using single source lightest path algorithm executions instead of BFSes.

4.1 Dataset and Experiments

We have experimented the modification of the DiFUB and 2-dSweep algorithms on several directed weighted real-world graphs: these graphs have been downloaded either from the weighted network dataset available at [25] or from the web site of the 9th DIMACS Implementation Challenge on shortest paths [7]. For the sake of brevity, we do not fully report these experimental results, but we limit ourselves to observe that, in most of the cases, the performances are very similar to the experiments on unweighted graphs. The only significant exceptions concern some of the road networks taken from the [7] dataset: in these cases, the number of performed single source shortest path computations is a quite large fraction (more than 50%) of the total number of nodes. Apart from these exceptions, our results turn out to be very promising.

5 Conclusion and Open Questions

In this paper we have described and experimented a new algorithm for computing the diameter of directed (weighted) graphs. Even though the algorithms has $O(nm)$ time complexity in the worst case, our experiments suggests that its execution for real-world networks requires time $O(m)$.

The performance of our algorithm depends on the choice of the starting node u (indeed, it could be interesting to experimentally analyze its behavior depending on this choice). Ideally, u should be a "center" of the graph G, that is, the maximum between the forward and the backward eccentricity of u should be close to the radius R of the graph (which is defined as $R = \min_{v \in V}\{\max\{ecc_F(v), ecc_B(v)\}\}$). Surprisingly, we have observed that in the case of real-world graphs, R is close to the minimum possible, that is $D/2$. This peculiar structural property affects the performance of our algorithm: since the minimum number of iterations performed by DiFUB is obtained whenever the starting node u is a center of the graph, we have that, in this case, the upper bound on the iterations is minimum and equal to $R - D/2 + 1$. Since the radius is approximately half the diameter and, as shown by our experiments, the double sweep seems to be very effective in order to find central nodes, peripheral nodes are discovered during the first iterations.

The main fundamental questions are now the following. Why the double sweep, both in the directed and in the undirected version, is so effective in finding tight lower bounds for the diameter and nodes with low eccentricity? Which is the topological underlying property that can lead us to these results? Why real world graphs exhibit this property? Some progress has been done by [8], but still a lot has to be done. Finally, it could be interesting to analyze a parallel implementation of the DiFUB algorithm. Indeed, the eccentricities of the nodes belonging to the same fringe set can be computed in parallel. Moreover, a variety of parallel BFS algorithms have been explored in the literature and can be integrated in the implementation of our algorithm.

References

1. Backstrom, L., Boldi, P., Rosa, M., Ugander, J., Vigna, S.: Four Degrees of Separation (2011) arXiv:1111.4570v1
2. Bansal, S., Khandelwal, S., Meyers, L.: Exploring biological network structure with clustered random networks. BMC Bioinformatics 10(1), 405+ (2009)
3. Boldi, P., Vigna, S.: The WebGraph Framework I: Compression Techniques. In: Proceedings of the 13th International World Wide Web Conference, pp. 595–601. ACM Press, Manhattan (2003)
4. Boldi, P., Rosa, M., Vigna, S.: Hyperanf: approximating the neighbourhood function of very large graphs on a budget. In: WWW, pp. 625–634 (2011)
5. Brandes, U., Erlebach, T.: Network Analysis: Methodological Foundations. Springer (2005)
6. Broder, A.Z., Kumar, R., Maghoul, F., Raghavan, P., Rajagopalan, S., Stata, R., Tomkins, A., Wiener, J.L.: Graph structure in the web. Computer Networks 33(1-6), 309–320 (2000)
7. 9th DIMACS Implementation Challenge - Shortest Paths (2006), http://www.dis.uniroma1.it/~challenge9/
8. Chepoi, V., Dragan, F., Estellon, B., Habib, M., Vaxès, Y.: Diameters, centers, and approximating trees of delta-hyperbolic geodesic spaces and graphs. In: Proceedings of the 24th Annual Symposium on Computational Geometry, SCG 2008, pp. 59–68. ACM, New York (2008)
9. Crescenzi, P., Grossi, R., Habib, M., Lanzi, L., Marino, A.: On Computing the Diameter of Real-World Undirected Graphs. Presented at Workshop on Graph Algorithms and Applications (Zurich–July 3, 2011) and selected for submission to the special issue of Theoretical Computer Science in honor of Giorgio Ausiello in the occasion of his 70th birthday (2011)
10. Crescenzi, P., Grossi, R., Imbrenda, C., Lanzi, L., Marino, A.: Finding the Diameter in Real-World Graphs. In: de Berg, M., Meyer, U. (eds.) ESA 2010, Part I. LNCS, vol. 6346, pp. 302–313. Springer, Heidelberg (2010)

11. Dijkstra, E.: A note on two problems in connexion with graphs. Numerische Mathematik 1, 269–271 (1959)
12. Havlin, S., Cohen, R.: Complex Networks: Structure, Robustness and Function. Cambridge University Press, Cambridge (2010)
13. Junker, B.O.H., Schreiber, F.: Analysis of Biological Networks. Wiley Series in Bioinformatics. Wiley Interscience (2008)
14. Kang, U., Tsourakakis, C.E., Appel, A.P., Faloutsos, C., Leskovec, J.: Hadi: Mining radii of large graphs. TKDD 5(2), 8 (2011)
15. Kang, U., Tsourakakis, C.E., Faloutsos, C.: PEGASUS: A Peta-Scale graph mining system implementation and observations. In: 2009 Ninth IEEE International Conference on Data Mining, pp. 229–238. IEEE (December 2009)
16. Latapy, M., Magnien, C.: Measuring Fundamental Properties of Real-World Complex Networks. CoRR abs/cs/0609115 (2006)
17. Leskovec, J., Faloutsos, C.: Sampling from large graphs. In: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2006, pp. 631–636. ACM, New York (2006)
18. 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 Mathematics 6(1), 29–123 (2009)
19. Mehlhorn, K., Meyer, U.: External-Memory Breadth-First Search with Sublinear I/O. In: Möhring, R.H., Raman, R. (eds.) ESA 2002. LNCS, vol. 2461, pp. 723–735. Springer, Heidelberg (2002)
20. Mislove, A., Marcon, M., Gummadi, K.P., Druschel, P., Bhattacharjee, B.: Measurement and analysis of online social networks. In: Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement, IMC 2007, pp. 29–42. ACM, New York (2007)
21. Newman, M.E.J.: The structure and function of complex networks. SIAM Review 45, 167–256 (2003)
22. Palmer, C.R., Gibbons, P.B., Faloutsos, C.: ANF: a Fast and Scalable Tool for Data Mining in Massive Graphs. In: Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 81–90 (2002)
23. SNAP: Stanford Network Analysis Package (SNAP) (2009), http://snap.stanford.edu
24. Takes, F.W., Kosters, W.A.: Determining the diameter of small world networks. In: CIKM, pp. 1191–1196 (2011)
25. Network datasets (2009), http://toreopsahl.com/datasets/
26. Wang, F., Moreno, Y., Sun, Y.: Structure of peer-to-peer social networks. Phys. Rev. E 73, 036123 (2006)
27. WebGraph: WebGraph (2001), http://webgraph.dsi.unimi.it/

Reoptimizing the Strengthened Metric TSP on Multiple Edge Weight Modifications*

Annalisa D'Andrea[1] and Guido Proietti[1,2]

[1] Dipartimento di Informatica, University of L'Aquila, Italy
[2] Istituto di Analisi dei Sistemi ed Informatica, CNR, Rome, Italy
{annalisa.dandrea,guido.proietti}@univaq.it

Abstract. We consider the following (re)optimization problem: Given a minimum-cost Hamiltonian cycle of a complete non-negatively real weighted graph $G = (V, E, c)$ obeying the *strengthened* triangle inequality (i.e., for some *strength factor* $\frac{1}{2} \leq \beta < 1$, we have that $\forall x, y, z \in V, c(x, y) \leq \beta(c(x, z) + c(y, z))$), and given a set of k edge weight modifications producing a new weighted graph still obeying the strengthened triangle inequality, find a minimum-cost Hamiltonian cycle of the modified graph. This problem is known to be NP-hard already for a single edge weight modification. However, in this case, if both the input and the modified graph obey the strengthened triangle inequality and the respective strength factors are fixed (i.e., independent of $|V|$), then it has been shown that the problem admits a PTAS (which just consists of either returning the old optimal cycle, or instead computing — for finitely many inputs — a new optimal solution from scratch, depending on the required accuracy in the approximation). In this paper we first extend the analysis of the PTAS to show its applicability for all $k = O(1)$, and then we provide a large set of experiments showing that, in most practical circumstances, altering (uniformly at random) even several edge weights does not affect the goodness of the old optimal solution.

1 Introduction

Optimization theory has always focused on the challenge of finding good feasible solutions for input instances that were practically relevant, but it has almost totally omitted to consider the possibility that a substantial part of the sought solution could be known in advance. This seemingly paradoxical perspective could instead be realistic in all those cases in which one has to cope with a limited set of input instances, that can be reasonably supposed to be temporally invariant, e.g., a railway system. Then, when the original setting undergoes a modification, even minimal, is it really necessary to re-compute a new solution from scratch by just forgetting the old one? Or does it make more sense to try

* This work was partially supported by the PRIN 2008 research project COGENT (COmputational and GamE-theoretic aspects of uncoordinated NeTworks), funded by the Italian Ministry of Education, University, and Research.

R. Klasing (Ed.): SEA 2012, LNCS 7276, pp. 111–122, 2012.

to exploit the knowledge we have about the old solution in order to find the new one in a more efficient way?

To get closer to the subject of this paper, imagine a scenario in which, given an instance of the classic *traveling salesman problem* (TSP), i.e., a complete non-negatively real weighted graph $G = (V, E, c)$, along with an optimal solution, i.e., a minimum-cost Hamiltonian cycle of G, this is subject to a small modification, e.g., the insertion or the deletion of a new node in the graph. Then the natural arising questions are the following: Is it still NP-hard to find an optimal solution for this problem? And in the positive case, what about its approximability? This was exactly the pioneering formalization of a reoptimization problem given in [4]. More precisely, the authors focused on the *metric TSP* (i.e., the TSP restricted to graphs whose weight function obeys the triangle inequality), and showed that the problem remains NP-hard. Moreover, they showed that the adaptation of the classic *cheapest insertion* heuristic (i.e., start from a subcycle, and among all nodes not inserted so far, choose a node whose insertion causes the lowest increase in the cost of the cycle) guarantees a 3/2-approximation factor, instead of the 2-factor we can get by recomputing a solution from scratch. Inspired by this work, in [9] the authors addressed another natural reoptimization variant of the metric TSP, namely that in which the instance undergoes the alteration of the weight of a single edge. Once again, the authors proved the NP-hardness of the problem, and therefore provided an extensive comparative analysis with the canonical metric TSP as far as the approximability of the problem was concerned. To this respect, the authors developed their study by classifying the approximability of TSP depending on the "metricity" of both the input and the modified instance. More formally, they proved that: (i) if the input and the modified instance are both *strengthened* metric (i.e., their respective weight functions obeys the triangle inequality up to a multiplying factor $1/2 \leq \beta < 1$), the problem admits a PTAS (this compares favorably with the APX-hardness of the counterpart);[1] (ii) if the input and the modified instance are both metric, the problem can be approximated within 7/4 (this compares favorably with the old-standing 3/2-approximation ratio guaranteed by the Christofidies algorithm [14] for the counterpart); and finally, (iii) if the input and the modified instance obey the *relaxed* triangle inequality up to a multiplying factor $1 < \beta < 3.34899$, the problem admits a $\beta^2 \frac{15\beta^2+5\beta-6}{13\beta^2+3\beta-6}$-approximation algorithm, which is better than its counterparts given in [6,11].

Further reoptimization versions of the metric TSP were developed for both the strengthened and the relaxed case [5,9,13]. Afterwards, other authors have faced the *Steiner tree* problem [7,12], or some classic optimization problems like *maximum independent set* and *minimum vertex cover* [8].

Our Results. In this paper we focus on the reoptimization version of the strengthened metric TSP. We first extend the analysis of the PTAS provided in [9] to show its applicability also when the number of modified edges k is $O(1)$. More precisely,

[1] Notice that $\beta = \frac{1}{2}$ corresponds to the trivial case where all edge weights are equal; thus, we will assume $\frac{1}{2} < \beta < 1$ for the remainder of the paper.

if we assume that the *original* and the *modified* graph, say G_O and G_M, obey the triangle inequality up to a strength factor of $1/2 \leq \beta_\mathsf{O} < 1$ and $1/2 \leq \beta_\mathsf{M} < 1$, respectively, and if we set $\beta_\mathsf{L} = \min\{\beta_\mathsf{O}, \beta_\mathsf{M}\}$ and $\beta_\mathsf{H} = \max\{\beta_\mathsf{O}, \beta_\mathsf{M}\}$, then we show that the simple returning of the given optimal cycle of the original instance, guarantees an approximation of $1 + \frac{2k{\beta_\mathsf{H}}^2 - k(1-\beta_\mathsf{L})(1-\beta_\mathsf{H})}{(1-\beta_\mathsf{L})(1-\beta_\mathsf{H})|V|}$, which can be made arbitrarily close to 1 for sufficiently large input graphs. To assess the practical relevance of our result, which suggests that in strengthened metric graphs the quality of optimal Hamiltonian cycles is resilient to multiple edge weight modifications, we provide a large set of experiments showing that, in most practical circumstances, altering (uniformly at random) even several edge weights does not affect the goodness of the old optimal solution.

2 Reoptimizing the Strengthened Metric TSP

We start by giving a formal definition of our problem. For an input graph G, let $\textsc{Opt}_G$ denote the cost of a minimum-cost Hamiltonian cycle of G. Then, we have the following:

Definition 1. *The* k-edge-modified strengthened metric TSP *(*k-EM-SMTSP*) is defined as follows: Given*

- *two complete weighted graphs $G_\mathsf{O} = (V, E, c_\mathsf{O})$, $G_\mathsf{M} = (V, E, c_\mathsf{M})$ such that G_O and G_M obey the triangle inequality up to a strength factor of β_O and β_M, respectively, with $\beta_\mathsf{O}, \beta_\mathsf{M} < 1$, and such that c_O and c_M coincide, except for $k \geq 1$ edges;*
- *a Hamiltonian cycle $\overline{C}$ of G_O such that $\sum_{e \in \overline{C}} c_\mathsf{O}(e) = \textsc{Opt}_{G_\mathsf{O}}$;*

find a Hamiltonian cycle C of G_M such that $\sum_{e \in C} c_\mathsf{M}(e) = \textsc{Opt}_{G_\mathsf{M}}$.

Notice that k-EM-SMTSP is NP-hard already for $k = 1$ [9]. Recall also that in a strengthened metric graph G, the following holds (see [10]):

- Let $c_{\max}$ and $c_{\min}$ denote the maximum and the minimum edge weight in G, respectively. Then,
$$\frac{c_{\max}}{c_{\min}} \leq \frac{2\beta^2}{1-\beta}. \qquad (1)$$
- Neighboring edges of G never differ by a factor of more than $\frac{1}{1-\beta}$.

From this, by following the proof of Lemma 16 in [9], we get

Lemma 1. *Let G_O and G_M be two weighted graphs such that G_O and G_M obey the triangle inequality up to a strength factor of β_O and β_M, respectively, with $\beta_\mathsf{O}, \beta_\mathsf{M} < 1$. For $i \in \{\mathsf{O}, \mathsf{M}\}$, let $c_{\max,i}$ and $c_{\min,i}$ denote the maximum and* *minimum weight of an edge in G_i, respectively. Let the edge weights in G_{O} and G_{M} agree except for $k < |V| - 1$ edges. Then,*

$$\begin{aligned} c_{\max,\mathsf{M}} &\le \frac{1}{1-\beta_{\mathsf{M}}} c_{\max,\mathsf{O}}; & c_{\min,\mathsf{M}} &\le \frac{1}{1-\beta_{\mathsf{O}}} c_{\min,\mathsf{O}}; \\ c_{\max,\mathsf{O}} &\le \frac{1}{1-\beta_{\mathsf{O}}} c_{\max,\mathsf{M}}; \quad \textit{and} & c_{\min,\mathsf{O}} &\le \frac{1}{1-\beta_{\mathsf{M}}} c_{\min,\mathsf{M}}. \end{aligned} \tag{2}$$

Proof. Let $e_1, e_2, ..., e_k$ be the edges such that $c_{\mathsf{O}}(e_1) \neq c_{\mathsf{M}}(e_1), ..., c_{\mathsf{O}}(e_k) \neq c_{\mathsf{M}}(e_k)$. Since $k < |V| - 1$, every edge e_i, $i \in [1, ..., k]$, will have at least one adjacent edge having the same weight in G_{O} as in G_{M}, and therefore bounded by $c_{\max,\mathsf{O}}$. Hence, we have $c_{\mathsf{M}}(e_i) \le \frac{1}{1-\beta_{\mathsf{M}}} c_{\max,\mathsf{O}}$, from which

$$c_{\max,\mathsf{M}} \le \max\{c_{\mathsf{M}}(e_1), ..., c_{\mathsf{M}}(e_k), c_{\max,\mathsf{O}}\} \le \frac{1}{1-\beta_{\mathsf{M}}} c_{\max,\mathsf{O}}.$$

Similarly, $c_{\mathsf{O}}(e_i) \ge (1 - \beta_{\mathsf{O}}) c_{\min,\mathsf{M}}$, and then

$$c_{\min,\mathsf{O}} \ge \min\{c_{\mathsf{O}}(e_1), ..., c_{\mathsf{O}}(e_k), c_{\min,\mathsf{M}}\} \ge (1 - \beta_{\mathsf{O}}) c_{\min,\mathsf{M}}.$$

The two remaining inequalities are symmetric. □

We are now ready to give our main result:

Theorem 1. *Let $(G_{\mathsf{O}}, G_{\mathsf{M}}, \overline{C})$ be an input instance of k-EM-SMTSP such that G_{O} and G_{M} obey the triangle inequality up to a strength factor of β_{O} and β_{M}, respectively, with $\beta_{\mathsf{O}}, \beta_{\mathsf{M}} < 1$. For $i \in \{O, N\}$, let $c_{\max,i}$ and $c_{\min,i}$ denote the maximum and minimum weight of an edge in G_i, respectively. Let the edge weights in G_{O} and G_{M} agree except for $k < |V| - 1$ edges. Let $\beta_{\mathsf{H}} := \max\{\beta_{\mathsf{O}}, \beta_{\mathsf{M}}\} < 1$ and $\beta_{\mathsf{L}} := \min\{\beta_{\mathsf{O}}, \beta_{\mathsf{M}}\}$. Then, it is a $\left(1 + \frac{2k\beta_{\mathsf{H}}^2 - k(1-\beta_{\mathsf{L}})(1-\beta_{\mathsf{H}})}{(1-\beta_{\mathsf{L}})(1-\beta_{\mathsf{H}})|V|}\right)$-approximation to simply output $\overline{C}$.*

Proof. First of all, observe that for $i \in \{\mathsf{L}, \mathsf{H}\}$ the following inequality holds:

$$\frac{\beta_i^2}{(1-\beta_i)} \le \frac{\beta_{\mathsf{H}}^2}{(1-\beta_{\mathsf{H}})(1-\beta_{\mathsf{L}})}. \tag{3}$$

Indeed we have $1 \le \frac{1}{(1-\beta_{\mathsf{L}})}$, and then by multiplying both sides by $\frac{\beta_{\mathsf{H}}^2}{(1-\beta_{\mathsf{H}})}$ we get

$$\frac{\beta_{\mathsf{H}}^2}{(1-\beta_{\mathsf{H}})} \le \frac{\beta_{\mathsf{H}}^2}{(1-\beta_{\mathsf{H}})(1-\beta_{\mathsf{L}})}.$$

Moreover

$$\beta_{\mathsf{L}}^2 \le \beta_{\mathsf{H}}^2 \le \frac{\beta_{\mathsf{H}}^2}{(1-\beta_{\mathsf{H}})}$$

from which

$$\frac{\beta_{\mathsf{L}}^2}{(1-\beta_{\mathsf{L}})} \le \frac{\beta_{\mathsf{H}}^2}{(1-\beta_{\mathsf{H}})(1-\beta_{\mathsf{L}})}.$$

Suppose now that $e_1, e_2, ..., e_k$ are the edges whose weights are altered. The proof is by cases.

Case 1: Let $c_{\mathsf{M}}(e_i) > c_{\mathsf{O}}(e_i)$ for $i = 1, ..., k$, and then $\textsc{Opt}_{G_{\mathsf{O}}} \leq \textsc{Opt}_{G_{\mathsf{M}}}$. Let $A = \{e_{i_1}, ..., e_{i_s}\} \subseteq \{e_1, e_2, ..., e_k\}$ be the subset of altered edges that are part of $\overline{C}$ (note that if $A = \varnothing$ then $\overline{C}$ is an optimal cycle in G_{M}), and let $c_{\mathsf{M}}(\overline{C}) = \sum_{e \in \overline{C}} c_{\mathsf{M}}(e)$. Then

$$\begin{aligned} c_{\mathsf{M}}(\overline{C}) &= \textsc{Opt}_{G_{\mathsf{O}}} + \sum_{e \in A} (c_{\mathsf{M}}(e) - c_{\mathsf{O}}(e)) \\ &\leq \textsc{Opt}_{G_{\mathsf{O}}} + |A|(c_{\max,\mathsf{M}} - c_{\min,\mathsf{O}}) \\ &\leq \textsc{Opt}_{G_{\mathsf{M}}} + |A|(c_{\max,\mathsf{M}} - c_{\min,\mathsf{O}}) \end{aligned}$$

and since $c_{\max,\mathsf{M}} \geq c_{\min,\mathsf{O}}$ and $|A| \leq k$

$$\begin{aligned} &\leq \textsc{Opt}_{G_{\mathsf{M}}} + kc_{\max,\mathsf{M}} - kc_{\min,\mathsf{O}} && (4) \\ &\overset{(2)}{\leq} \textsc{Opt}_{G_{\mathsf{M}}} + \frac{k}{1-\beta_{\mathsf{M}}} c_{\max,\mathsf{O}} - kc_{\min,\mathsf{O}} \\ &\overset{(1)}{\leq} \textsc{Opt}_{G_{\mathsf{M}}} + \frac{k}{(1-\beta_{\mathsf{M}})} \frac{2\beta_{\mathsf{O}}^2}{(1-\beta_{\mathsf{O}})} c_{\min,\mathsf{O}} - kc_{\min,\mathsf{O}}. && (5) \end{aligned}$$

But we also have

$$\begin{aligned} c_{\mathsf{M}}(\overline{C}) &\overset{(4)}{\leq} \textsc{Opt}_{G_{\mathsf{M}}} + kc_{\max,\mathsf{M}} - kc_{\min,\mathsf{O}} \\ &\overset{(1)}{\leq} \textsc{Opt}_{G_{\mathsf{M}}} + \frac{2k\beta_{\mathsf{M}}^2}{(1-\beta_{\mathsf{M}})} c_{\min,\mathsf{M}} - kc_{\min,\mathsf{O}} \\ &\overset{(2)}{\leq} \textsc{Opt}_{G_{\mathsf{M}}} + \frac{2k\beta_{\mathsf{M}}^2}{(1-\beta_{\mathsf{M}})} \frac{1}{(1-\beta_{\mathsf{O}})} c_{\min,\mathsf{O}} - kc_{\min,\mathsf{O}}. && (6) \end{aligned}$$

The combination of (5) and (6) yields

$$\begin{aligned} c_{\mathsf{M}}(\overline{C}) &\leq \textsc{Opt}_{G_{\mathsf{M}}} + \frac{\min\{2k\beta_{\mathsf{M}}^2, 2k\beta_{\mathsf{O}}^2\}}{(1-\beta_{\mathsf{M}})(1-\beta_{\mathsf{O}})} c_{\min,\mathsf{O}} - kc_{\min,\mathsf{O}} \\ &\leq \textsc{Opt}_{G_{\mathsf{M}}} + \left(\frac{2k\beta_{\mathsf{L}}^2}{(1-\beta_{\mathsf{H}})(1-\beta_{\mathsf{L}})} - k \right) \frac{\textsc{Opt}_{G_{\mathsf{M}}}}{|V|} \\ &\leq \left(1 + \frac{2k\beta_{\mathsf{H}}^2 - k(1-\beta_{\mathsf{H}})(1-\beta_{\mathsf{L}})}{(1-\beta_{\mathsf{H}})(1-\beta_{\mathsf{L}})|V|} \right) \textsc{Opt}_{G_{\mathsf{M}}}. \end{aligned}$$

Case 2: Now suppose that $c_{\mathsf{M}}(e_i) < c_{\mathsf{O}}(e_i)$ for $i = 1, ..., k$. Let $A_{D_1} = \{e_{i_1}, ..., e_{i_s}\} \subseteq \{e_1, e_2, ..., e_k\}$ be the subset of altered edges that are part of $\overline{C}$, and let $A_{D_2} = \{e_1, e_2, ..., e_k\} \backslash A_{D_1}$. For $i \in \{1, 2\}$, let $\varDelta A_{D_i} = \sum_{e \in A_{D_i}} (c_{\mathsf{O}}(e) - c_{\mathsf{M}}(e))$. Note that if $A_{D_1} = \{e_1, e_2, ..., e_k\}$ then $\overline{C}$ is an optimal cycle in G_{M}. Observe that

$$\begin{aligned} \textsc{Opt}_{G_{\mathsf{O}}} &\leq \textsc{Opt}_{G_{\mathsf{M}}} + |\varDelta A_{D_1} + \varDelta A_{D_2}| \\ &= \textsc{Opt}_{G_{\mathsf{M}}} + \sum_{e \in A_{D_1}} (c_{\mathsf{O}}(e) - c_{\mathsf{M}}(e)) + \sum_{e \in A_{D_2}} (c_{\mathsf{O}}(e) - c_{\mathsf{M}}(e)). && (7) \end{aligned}$$

Hence

$$\begin{aligned}
c_{\mathsf{M}}(\overline{C}) &= \textsc{Opt}_{G_{\mathsf{O}}} + \sum_{e\in A_{D_1}} (c_{\mathsf{M}}(e) - c_{\mathsf{O}}(e)) \\
&\overset{(7)}{\leq} \textsc{Opt}_{G_{\mathsf{M}}} + \sum_{e\in A_{D_1}} (c_{\mathsf{O}}(e) - c_{\mathsf{M}}(e)) + \sum_{e\in A_{D_2}} (c_{\mathsf{O}}(e) - c_{\mathsf{M}}(e)) \\
&+ \sum_{e\in A_{D_1}} (c_{\mathsf{M}}(e) - c_{\mathsf{O}}(e)) \\
&\leq \textsc{Opt}_{G_{\mathsf{M}}} + |A_{D_2}|(c_{\max,\mathsf{O}} - c_{\min,\mathsf{M}})
\end{aligned}$$

and since $c_{\max,\mathsf{O}} > c_{\min,\mathsf{M}}$ and $|A_{D_2}| \leq k$

$$\begin{aligned}
&\leq \textsc{Opt}_{G_{\mathsf{M}}} + kc_{\max,\mathsf{O}} - kc_{\min,\mathsf{M}} && (8)\\
&\overset{(2)}{\leq} \textsc{Opt}_{G_{\mathsf{M}}} + \frac{k}{(1-\beta_{\mathsf{O}})} c_{\max,\mathsf{M}} - kc_{\min,\mathsf{M}} \\
&\overset{(1)}{\leq} \textsc{Opt}_{G_{\mathsf{M}}} + \frac{k}{(1-\beta_{\mathsf{O}})} \frac{2\beta_{\mathsf{M}}^2}{(1-\beta_{\mathsf{M}})} c_{\min,\mathsf{M}} - kc_{\min,\mathsf{M}}. && (9)
\end{aligned}$$

But we also have

$$\begin{aligned}
c_{\mathsf{M}}(\overline{C}) &\overset{(8)}{\leq} \textsc{Opt}_{G_{\mathsf{M}}} + kc_{\max,\mathsf{O}} - kc_{\min,\mathsf{M}} \\
&\overset{(1)}{\leq} \textsc{Opt}_{G_{\mathsf{M}}} + \frac{2k\beta_{\mathsf{O}}^2}{(1-\beta_{\mathsf{O}})} c_{\min,\mathsf{O}} - kc_{\min,\mathsf{M}} \\
&\overset{(2)}{\leq} \textsc{Opt}_{G_{\mathsf{M}}} + \frac{2k\beta_{\mathsf{O}}^2}{(1-\beta_{\mathsf{O}})} \frac{1}{(1-\beta_{\mathsf{M}})} c_{\min,\mathsf{M}} - kc_{\min,\mathsf{M}}. && (10)
\end{aligned}$$

The combination of (9) and (10) yields

$$\begin{aligned}
c_{\mathsf{M}}(\overline{C}) &\leq \textsc{Opt}_{G_{\mathsf{M}}} + \frac{\min\{2k\beta_{\mathsf{M}}^2, 2k\beta_{\mathsf{O}}^2\}}{(1-\beta_{\mathsf{M}})(1-\beta_{\mathsf{O}})} c_{\min,\mathsf{M}} - kc_{\min,\mathsf{M}} \\
&\leq \textsc{Opt}_{G_{\mathsf{M}}} + \left(\frac{2k\beta_{\mathsf{L}}^2}{(1-\beta_{\mathsf{H}})(1-\beta_{\mathsf{L}})} - k \right) c_{\min,\mathsf{M}} \\
&\leq \textsc{Opt}_{G_{\mathsf{M}}} + \left(\frac{2k\beta_{\mathsf{L}}^2}{(1-\beta_{\mathsf{H}})(1-\beta_{\mathsf{L}})} - k \right) \frac{\textsc{Opt}_{G_{\mathsf{M}}}}{|V|} \\
&\leq \left(1 + \frac{2k\beta_{\mathsf{H}}^2 - k(1-\beta_{\mathsf{H}})(1-\beta_{\mathsf{L}})}{(1-\beta_{\mathsf{H}})(1-\beta_{\mathsf{L}})|V|} \right) \textsc{Opt}_{G_{\mathsf{M}}}.
\end{aligned}$$

Case 3: Now suppose that some of the edges whose weights are altered increase their weight, while others decrease it. Let $A_I = \{f_1, f_2, ..., f_r\}$ be the edges such that $c_{\mathsf{M}}(f_i) > c_{\mathsf{O}}(f_i)$ for $i = 1, ..., r$, while let $A_D = \{g_1, g_2, ..., g_{k-r}\}$

be the edges for which $c_{\mathsf{M}}(g_i) < c_{\mathsf{O}}(g_i)$ for $i = 1, ..., k-r$. Let $A_{I_1} = \{f_{i_1}, ..., f_{i_s}\} \subseteq A_I$, and $A_{D_1} = \{g_{z_1}, ..., g_{z_t}\} \subseteq A_D$ be the subsets of altered edges that belong to $\overline{C}$, and let $A_{D_2} = A_D \setminus A_{D_1}$. Note that if $A_{I_1} = \varnothing$ and $A_{D_1} = \{e_1, e_2, ..., e_k\}$, then $\overline{C}$ is an optimal cycle in G_{M}. Let $G_I = (V, E, c_I)$ be the graph obtained from G_{O} by modifying only the edges that belong to A_I, i.e., such that $c_I(f_i) = c_{\mathsf{M}}(f_i) > c_{\mathsf{O}}(f_i)$ for $i = 1, ..., r$, and $c_I(e) = c_{\mathsf{O}}(e)$ for any $e \in E \setminus A_I$. Then, $\textsc{Opt}_{G_{\mathsf{O}}} \leq \textsc{Opt}_{G_I}$. Finally, let $\varDelta A_D = \sum_{g \in A_D} c_{\mathsf{O}}(g) - \sum_{g \in A_D} c_{\mathsf{M}}(g)$. Observe that

$$\begin{aligned}\textsc{Opt}_{G_I} &\leq \textsc{Opt}_{G_{\mathsf{M}}} + |\varDelta A_D| = \textsc{Opt}_{G_{\mathsf{M}}} + \sum_{g \in A_D} (c_{\mathsf{O}}(g) - c_{\mathsf{M}}(g)) \\ &= \textsc{Opt}_{G_{\mathsf{M}}} + \sum_{g \in A_{D_1}} (c_{\mathsf{O}}(g) - c_{\mathsf{M}}(g)) + \sum_{g \in A_{D_2}} (c_{\mathsf{O}}(g) - c_{\mathsf{M}}(g)). \qquad (11)\end{aligned}$$

Then, we have that

$$\begin{aligned}c_{\mathsf{M}}(\overline{C}) &= \textsc{Opt}_{G_{\mathsf{O}}} + \sum_{f \in A_{I_1}} (c_{\mathsf{M}}(f) - c_{\mathsf{O}}(f)) + \sum_{g \in A_{D_1}} (c_{\mathsf{M}}(g) - c_{\mathsf{O}}(g)) \\ &\leq \textsc{Opt}_{G_I} + \sum_{f \in A_{I_1}} (c_{\mathsf{M}}(f) - c_{\mathsf{O}}(f)) + \sum_{g \in A_{D_1}} (c_{\mathsf{M}}(g) - c_{\mathsf{O}}(g)) \\ &\overset{(11)}{\leq} \textsc{Opt}_{G_{\mathsf{M}}} + \sum_{g \in A_{D_1}} (c_{\mathsf{O}}(g) - c_{\mathsf{M}}(g)) + \sum_{g \in A_{D_2}} (c_{\mathsf{O}}(g) - c_{\mathsf{M}}(g)) \\ &\quad + \sum_{f \in A_{I_1}} (c_{\mathsf{M}}(f) - c_{\mathsf{O}}(f)) + \sum_{g \in A_{D_1}} (c_{\mathsf{M}}(g) - c_{\mathsf{O}}(g)) \\ &\leq \textsc{Opt}_{G_{\mathsf{M}}} + |\varDelta A_{I_1}|(c_{\max,\mathsf{M}} - c_{\min,\mathsf{O}}) + |\varDelta A_{D_2}|(c_{\max,\mathsf{O}} - c_{\min,\mathsf{M}})\end{aligned}$$

and since $|A_{I_1}| \leq r$, $c_{\max,\mathsf{M}} > c_{\min,\mathsf{O}}$, $|A_{D_2}| \leq k - r$, and $c_{\max,\mathsf{O}} > c_{\min,\mathsf{M}}$

$$\leq \textsc{Opt}_{G_{\mathsf{M}}} + rc_{\max,\mathsf{M}} - rc_{\min,\mathsf{O}} + (k-r)c_{\max,\mathsf{O}} - (k-r)c_{\min,\mathsf{M}}. \qquad (12)$$

We now analyze the various subcases, depending on the four feasible orders of $c_{\min,\mathsf{M}}, c_{\max,\mathsf{M}}, c_{\min,\mathsf{O}}$, and $c_{\max,\mathsf{O}}$.

Case 3.1: Let $c_{\min,\mathsf{M}} \leq c_{\min,\mathsf{O}} \leq c_{\max,\mathsf{O}} \leq c_{\max,\mathsf{M}}$. Then, from (12) we have

$$\begin{aligned}c_{\mathsf{M}}(\overline{C}) &\leq \textsc{Opt}_{G_{\mathsf{M}}} + kc_{\max,\mathsf{M}} - rc_{\min,\mathsf{M}} - (k-r)c_{\min,\mathsf{M}} \\ &\overset{(1)}{\leq} \textsc{Opt}_{G_{\mathsf{M}}} + \frac{2k\beta_{\mathsf{M}}^2}{(1-\beta_{\mathsf{M}})} c_{\min,\mathsf{M}} - kc_{\min,\mathsf{M}} \\ &\overset{(3)}{\leq} \textsc{Opt}_{G_{\mathsf{M}}} + \left(\frac{2k\beta_{\mathsf{H}}^2}{(1-\beta_{\mathsf{H}})(1-\beta_{\mathsf{L}})} - k \right) c_{\min,\mathsf{M}} \\ &\leq \left(1 + \frac{2k\beta_{\mathsf{H}}^2 - k(1-\beta_{\mathsf{H}})(1-\beta_{\mathsf{L}})}{(1-\beta_{\mathsf{H}})(1-\beta_{\mathsf{L}})|V|} - k \right) \textsc{Opt}_{G_{\mathsf{M}}}.\end{aligned}$$

Case 3.2: Let $c_{\min,\mathsf{O}} \leq c_{\min,\mathsf{M}} \leq c_{\max,\mathsf{M}} \leq c_{\max,\mathsf{O}}$. Then, from (12) we have

$$\begin{aligned} c_{\mathsf{M}}(\overline{C}) &\leq \textsc{Opt}_{G_{\mathsf{M}}} + kc_{\max,\mathsf{O}} - rc_{\min,\mathsf{O}} - (k-r)c_{\min,\mathsf{O}} \\ &\overset{(1)}{\leq} \textsc{Opt}_{G_{\mathsf{M}}} + \frac{2k\beta_{\mathsf{O}}^2}{(1-\beta_{\mathsf{O}})} c_{\min,\mathsf{O}} - kc_{\min,\mathsf{O}} \\ &\overset{(3)}{\leq} \textsc{Opt}_{G_{\mathsf{M}}} + \left(\frac{2k\beta_{\mathsf{H}}^2}{(1-\beta_{\mathsf{H}})(1-\beta_{\mathsf{L}})} - k \right) c_{\min,\mathsf{O}} \\ &\leq \textsc{Opt}_{G_{\mathsf{M}}} + \left(\frac{2k\beta_{\mathsf{H}}^2}{(1-\beta_{\mathsf{H}})(1-\beta_{\mathsf{L}})} - k \right) c_{\min,\mathsf{M}} \\ &\leq \left(1 + \frac{2k\beta_{\mathsf{H}}^2 - k(1-\beta_{\mathsf{H}})(1-\beta_{\mathsf{L}})}{(1-\beta_{\mathsf{H}})(1-\beta_{\mathsf{L}})|V|} - k \right) \textsc{Opt}_{G_{\mathsf{M}}}. \end{aligned}$$

Case 3.3: Let $c_{\min,\mathsf{O}} \leq c_{\min,\mathsf{M}} \leq c_{\max,\mathsf{O}} \leq c_{\max,\mathsf{M}}$. Then, from (12) we have

$$c_{\mathsf{M}}(\overline{C}) \leq \textsc{Opt}_{G_{\mathsf{M}}} + kc_{\max,\mathsf{M}} - rc_{\min,\mathsf{O}} - (k-r)c_{\min,\mathsf{O}}$$

and then the rest of the proof follows the steps after inequality (4).

Case 3.4: Let $c_{\min,\mathsf{M}} \leq c_{\min,\mathsf{O}} \leq c_{\max,\mathsf{M}} \leq c_{\max,\mathsf{O}}$. Then, from (12) we have

$$c_{\mathsf{M}}(\overline{C}) \leq \textsc{Opt}_{G_{\mathsf{M}}} + kc_{\max,\mathsf{O}} - rc_{\min,\mathsf{M}} - (k-r)c_{\min,\mathsf{M}}$$

and then the rest of the proof follows the steps after inequality (8).

□

The above result allows to construct the following algorithm, similar to that given in [9]:

1. Compute $\beta_{\mathsf{H}}, \beta_{\mathsf{L}}$ and $|V|$ for the input instance;
2. By Theorem 1, compute the approximation which can be guaranteed by simply returning $\overline{C}$;
3. If it is good enough, then output $\overline{C}$, otherwise perform exhaustive search for an optimal solution.

Theorem 2. *If β_{O} and β_{M} do not depend on $|V|$, and $k = O(1)$, the above algorithm is a* PTAS *for k-*EM-SMTSP.

Proof. From the assumptions

$$c = \frac{2k\beta_{\mathsf{H}}{}^2 - k(1-\beta_{\mathsf{L}})(1-\beta_{\mathsf{H}})}{(1-\beta_{\mathsf{L}})(1-\beta_{\mathsf{H}})}$$

is constant (and can actually be computed in $O(|V|^3)$). Then, given an instance of k-EM-SMTSP, for any $\varepsilon > 0$, we proceed as follows:

1. if $|V| \geq \frac{c}{\varepsilon}$, we know from Theorem 1 that returning $\overline{C}$ guarantees an approximation factor of $1 + \frac{c}{|V|} \leq 1 + \varepsilon$, and thus this case costs $O(1)$ time;
2. otherwise, we perform exhaustive search for a new optimal solution, and this costs $O(|V|!) = O(|V|^{\frac{c}{\varepsilon}+1})$ time.

From this, the claim follows. □

3 Experimental Results

To assess experimentally the obtained theoretical results, it was necessary to create a set of input instances for strengthened metric graphs, along with an optimal solution for each of them. For the computation of an optimal solution, it was used the well known *Concorde TSP Solver* [3], the fastest solver for TSP to date. The site contains also a library, *TSPLIB* [1], which provides a set of instances for TSP coming from various sources. In particular, TSPLIB provides three strengthened metric instances, i.e., *si175*, *si535*, and *si1032* (notice that all of them are synthetic instances having a number of nodes as specified by their name). Additionally, by using Google Maps© to get the latitude and the longitude of any given site on the Earth, we have built three more strengthened metric instances, in which selected nodes are specified by their coordinates, and edge weights are given by the geodetic distance between the nodes. In this way we managed to build three instances: (i) the set of sites which are part of the *World Heritage List* [2], that we called precisely *UNESCO* (this instance was selected by imaging a virtual tour of the most beautiful places in the World); (ii) and (iii) the set of nuclear power plants operating in Europe and America (both North and South),respectively, and that we called *EuNPP* and *AmNPP* (these two instances were selected by imaging a drone that flies over these sites to make aerial monitoring). Instance *UNESCO* is composed by 193 nodes,[2] and has a strength factor of 0.99994. Instance *EuNPP* consists of 79 nodes, and has a strength factor of 0.99987. Finally, *AmNPP* consists of 74 nodes, and has a strength factor of 0.99990. Obviously, also for these instances we have generated the optimal solution through the use of the Concorde TSP Solver.

On the above mentioned six instances we simulated a variation of the edge weights as follows. We extracted randomly a number between 0 and 1, and: (i) if this number was greater than 0.5, then we selected at random an edge belonging to the optimal cycle, and we incremented its weight by a random value between 1 and its current weight; otherwise (ii) we selected at random an edge not belonging to the optimal cycle, and we decremented its weight by a random value between 1 and the weight of the edge itself minus 1. In this way, we avoided the favorable (but meaningless) cases of either reducing the weight of edges belonging to the optimal original cycle, or increasing the weight of edges not belonging to it.

[2] We have not included all the 936 World Heritage sites, because many of them are very close, if not overlapping.

We then performed a series of tests by changing the weight of $1, 2, 3, 5$ and 10 edges (the case of modifying just one edge is included in order to perform a comparison with the approximation ratio given in [9]). All over these experiments the value of β_{M} was always not less than the value of β_{O}. This can be easily explained, since on one hand the instances can have more than one triple of vertices matching the strength factor of the associated graph, and on the other hand reducing β_{O} is possible only by altering exactly the weights of the edges belonging to these triples.

In the following we provide a series of charts showing, in addition to the estimated approximation ratio given by Theorem 1, the actual approximation which is achieved by returning the optimal cycle of the original graph. Notice that we have run ten different executions for each fixed number of modified edges, and we sorted increasingly the obtained ratios. Notice also that for the last three instances the strength factor of the original graph was very close to 1, and this prevented us (although a very large number of attempts) to generate randomly a modified graph obeying the strengthened triangle inequality for the modification of 10 edges. From the set of experiments, it emerges clearly the stability of the quality of the original optimal cycle, which performs much better than that we theoretically expect.

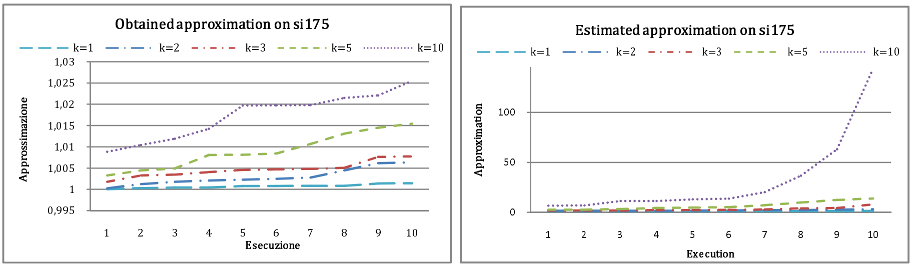


Fig. 1. Results obtained for *si175* (175 nodes, $\beta_{\mathsf{O}} = 0.84375$, $0.84375 \leq \beta_{\mathsf{M}} \leq 0.99492$)

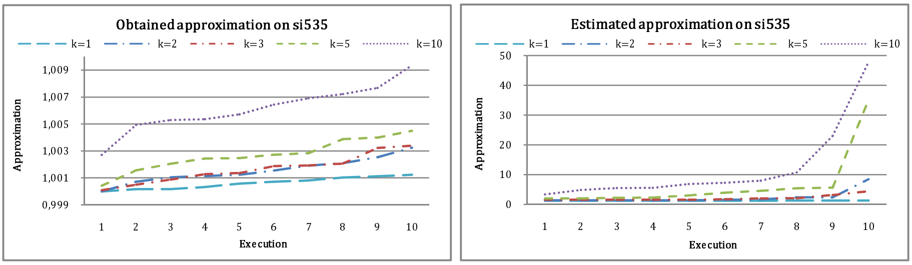


Fig. 2. Results obtained for *si535* (535 nodes, $\beta_{\mathsf{O}} = 0.87958$, $0.87958 \leq \beta_{\mathsf{M}} \leq 0.99549$)

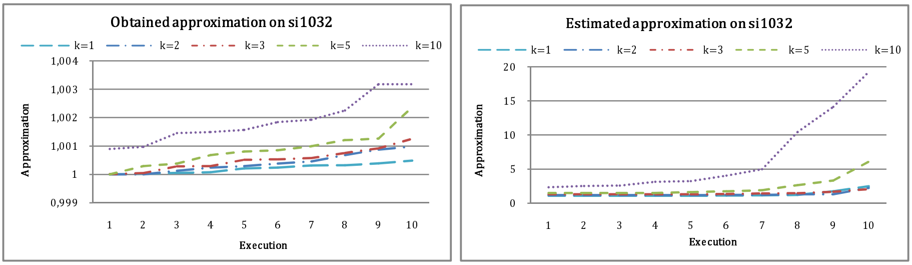


Fig. 3. Results obtained for *si1032* (1032 nodes, $\beta_O = 0.87888$, $0.87888 \leq \beta_M \leq 0.99173$)

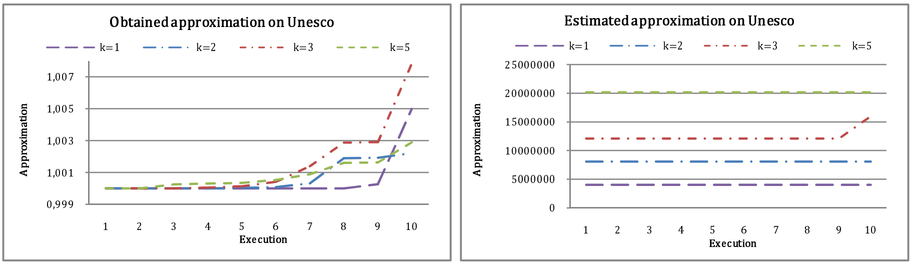


Fig. 4. Results obtained for *UNESCO* (193 nodes, $\beta_O = 0.99994$, $0.99994 \leq \beta_M \leq 0.99996$)

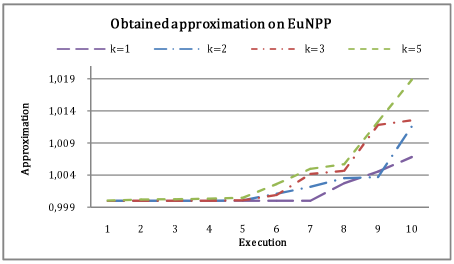


Fig. 5. Results obtained for *EuNPP* (79 nodes, $\beta_O = \beta_M = 0.99987$). The estimated approximation is 1629366, 3258732, 4888097, 8146828, for $k = 1, 2, 3, 5$, respectively

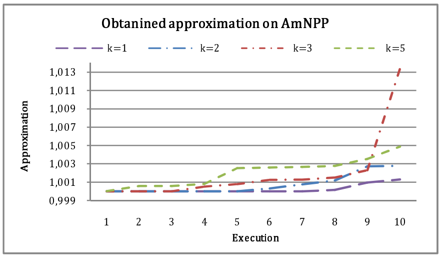


Fig. 6. Results obtained for *AmNPP* (74 nodes, $\beta_O = \beta_M = 0,99990$). The estimated approximation is 2845826, 56916512, 8537476, 14229126, for $k = 1, 2, 3, 5$, respectively

References

1. TSPLIB, http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/
2. UNESCO World Heritage Sites, http://en.wikipedia.org/wiki/World_Heritage_Sites
3. Concorde TSP Solver (2003), http://www.tsp.gatech.edu/concorde
4. Archetti, C., Bertazzi, L., Speranza, M.G.: Reoptimizing the traveling salesman problem. Networks 42(3), 154–159 (2003)
5. Ausiello, G., Escoffier, B., Monnot, J., Paschos, V.T.: Reoptimization of Minimum and Maximum Traveling Salesman's Tours. In: Arge, L., Freivalds, R. (eds.) SWAT 2006. LNCS, vol. 4059, pp. 196–207. Springer, Heidelberg (2006)
6. Bender, M.A., Chekuri, C.: Performance Guarantees for the TSP with a Parameterized Triangle Inequality. In: Dehne, F., Gupta, A., Sack, J.-R., Tamassia, R. (eds.) WADS 1999. LNCS, vol. 1663, pp. 80–85. Springer, Heidelberg (1999)
7. Bilò, D., Böckenhauer, H.-J., Hromkovič, J., Královič, R., Mömke, T., Widmayer, P., Zych, A.: Reoptimization of Steiner Trees. In: Gudmundsson, J. (ed.) SWAT 2008. LNCS, vol. 5124, pp. 258–269. Springer, Heidelberg (2008)
8. Bilò, D., Widmayer, P., Zych, A.: Reoptimization of Weighted Graph and Covering Problems. In: Bampis, E., Skutella, M. (eds.) WAOA 2008. LNCS, vol. 5426, pp. 201–213. Springer, Heidelberg (2009)
9. Böckenhauer, H.-J., Forlizzi, L., Hromkovič, J., Kneis, J., Kupke, J., Proietti, G., Widmayer, P.: On the approximability of TSP on local modifications of optimally solved instances. Algorithmic Operations Research 2, 83–93 (2007)
10. Böckenhauer, H.-J., Hromkovič, J., Klasing, R., Seibert, S., Unger, W.: Approximation algorithms for TSP with sharpened triangle inequality. Information Processing Letters 75, 133–138 (2000)
11. Böckenhauer, H.-J., Hromkovič, J., Klasing, R., Seibert, S., Unger, W.: Towards the notion of stability of approximation for hard optimization tasks and the traveling salesman problem. Theoretical Computer Science 285(1), 3–24 (2002)
12. Böckenhauer, H.-J., Hromkovič, J., Královic, R., Mömke, T., Rossmanith, P.: Reoptimization of Steiner trees: Changing the terminal set. Theoretical Computer Science 410(36), 3428–3435 (2009)
13. Böckenhauer, H.-J., Komm, D.: Reoptimization of the metric deadline TSP. J. Discrete Algorithms 8(1), 87–100 (2010)
14. Christofides, N.: Worst-case analysis of a new heuristic for the traveling salesman problem. Technical report, Graduate School of Industrial Administration, Carnegy–Mellon University (1976)

Engineering a New Loop-Free Shortest Paths Routing Algorithm*

Gianlorenzo D'Angelo[1], Mattia D'Emidio[2],
Daniele Frigioni[2], and Vinicio Maurizio[2]

[1] MASCOTTE Project I3S(CNRS/UNSA)/INRIA
gianlorenzo.d_angelo@inria.fr
[2] Department of Electrical and Information Engineering, University of L'Aquila
{mattia.demidio,daniele.frigioni}@univaq.it,
vinicio.maurizio@cc.univaq.it

Abstract. We present LFR (Loop Free Routing), a new loop-free distance vector routing algorithm, which is able to update the shortest paths of a distributed network with n nodes in fully dynamic scenarios. If Φ is the total number of nodes *affected* by a set of updates to the network, and ϕ is the maximum number of destinations for which a node is affected, then LFR requires $O(\Phi \cdot \Delta)$ messages and $O(n + \phi \cdot \Delta)$ space per node, where Δ is the maximum degree of the nodes of the network.

We experimentally compare LFR with DUAL, one of the most popular loop-free distance vector algorithms, which is part of CISCO's EIGRP protocol and requires $O(\Phi \cdot \Delta)$ messages and $\Theta(n \cdot \Delta)$ space per node. The experiments are based on both real-world and artificial instances and show that LFR is always the best choice in terms of memory requirements, while in terms of messages LFR outperforms DUAL on real-world instances, whereas DUAL is the best choice on artificial instances.

1 Introduction

Updating shortest paths in a distributed network whose topology dynamically changes over the time is considered crucial in today's communication networks. This problem has been widely studied in the literature, and the solutions found can be classified as *distance-vector* and *link-state*.

Distance-vector algorithms require that a node knows the distance from each of its neighbors to every destination and stores them in a data structure called *routing table*; a node uses its own routing table to compute the distance and the next node in the shortest path to each destination. Most of the known distance-vector solutions (e.g., see [6,9,10,15,16]) are based on the classical Distributed Bellman-Ford method (DBF), originally introduced in the Arpanet [12], which is implemented in the RIP protocol. The convergence of DBF can be very slow (possibly infinite) due to the well-known *looping* phenomenon which occurs when

* Support for the IPv4 Routed/24 Topology Dataset is provided by NSF, US Department of Homeland Security, WIDE Project, Cisco Systems, and CAIDA.

R. Klasing (Ed.): SEA 2012, LNCS 7276, pp. 123–134, 2012.

a path induced by the routing table entries visits the same node more than once before reaching the intended destination. Furthermore, if the nodes of the network are not synchronized, even though no change occurs in the network, the overall number of messages sent by DBF is exponential in the size of the network (e.g., see [1]).

Link-state algorithms, as for example the *Open Shortest Path First (OSPF)* protocol widely used in the Internet (e.g., see [13]), require that a node must know the entire network topology to compute its distance to any destination, usually running the centralized Dijkstra's algorithm. Link-state algorithms are free of looping, but each node needs to receive and store up-to-date information on the entire network topology after a change, thus requiring quadratic space per node. This is achieved by broadcasting each change of the network topology to all nodes [13,18] and by using a centralized algorithm for shortest paths.

Related Works. In the last years, there has been a renewed interest in devising new efficient light-weight loop-free distributed shortest paths solutions for large-scale Ethernet networks (see, e.g., [3,7,8,17,19,20]), where usually the routing devices have limited storage capabilities, and hence distance-vector algorithms seem to be an attractive alternative to link-state solutions. For example, in [17] a new technique has been introduced, named DIV, which is not a routing algorithm by itself, rather it can run on top of any routing algorithm to guarantee loop freedom. A distance vector algorithm has been recently introduced in [5] and successively developed in [4], where it has been named DUST (Distributed Update of Shortest paThs), which suffers of looping, although it has been designed to heuristically reduce the cases where this phenomenon occurs.

Despite the renewed interest of the last years, the most important distance vector algorithm in the literature is surely DUAL (Diffuse Update ALgorithm) [9], which is free of looping and is indeed part of CISCO's widely used EIGRP protocol. DUAL has been experimentally tested in [4] against DUST and DBF in various artificial and real-world scenarios. It has been shown that DUST is always the best choice in terms of space per node. In terms of messages, DUST is the best choice on those real-world topologies in which it does not fall in looping, while DUAL is better than DUST and DBF in all other cases.

Results of the Paper. We propose a new loop-free distance vector algorithm, named LFR (Loop Free Routing), which is able to update the shortest paths of a distributed network subject to arbitrary modifications on the edges of the network. Let us denote by n the number of nodes in the network, by $\varDelta$ the maximum node degree, by $\varPhi$ the number of nodes *affected* by a sequence of updates on the edges of the network, that is the nodes changing their routing table during that sequence, and by ϕ the maximum number of destinations for which a node is affected. Then LFR requires $O(\varPhi \cdot \varDelta)$ messages and $O(\varPhi)$ steps to converge and requires $O(n + \phi \cdot \varDelta)$ space per node. Compared with DUAL, LFR sends the same number of messages but requires less memory per node. In fact, DUAL requires $O(\varPhi \cdot \varDelta)$ messages and $O(\varPhi)$ steps to converge and $\varTheta(n \cdot \varDelta)$ space per node. Compared with DUST, LFR is better in terms of both number of messages sent and memory requirement per node. In fact, the number of messages sent by DUST cannot be bounded, as it suffers of looping, and its space requirement per node is $O(n \cdot \Delta)$.

From the experimental point of view, we conducted an extensive study with the aim of comparing the performances of the loop-free algorithms LFR and DUAL also in practical cases. Our simulations were performed in the OMNeT++ simulation environment [14]. As input to the algorithms, we used both real-world and artificial networks. In detail, we considered some of the Internet topologies of the *CAIDA IPv4 topology dataset* [11] (CAIDA - Cooperative Association for Internet Data Analysis provides data and tools for the analysis of the Internet infrastructure) and *Erdös-Rényi* random graphs [2]. The results of our experiments can be summarized as follows: in real-world networks LFR outperforms DUAL in terms of both number of messages and space occupancy per node. In the experiments, we observe that this is in part due to the topological structure of the CAIDA instances which are sparse and contain a high number of nodes of small degree. Therefore, we considered also *Erdös-Rényi* random instances with a variable degree of density. In this case, DUAL sends a number of messages smaller than that of LFR. However, the space requirements of DUAL grow drastically in these random networks while that of LFR are the same of real-word networks. Since CAIDA instances used in the experiments follow a power-law node degree distribution, we embedded the two algorithms in DLP (Distributed Leaf Pruning), a general framework recently proposed in [7] which can run on top of any routing algorithm and is able to reduce the number of messages sent by such algorithms in this kind of graphs. These further experiments show that the good performances of LFR in real-world instances improve when it is used in combination with DLP.

2 Preliminaries

We consider a network made of processors linked through communication channels that exchange data using a message passing model, in which: each processor can send messages only to its neighbors; messages are delivered to their destination within a finite delay but they might be delivered out of order; there is no shared memory among the nodes; the system is asynchronous, that is, a sender of a message does not wait for the receiver to be ready to receive the message.

Graph Notation. We represent the network by an undirected weighted graph $G = (V, E, w)$, where V is a finite set of n nodes, one for each processor, E is a finite set of m edges, one for each communication channel, and w is a weight function $w : E \rightarrow \mathbb{R}^+ \cup \{\infty\}$ on the edges. An edge in E that links nodes u and v is denoted as $\{u, v\}$. Given $v \in V$, $N(v)$ denotes the set of neighbors of v. A *shortest path* between nodes u and v is a path from u to v with the minimum weight. The *distance* $d(u, v)$ from u to v is the weight of a shortest path from u to v. Given two nodes $u, v \in V$, the *via* from u to v is the set of neighbors of u that belong to a shortest path from u to v. Formally: $via(u, v) \equiv \{z \in N(u) \mid d(u, v) = w(u, z) + d(z, v)\}$. We denote as $w^t()$, $d^t()$ and $via^t()$ an edge weight, a distance

```
Procedure: Update($u, s, \mathtt{D}_u[s]$)
Input: Node $v$ receives the message $update(u, s, \mathtt{D}_u[s])$ from $u$
if $STATE_v[s] = false$ then
    if $\mathtt{D}_v[s] > \mathtt{D}_u[s] + w(u, v)$ then Decrease($u, s, \mathtt{D}_u[s]$);
    else if $\mathtt{D}_v[s] < \mathtt{D}_u[s] + w(u, v)$ then Increase($u, s, \mathtt{D}_u[s]$);
```

Fig. 1. Pseudocode of procedure Update

and a via in G at time instant t, respectively. We denote a sequence of update operations on the edges of G by $\mathcal{C} = (c_1, c_2, ..., c_k)$. Assuming $G_0 \equiv G$, we denote as G_i, $0 \leq i \leq k$, the graph obtained by applying c_i to G_{i-1}. We consider the case in which c_i occurs at time $t_i \geq t_{i-1}$ and either increases or decreases the weight of $\{x_i, y_i\}$ by a quantity $\epsilon_i > 0$. The extension to *delete* and *insert* operations is straightforward. In what follows, given a sequence $\mathcal{C} = (c_1, c_2, ..., c_k)$ of update operations, we denote as $\phi_{c_i,s}$ the set of nodes that change the distance or the via to s as a consequence of c_i, formally: $\phi_{c_i,s} = \{v \in V \mid d^{t_i}(v, s) \neq d^{t_{i-1}}(v, s)$ or $via^{t_i}(v, s) \neq via^{t_{i-1}}(v, s)\}$. If $v \in \cup_{i=1}^{k} \cup_{s \in V} \phi_{c_i,s}$ we say that v is *affected* by c_i. We denote as $\Phi = \sum_{i=1}^{k} \sum_{s \in V} |\phi_{c_i,s}|$. Furthermore, given a generic destination s in V, we denote as $\phi_s = \cup_{i=1}^{k} \phi_{c_i,s}$ and by $\phi = \max_s |\phi_s|$. Note that a node can be affected for at most ϕ different destinations.

Conditions for Loop Freedom. A distance vector algorithm can be designed to be loop-free by using sufficient conditions as for example those described in [9]. In particular, we focus on snc (Source Node Condition), which can be implemented and work in combination with a distance vector algorithm that maintains at least the *routing table* and the so-called *topology table*. The routing table of a node v has two entries for each $s \in V$: the estimated distance $\mathtt{D}_v[s]$ between v and s in G; the node $\mathtt{VIA}_v[s] \in via(v, s)$ representing the via from v to s in G. We denote as $\mathtt{D}_v[s](t)$ and $\mathtt{VIA}_v[s](t)$ the estimated distance, and the estimated via at a certain time t. The topology table of v has to contain enough information for v to compute, for each $u \in N(v)$ and for each $s \in V$, the quantity $\mathtt{D}_u[s]$. These values are used in snc to determine whether a path is free of loops as follows: if, at time t, v needs to change $\mathtt{VIA}_v[s]$ for some $s \in V$, it can select as $\mathtt{VIA}_v[s](t)$ any neighbor $k \in N(v)$ satisfying the following *loop-free test*: $\mathtt{D}_k[s](t) + w^t(v, k) = \min_{v_i \in N(v)}\{\mathtt{D}_{v_i}[s](t) + w^t(v_i, v)\}$ and $\mathtt{D}_k[s](t) < \mathtt{D}_v[s](t)$. If no such neighbor exists, then $\mathtt{VIA}_v[s]$ does not change. Let $\mathtt{VIA}_G[s](t)$ be the directed subgraph of G induced by the set $\mathtt{VIA}_v[s](t)$, for each $v \in V$. In [9] it is proved that if $\mathtt{VIA}_G[s](t_0)$ is loop-free and snc is used when nodes change their via, then $\mathtt{VIA}_G[s](t)$ remains loop-free, for each time $t \geq t_0$.

3 The New Algorithm

In this section, we describe LFR which consists of four procedures named Update, Decrease, Increase and SendFeasibleDist, which are reported in Fig.s 1, 2, 3 and 4, resp. The algorithm is described wrt a source $s \in V$, and it starts every time a weight change $c_i \in \mathcal{C} = (c_1, c_2, ..., c_k)$ occurs on edge $\{x_i, y_i\}$.

```
Procedure: DECREASE($u, s, \mathtt{D}_u[s]$)
1 $\mathtt{D}_v[s] := \mathtt{D}_u[s] + w(u, v)$; $\mathtt{UD}_v[s] := \mathtt{D}_v[s]$; $\mathtt{VIA}_v[s] := u$;
2 foreach $k \in N(v) \setminus \{\mathtt{VIA}_v[s]\}$ do
3     send $update(v, s, \mathtt{D}_v[s])$ to $k$ ;
```

Fig. 2. Pseudo-code of procedure Decrease

Data Structures. LFR stores, for each node v, the arrays $\mathtt{D}_v[s]$ and $\mathtt{VIA}_v[s]$ plus, for each $s \in V$, the following data structures: $\mathtt{STATE}_v[s]$, which represents the state of node v wrt source s, v is in *active* state and $\mathtt{STATE}_v[s] = true$, if and only if it is performing procedure Increase or procedure SendFeasibleDist with respect to s; $\mathtt{UD}_v[s]$ which represents the distance from v to s through $\mathtt{VIA}_v[s]$. In particular, if v is active $\mathtt{UD}_v[s]$ is always greater or equal to $\mathtt{D}_v[s]$, otherwise they coincide; in addition, node v stores a temporary data structure $\mathtt{tempD}_v$ needed to implement the topology table. $\mathtt{tempD}_v$ is allocated for a certain s only when needed, that is when v is active wrt s, and it is deallocated when v turns back in passive state wrt s. The entry $\mathtt{tempD}_v[u][s]$ contains $\mathtt{UD}_u[s]$, for each $u \in N(v)$, and hence $\mathtt{tempD}_v$ requires Δ space per node for each source for which v is active. The number of nodes for which v is active is at most ϕ, thus giving an $O(n + \phi \cdot \Delta)$ bound for the space requirement per node, which is better than DUAL. In Section 4 we will show that also in real practical cases the space per node needed by LFR is always smaller than that of DUAL.

Description of LFR**.** Before LFR starts, at time $t < t_1$, we assume that, for each $v, s \in V$, $\mathtt{D}_v[s](t)$ and $\mathtt{VIA}_v[s](t)$ are correct, that is $\mathtt{D}_v[s](t) = d^t(v, s)$ and $\mathtt{VIA}_v[s](t) \in via^t(v, s)$. We focus the description on a source $s \in V$ and we assume that each node $v \in V$, at time t, is passive wrt s. The algorithm starts when the weight of an edge $\{x_i, y_i\}$ changes. As a consequence, x_i (y_i resp.) sends to y_i (x_i resp.) message $update(x_i, s, \mathtt{D}_{x_i}[s])$ ($update(y_i, s, \mathtt{D}_{y_i}[s])$ resp.). Messages received at a node wrt a source s are stored in a queue and processed in a FIFO order to guarantee mutual exclusion. If an arbitrary node v receives $update(u, s, \mathtt{D}_u[s])$ from $u \in N(v)$, then it performs procedure Update in Fig. 1. Basically, Update compares $\mathtt{D}_v[s]$ with $\mathtt{D}_u[s] + w(u, v)$ to determine whether v needs to update its estimated distance and its estimated via to s. If node v is active, the processing of the message is postponed by enqueueing it into the FIFO queue associated to s. Otherwise, if $\mathtt{D}_v[s] > \mathtt{D}_u[s] + w(u, v)$ (Line 2), then v performs procedure Decrease, while if $\mathtt{D}_v[s] < \mathtt{D}_u[s] + w(u, v)$ (Line 3) v performs procedure Increase. Finally, if node v is passive and $\mathtt{D}_v[s] = \mathtt{D}_u[s] + w(u, v)$ then there is more than one shortest path from v to s. In this case the message is discarded and the procedure ends.

When a node v performs procedure Decrease, it simply updates $\mathtt{D}$, $\mathtt{UD}$ and $\mathtt{VIA}$ by using the updated information provided by u. Then, the update is forwarded to all neighbors of v with the exception of $\mathtt{VIA}_v[s]$ (Line 3).

When a node v performs procedure Increase, it first verifies whether the update has been received from $\mathtt{VIA}_v[s]$ or not (Line 1). In fact, only in the affirmative case v changes its distance to s and needs to find a new via. To this

```
Procedure: INCREASE($u, s, \mathtt{D}_u[s]$)
1  if $\mathit{VIA}_v[s] = u$ then
2      $\mathtt{STATE}_v[s] := \mathit{true}$;
3      Allocate $\mathtt{tempD}_v[\cdot][s]$;
4      $\mathtt{tempD}_v[u][s] := \mathtt{D}_u[s]$;
5      $\mathtt{UD}_v[s] := \mathtt{tempD}_v[u][s] + w(u, v)$;
6      foreach $v_i \in N(v) \setminus \{\mathit{VIA}_v[s]\}$ do
7          receive $\mathtt{UD}_{v_i}[s]$ and store it in $\mathtt{tempD}_v[v_i][s]$ by sending $\mathit{get.dist}(v, s, \mathtt{UD}_v[s])$
           to $v_i$;
8      $\mathtt{Dmin} := \min_{u \in N(v)}\{\mathtt{tempD}_v[u][s] + w(u, v)\}$;
9      $\mathtt{VIAmin} := \arg\min_{u \in N(v)}\{\mathtt{tempD}_v[u][s] + w(u, v)\}$;
10     if $\mathit{tempD}_v[\mathit{VIAmin}][s] \geq D_v[s]$ then
11         foreach $v_i \in N(v) \setminus \{\mathit{VIA}_v[s]\}$ do
12             receive loop-free distance $\mathtt{UD}_{v_i}[s]$ and store it in $\mathtt{tempD}_v[v_i][s]$ by sending
               $\mathit{get.feasible.dist}(v, s, \mathtt{UD}_v[s])$ to $v_i$;
13         $\mathtt{Dmin} := \min_{u \in N(v)}\{\mathtt{tempD}_v[u][s] + w(u, v)\}$;
14         $\mathtt{VIAmin} := \arg\min_{u \in N(v)}\{\mathtt{tempD}_v[u][s] + w(u, v)\}$;
15     Deallocate $\mathtt{tempD}_v[\cdot][s]$;
16     $\mathtt{D}_v[s] := \mathtt{Dmin}$;
17     $\mathtt{UD}_v[s] := \mathtt{D}_v[s]$;
18     $\mathtt{VIA}_v[s] := \mathtt{VIAmin}$;
19     foreach $k \in N(v)$ do send $\mathit{update}(v, s, \mathtt{D}_v[s])$ to $k$;
20     $\mathtt{STATE}_v[s] := \mathit{false}$;
```

Fig. 3. Pseudo-code of procedure INCREASE

aim, node v becomes active, allocates the temporary data structure $\mathtt{tempD}_v$, and sets $\mathtt{UD}_v[s]$ to the current distance through $\mathtt{VIA}_v[s]$ (Lines 2–5). At this point, v first performs the so called Local-Computation (Lines 6–9), which involves all the neighbors of v. If the Local-Computation does not succeed, then node v initiates the so called Global-Computation (Lines 11–14), which involves in the worst case all nodes of the network.

In the Local-Computation, node v sends *get.dist* messages, carrying $\mathtt{UD}_v[s]$, to all its neighbors, except u. A neighbor $k \in N(v)$ that receives a *get.dist* message, immediately replies with the value $\mathtt{UD}_k[s]$, and if k is active, it updates $\mathtt{tempD}_k[v][s]$ to $\mathtt{UD}_v[s]$. When v receives these values from its neighbors, it stores them in $\mathtt{tempD}_v$, and it uses them to compute the minimal estimated distance $\mathtt{Dmin}$ to s and the neighbor $\mathtt{VIAmin}$ which gives such a distance (Lines 8–9).

At the end of the Local-Computation v checks whether a feasible via exists, according to SNC, by executing the loop-free test at line 10. If the test does not succeed, then v initiates the Global-Computation (Line 11), in which it entrusts the neighbors the task of finding a loop-free path. In this phase, v sends *get.feasible.dist*$(v, s, \mathtt{UD}_v[s])$ message to each of its neighbors. This message carries the value of the estimated distance through its current via. This distance is not guaranteed to be minimum but it is guaranteed to be loop-free. When v receives the answers to *get.feasible.dist* messages from its neighbors, again it stores them in $\mathtt{tempD}_v$ and it uses them to compute the minimal estimated distance $\mathtt{Dmin}$ to s and the neighbor $\mathtt{VIAmin}$ which gives such a distance (Lines 13–14).

```
Procedure: SendFeasibleDist$(u, s, \overline{D})$
Input: Node $v$ receives $get.feasible.dist(u, s, \mathtt{UD}_u[s])$ from $u$
1  if $VIA_v[s] = u$ and $STATE_v[s] = false$ then
2      $\mathtt{STATE}_v[s] := true$;
3      Allocate $\mathtt{tempD}_v[\cdot][s]$;
4      $\mathtt{tempD}_v[u][s] := \overline{D}$;
5      $\mathtt{UD}_v[s] := \mathtt{tempD}_v[u][s] + w(u, v)$;
6      foreach $v_i \in N(v) \setminus \{VIA_v[s]\}$ do
7          receive $\mathtt{UD}_{v_i}[s]$ and store it in $\mathtt{tempD}_v[v_i][s]$ by sending $get.dist(v, s, \mathtt{UD}_v[s])$
           to $v_i$;
8      $\mathtt{Dmin} := \min_{u \in N(v)}\{\mathtt{tempD}_v[u][s] + w(u, v)\}$;
9      $\mathtt{VIAmin} := \arg\min_{u \in N(v)}\{\mathtt{tempD}_v[u][s] + w(u, v)\}$;
10     if $tempD_v[VIAmin][s] \geq D_v[s]$ then
11         foreach $v_i \in N(v) \setminus \{VIA_v[s]\}$ do
12             receive loop-free distance $\mathtt{UD}_{v_i}[s]$ and store it in $\mathtt{tempD}_v[v_i][s]$ by
               sending $get.feasible.dist(v, s, \mathtt{UD}_v[s])$ to $v_i$;
13         $\mathtt{Dmin} := \min_{u \in N(v)}\{\mathtt{tempD}_v[u][s] + w(u, v)\}$;
14         $\mathtt{VIAmin} := \arg\min_{u \in N(v)}\{\mathtt{tempD}_v[u][s] + w(u, v)\}$;
15     send Dmin to $\mathtt{VIA}_v[s]$;
16     Deallocate $\mathtt{tempD}_v[\cdot][s]$;
17     $\mathtt{D}_v[s] := \mathtt{Dmin}$;
18     $\mathtt{UD}_v[s] := \mathtt{D}_v[s]$;
19     $\mathtt{VIA}_v[s] := \mathtt{VIAmin}$;
20     foreach $k \in N(v)$ do send $update(v, s, \mathtt{D}_v[s])$ to $k$;
21     $\mathtt{STATE}_v[s] := false$;
22 else
23     if $STATE_v[s] = true$ then
24         $\mathtt{tempD}_v[u][s] := \overline{D}$;
25     send $\mathtt{UD}_v[s]$ to $u$ ;
```

Fig. 4. Pseudo-code of procedure SendFeasibleDist

At this point v has surely found a feasible via to s and it deallocates $\mathtt{tempD}_v$, updates $\mathtt{D}_v[s]$, $\mathtt{UD}_v[s]$ and $\mathtt{VIA}_v[s]$ and propagates the change by sending *update* messages to its neighbors (Lines 15–19). Finally, v turns back in passive state.

A node $k \in N(v)$ which receives a *get.feasible.dist* message performs the procedure SendFeasibleDist. If $\mathtt{VIA}_k[s] = v$ and k is passive (Line 1), then procedure SendFeasibleDist behaves similarly to procedure Increase, notice indeed that Lines 2–21 of SendFeasibleDist are basically identical to Lines 2–20 of Increase. The only difference is Line 15 of SendFeasibleDist which is not present in Increase, and represents the answer to the *get.feasible.dist* message. However, within SendFeasibleDist the Local-Computation and the Global-Computation are performed with the aim of sending a reply with an estimated loop-free distance in addition to that of updating the routing table. In particular, node k needs to provide to v a new loop-free distance. To this aim, node k becomes active, allocates the temporary data structure $\mathtt{tempD}_k$, and sets $\mathtt{UD}_k[s]$ to the current distance through $\mathtt{VIA}_v[s]$ (Lines 2–5). Then, as in procedure

Increase, k first performs the Local-Computation (Lines 6–9), which involves all the neighbors of k. If the Local-Computation does not succeed, that is the snc is violated, then node k initiates the Global-Computation (Lines 11–14), which involves in the worst case all nodes of the network. At this point k has surely found an estimated distance to s which is guaranteed to be loop-free and hence, differently from Increase, it sends this value to its current via v (Line 15) as answer to the *get.feasible.dist* message. Now, as in procedure Increase, node k can deallocate $\texttt{tempD}_v$, update its local data structures $\texttt{D}_v[s]$, $\texttt{UD}_v[s]$ and $\texttt{VIA}_v[s]$, and propagate the change by sending *update* messages to all its neighbors (Lines 15–19). Finally, v turns back in passive state.

Differently from the *get.dist* case, k immediately replies with $\texttt{UD}_k[s]$ only if $\texttt{VIA}_k[s] \neq v$ or $\texttt{STATE}_v[s] = \mathit{false}$ (Line 25). If $\texttt{VIA}_k[s] \neq v$, that is k does not use v to reach s, and k is active with respect to s, then it updates $\texttt{tempD}_k[v][s]$ with the received most recent value (Line 24) before sending to v the distance to s through its current via. This is done to send to v the most up to date value.

The next theorems, whose proofs will be given in the full paper, show the loop-freedom, correctness, and complexity of LFR. As highlighted in [9], the complexity of a distributed algorithm in the asynchronous model depends on the time needed by processors to execute the local procedures of the algorithm and on the delays incurred in the communication among nodes. Moreover, these parameters influence the scheduling of the distributed computation and hence the number of messages sent. For these reasons, we consider the *realistic* case where the weight of an edge models the time needed to traverse such edge and all the processors require one unit of time to process a procedure. In this way, the distance between two nodes models the minimum time that such nodes need to communicate. We then measure the time complexity by the number of times that a processor performs a procedure. Note that, in the *realistic* case, DUAL has the same worst case complexity of LFR.

Theorem 1 (Loop-Freedom). *Let s be a node in G and let $\mathcal{C} = (c_1, c_2, ..., c_k)$ be a sequence of edge weight changes on G, if $\texttt{VIA}_G[s](t_0)$ is loop-free, then $\texttt{VIA}_G[s](t)$ is loop-free for each $t \geq t_0$.*

Theorem 2 (Correctness). *There exists a time $t_F \geq t_k$ such that, for each pair of nodes $v, s \in V$, and for each time $t \geq t_F$, $\texttt{D}_v[s](t) = d^{t_k}(v, s)$ and $\texttt{VIA}_t[v, s] \in via^{t_k}(v, s)$.*

Theorem 3 (Complexity). *Given a sequence of weight change operations $\mathcal{C} = (c_1, c_2, ..., c_k)$,* LFR *requires $O(n + \phi \cdot \Delta)$ space per node and sends $O(\Phi \cdot \Delta)$ messages and needs $O(\Phi)$ steps to converge.*

4 Experimental Analysis

In this section, we report the results of our experimental study on LFR and DUAL. Our experiments have been performed on a workstation equipped with a Quad-core 3.60 GHz Intel Xeon X5687 processor, with 12MB of internal cache and 24 GB of main memory, and consist of simulations within the OMNeT++

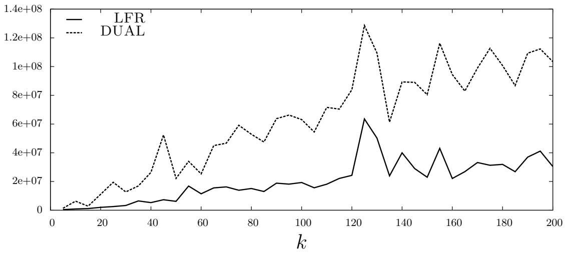


Fig. 5. Number of messages sent by LFR and DUAL on $G_{IP-8000}$

4.0p1 environment [14]. The program has been compiled with GNU g++ compiler 4.4.3 under Linux (Kernel 2.6.32).

Executed Tests. For the experiments we used both the real-world instances of the *CAIDA IPv4 topology dataset* [11] and *Erdős-Rényi* random graphs [2]. CAIDA is an association which provides data and tools for the analysis of the Internet infrastructure. We parsed the files in the CAIDA dataset to obtain a weighted undirected graph G_{IP} where nodes represent routers, edges represent links among routers and weights are given by Round Trip Times (RTT). As the graph G_{IP} consists of almost 35000 nodes, we cannot use it for the experiments, due to the memory requirements of DUAL. Hence, we performed our tests on connected subgraphs of G_{IP}, with a variable number of nodes and edges, induced by the settled nodes of a breadth first search starting from a node taken at random. We generated a set of different tests, each consisting of a dynamic graph characterized by a subgraph of G_{IP} with $n \in \{1200, 5000, 8000\}$ nodes (we denote an n nodes subgraph of G_{IP} with G_{IP-n}) and a set of k concurrent edge updates, where k assumes values in $\{5, 10, \ldots, 200\}$. An edge update consists of multiplying the weight of a random selected edge by a value randomly chosen in $[0.5, 1.5]$. For each test configuration (a dynamic graph with a fixed value of k) we performed 5 different experiments and we report average values. We performed the experiments in the *realistic* case, that is we considered the RTT as the time delay for receiving packets.

Graphs G_{IP} turns out to be very sparse (i.e., $m/n \approx 1.3$), so it is worth analyzing LFR and DUAL also on graphs denser than G_{IP}. To this aim we considered Erdős-Rényi random graphs [2]. In detail, we randomly generated a set of different tests, where a test consists of a dynamic graph characterized by: an Erdős-Rényi random graph G_{ER} of 2000 nodes; the density *dens* of the graph, computed as the ratio between m and the number of the edges of the n-complete graph; and the number k of edge update operations. We chose different values of *dens* in $[0.01, 0.61]$ and $k = 200$. Edge weights are randomly chosen in $[1, 10000]$. Edge updates are randomly chosen as in the CAIDA tests. For each test configuration, we performed 5 different experiments and we report average values. As in G_{IP}, time delays are equal to the edge weights.

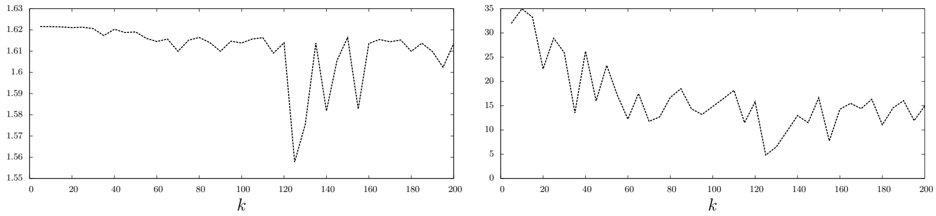


Fig. 6. Ratio between the average space occupancy of DUAL and LFR (left); ratio between the maximum space occupancy of DUAL and LFR (right) on $G_{IP-8000}$

Analysis. In Fig. 5 we report the number of messages sent by LFR and DUAL on $G_{IP-8000}$. The diagram shows that LFR always outperforms DUAL. The ratio between the number of messages sent by DUAL and LFR is between 2.02 and 7.61. The results of our experiments on $G_{IP-1200}$ and $G_{IP-5000}$ give similar results. These good performances of LFR are due in part to the topological structure of G_{IP} in which the average node degree is almost one. In fact, LFR uses *get.dist* messages in order to know the estimated distances of its neighbors. It follows that the number of *get.dist* messages sent by a node is proportional to the node degree, and hence the contribution to the message complexity of LFR of these messages is basically irrelevant. Note that, DUAL does not need to use *get.dist* messages as it stores, for each node, the estimated distances of its neighbors. We can hence conclude that the better performance of LFR on G_{IP} with respect to DUAL are basically due to the different way in which the two algorithms manage the distributed computation of shortest paths.

To conclude our analysis on G_{IP}, we considered the space occupancy per node of the implemented algorithms (recall that DUAL and LFR require $O(n \cdot \Delta)$ and $O(n + \phi \cdot \Delta)$ space per node, resp.). The experimental results on the space occupancy are reported in Fig. 6. Since in these sparse graphs the average degree is almost one, the average space occupancy of the two algorithms are almost equivalent although LFR is always slightly better than DUAL (Fig. 6 (left)). If we consider the maximum space occupancy, that is the space occupied by the node with the highest space requirements, then LFR is by far better than DUAL (Fig. 6 (right)). In fact DUAL requires a maximum space occupancy per node which is between 4.81 and 34.97 times the maximum space occupancy required by LFR. This is due to the fact that the topology table, implemented by the `tempD` data structure, is allocated by LFR only when needed.

As already observed, graph G_{IP} and its subgraphs have a high number of nodes with small degree. For instance, $G_{IP-8000}$ has 3072 degree-one nodes which corresponds to 38.4% of the nodes of the graph. Indeed, these graphs follow a power-law degree distribution. Hence, we embedded LFR and DUAL in DLP [7], a recently proposed framework which is able to reduce the number of messages sent by any distance vector algorithm in this kind of graphs by avoiding any computation involving degree-one nodes. The algorithms obtained by combining LFR and DUAL with DLP are denoted as LFR-DLP and DUAL-DLP,

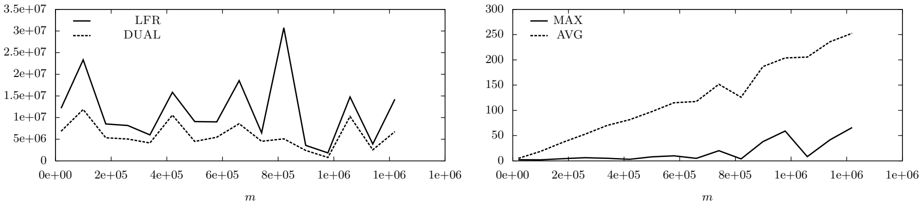


Fig. 7. Experiments on G_{ER} with $k = 200$ and $dens = 0.01, 0.05, ..., 0.61$: number of messages sent (left); ratio between the space occupancies, maximum (MAX) and average (AVG), of DUAL and LFR (right)

resp. Our experiments on these algorithms show that the use of DLP reduced the number of messages sent by a factor in $[1.96, 2.78]$ in LFR and $[2, 17, 8.33]$ in DUAL. This allows us to state that the good performance of LFR obtained in real-world instances are confirmed if it is used in combination with DLP. The ratio between the number of messages sent by DUAL-DLP and LFR-DLP is very similar to the ratio between the number of messages sent by DUAL and LFR since it always lies between 1.47 and 6.25. Finally, our experiments show that the space overhead resulting from the application of DLP is irrelevant. In fact, the ratio between the average space occupancy of DUAL and DUAL-DLP is 1.04 while the ratio between the average space occupancy of LFR and LFR-DLP is in $[1.04, 1.06]$. Since the performances of LFR and DUAL on the CAIDA graphs are in part influenced by the topological structure of such graphs, it is worth investigating how these algorithms perform also on dense graphs. To this aim we considered Erdős-Rényi random graphs G_{ER} with 2000 nodes, 200 weight changes and $dens$ ranging from 0.01 to 0.61, which leads to a number of edges ranging from about 20000 to about 1200000. Fig. 7 (left) shows the number of messages sent by LFR and DUAL on these instances. Contrarily to the case of G_{IP}, DUAL is better than LFR on G_{ER}. In fact, in most of the cases DUAL sends half the number of messages sent by LFR. This is due to the high number of *get.dist* messages sent by LFR which in these dense graphs are of course relevant. However, from the space occupancy point of view, we notice that the space requirements of DUAL increase more than those of LFR with the node degree, as highlighted in Fig. 7 (right). In detail, Fig. 7 (right) shows the ratio between the average space occupancy per node of DUAL and that of LFR in G_{ER} and the ratio between the maximum space occupancy per node of DUAL and that of LFR in G_{ER}. The average space occupancy ratio grows almost linearly with m, as the space occupancy of LFR depends on the degree with a factor ϕ while that of DUAL is proportional to the node degree with a factor n. Similar observations hold for the maximum space occupancy.

In conclusion, our experiments show that LFR is always the best choice in terms of memory requirements, while in terms of messages LFR outperforms DUAL on real-world instances and DUAL is the best choice on artificial ones.

References

1. Awerbuch, B., Bar-Noy, A., Gopal, M.: Approximate distributed bellman-ford algorithms. IEEE Trans. on Communications 42(8), 2515–2517 (1994)
2. Bollobás, B.: Random Graphs. Cambridge University Press (2001)
3. Cicerone, S., D'Angelo, G., Di Stefano, G., Frigioni, D.: Partially dynamic efficient algorithms for distributed shortest paths. Theoretical Computer Science 411, 1013–1037 (2010)
4. Cicerone, S., D'Angelo, G., Di Stefano, G., Frigioni, D., Maurizio, V.: Engineering a new algorithm for distributed shortest paths on dynamic networks. Algorithmica To appear Prel. version in [5]
5. Cicerone, S., D'Angelo, G., Di Stefano, G., Frigioni, D., Maurizio, V.: A New Fully Dynamic Algorithm for Distributed Shortest Paths and Its Experimental Evaluation. In: Festa, P. (ed.) SEA 2010. LNCS, vol. 6049, pp. 59–70. Springer, Heidelberg (2010)
6. Cicerone, S., Di Stefano, G., Frigioni, D., Nanni, U.: A fully dynamic algorithm for distributed shortest paths. Theoretical Comp. Science 297(1-3), 83–102 (2003)
7. D'Angelo, G., D'Emidio, M., Frigioni, D., Maurizio, V.: A Speed-Up Technique for Distributed Shortest Paths Computation. In: Murgante, B., Gervasi, O., Iglesias, A., Taniar, D., Apduhan, B.O. (eds.) ICCSA 2011, Part II. LNCS, vol. 6783, pp. 578–593. Springer, Heidelberg (2011)
8. Elmeleegy, K., Cox, A.L., Ng, T.S.E.: On count-to-infinity induced forwarding loops in ethernet networks. In: Proceedings IEEE INFOCOM, pp. 1–13 (2006)
9. Garcia-Lunes-Aceves, J.J.: Loop-free routing using diffusing computations. IEEE/ACM Trans. on Networking 1(1), 130–141 (1993)
10. Humblet, P.A.: Another adaptive distributed shortest path algorithm. IEEE Trans. on Communications 39(6), 995–1002 (1991)
11. Hyun, Y., Huffaker, B., Andersen, D., Aben, E., Shannon, C., Luckie, M., Claffy, K.: The CAIDA IPv4 routed/24 topology dataset, http://www.caida.org/data/active/ipv4_routed_24_topology_dataset.xml
12. McQuillan, J.: Adaptive routing algorithms for distributed computer networks. Technical Report BBN Report 2831, Cambridge, MA (1974)
13. Moy, J.T.: OSPF: Anatomy of an Internet routing protocol. Addison-Wesley (1998)
14. OMNeT++. Discrete event simulation environment, http://www.omnetpp.org.
15. Orda, A., Rom, R.: Distributed shortest-path and minimum-delay protocols in networks with time-dependent edge-length. Distr. Computing 10, 49–62 (1996)
16. Ramarao, K.V.S., Venkatesan, S.: On finding and updating shortest paths distributively. Journal of Algorithms 13, 235–257 (1992)
17. Ray, S., Guérin, R., Kwong, K.-W., Sofia, R.: Always acyclic distributed path computation. IEEE/ACM Trans. on Networking 18(1), 307–319 (2010)
18. Rosen, E.C.: The updating protocol of arpanet's new routing algorithm. Computer Networks 4, 11–19 (1980)
19. Yao, N., Gao, E., Qin, Y., Zhang, H.: Rd: Reducing message overhead in DUAL. In: Proceedings 1st International Conference on Network Infrastructure and Digital Content (IC-NIDC 2009), pp. 270–274. IEEE Press (2009)
20. Zhao, C., Liu, Y., Liu, K.: A more efficient diffusing update algorithm for loop-free routing. In: 5th International Conference on Wireless Communications, Networking and Mobile Computing (WiCom 2009), pp. 1–4. IEEE Press (2009)

Fully Dynamic Maintenance of Arc-Flags in Road Networks

Gianlorenzo D'Angelo[1], Mattia D'Emidio[2],
Daniele Frigioni[2], and Camillo Vitale[2]

[1] MASCOTTE Project, I3S(CNRS/UNSA)/INRIA, France
gianlorenzo.d_angelo@inria.fr

[2] Dept. of Electrical and Information Engineering, University of L'Aquila, Italy
{mattia.demidio,daniele.frigioni}@univaq.it, camillo.vitale@gmail.com

Abstract. The problem of finding best routes in road networks can be solved by applying Dijkstra's shortest paths algorithm. Unfortunately, road networks deriving from real-world applications are huge yielding unsustainable times to compute shortest paths. For this reason, great research efforts have been done to accelerate Dijkstra's algorithm on road networks. These efforts have led to the development of a number of *speed-up techniques*, as for example *Arc-Flags*, whose aim is to compute additional data in a preprocessing phase in order to accelerate the shortest paths queries in an on-line phase. The main drawback of most of these techniques is that they do not work well in dynamic scenarios.

In this paper we propose a new algorithm to update the Arc-Flags of a graph subject to edge weight decrease operations. To check the practical performances of the new algorithm we experimentally analyze it, along with a previously known algorithm for edge weight increase operations, on real-world road networks subject to fully dynamic sequences of operations. Our experiments show a significant speed-up in the updating phase of the Arc-Flags, at the cost of a small space and time overhead in the preprocessing phase.

1 Introduction

The problem of finding best connections in transportation networks has received a lot of attention in the last years. If travel times are assigned to the edges of the graph representing the network, this problem can be easily solved by applying Dijkstra's algorithm to find the shortest path between two points. Unfortunately, transportation networks deriving from real-world applications tend to be huge yielding unsustainable times to compute shortest paths. For this reason, great research efforts have been done over the last decade to accelerate Dijkstra's algorithm on typical instances of transportation networks, such as road or railway networks (see [5] for a recent overview). These research efforts have led to the development of a number of *speed-up techniques*, whose aim is to compute additional data in a preprocessing phase in order to accelerate the shortest paths queries during an on-line phase. However, most of the speed-up techniques developed in the literature do not work well in dynamic scenarios, when edge weights

R. Klasing (Ed.): SEA 2012, LNCS 7276, pp. 135–147, 2012.

changes occur to the network due to traffic jams or delays of trains. In other words, the correctness of these speed-up techniques relies on the fact that the network does not change between two queries. Unfortunately, such situations arise frequently in practice. In order to keep the shortest paths queries correct, the preprocessed data need to be updated. The easiest way is to recompute the preprocessed data from scratch after each change to the network. This is in general unfeasible since even the fastest methods need too much time. In general, the typical update operations that can occur on a network can be modelled as insertions and deletions of edges and edge weight changes (weight decrease and weight increase). When arbitrary sequences of the above operations are allowed, we refer to the *fully dynamic problem*, otherwise we refer to the *partially dynamic problem*; if only insertions and weight decrease (deletion and weight increase, respectively) operations are allowed, then the partially dynamic problem is known as the *incremental* (*decremental*, respectively) problem.

Related Works. We refer here only to papers on the dynamic case and refer to [5] as a survey for the static case. A number of efforts have been done in the last years to accelerate the computation of shortest paths in dynamic scenarios [1–4, 6, 13, 15]. The first of these techniques was Geometric Containers [15], whose key idea is to allow suboptimal containers after a few updates. However, this approach yields a loss in query performance. The same holds for the dynamic variant of Arc-Flags [9] proposed in [2], where, after a number of updates, the query performances get worse yielding only a low speed-up over Dijkstra's algorithm. In [13] the authors combine ideas from highway hierarchies [12] and overlay graphs [14] yielding very good query times in dynamic road networks. The ALT algorithm, introduced in [7] works considerably well in dynamic scenarios where edge weights can increase their value. Also in this case, query performances get worse if too many edges weights change [6]. Summarizing, all above techniques work in a dynamic scenario as long as the number of updates is small. As soon as the number of updates is greater than a certain value, it is better to repeat the preprocessing from scratch. To our knowledge, the only other dynamic technique known in the literature with no loss in query performance is that in [13]. In [4], a very practically efficient algorithm has been given to compute shortest paths in continental road graphs with arbitrary metrics, whose efficiency is also due to the use of parallelism. This algorithm is fast enough to be used in dynamic scenarios for the recomputation from scratch of shortest paths. Recently, a data structure named *Road-Signs* has been introduced in [3] to compute and update the Arc-Flags of a graphs. In detail, in [3] the authors define an algorithm to preprocess Road-Signs and a decremental algorithm to update them each time that a *weight increase* operation occurs on an edge of the graph. As the updating algorithm is able to correctly update Arc-Flags, there is no loss in query performance. They also experimentally analyze this algorithm in real-world road networks showing that it yields a significant speed-up in the updating phase of Arc-Flags with respect to the recomputation from-scratch, at the price of a small space and time overhead in the preprocessing phase. However, the solution in [3] is not able to update Arc-Flags in a fully dynamic scenario.

Contribution of the Paper. We propose a new *incremental* algorithm which is able to update the Arc-Flags of a graph by updating Road-Signs, during a sequence of *weight decrease* operations. Since the new incremental algorithm uses the same data structures of the decremental solution of [3], then the combination of these two solutions can be used to update Arc-Flags in a fully dynamic scenario. To check the practical usefulness of this combination we implemented the two algorithms and performed an extensive experimental study against the recomputation from scratch of Arc-Flags on fully dynamic sequences of weight increase and weight decrease operations on real world road networks. The results of our experiments can be summarized as follows: in comparison to the recomputation from-scratch of Arc-Flags, we obtained a significant speed-up in the updating phase, at the cost of a small space and time overhead in the preprocessing phase. In detail, we experimentally show that the fully dynamic algorithm is able to update the Arc-Flags between 40 and 431 times faster than the recomputation from scratch in average. However, in order to compute and store the Road-Signs, we need an overhead in the preprocessing phase and in the space occupancy. We experimentally show that such an overhead is small compared to the speed-up gained in the updating phase. In fact, the preprocessing requires, in average, 2.10 and 2.36 times the time and the space required by Arc-Flags, respectively.

2 Preliminaries

Road Graphs. A *road graph* is a weighted directed graph $G = (V, E, w)$, used to model real road networks, where nodes represent points on the network, edges represent road segments between two points and the weight function $w : E \to \mathbb{R}^+$ represents an estimate of the travel time needed to traverse road segments. Given G, we denote as $\bar{G} = (V, \bar{E})$ the *reverse graph* of G where $\bar{E} = \{(v, u) \mid (u, v) \in E\}$. A *minimal travel time route* between two crossings S and T in a road network corresponds to a *shortest path* from the node s representing S and the node t representing T in the corresponding road graph. The total weight of a shortest path between nodes s and t is called *distance* and it is denoted as $d(s, t)$. A partition of V is a family $\mathcal{R} = \{R_1, R_2, \ldots, R_r\}$ of subsets of V called *regions*, such that each node $v \in V$ is contained in exactly one region. Given $v \in R_k$, v is a *boundary node* of R_k if there exists an edge $(u, v) \in E$ such that $u \notin R_k$. Minimal routes in road networks can be computed by shortest paths algorithms such as Dijkstra's one. In order to perform an s-t query, the algorithm grows a shortest path tree starting from s and stopping as soon as it visits t. A simple variation of Dijkstra's algorithm is *bidirectional Dijkstra* which grows two shortest path trees starting from both s and t. In detail, the algorithm performs a visit of G starting from s and a visit of $\bar{G}$ starting from t. The algorithm stops as soon the two visits meet at some node in the graph.

Arc-Flags. The preprocessing phase of Arc-Flags first computes a partition $\mathcal{R} = \{R_1, R_2, \ldots, R_r\}$ of V and then associates a *label* to each edge (u, v) in E. A label contains, for each region $R_k \in \mathcal{R}$, a *flag* $A_k(u, v)$ which is true if and only if a shortest path in G towards a node in R_k starts with (u, v). The set of flags of an edge (u, v) is called *Arc-Flags* label of (u, v). The preprocessing phase associates also Arc-Flags to edges in the reverse graph $\bar{G}$. The query phase of Arc-Flags consists of a modified version of the bidirectional Dijkstra's algorithm: the forward search only considers those edges for which the flag of the target node's region is true, while the backward search follows only those edges having a true flag for the source node's region. The main advantage of Arc-Flags is its easy query algorithm combined with an excellent query performance. However, preprocessing is very time-consuming since it grows a full shortest path tree from each boundary node of each region. This is unpractical in dynamic scenarios where, in order to keep correctness of queries, the preprocessing phase has to be performed from scratch any time that an edge weight changes.

Road-Signs. Given a road graph $G = (V, E, w)$, a partition $\mathcal{R} = \{R_1, R_2, \ldots, R_r\}$ of V in regions, an edge $(u, v) \in E$ and a region $R_k \in \mathcal{R}$, the Road-Sign $RS_k(u, v)$ of (u, v) to R_k is the subset of boundary nodes b of R_k, such that there exists a shortest path from u to b that contains (u, v). The Road-Signs of (u, v) are represented as a boolean vector, whose size is the overall number of boundary nodes, where the i-th element is true if and only if the i-th boundary node is contained in $RS_k(u, v)$, for some region R_k. Hence, such a data structure requires $O(|E| \cdot |B|)$ memory, where B is the set of boundary nodes of G induced by $\mathcal{R}$. Notice that, in [3] it has been also proposed a technique to compact road signs that requires $O((|E| - |V|) \cdot |B|)$ overall space. The Road-Signs of G can be computed in the preprocessing phase of Arc-Flags. Given an edge (u, v) and a region R_k, $A_k(u, v)$ is set to true if and only if (u, v) is an edge in at least one of the shortest path trees grown for the boundary nodes of R_k. Such a procedure can be generalized to compute also Road-Signs. In fact, it is enough to add the boundary node b to $RS_k(u, v)$ if (u, v) is in the tree grown for b.

3 Incremental Update of Arc-Flags

Given a road graph $G = (V, E, w)$ and a partition $\mathcal{R} = \{R_1, R_2, \ldots, R_r\}$ of V in regions, we consider the problem of updating the Arc-Flags of G in a dynamic scenario where a sequence of only *weight decrease* operations $C = (c_1, c_2, \ldots, c_h)$ occur on G. We denote as $G_i = (V, E, w_i)$ the graph obtained after i weight decrease operations, $0 \leq i \leq h$, $G_0 \equiv G$. Each operation c_i decreases the weight of one edge $e_i = (x_i, y_i)$ of an amount $\gamma_i > 0$, such that $w_i(e_i) = w_{i-1}(e_i) - \gamma_i > 0$ and $w_i(e) = w_{i-1}(e)$, for each edge $e \neq e_i$ in E. Our algorithm is based on the following proposition given in [3], which provides a straightforward method to compute the Arc-Flags of a graph given the Road-Signs of such graph.

Proposition 1. [3] *Given $G = (V, E, w)$, a partition $\mathcal{R} = \{R_1, R_2, \ldots, R_r\}$ of V, an edge $(u, v) \in E$ and a region $R_k \in \mathcal{R}$, the following conditions hold:*
(i) if $u, v \in R_k$, then $A_k(u, v) = true$;
(ii) if $RS_k(u, v) \neq \emptyset$, then $A_k(u, v) = true$;
(iii) if u or v is not in R_k and $RS_k(u, v) = \emptyset$, then $A_k(u, v) = false$.

Hence in what follows, we describe how our solution updates Road-Signs. The algorithm is denoted as IncRS and its pseudo-code is given in Figure 1. The algorithm is based on the observation that, when c_i occurs, all the shortest paths which contain (x_i, y_i) in G_{i-1} (i.e. before c_i) are shortest paths also in G_i (i.e. after c_i). Therefore, if some boundary node for some region R_k belongs to $RS_k(x_i, y_i)$ before c_i it belongs also to $RS_k(x_i, y_i)$ after c_i and no update is needed for it. However, it could happen that the shortest path from x_i to some boundary node b of R_k in G_{i-1} does not contain (x_i, y_i) but, after the weight decrease, the new shortest path in G_i contains (x_i, y_i). In this case, b needs to be added to $RS_k(x_i, y_i)$ and removed from $RS_k(x_i, w)$, where (x_i, w) is the edge outgoing x_i whose road signs b belongs to. Same arguments can be applied to the incoming edges of x_i, (z, x_i): if a boundary node belongs to $RS_k(z, x_i)$ and $RS_k(x_i, y_i)$ before c_i, then it belongs also to $RS_k(z, x_i)$ after c_i; if a boundary node b is in $RS_k(x_i, y_i)$ (because it was already in $RS_k(x_i, y_i)$ or because it has been added to it as a consequence of c_i) and it does not belong to $RS_k(z, x_i)$ before c_i, then it might be added to $RS_k(z, x_i)$ in the case that the shortest path from z to b in G_i contains the sub-path (z, x_i, y_i). We iteratively apply the same arguments to the other edges of the graph, starting from x_i and traversing its incoming edges. Note that, if at some point of the iteration we find out that the shortest path from a node z to some boundary node b does not decrease, then we do not need to add or remove b to any incoming edge of z. This allows us to reduce the search space of the algorithm.

IncRS works in two phases. In the first phase (lines 1–8) $RS_k(x_i, y_i)$ is updated by possibly adding new boundaries $b \notin RS_k(x_i, y_i)$ to it. This phase is performed for each $b \notin RS_k(x_i, y_i)$ separately (line 1). In the pseudo-code, when needed, we store distances between a node u and a boundary b in data structure $D[u, b]$ and we use an heap H to compute the minimum among the computed distances. Since each boundary node is processed separately, these data structures are overwritten at each computation, hence requiring $O(n)$ space in the worst case. In detail, at lines 2 and 3–4 we compute the distances from y_i to b and from node $\bar{r}$ such that $b \in RS_k(x_i, \bar{r})$ to b, respectively. Then, at line 5, we check whether the weight of the path passing through y_i is smaller than that passing through $\bar{r}$ (that is the shortest path from x_i to b in G_{i-1}). In the affirmative case, we update the road signs by adding b to $RS_k(x_i, y_i)$ and removing b from $RS_k(x_i, \bar{r})$ (lines 6–7). Moreover, we store the new distance from x_i to b in $D[x_i, b]$ (line 8) in order to use it in the next phase. In the second phase (lines 9–22) the road signs are updated for each $b \in RS_k(x_i, y_i)$ separately. The updating is done by mimicking the Dijkstra's algorithm, that is by greedily visiting the reverse graph starting from x_i and stopping when a node does not need to update the road signs of its outgoing edges wrt b. At line 11, H is initialized by inserting x_i into it using $D[x_i, b]$ as key. If b was already in $RS_k(x_i, y_i)$, then $D[x_i, b]$ has not been computed at line 8 and hence it is computed at line 10. Until H is not empty (line 12), a node z and its distance $D[z, b]$ is extracted

```
**Procedure**: IncRS($G_{i-1}, c_i, R_k$)
**Input**  : Graph $G_{i-1}$, weight decrease $c_i$ on edge $(x_i, y_i)$ and region $R_k \in \mathcal{R}$
**Output**: Updated $RS_k(u, v)\ \forall\ (u, v) \in E$

 1 foreach $b \notin RS_k(x_i, y_i)$ do
 2     Compute $D[y_i, b]$;
 3     Let $\bar{r} : (x_i, \bar{r}) \in E$ and $b \in RS_k(x_i, \bar{r})$;
 4     Compute $D[\bar{r}, b]$;
 5     if $w(x_i, y_i) + D[y_i, b] < w(x_i, \bar{r}) + D[\bar{r}, b]$ then
 6         $RS_k(x_i, \bar{r}) := RS_k(x_i, \bar{r}) \setminus \{b\}$;
 7         $RS_k(x_i, y_i) := RS_k(x_i, y_i) \cup \{b\}$;
 8         $D[x_i, b] := w(x_i, y_i) + D[y_i, b]$;
 9 foreach $b \in RS_k(x_i, y_i)$ do
10     Compute $D[x_i, b]$ if it has not been already computed at line 8;
11     H.$Insert(x_i, D[x_i, b]))$;
12     while H $\neq \emptyset$ do
13         $(z, D[z, b]) :=$ H.$Delete\_Min()$;
14         foreach $z_i : (z_i, z) \in E$ do
15             Let $\bar{r} : (z_i, \bar{r}) \in E$ and $b \in RS_k(z_i, \bar{r})$;
16             Compute $D[\bar{r}, b]$;
17             if $w(z_i, z) + D[z, b] < w(z_i, \bar{r}) + D[\bar{r}, b]$ or $\bar{r} = z$ then
18                 $RS_k(z_i, \bar{r}) := RS_k(z_i, \bar{r}) \setminus \{b\}$;
19                 $RS_k(z_i, z) := RS_k(z_i, z) \cup \{b\}$;
20                 $D[z_i, b] := w(z_i, z) + D[z, b]$;
21                 if $z_i \in$ H then H.$Decrease\_Key(z_i, D[z_i, b]))$;
22                 else H.$Insert(z_i, D[z_i, b]))$;
```

Fig. 1. Pseudo-code of algorithm IncRS

from H. Then, for each node z_i such that $(z_i, z) \in E$, at lines 14–22 we perform the same steps done for x_i and (x_i, y_i): we compute the distance from $\bar{r}$ such that $b \in RS_k(z_i, \bar{r})$ to b; we check whether the weight of the path from z_i to b passing through z is smaller than that passing through $\bar{r}$ and, in the affirmative case, we update the road signs by adding b to $RS_k(z_i, z)$ and removing it from $RS_k(z_i, \bar{r})$; finally, we store the new distance from z_i to b in $D[z_i, b]$ and insert z_i into H or decrease its key if it already belongs to H.

The following theorem states the correctness of IncRS.

Theorem 1. *Given $G = (V, E, w)$ and a partition $\mathcal{R} = \{R_1, R_2, \ldots, R_r\}$ of V, for each $(u, v) \in E$ and $R_k \in \mathcal{R}$,* IncRS *correctly updates $RS_k(u, v)$ and $A_k(u, v)$ after a weight decrease operation on an edge of G.*

Proof. Let us consider a region $R_k \in \mathcal{R}$ and a weight decrease operation c_i on edge (x_i, y_i). From Proposition 1, it is enough to show that IncRS correctly updates $RS_k(u, v)$ after c_i, for each $(u, v) \in E$. Given an edge (u, v), we denote as $RS_k(u, v)$ and $RS'_k(u, v)$ the road-signs of (u, v) before and after c_i, respectively.

As c_i decreases the weight of (x_i, y_i), then $RS_k(x_i, y_i) \subseteq RS'_k(x_i, y_i)$, moreover for each $(u, v) \in E$, $RS_k(u, v)$ can be modified by adding or removing boundary nodes in $RS'_k(x_i, y_i)$, that is $RS'_k(u, v) \setminus RS_k(u, v) \subseteq RS'_k(x_i, y_i)$ and $RS_k(u, v) \setminus RS'_k(u, v) \subseteq RS'_k(x_i, y_i)$. It follows that phase one (lines 1–8) correctly updates the road-signs of edges outgoing from x_i. The road-signs of the remaining edges are updated in phase two, whose correctness is shown separately for each boundary node $b \in RS'_k(x_i, y_i)$ and derives from the following facts.

F1 *The nodes are extracted from* H *at line 13 in non-decreasing order of keys.* Let us consider two nodes u and v extracted from H at times t_u and t_v with keys $D[u, b]$ and $D[v, b]$, respectively. By contradiction, suppose that $t_u < t_v$ and $D[u, b] > D[v, b]$. Since at line 13 the node with the minimum key is extracted, at time t_u, $D[u, b]$ was minimum and hence either u has been extracted into H after t_u or its key has been decreased after t_u. In both cases, the algorithm passed the test at line 17 which implies that there exists a node v_1 and a time $t_{v_1} < t_v$ such that $D[v_1, b] < D[v, b] < D[u, b]$, where $D[v_1, b]$ is the key of v_1 at time t_{v_1} when v_1 is extracted from H. If we iteratively repeat this arguments for node v_1, we obtain a sequence of nodes $v \equiv v_0, v_1, v_2, \ldots, v_k$, where $v_k \equiv x_i$, such that, if we denote as $D[v_j, b]$ the key of v_j when it is extracted from H at time t_{v_j}, then $D[v_{j+1}, b] < D[v_j, b]$ and $t_{v_{j+1}} < t_{v_j}$ for each $j = 0, 1, \ldots k - 1$. For the condition at line 12, one of the nodes in the sequence v_j has to belong to H with key $D[v_j, b] < D[u, b]$ at time t_u which contradicts the fact that, at time t_u, node u is the node with the minimum key.

F2 *A node is extracted from* H *at line 13 at most once.* Suppose that a node u is extracted from H at two different times $t_1 < t_2$. Then, node u has been inserted into H at two different times, denoted as $\bar{t}_1$ and $\bar{t}_2$, when it does not belong to H. It follows that $[\bar{t}_1, t_1] \cap [\bar{t}_2, t_2] = \emptyset$. Further, let us denote as $D_{t_1}[u, b]$ ($D_{t_2}[u, b]$, resp.) the key of u at time $\bar{t}_1$ ($\bar{t}_2$, resp.) which is equal to that at time t_1 (t_2, resp.). Let us consider the two (possibly different) nodes v_1 and v_2 which are extracted from H immediately before times $\bar{t}_1$ and $\bar{t}_2$, respectively. Let us analyze the keys extracted from H. At time $\bar{t}_1$, v_1 is extracted with key $D[v_1, b]$, $D[u, b]$ is set to $D_{t_1}[u, b] = w(u, v_1) + D[v_1, b]$ and b is added to $RS_k(u, v_1)$. At time t_1, u is extracted with key $D_{t_1}[u, b]$. At time $\bar{t}_2$, v_2 is extracted with key $D[v_2, b]$, and $D[u, b]$ is set to $D_{t_2}[u, b] = w(u, v_2) + D[v_2, b]$, it follows that the test at line 17 returned true and, as $b \in RS_k(u, v_1)$, $w(u, v_2) + D[v_2, b] < w(u, v_1) + D[v_1, b]$. Hence $D_{t_2}[u, b] < D_{t_1}[u, b]$. At time t_2, u is extracted with key $D_{t_2}[u, b]$ and, since $t_1 < t_2$, this contradicts Fact F1.

F3 *For each edge (u, v) such that $RS'_k(u, v) \neq RS_k(u, v)$, u is inserted into* H*.* We show a stronger statement that is: if a node changes its distance to $b \in RS'_k(u, v)$ it is inserted into H. By contradiction, let us consider the node u such that: it changes its distance to b, it is not inserted into H, and its distance to b after c_i is minimal among the nodes with the same properties. By this last property, the node v on the shortest path from u to b after c_i is inserted into H. When v is extracted from H, either b belongs to $RS_k(u, v)$ or $w(u, v) + D[v, b] < w(u, \bar{r}) + D[\bar{r}, b]$, where $\bar{r}$ is the node such that $b \in RS_k(u, \bar{r})$. In both cases, u is inserted into H at line 22.

Table 1. Tested road graphs. 1st col.: the graph; 2nd and 3rd col.s: number of nodes and edges in the graph; 4th–7th col.s: percentage of edges into categories: motorways (MOT), national roads (NAT), regional roads (REG), and urban streets (URB)

graph	**n. of nodes**	**n. of edges**	%MOT	%NAT	%REG	%URB
LUX	30 647	75 576	0.55	1.95	14.77	82.71
DNK	469 110	1 090 148	24.02	3.06	0.48	72.45
BEL	458 403	1 164 046	22.90	2.92	0.52	73.62
AUT	722 512	1 697 902	27.60	5.33	1.71	65.21
ESP	695 399	1 868 838	33.22	6.34	1.51	58.87
NED	892 027	2 278 824	0.40	0.56	5.16	93.86
SWE	1 546 705	3 484 378	19.54	2.86	0.45	77.10

F4 *When a node u is extracted from* H *at line 13, for each* $(u, v) \in E$, $RS_k(u, v)$ *is correctly updated.*
By contradiction, let us consider the first node u whose outgoing edges have wrong road-signs when u is extracted from H. Let us consider the node v such that $b \in RS_k(u, v)$ when u is extracted, that is v is the node that was extracted from H immediately before the last time that either u is inserted into H or its key is decreased. As u is the first node whose outgoing edges have wrong road-signs when it is extracted from H, then the road-signs of the edges outgoing from v are correctly updated. Moreover, also the road-signs of edges outgoing from w, for each $(u, w) \in E$ are correctly updated. In fact two cases may arise: if w changes the road-signs of its outgoing edges then, by Fact F3, it is inserted into H, by facts F1 and F2, it is extracted before u and hence, by hypothesis, it correctly updates the road-signs of its outgoing edges; otherwise the road-signs of its outgoing edges are already correct. In any case, when u is inserted into H the distances used in the test of line 17 are correctly computed and hence the road signs are correctly updated. □

From a theoretical point of view, IncRS requires a computational complexity which is, in the worst case, comparable to that of the recomputation from scratch of Arc-Flags. However, IncRS focuses the computation only on the nodes that change shortest paths to a subset of the boundary nodes (and possibly on the neighbors of such nodes). In contrast, the recomputation from scratch computes all the shortest paths from any boundary node to each other node of the network. This difference is difficult to be captured by a worst case analysis and this motivates the experimental study of the next section.

4 Experimental Study

In this section, we compare the performances of the incremental algorithm proposed in this paper and the decremental algorithm of [3], whose combination is named here DynRS, on fully dynamic sequences of weight change operations against the recomputation from scratch of Arc-Flags. We used the implementation of Arc-Flags of [2]. Furthermore, it has been shown in [10] that the best

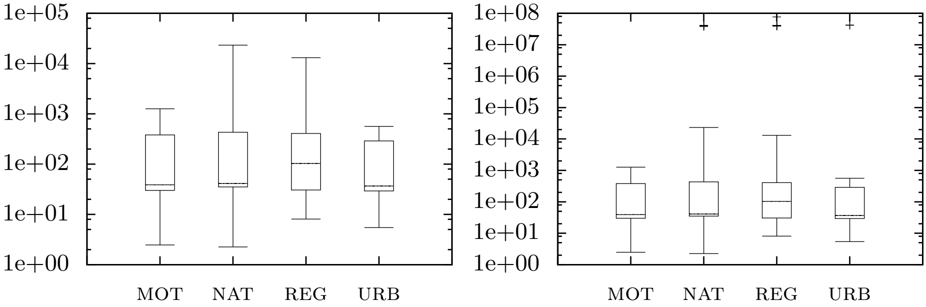


Fig. 2. Speed-up factors of the fully dynamic algorithm for the road network of Sweden, without (left) and with (right) outliers

query performances for Arc-Flags are achieved when partitions are computed by using arc-separator algorithms. For this reason, we used arc-separators obtained by the METIS library [8]. Our experiments are performed on a workstation equipped with a Quad-core 3.60 GHz Intel Xeon X5687 processor, with 12MB of internal cache and 24 GB of main memory. The program has been compiled with GNU g++ compiler 4.4.3 under Linux (kernel 2.6.32).

We consider seven road graphs available from PTV [11] representing the road networks of Luxembourg, Denmark, Belgium, Austria, Spain, Netherlands and Sweden, denoted as LUX, DNK, BEL, AUT, ESP, NED and SWE, respectively. In each graph, edges are classified into four categories according to their speed limits: motorways (MOT), national roads (NAT), regional roads (REG) and urban streets (URB). The main characteristics of these graphs are reported in Table 1. We consider fully dynamic sequences of updates simulating disruptions on road networks built as follows. The most significant operation that can occur on a road segment is the increase of a weight, which simulates a delay in the travel time on that segment due, for instance, to a traffic jam. This operation is usually followed by a weight decrease on the same road segment which simulates the restore from the delay. Hence, for each operation in the sequence that increases the weight of an edge (x_i, y_i) of a quantity γ_i, there is a corresponding subsequent operation which decreases the weight of edge (x_i, y_i) of the same amount γ_i. We execute, for each graph considered and for each road category, random sequences of 100 weight-change operations as described above. The weight-change amount for each operation is chosen uniformly at random in $[25\%, 75\%]$ of the weight of the edge involved in that operation. As a performance indicator, we choose the time used by the algorithms to complete a single update during the execution of a sequence. We measure, as speed-up factor, the ratio between the time required by the recomputation from scratch of Arc-Flags and that required by DynRS. The results are reported in Fig. 2–3, and in Table 2.

Fig.s 2 and 3 show two box-plot diagrams representing the values of the speed-up factors obtained for SWE and NED. For each category, we represent minimum

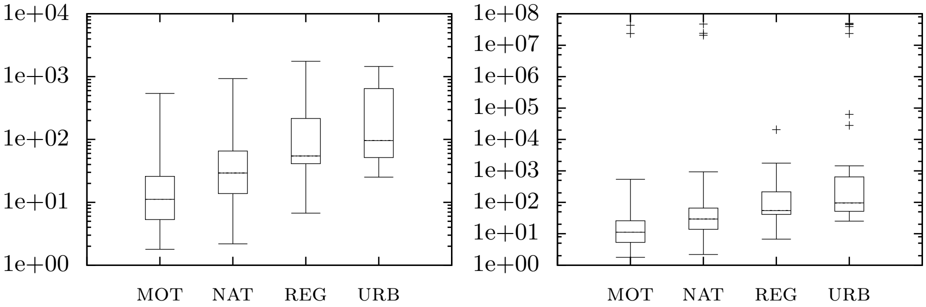


Fig. 3. Speed-up factors of the fully dynamic algorithm for the road network of the Netherlands, without (left) and with (right) outliers

value, first quartile, median value, third quartile, and maximum value. In both figures, the diagram on the left does not show outlier values while the diagram on the right does. Outlier values occur when DynRS performs much better than Arc-Flags because the number of Road-Signs changed is very small. We consider a test as outlier if the speed-up factor is 1000 times the speed-up factor median value. Even without outliers, the speed-up gained by DynRS is high.

Notice that, in the case of SWE (Fig. 2), the speed-up factors are quite similar on the different categories, thus highlighting an independency from categories. This is the typical behavior of road networks, as shown also for BEL, DNK, ESP and AUT in Table 2, where we report the *average* time of the recomputation from-scratch of Arc-Flags and the *average* time of DynRS, the *average ratios* between these quantities and the speed-up factors. The only exceptions are LUX and NED, where the percentage of motorways is very low. This is the reason why we highlight the behavior of DynRS on NED in Fig. 3, where the speed-up factor reaches the highest values when update operations occur on urban edges, while it is smaller when they occur on motorway edges. In fact, when an update operation occurs on urban edges, the number of shortest paths that change is small compared to the case that an update operation occurs on motorways edges. This implies that DynRS, which selects the nodes that change such shortest paths and focus the computation only on such nodes, performs better than the recomputation from-scratch of the shortest paths from any boundary node.

We note that, the speed-up factor increases with the size of the network. This can be explained by the fact that, when an edge update operation occurs, it affects only a part of the graph, hence only a subset of the edges in the graph need to update their Arc-Flags or Road-Signs. In most of the cases, this part is small compared to the size of the network and, with high probability, it corresponds to the subnetwork close to the edge increased or closely linked to it. In other words, it is unlike that a traffic jam in a certain part of the network affects the shortest paths of another part which is far or not linked to the first one. Clearly, this fact is more evident when the road network is big. In conclusion, it is evident from

Table 2. Avg update times and speed-up factors of DynRS. 1st col.: graph; 2nd col.: category where the weight changes occur; 3rd and 4th col.s: avg computational time for Arc-Flags and DynRS, resp.; 5th col.: ratio between the values of the 3rd and the 4th col.s; 6th col.: avg speed-up factors of DynRS against Arc-Flags

graph	**cat.**	**AF (s)**		DynRS (s)		**ratio**		**avg. speed-up**	
LUX	MOT	5.50	5.52	2.28	1.57	2.42	6.16	11.09	40.32
	NAT	5.51		2.64		2.09		32.81	
	REG	5.56		0.99		5.61		24.79	
	URB	5.52		0.38		14.50		92.60	
DNK	MOT	542.55	542.46	17.05	14.90	31.84	37.40	449.10	431.41
	NAT	542.83		11.24		48.31		430.16	
	REG	542.22		16.16		33.61		330.64	
	URB	542.25		15.13		35.84		515.72	
BEL	MOT	644.48	644.34	23.16	23.73	27.83	29.18	195.53	238.23
	NAT	644.30		28.93		22.26		195.81	
	REG	644.31		28.00		23.03		421.83	
	URB	644.28		14.85		43.58		139.73	
AUT	MOT	935.16	940.83	39.31	26.25	23.76	39.04	108.12	166.10
	NAT	934.16		19.08		48.99		185.48	
	REG	956.62		27.70		33.74		183.09	
	URB	937.38		18.89		49.65		187.71	
ESP	MOT	736.81	737.31	22.21	21.41	33.15	34.91	392.19	373.98
	NAT	737.30		21.44		34.39		562.27	
	REG	736.50		24.46		30.12		264.78	
	URB	736.32		17.54		41.98		276.66	
NED	MOT	1 607.36	1 606.29	206.36	90.55	7.78	41.31	31.45	169.79
	NAT	1 606.22		107.27		14.97		107.20	
	REG	1 609.60		30.85		52.03		181.84	
	URB	1 601.96		17.71		90.45		358.67	
SWE	MOT	2 681.54	2 681.16	113.90	76.12	23.65	38.14	180.36	316.64
	NAT	2 681.94		68.98		38.87		519.96	
	REG	2 678.81		52.79		50.75		394.99	
	URB	2 682.34		68.82		39.29		171.24	

Table 2, that DynRS outperforms the recomputation from-scratch by far and that it requires reasonable computational time.

Regarding the preprocessing phase, in Table 3 we report the computational time and the space occupancy required by Arc-Flags and DynRS. Table 3 shows that, for computing Road-Signs along with Arc-Flags, we need about twice the computational time required for computing only Arc-Flags, which is a small overhead compared to the speed-up gained in the updating phase. The same observation can be done regarding the space occupancy. In fact, Table 3 also shows that the space required for storing both Road-Signs and Arc-Flags is between 1.44 and 3.48 times that required to store only Arc-Flags.

Table 3. Preprocessing time and space. 1st col.: graph; 2nd col.: number of regions; 3rd col.: preprocessing time, in seconds, of Arc-Flags; 4th col.: preprocessing time, in seconds, of Arc-Flags and Road-Signs; 5th col.: ratio between the values reported in the 4th and the 3rd column; 6th col.: space required, in Bytes, to store Arc-Flags; 7th col.: space required, in Bytes, to store Arc-Flags and Road-Signs; 8th col.: ratio between the values of the 7th and the 6th column

graph	reg.	AF (s)	AF+RS (s)	t. ratio	AF (B)	AF+RS (B)	s. ratio
LUX	64	5.52	11.79	2.14	1 209 216	1 744 531	1.44
DNK	128	542.46	1 104.16	2.04	17 442 368	43 932 508	2.52
BEL	128	644.34	1 373.11	2.13	18 624 736	64 759 010	3.48
AUT	128	940.83	1 995.93	2.12	27 166 432	78 720 055	2.90
ESP	128	737.31	1 446.65	1.96	29 901 408	67 664 666	2.26
NED	128	1 606.29	3 214.34	2.00	36 461 184	64 612 836	1.77
SWE	128	2 681.16	5 986.62	2.23	55 750 048	163 151 588	2.93

References

1. Berger, A., Grimmer, M., Müller-Hannemann, M.: Fully Dynamic Speed-Up Techniques for Multi-criteria Shortest Path Searches in Time-Dependent Networks. In: Festa, P. (ed.) SEA 2010. LNCS, vol. 6049, pp. 35–46. Springer, Heidelberg (2010)
2. Berrettini, E., D'Angelo, G., Delling, D.: Arc-flags in dynamic graphs. In: Proc. of the 9th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS 2009), Schloss Dagstuhl, Germany (2009)
3. D'Angelo, G., Frigioni, D., Vitale, C.: Dynamic Arc-Flags in Road Networks. In: Pardalos, P.M., Rebennack, S. (eds.) SEA 2011. LNCS, vol. 6630, pp. 88–99. Springer, Heidelberg (2011)
4. Delling, D., Goldberg, A.V., Pajor, T., Werneck, R.F.: Customizable Route Planning. In: Pardalos, P.M., Rebennack, S. (eds.) SEA 2011. LNCS, vol. 6630, pp. 376–387. Springer, Heidelberg (2011)
5. Delling, D., Sanders, P., Schultes, D., Wagner, D.: Engineering Route Planning Algorithms. In: Lerner, J., Wagner, D., Zweig, K.A. (eds.) Algorithmics. LNCS, vol. 5515, pp. 117–139. Springer, Heidelberg (2009)
6. Delling, D., Wagner, D.: Landmark-Based Routing in Dynamic Graphs. In: Demetrescu, C. (ed.) WEA 2007. LNCS, vol. 4525, pp. 52–65. Springer, Heidelberg (2007)
7. Goldberg, A.V., Harrelson, C.: Computing the Shortest Path: A* Search Meets Graph Theory. In: 16th Annual ACM–SIAM Symp. on Discrete Algorithms (SODA 2005), pp. 156–165 (2005)
8. Karypis, G.: METIS - A Family of Multilevel Partitioning Algorithms (2007)
9. Lauther, U.: An extremely fast, exact algorithm for finding shortest paths. Static Networks with Geographical Background 22, 219–230 (2004)
10. Möhring, R.H., Schilling, H., Schütz, B., Wagner, D., Willhalm, T.: Partitioning Graphs to Speedup Dijkstra's Algorithm. ACM J. Exp. Algorithmics 11, 2.8 (2006)
11. PTV AG - Planung Transport Verkehr (2008), http://www.ptv.de
12. Sanders, P., Schultes, D.: Engineering Highway Hierarchies. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 804–816. Springer, Heidelberg (2006)

13. Schultes, D., Sanders, P.: Dynamic Highway-Node Routing. In: Demetrescu, C. (ed.) WEA 2007. LNCS, vol. 4525, pp. 66–79. Springer, Heidelberg (2007)
14. Schulz, F., Wagner, D., Zaroliagis, C.: Using Multi-Level Graphs for Timetable Information in Railway Systems. In: Mount, D.M., Stein, C. (eds.) ALENEX 2002. LNCS, vol. 2409, pp. 43–59. Springer, Heidelberg (2002)
15. Wagner, D., Willhalm, T., Zaroliagis, C.: Geometric Containers for Efficient Shortest-Path Computation. ACM J. Exp. Algorithmics 10, 1.3 (2005)

A More Reliable Greedy Heuristic for Maximum Matchings in Sparse Random Graphs*

Martin Dietzfelbinger[1], Hendrik Peilke[2,**], and Michael Rink[1]

[1] Fakultät für Informatik und Automatisierung, Technische Universität Ilmenau
{martin.dietzfelbinger,michael.rink}@tu-ilmenau.de
[2] IBYKUS AG für Informationstechnologie, Erfurt, Germany
Hendrik.Peilke@Ibykus.de

Abstract. We propose a new greedy algorithm for the maximum cardinality matching problem. We give experimental evidence that this algorithm is likely to find a maximum matching in random graphs with constant expected degree $c > 0$, independent of the value of c. This is contrary to the behavior of commonly used greedy matching heuristics which are known to have some range of c where they probably fail to compute a maximum matching.

1 Introduction

Maximum Cardinality Matchings. Consider an undirected graph $G = (V, E)$ with node set V, $|V| = n$, and edge set $E \subseteq \binom{V}{2}$, $|E| = m$. A matching M in G is a subset of E with the property that the edges in M are pairwise disjoint. The problem of finding a matching with the largest possible cardinality, a so called *maximum matching*, has been a subject of study for decades. The first polynomial time algorithm for this problem was given in 1965 by Edmonds [11]. A straightforward implementation of this algorithm has running time $O(n^2 \cdot m)$. Many other polynomial time algorithms followed, eventually reducing the running time to $O(n^{1/2} \cdot m)$, as, e.g., the algorithm of Micali and Vazirani [16,21]. For dense graphs, i.e., graphs with $m = \Theta(n^2)$ edges, this was the best known until 2004 when Mucha and Sankowski [17] gave an algorithm that has (expected) running time dominated by the time for multiplying two $n \times n$ matrices, which is $O(n^\omega)$, with $\omega < 2.376$ [8].

Heuristics. Usually matching algorithms, notably augmenting path algorithms, are allowed to be initialized with a non-empty matching which is then iteratively improved to a maximum matching. Hence a large enough initial matching determined with some fast heuristic approach can decrease the running time of an exact algorithm significantly. Apart from the use of heuristics in the preprocessing phase of exact algorithms there is an interest in graph classes where heuristics, especially fast greedy algorithms, are likely to obtain maximum matchings. On

* Research supported by DFG grant DI 412/10-2.

** M.Sc. student at the Technische Universität Ilmenau while this work was done.

R. Klasing (Ed.): SEA 2012, LNCS 7276, pp. 148–159, 2012.

such classes heuristics can replace the (overall) exact algorithms if the heuristics are faster or at least equally fast but easier to implement.

Sparse Random Graphs. A well studied graph class in this context is the class of random graphs with constant expected degree c. Let $G(n;c)$ be a random (general) graph with n nodes where each of the $\binom{n}{2}$ possible edges is present with probability $p = c/(n-1)$, and let $B(n/2, n/2; c)$ be a random bipartite graph with n nodes where each of the $n^2/4$ possible edges is present with probability $p = 2{\cdot}c/n$. Bast et al. [3] showed that if $c > c_0$ for $c_0 = 32.67$ in the case of general graphs, and $c_0 = 8.83$ in the case of bipartite graphs, then with high probability every non-maximum matching in $G(n;c)$ and $B(n/2, n/2; c)$ has an augmenting path of length $O(\log n)$. (Note that this also holds for $c \in (0,1)$, and indeed it is conjectured that the statement is true for all constant $c > 0$, in both cases.) Hence matching algorithms using shortest augmenting paths like the algorithm of Micali and Vazirani [16] for general graphs and the algorithm of Hopcroft and Karp [13] for bipartite graphs have (expected) running time $O(n{\cdot}\log n)$ on sparse random graphs. Chebolu et al. [7] gave an algorithm with (expected) running time $O(n)$. The first, simple phase of their algorithm uses the well-known Karp-Sipser heuristic. (The second part is much more complex.) Karp and Sipser [14] proved that with high probability their greedy algorithm produces a matching which is within $o(n)$ of the maximum for every constant $c > 0$. This result was improved by Aronson et al. [2] who showed that actually for $c < e$ the Karp-Sipser algorithm finds a maximum matching with high probability, and for $c > e$ the size of the matching found is within $n^{1/5+o(1)}$ of the maximum with high probability. For practical purposes Karp and Sipser suggested a different greedy algorithm, Algorithm 1 of [14], that turns out to give better results in their experiments but seems to be more complicated to analyze because it utilizes an operation called "contraction of nodes". This operation plays a role in some of the algorithms discussed below.

"Critical Region". In an experimental study Magun [15] compared the performance of several greedy matching algorithms in the style of the algorithms given in [14] on sparse random graphs. It turned out that there are good greedy algorithms that are likely to give maximum matchings for a wide range of c, but even the best algorithm in this study often fails to find a maximum matching in the range of about $2.6 \le c \le 3.8$ (where the lower bound is likely to converge to $e \approx 2.718$ for n large enough). Hence there is some region for c that seems critical for known greedy matching heuristics.

1.1 Our Results

We describe a new greedy matching algorithm and give experimental evidence that this algorithm is likely to compute a maximum matching in sparse random graphs for all ranges of c and large enough n; in particular, it seems to overcome the critical region that shows up in the results of [15]. The algorithm is motivated by the "selfless algorithm" of Sanders [18], for orienting undirected graphs such that the maximum in-degree is below a given constant.

Drawback. In comparison to the common greedy heuristics discussed above the running time of our algorithm is larger and more affected by the expected degree c. Hence, we propose using a combined algorithm that falls back on our approach solely for the critical region.

1.2 Overview of the Paper

In the next section we consider several common greedy matching heuristics and give some motivation for our new approach. Following that, in the main part of the paper we describe the experiments and discuss the results.

2 Greedy Matching Heuristics

In this section we give a brief description of the greedy matching algorithms considered here. The structure of this section is similar to Section 3 of [15].

Basic Structure. The algorithms work recursively. Let $G_0 = G$ be the input graph. Consider some arbitrary recursion level $l \geq 0$. Let G_l be the current graph, and let d be the minimum degree of G_l. There are two cases:

$d \leq 2$. Apply an "optimal reduction step" on G_l, i.e., depending on d, remove nodes and edges from G_l to obtain G_{l+1}.

$d \geq 3$. Apply a "heuristic reduction step" on G_l, i.e., choose an edge $e = \{u, v\}$ from G_l with the highest priority according to some heuristic order of priority, and remove u and v and all incident edges from G_l to obtain G_{l+1}.

Now run the algorithm recursively on G_{l+1}, which will return a matching M_{l+1} for G_{l+1}. Finally, add an edge to M_{l+1} to obtain a matching M_l for G_l. An optimal step will never decrease the size of a maximum matching, while a heuristic step might do that.

Optimal Steps. The two optimal steps that we consider are commonly known as "degree 1 reduction" and "degree 2 reduction". They are based on the following facts proved by Karp and Sipser in [14].

Fact 1. Let $G = (V, E)$ be a graph. If there exists a node $u \in V$ with degree $\deg(u) = 1$, adjacent to a node $v \in V$, then there exists a maximum matching M in G with $\{u, v\} \in M$.

Fact 2. Let $G = (V, E)$ be a graph. If there exists a node $u \in V$ with degree $\deg(u) = 2$, adjacent to nodes $v_1, v_2 \in V$, then there exists a maximum matching M in G with either $\{u, v_1\} \in M$ or $\{u, v_2\} \in M$.

For any subset V' of the nodes of G let $G \setminus V'$ be the subgraph of G that is induced by all nodes of $V \setminus V'$. (Note that if e is an edge of G, then $G \setminus e$ means that the two nodes of e are removed.) Let $G \circ V'$ be the graph that results from G by contracting all nodes of V' into a single node and removing all multiple edges and self-loops. Using these definitions we can state the optimal degree reduction steps as follows.

degree 1 reduction: Randomly choose a node u from G_l with degree $\deg(u) = 1$, incident to an edge e. Shrink the graph G_l via $G_{l+1} \leftarrow G_l \setminus e$. Enlarge the matching M_{l+1} given by the recursive call via $M_l \leftarrow M_{l+1} \cup \{e\}$.

degree 2 reduction: Randomly choose a node u from G_l with $\deg(u) = 2$, adjacent to nodes v_1, v_2. Contract the three nodes into a single node v via $G_{l+1} \leftarrow G_l \circ \{u, v_1, v_2\}$ and store how v was constructed. If an edge $e = \{v, w\}$ is part of the matching M_{l+1} given by the recursive call, then to obtain the matching M_l either replace e with $\{v_1, w\}$ in M_{l+1} and add $\{u, v_2\}$ to M_{l+1}, or replace e with $\{v_2, w\}$ in M_{l+1} and add $\{u, v_1\}$ to M_{l+1}.

In the following we will use "OPT(1)" and "optimal degree 1 reduction", as well as "OPT(1,2)" and "optimal degree 1 and optimal degree 2 reduction" synonymously.

Heuristic Steps. The procedure for the heuristic step is similar to the degree 1 reduction step. First choose an edge e, then shrink the graph via $G_{l+1} \leftarrow G_l \setminus e$, and finally enlarge the matching M_{l+1} obtained by the recursive call on G_{l+1} via $M_l \leftarrow M_{l+1} \cup \{e\}$. The choice of the edge is based on a priority order of the edges, where the priorities are calculated using properties in the neighborhood of the nodes. We consider the following heuristics.

random edge: Randomly choose an edge $e \in E$.

double minimum degree: Randomly choose a node $u \in V$ among the nodes with smallest degree. Randomly choose an edge $e = \{u, v\} \in E$ where v is among the neighbors of u that have smallest degree.

minimum expected potential, minimum degree: Randomly choose a node $u \in V$ among the nodes with smallest potential $\pi(u)$, where

$$\pi(u) = \sum\nolimits_{\{u,v\} \in E} \frac{1}{\deg(v)} \; .$$

Then randomly choose an edge $e = \{u, v\} \in E$ where v is among the neighbors of u that have smallest degree.

Simply choosing an edge at random can be seen as all edges having the same priority, which disregards the structure of the graph. The idea of choosing a node of low degree is that the lower the degree the fewer the possibilities of the node u to be covered by a matching. This is taken one step further in the third heuristic by calculating the values $\pi(u)$. If each neighbor v of a node u randomly declares one of its incident edges to be the only edge that is allowed to cover v in a matching then the value $\pi(u)$ is the expected number of potential matching edges that could cover u. As before, the lower the number of possibilities the more urgent it is to include the node in a matching edge.

In the following we will use interchangeably: "HEU(rand)" and "random edge heuristic", "HEU(deg,deg)" and "double minimum degree heuristic", as well as "HEU(pot,deg)" and "minimum expected potential, minimum degree heuristic".

Algorithms. We list six matching algorithms whose performance is examined in our experiments, where the last two algorithms are new. The names of the algorithms are generic, describing their structure as combination of the utilized optimal and heuristic steps. If an algorithm uses OPT(1,2) then the degree 1 reduction step is always preferred to the degree 2 reduction step.

OPT(1):HEU(rand) This algorithm is commonly known as Karp-Sipser algorithm as it was first analyzed by Karp and Sipser in [14, Algorithm 2]. If the expected degree c of a sparse random graph is below e then the algorithm finds a maximum matching (with high probability) and if c is larger than e then the matching is within $n^{1/5+o(1)}$ of the maximum cardinality (with high probability), see [2].

OPT(1,2):HEU(rand) This is a variant of the Karp-Sipser algorithm using in addition the degree 2 reduction step, which was also proposed in [14]. It is included to investigate the effect of the degree 2 reduction.

OPT(1):HEU(deg,deg) This algorithm is recommended in the experimental study [15] as the most practical algorithm, see [15, Conclusion]. Note that the optimal degree 1 reduction does not need to be implemented separately since it is performed implicitly by the heuristic step.

OPT(1,2):HEU(deg,deg) This is one of the two algorithms proposed in [15] that offer the highest quality of solution. The other one (called BlockRed) is more complicated, using an additional optimal reduction, but has very similar performance. It was demonstrated experimentally that both algorithms are likely to compute a maximum matching in sparse random graphs when $c < 2.6$ or $c > 3.8$, but fail to do so for other values of c. Moreover, in the "critical region" $2.6 \leq c \leq 3.8$ the number of edges that are missing from a matching with maximum cardinality is increasing with increasing n.

OPT(1):HEU(pot,deg) This is the first new algorithm. It is a straightforward adaptation of the selfless algorithm proposed by Sanders in [18] for determining an orientation of the edges of an undirected graph. The selfless algorithm has been proven to be optimal in the sense that with high probability it obtains an orientation of the edges of an undirected sparse random graph that gives maximum in-degree k, if the average degree c is such that such an orientation is likely to exist, see [6].

OPT(1,2):HEU(pot,deg) This is the second new algorithm and the outcome of our search for an algorithm that has probably no critical region. As shown in the following experiments the additional use of the degree 2 reduction is essential.

Note that the recursive structure of the algorithms can easily be transformed into an iterative structure, if there is no degree 2 reduction or one only needs to compute the size of a maximum matching, since in both cases there is no need to undo the contraction of nodes.

Algorithm OPT(1,2):HEU(pot,deg) is the heuristic we propose for computing maximum cardinality matchings in sparse random graphs. Therefore its pseudocode (Algorithm 1) is given below for completeness.

Algorithm 1. `OPT(1,2):HEU(pot,deg)[`G`: graph]`

```
Input: simple graph $G = (V, E)$ with node set $V$ and edge set $E$
Output: matching $M$
$M \leftarrow \emptyset$;
if $E \neq \emptyset$ then
    $d \leftarrow$minimum degree of all nodes in $V$;
    if $d = 1$ then
        $u \leftarrow$ random node from $V$ with deg($u$) $= 1$;
        $v \leftarrow$ neighbor of $u$;
        $M \leftarrow$ OPT(1,2):HEU(pot,deg)[$G \setminus \{u, v\}$];
        $M \leftarrow M \cup \{\{u, v\}\}$;
    else if $d = 2$ then
        $u \leftarrow$ random node from $V$ with deg($u$) $= 2$;
        $\{v_1, v_2\} \leftarrow$ set of 2 neighbors of $u$;
        $v \leftarrow \{u, v_1, v_2\}$;
        $M \leftarrow$ OPT(1,2):HEU(pot,deg)[$G \circ \{u, v_1, v_2\}$];
        if $v$ is not matched in $M$ then $M \leftarrow M \cup \{\{u, v_1\}\}$;
        else
            $w \leftarrow$ matching neighbor of $v$;
            $M \leftarrow M \setminus \{\{v, w\}\}$;
            if $\{v_1, w\} \in E$ then $M \leftarrow M \cup \{\{v_1, w\}, \{u, v_2\}\}$;
            else $M \leftarrow M \cup \{\{v_2, w\}, \{u, v_1\}\}$;
    else
        $\pi \leftarrow$minimum potential of all nodes in $V$;
        $u \leftarrow$ random node from $V$ with $\pi(u) = \pi$;
        $N \leftarrow$ set of neighbors of $u$;
        $v \leftarrow$ random node from $N$ with minimum degree;
        $M \leftarrow$ OPT(1,2):HEU(pot,deg)[$G \setminus \{u, v\}$];
        $M \leftarrow M \cup \{\{u, v\}\}$;
return $M$;
```

3 Experiments

We examine the performance of the six greedy matching algorithms, described in the last section, on random general graphs $G(n; c)$ and random bipartite graphs $B(n/2, n/2; c)$ with n nodes and constant expected average degree c. We cover parameter ranges $n \in \{10^4, 10^5, 10^6\}$ and $c \in [1, 10]$, where parameter c is iteratively increased via $c = 1 + i \cdot 0.1$, for $i = 0, 1, \ldots, 90$.

Remark 1. It is well known that for $c > 1$ the size of the largest component of the random graphs considered here is linear in n with high probability (the "giant component").

Construction of Random Graphs. Let $N = \binom{n}{2}, p = c/(n-1)$ for random general graphs $G(n; c)$, and let $N = n^2/4, p = 2 \cdot c/n$ for random bipartite graphs $B(n/2, n/2; c)$. For fixed parameters (n, c) the construction of a random graph $G = (V, E)$ is done as follows. We start with the node set $V = \{1, 2, \ldots, n\}$ and an empty edge set E. If $n = 10^4$ then each of the N possible edges is generated and added to E with probability p independently of all other edges. If $n \in \{10^5, 10^6\}$ then, in order to keep the construction time manageable, we first determine the number of edges X, which is expected to be linear in n, and then randomly choose X edges from the set of N possible edges. The number of edges follows a binomial distribution $X \sim \text{Bin}(N, p)$. To determine a realization x of X, we obtain a realization y of a standard normal random variable $Y \sim \text{Nor}(0, 1)$ using the polar method [20, Section 2.3.1]. The value $\tilde{x} = \text{round}(y \cdot \sqrt{N \cdot p(1-p)} + N \cdot p)$ is used as an approximation of x. As long as $\tilde{x}$ is not feasible the calculation is repeated with new realizations of N.

Measurements. For each pair of parameters (n, c) we constructed 100 random graphs (bipartite and general) and measured the following quantities for each of the six heuristics:

- the failure rate λ. This is the fraction of graphs where the matching obtained by the heuristic is not a maximum matching.
- the average number of "lost edges" ρ, which we define as the average number of edges missing from a maximum matching, conditioned on the event that a failure occurs. If no failure occurs we let $\rho = 0$.

To get insight in how the parameter c might influence the running time of our new algorithm we did additional experiments using random graphs with $n = 10^6$ nodes. For each c we constructed 10 random graphs (bipartite and general) and measured the following quantities for OPT(1,2):HEU(pot,deg):

- the average running time $\bar{t}$ needed to obtain a matching, as well as the corresponding sample standard deviation.
- the average fraction of: degree 1 reduction steps $\overline{\#o1}$, degree 2 reduction steps $\overline{\#o2}$, and heuristic steps $\overline{\#h}$.

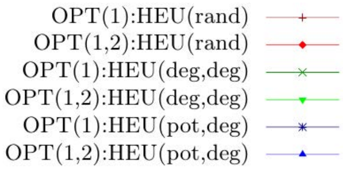
System. The source code for the graph generators as well as for the algorithms is written in C++ and compiled with g++ version 4.5.1. The experiments regarding the running time ran on an Intel Xeon CPU E5450 (using one core) under openSUSE with kernel 2.6.37.6-0.11-desktop.

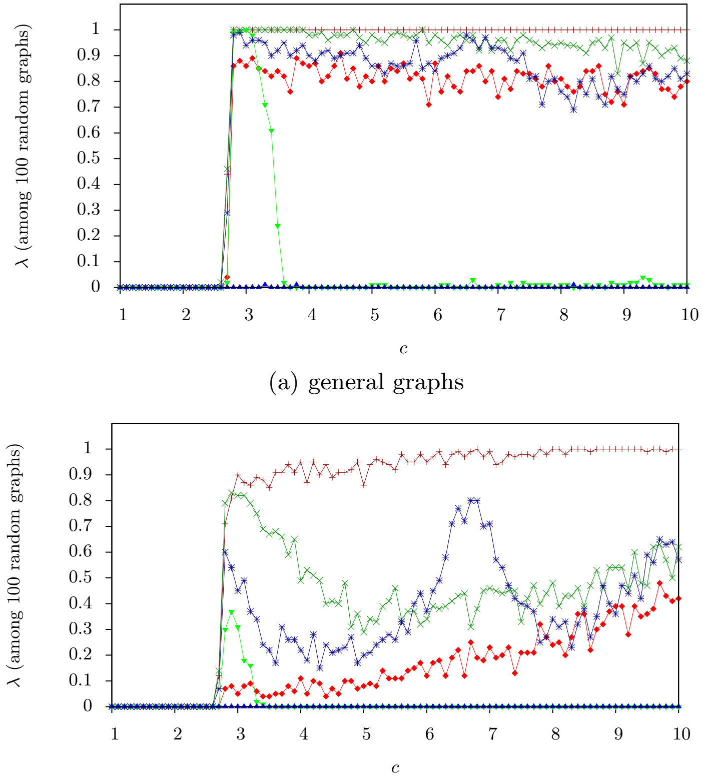
"Random" Numbers. For the necessary random choices for the algorithms as well as for the construction of the random graphs we used the pseudo random number generator MT19937 "Mersenne Twister" of the GNU Scientific Library [12].

3.1 Results

Here we consider results from the matching heuristics given in Section 2.

Failure Rates. Figure 1 gives the failure rates on general and bipartite random graphs with $n = 10^6$ nodes and expected degree c ranging from 1 to 10. The legend for both plots is given to the right. Figures depicting the failure rates for graphs with 10^4 and 10^5 nodes are given in the full version of the paper [10]. The results are qualitatively similar to the results for $n = 10^6$.





(a) general graphs

(b) bipartite graphs

Fig. 1. Failure rates on graphs with $n = 10^6$ nodes

For $1 \leq c \leq 2.5$ no failure occurred in any of the algorithms. Our new algorithm OPT(1,2):HEU(pot,deg) never failed on bipartite graphs and failed three times on general graphs, for $c \in \{3.3, 3.8, 8.2\}$ with failure rate $\lambda = 1/100$. For the other algorithms we observed the following behavior.

- For general graphs at $c = 2.8$ all of them have a failure rate λ of at least 0.86. For OPT(1,2):HEU(deg,deg) we could replicate the behavior, observed in [15], that for $c \leq 2.6$ and $c \geq 3.7$ the failure rate of the algorithm is almost zero while for the other values of c the failure rate is very high, reaching its peak with $\lambda = 1$ at $c = 3.0$. For the other heuristics λ stays quite high beyond $c = 2.8$.

- For bipartite graphs the situation is different. The failure rates go up only beyond 2.6 and the qualitative behavior varies widely among the different heuristics. For OPT(1,2):HEU(deg,deg) we observed a critical region of $2.8 < c < 3.5$ but with a less pronounced failure rate, reaching its peak at $c = 2.9$ with $\lambda = 0.37$. For all other algorithms the failure rate seems to increase for c beyond 8.

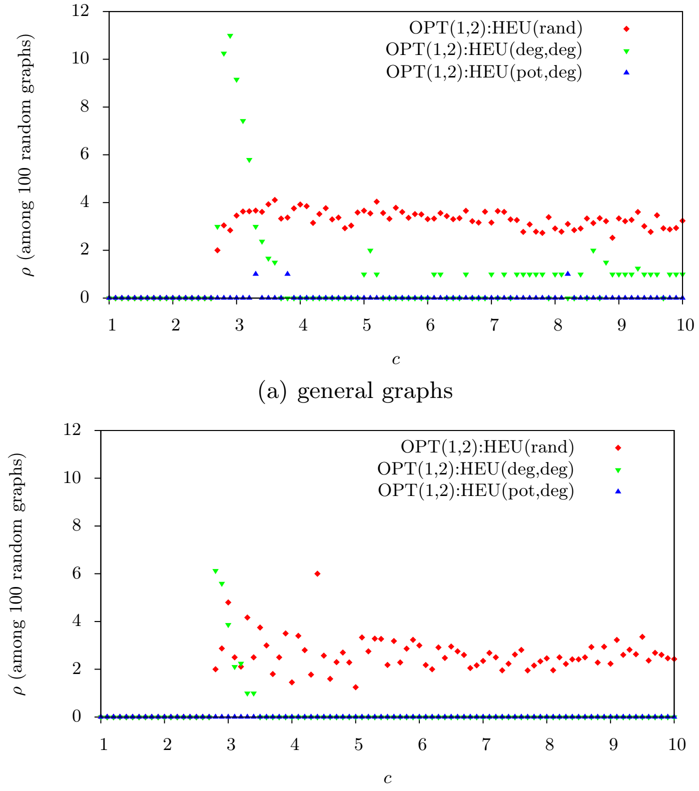
In [2], it is proved that OPT(1):HEU(rand) is likely to find a maximum matching for $c < e \approx 2.718$ mainly due to the optimal degree 1 reduction steps (so called e-phenomenon). Our results indicate that including degree 2 reductions does not influence this bound much. Overall, the heuristics with degree 2 reduction more often give a maximum matching than their counterparts that can only utilize degree 1 reduction. In terms of the difference of the failure rates this effect is smallest for OPT(1):HEU(rand) and OPT(1,2):HEU(rand) on general random graphs. The best algorithms in terms of quality of solution are OPT(1,2):HEU(deg,deg) and OPT(1,2):HEU(pot,deg).

Edges Lost if Failure Occurs. Unlike before, we are only interested in the algorithms using degree 2 reduction, since on average they obtain the largest matchings. Figure 2 gives the average number of lost edges conditioned on the event that a failure occurs, for general and bipartite random graphs with $n = 10^6$ nodes and expected degree c ranging from 1 to 10. The figures for the number of lost edges for graphs with 10^4 and 10^5 nodes are given in [10]. The results are qualitatively similar to the results for $n = 10^6$.

The number of lost edges for heuristic OPT(1,2):HEU(rand) and for heuristic OPT(1,2):HEU(deg,deg), within their critical ranges, increases with increasing n, cf. [10]. In the range of $2.8 \leq c \leq 10$ the mean over the values ρ for heuristic OPT(1,2):HEU(rand) is higher for the general graph scenario than for the bipartite graph scenario, while the variance of ρ is lower. Outside its critical range the double minimum degree heuristic OPT(1,2):HEU(deg,deg) loses zero or one edges for most of the average degrees c on general graphs, and no edge on bipartite graphs. Our new algorithm OPT(1,2):HEU(pot,deg) loses one edge only in three cases.

Running Time Behavior. Figure 3 shows the average running time $\bar{t}$ of algorithm OPT(1,2):HEU(pot,deg) for calculating a matching, as well as the corresponding average fraction of degree 1 reduction steps $\overline{\#o1}$, degree 2 reduction steps $\overline{\#o2}$, and heuristic steps $\overline{\#h}$, on general random graphs with 10^6 nodes. The run-time behavior on bipartite random graphs of this size is qualitatively and quantitatively quite similar and given in [10]. The failure rate was zero in these experiments.

The average running time exhibits a non-linear increase. In a first phase, for $1 \leq c \leq 2.7$, the slope is linear and quite low. This is because in this range the running time is dominated by the fraction of degree 1 reduction steps $\overline{\#o1}$, which



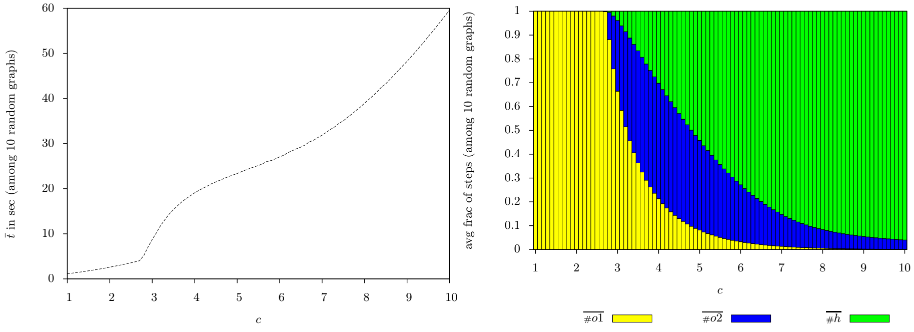
(a) general graphs

(b) bipartite graphs

Fig. 2. Average number of lost edges (if $\lambda > 0$) for graphs with $n = 10^6$ nodes

makes up more than 99 percent of all reduction steps. It follows a second phase starting with a sudden increase of $\bar{t}$ which starts to flatten soon at c about 3.5. This goes along with a strong decrease of $\overline{\#o1}$ and increase of $\overline{\#o2}$ and $\overline{\#h}$. The next slight increase of the slope seems to be between $c = 6$ and $c = 7$ when $\overline{\#o1}$ falls below 0.03 and the fraction of heuristic steps $\overline{\#h}$ is more than 0.7, which indicates the begin of a third phase. The slope in this phase is larger than in the first phase and seems to be slightly non-linear. The sample standard deviation of the running time is very low for the first phase and then increases slightly with increasing c; we observed a maximum of about 0.46 sec for general random graphs and of about 0.32 sec for bipartite random graphs.

Remark 2. To give an idea how to relate the running time of our implementation of the heuristic OPT(1,2):HEU(pot,deg) to the running time of an exact matching algorithm, we give a rough comparison with the maximum cardinality matching algorithm of the Boost C++ library [1], which is an implementation of Edmonds' algorithm [11] that follows closely [19]. On random general graphs, using an initial matching from the OPT(1,2):HEU(rand) heuristic the exact



(a) average running time

(b) average fraction of steps (stacked histogram)

Fig. 3. Run-time behavior of algorithm OPT(1,2):HEU(pot,deg) on general random graphs with 10^6 nodes

algorithm was roughly 1.5 till 7.6 times faster (the gain increased with increasing c), and using an empty initial matching the exact algorithm was about 59.4 times slower for $c = 4$ and about 40.4 times slower for $c = 5$.

4 Summary and Future Work

We proposed a new greedy algorithm to solve the maximum cardinality matching problem on random graphs with constant expected degree c, and found in experiments that this algorithm has a very low failure rate for a broad range of c. It is an open problem to prove that this behavior is to be expected.

As suggested by a reviewer, an interesting possible future work is to study the performance of our heuristic on other classes of sparse random graphs that have the property that with high probability

- there exists a matching of size $(1 - o(1)) \cdot n/2$, that is an almost perfect matching, as proved in [4] for graphs with a fixed log-concave degree profile.
- there exists no almost perfect matching, see e.g. [5, Appendix].

The algorithm itself is an adaptation of the selfless algorithm of Sanders [18] for orienting graphs, which was successfully generalized to orienting hypergraphs before, see [9]. It seems possible that the "selfless approach" can be used as generic building block for other greedy algorithms on random graphs too, like, e.g., graph coloring, which would be interesting to investigate.

Acknowledgment. We thank the referees for their remarks that helped in improving the paper.

References

1. Boost C++ Libraries (version 1.44.0), http://www.boost.org/
2. Aronson, J., Frieze, A.M., Pittel, B.: Maximum matchings in sparse random graphs: Karp-Sipser revisited. Random Struct. Algorithms 12(2), 111–177 (1998)
3. Bast, H., Mehlhorn, K., Schäfer, G., Tamaki, H.: Matching Algorithms Are Fast in Sparse Random Graphs. Theory Comput. Syst. 39(1), 3–14 (2006)
4. Bohman, T., Frieze, A.M.: Karp-Sipser on Random Graphs with a Fixed Degree Sequence. Combinatorics, Probability & Computing 20(5), 721–741 (2011)
5. Bordenave, C., Lelarge, M., Salez, J.: Matchings on infinite graphs. CoRR arXiv:1102.0712 (2011)
6. Cain, J.A., Sanders, P., Wormald, N.C.: The Random Graph Threshold for k-orientiability and a Fast Algorithm for Optimal Multiple-Choice Allocation. In: Proc. 18th SODA, pp. 469–476. ACM-SIAM (2007)
7. Chebolu, P., Frieze, A.M., Melsted, P.: Finding a Maximum Matching in a Sparse Random Graph in $O(n)$ Expected Time. J. ACM 57(4) (2010)
8. Coppersmith, D., Winograd, S.: Matrix Multiplication via Arithmetic Progressions. J. Symb. Comput. 9(3), 251–280 (1990)
9. Dietzfelbinger, M., Goerdt, A., Mitzenmacher, M., Montanari, A., Pagh, R., Rink, M.: Tight Thresholds for Cuckoo Hashing via XORSAT. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010, Part I. LNCS, vol. 6198, pp. 213–225. Springer, Heidelberg (2010)
10. Dietzfelbinger, M., Peilke, H., Rink, M.: A More Reliable Greedy Heuristic for Maximum Matchings in Sparse Random Graphs. CoRR arXiv:1203.4117 (2012)
11. Edmonds, J.: Paths, trees, and flowers. Canadian Journal of Mathematics 17, 449–467 (1965)
12. Galassi, M., Davies, J., Theiler, J., Gough, B., Jungman, G., Alken, P., Booth, M., Rossi, F.: GNU Scientific Library Reference Manual - Edition 1.15, for GSL Version 1.15 (2011), http://www.gnu.org/software/gsl/manual/
13. Hopcroft, J.E., Karp, R.M.: An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs. SIAM J. Comput. 2(4), 225–231 (1973)
14. Karp, R.M., Sipser, M.: Maximum Matchings in Sparse Random Graphs. In: Proc. 22nd FOCS, pp. 364–375. IEEE Computer Society (1981)
15. Magun, J.: Greedy Matching Algorithms: An Experimental Study. ACM Journal of Experimental Algorithmics 3, 6 (1998)
16. Micali, S., Vazirani, V.V.: An $O(\sqrt{|v|}\cdot|E|)$ Algorithm for Finding Maximum Matching in General Graphs. In: Proc. 21st FOCS, pp. 17–27. IEEE Computer Society (1980)
17. Mucha, M., Sankowski, P.: Maximum Matchings via Gaussian Elimination. In: Proc. 45th FOCS, pp. 248–255. IEEE Computer Society (2004)
18. Sanders, P.: Algorithms for Scalable Storage Servers. In: Van Emde Boas, P., Pokorný, J., Bieliková, M., Štuller, J. (eds.) SOFSEM 2004. LNCS, vol. 2932, pp. 82–101. Springer, Heidelberg (2004)
19. Tarjan, R.E.: Data Structures and Network Algorithms. SIAM, Philadelphia (1983)
20. Thomas, D.B., Luk, W., Leong, P.H., Villasenor, J.D.: Gaussian Random Number Generators. ACM Comput. Surv. 39 (2007)
21. Vazirani, V.V.: A Theory of Alternating Paths and Blossoms for Proving Correctness of the $O(\sqrt{V}E)$ General Graph Maximum Matching Algorithm. Combinatorica 14(1), 71–109 (1994)

Branch Mispredictions Don't Affect Mergesort

Amr Elmasry[1], Jyrki Katajainen[1,2], and Max Stenmark[2]

[1] Department of Computer Science, University of Copenhagen,
Universitetsparken 1, 2100 Copenhagen East, Denmark
[2] Jyrki Katajainen and Company,
Thorsgade 101, 2200 Copenhagen North, Denmark

Abstract. In quicksort, due to branch mispredictions, a skewed pivot-selection strategy can lead to a better performance than the exact-median pivot-selection strategy, even if the exact median is given for free. In this paper we investigate the effect of branch mispredictions on the behaviour of mergesort. By decoupling element comparisons from branches, we can avoid most negative effects caused by branch mispredictions. When sorting a sequence of n elements, our fastest version of mergesort performs $n \log_2 n + O(n)$ element comparisons and induces at most $O(n)$ branch mispredictions. We also describe an in-situ version of mergesort that provides the same bounds, but uses only $O(\log_2 n)$ words of extra memory. In our test computers, when sorting integer data, mergesort was the fastest sorting method, then came quicksort, and in-situ mergesort was the slowest of the three. We did a similar kind of decoupling for quicksort, but the transformation made it slower.

1 Introduction

Branch mispredictions may have a significant effect on the speed of programs. For example, Kaligosi and Sanders [8] showed that in quicksort [6] it may be more advantageous to select a skewed pivot instead of finding a pivot close to the median. The reason for this is that for a comparison against the median the outcome has a fifty percent chance of being smaller or larger, whereas the outcome of comparisons against a skewed pivot is easier to predict. All in all, a skewed pivot will lead to a better branch prediction and—possibly—a decrease in computation time. In a same vein, Brodal and Moruz [3] showed that skewed binary search trees can perform better than perfectly balanced search trees.

In this paper we tackle the following question posed in [8]. Given a random permutation of the integers $\{0, 1, \ldots, n-1\}$, does there exist a faster in-situ sorting algorithm than quicksort with skewed pivots for this particular type of input? We use the word *in-situ* to indicate that the algorithm is allowed to use $O(\log_2 n)$ extra words of memory (as any careful implementation of quicksort).

It is often claimed that quicksort is faster than mergesort. To check the correctness of this claim, we performed some simple benchmarks for the quicksort (`std::sort`) and mergesort (`std::stable_sort`) programs available at the GNU implementation (`g++` version 4.6.1) of the C++ standard library; `std::sort` is

R. Klasing (Ed.): SEA 2012, LNCS 7276, pp. 160–171, 2012.

Table 1. The execution time [ns], the number of conditional branches, and the number of mispredictions on two of our computers (Per and Ares), each per $n \log_2 n$, for the quicksort and mergesort programs taken from the C++ standard library

Program	`std::sort`				`std::stable_sort`			
	Time		**Branches**	**Mispredicts**	**Time**		**Branches**	**Mispredicts**
n	**Per**	**Ares**			**Per**	**Ares**		
2^{10}	6.5	5.3	1.47	0.45	6.2	5.0	2.05	0.14
2^{15}	6.2	5.2	1.50	0.43	5.9	4.7	2.02	0.09
2^{20}	6.2	5.1	1.50	0.43	6.3	4.7	2.01	0.07
2^{25}	6.1	5.1	1.51	0.43	6.1	4.6	2.01	0.05

an implementation of Musser's introsort [13] and `std::stable_sort` is an implementation of bottom-up mergesort. In our test environment[1], for integer data, the two programs had the same speed within the measurement accuracy (see Table 1). An inspection of the assembly-language code produced by `g++` revealed that in the performance-critical inner loop of `std::stable_sort` all element comparisons were followed by conditional moves. A *conditional move* is written in C as **if** `(a ◁ b) x = y`, where `a`, `b`, `x`, and `y` are some variables (or constants), and ◁ is some comparison operator. This instruction, or some of its restricted forms, is supported as a hardware primitive by most computers. By using a branch-prediction profiler (`valgrind`) we could confirm that the number of branch mispredictions per $n \log_2 n$—referred to as the *branch-misprediction ratio*—was much lower for `std::stable_sort` than for `std:.sort`. Based on these initial observations, we concluded that mergesort is a noteworthy competitor to quicksort.

Our main results in this paper are:

1. We optimize (reduce the number of branches executed and branch mispredictions induced) the standard-library mergesort so that it becomes faster than quicksort for our task in hand (Section 2).
2. We describe an in-situ version of this optimized mergesort (Section 3). Even though an ideal translation of its inner loop only contains 18 assembly-language instructions, in our experiments it was slower than quicksort.

[1] The experiments discussed in the paper were carried out on two computers:
Per: Model: Intel® Core$^{\text{TM}}$2 CPU T5600 @ 1.83GHz; main memory: 1 GB; L2 cache: 8-way associative, 2 MB; cache line: 64 B.
Ares: Model: Intel® Core$^{\text{TM}}$ i3 CPU M 370 @ 2.4GHz × 4; main memory: 2.6 GB; L2 cache: 12-way associative, 3 MB; cache line: 64 B.
Both computers run under Ubuntu 11.10 (Linux kernel 3.0.0-15-generic) and `g++` compiler (`gcc` version 4.6.1) with optimization level `-O3` was used. According to the documentation, at optimization level `-O3` this compiler always attempted to transform conditional branches into branch-less equivalents. Micro-benchmarking showed that in Per conditional moves were faster than conditional branches when the result of the branch condition was unpredictable. In Ares the opposite was true. All execution times were measured using `gettimeofday` in `sys/time.h`. For a problem of size n, each experiment was repeated $2^{26}/n$ times and the average was reported.

3. We eliminated all branches from the performance-critical loop of quicksort (Section 4). After this transformation the program induces $O(n)$ branch mispredictions on the average. However, in our experiments the branch-optimized versions of quicksort were slower than `std::sort`.
4. We made a number of experiments for quicksort with skewed pivots (Section 4). We could repeat the findings reported in [8], but the performance improvement obtained by selecting a skewed pivot was not very large. For our mergesort programs the branch-misprediction ratio is significantly lower than that reported for quicksort with skewed pivots in [8].

We took the idea of decoupling element comparisons from branches from Mortensen [12]. He described a variant of mergesort that performs $n \log_2 n + O(n)$ element comparisons and induces $O(n)$ branch mispredictions. However, the performance-critical loop of the standard-library mergesort only contains 14 assembly-language instructions, whereas that of Mortensen's program contains more. This could be a reason why Mortensen's implementation is slower than the standard-library implementation. Our key improvement is to keep the instruction count down while doing the branch optimization.

The idea of decoupling element comparisons from branches was also used by Sanders and Winkel [15] in their samplesort. The resulting program performs $n \log_2 n + O(n)$ element comparisons and induces $O(n)$ branch mispredictions in the expected case. As for Mortensen's mergesort, samplesort needs $O(n)$ extra space. Using the technique described in [15], one can modify heapsort such that it will achieve the same bound on the number of branch mispredictions in addition to its normal characteristics. In particular, heapsort is fully in-place but it suffers from a bad cache behaviour [5].

Brodal and Moruz [2] proved that any comparison-based program that sorts a sequence of n elements using $O(\beta n \log_2 n)$ element comparisons, for $\beta > 1$, must induce $\Omega(n \log_\beta n)$ branch mispredictions. However, this result only holds under the assumption that every element comparison is followed by a conditional branch depending on the outcome of the comparison. In particular, after decoupling element comparisons from branches the lower bound is no more valid.

The branch-prediction features of a number of sorting programs were experimentally studied in [1]; also a few optimizations were made to known methods. In a companion paper [5] it is shown that any program can be transformed into a form that induces only $O(1)$ branch mispredictions. The resulting programs are called *lean*. However, for a program of length κ, the transformation may make the lean counterpart a factor of κ slower. In practice, the slowdown is not that big, but the experiments showed that lean programs were often slower than moderately branch-optimized programs. In [5], lean versions of mergesort and heapsort are presented. In the current paper, we work towards speeding up mergesort even further and include quicksort in the study.

A reader who is unfamiliar with the branch-prediction techniques employed at the hardware level should recap the basic facts from a textbook on computer organization (e.g. [14]). In our theoretical analysis, we assume that the branch predictor used is static. A typical static predictor assumes that forward branches are not taken and backward branches are taken. Hence, for a conditional branch at the end of a loop the prediction is correct except for the last iteration when stepping out of the loop.

2 Tuned Mergesort

In the C++ standard library shipped with our compiler, `std::stable_sort` is an implementation of bottom-up mergesort. First, it divides the input into chunks of seven elements and sorts these chunks using insertionsort. Second, it merges the sequences sorted so far pairwise, level by level, starting with the sorted chunks, until the whole sequence is sorted. If possible, it allocates an extra buffer of size n, where n is the size of the input, then it alternatively moves the elements between the input and the buffer, and produces the final output in the place of the original input. If no extra memory is available, it reverts to an in-situ sorting strategy, which is asymptotically slower than the one using extra space.

One reason for the execution speed is a tight (compact) inner loop. We reproduce it in a polished form below on the left together with its assembly-language translation on the right. When illustrating the assembly-language translations, we use pure C [10], which is a glorified assembly language with the syntax of C [11]. In the following code extracts, `p` and `q` are iterators pointing to the current elements of the two input sequences, `r` is an iterator indicating the current output position, `t1` and `t2` are iterators indicating the first positions beyond the input sequences, and `less` is the comparator used in element comparisons. The additional variables are temporary: `s` and `t` store iterators, `x` and `y` elements, and `done` and `smaller` Boolean values.

```
while (p != t1 && q != t2) {
  if (less(*q, *p)) {
    *r = *q;
    ++q;
  }
  else {
    *r = *p;
    ++p;
  }
  ++r;
}
```

```
test:
  done = (q == t2);
  if (done) goto exit;
entrance:
  x = *p;
  s = p + 1;
  y = *q;
  t = q + 1;
  smaller = less(y, x);
  if (smaller) q = t;
  if (! smaller) p = s;
  if (! smaller) y = x;
  *r = y;
  ++r;
  done = (p == t1);
  if (! done) goto test;
exit:
```

Since the two branches of the **if** statement are so short and symmetric, a good compiler will compile them using conditional moves. The assembly-language translation corresponding to the pure-C code was produced by the `g++` compiler. As shown on the right above, the output contained 14 instructions.

By decoupling element comparisons from branches, each merging phase of two subsequences induces at most $O(1)$ branch mispredictions. In total, the merge procedure is invoked $O(n)$ times, so the number of branch mispredictions induced is $O(n)$. Other characteristics of bottom-up mergesort remain the same; it performs $n \log_2 n + O(n)$ element comparisons and element moves.

Table 2. The execution time [ns], the number of conditional branches, and the number of mispredictions, each per $n \log_2 n$, for bottom-up mergesort taken from the C++ standard library and our optimized mergesort

Program	`std::stable_sort`				**Optimized mergesort**			
	Time		**Branches**	**Mispredicts**	**Time**		**Branches**	**Mispredicts**
n	**Per**	**Ares**			**Per**	**Ares**		
2^{10}	6.2	5.0	2.05	0.14	4.4	3.5	0.75	0.06
2^{15}	5.9	4.7	2.02	0.09	4.4	3.5	0.66	0.04
2^{20}	6.3	4.7	2.01	0.07	5.2	3.7	0.62	0.03
2^{25}	6.1	4.6	2.01	0.05	5.2	3.7	0.60	0.02

To reduce the number of branches executed and the number of branch mispredictions induced even further, we implemented the following optimizations:

- Handle small subproblems differently: Instead of using insertionsort, we sort each chunk of size four with a straight-line code that has no branches. In brief, we simulate a sorting network for four elements using conditional moves. Insertionsort induces one branch misprediction per element, whereas our routine only induces $O(1)$ branch mispredictions in total.
- Unfold the main loop responsible for merging: When merging two subsequences, we move four elements to the output sequence in each iteration. We do this as long as each of the two subsequences to be merged has at least four elements. Hereafter in this performance-critical loop the instructions involved in the conditional branches, testing whether or not one of the input subsequences is exhausted, are only executed every fourth element comparison. If one or both subsequences have less than four elements, we handle the remaining elements by a normal (not-unfolded) loop.

To see whether or not our improvements are effective in practice, we tested our optimized mergesort against `std::stable_sort`. Our results are reported in Table 2. From these results, it is clear that even improvements in the linear term can be significant for the efficiency of a sorting program.

3 Tuned In-Situ Mergesort

Since the results for mergesort were so good, we set ourselves a goal to show that some variation of the in-place mergesort algorithm of Katajainen et al. [9] will be faster than quicksort. We have to admit that this goal was too ambitious, but we came quite close. We should also point out that, similar to quicksort, the resulting sorting algorithm is no more stable.

The basic step used in [9] is to sort half of the elements using the other half as a working area. This idea could be utilized in different ways. We rely on the simplest approach: Before applying the basic step, partition the elements around the median. In principle, the standard-library routine `std::nth_element` can accomplish this task by performing a quicksort-type partitioning. After partitioning and sorting, the other half of the elements can be handled recursively. We stop the recursion when the number of remaining elements is less than $n/\log_2 n$ and use introsort to handle them. An iterative procedure-level description of this sorting program is given below. Its interface is the same as that for `std::sort`.

```
template <typename iterator, typename comparator>
void sort(iterator p, iterator r, comparator less) {
  typedef typename std::iterator_traits<iterator>::difference_type index;
  index n = r - p;
  index threshold = n / ilogb(2 + n);
  while (n > threshold) {
    iterator q_1 = p + n / 2;
    iterator q_2 = r - n / 2;
    converse_relation<comparator> greater(less);
    std::nth_element(p, q_1, r, greater);
    mergesort(p, q_1, q_2, less);
    r = q_1;
    n = r - p;
  }
  std::sort(p, r, less);
}
```

Most of the work is done in the basic steps, and each step only uses $O(1)$ extra space in addition to the input sequence. Compared to normal mergesort, the inner loop is not much longer. In the following code extracts, the variables have the same meaning as those used in tuned mergesort: `p`, `q`, `r`, `s`, `t`, `t1`, and `t2` store iterators; `x` and `y` elements; and `done` and `smaller` Boolean values.

```
while (p != t1 && q != t2) {
  if (less(*q, *p)) {
    s = q;
    ++q;
  }
  else {
    s = p;
    ++p;
  }
  x = *r;
  *r = *s;
  *s = x;
  ++r;
}
```

```
test:
  done = (q == t2);
  if (done) goto exit;
entrance:
  x = *p;
  s = p + 1;
  y = *q;
  t = q + 1;
  smaller = less(y, x);
  if (smaller) s = t;
  if (smaller) q = t;
  if (! smaller) p = s;
  if (! smaller) y = x;
  x = *r;
  *r = y;
  --s;
  *s = x;
  ++r;
  done = (p == t1);
  if (! done) goto test;
exit:
```

As shown on the right above, an ideal translation of the loop contains 18 assembly-language instructions, which is only four more than that required by the inner loop of mergesort. Because of register spilling, the actual code produced by the `g++` compiler was a bit longer; it contained 26 instructions. Again, the two branches of the **if** statement were compiled using conditional moves.

For an input of size m, the worst-case cost of `std::nth_element` and `std::sort` is $O(m)$ and $O(m \log_2 m)$, respectively [13]. Thus, the overhead caused by these subroutines is linear in the input size. Both of these routines require at most a logarithmic amount of extra space. To sum up, we rely on standard library components and ensure that our program only induces $O(n)$ branch mispredictions.

Table 3. The execution time [ns], the number of conditional branches, and the number of mispredictions, each per $n \log_2 n$, for two in-situ variants of mergesort

Program	**In-situ `std::stable_sort`**				**In-situ mergesort**			
	Time		**Branches**	**Mispredicts**	**Time**		**Branches**	**Mispredicts**
n	**Per**	**Ares**			**Per**	**Ares**		
2^{10}	49.2	29.7	9.0	2.08	7.3	5.7	1.93	0.26
2^{15}	57.6	35.0	11.1	2.38	7.1	5.6	1.94	0.15
2^{20}	62.7	38.5	12.9	2.53	7.4	5.7	1.92	0.11
2^{25}	68.0	41.3	14.4	2.62	7.6	5.7	1.92	0.09

In our experiments, we compared our in-situ mergesort against the space-economical mergesort provided by the C++ standard library. The library routine is recursive, so (due to the recursion stack) it requires a logarithmic amount of extra space. The performance difference between the two programs is stunning, as seen in Table 3. We admit that this comparison is unfair; the library routine promises to sort the elements stably, whereas our in-situ mergesort does not. However, this comparison shows how well our in-situ mergesort performs.

4 Comparison to Quicksort

In the C++ standard library shipped with our compiler, `std::sort` is an implementation of introsort [13], which is a variant of median-of-three quicksort [6]. Introsort is half-recursive, it coarsens the base case by leaving small subproblems (of size 16 or smaller) unsorted, it calls insertionsort to finalize the sorting process, and it calls heapsort if the recursion depth becomes too large. Since introsort is known to be fast, it was natural to use it as our starting point.

The performance-critical loop of quicksort is tight as shown on the left below; `p` and `r` are iterators indicating how far the partitioning process has proceeded from the beginning and the end, respectively; `v` is the pivot, and `less` is the comparator used in element comparisons; the four additional variables are temporary: `x` and `y` store elements, and `smaller` and `cross` Boolean values.

```
while (true) {
  while (less(*p, v)) {
    ++p;
  }
  --r;
  while (less(v, *r)) {
    --r;
  }
  if (p >= r) {
    return p;
  }
  x = *p;
  *p = *r;
  *r = x;
  ++p;
}
```

```
  --p;
  goto first_loop;
swap:
  *p = y;
  *r = x;
first_loop:
  ++p;
  x = *p;
  smaller = less(x, v);
  if (smaller) goto first_loop;
second_loop:
  --r;
  y = *r;
  smaller = less(v, y);
  if (smaller) goto second_loop;
  cross = (p < r);
  if (cross) goto swap;
  return p;
```

In the assembly-language translation displayed in pure C [10] on the right above, both of the innermost **while** loops contain four instructions and, after rotating the instructions of the outer loop such that the conditional branch becomes its last instruction, the outer loop contains four instructions as well. Due to instruction scheduling and register allocation, the picture was a bit more complicated for the code produced by the compiler, but the simplified code displayed to the right above is good enough for our purposes.

For the basic version of mergesort, the number of instructions executed per $n \log_2 n$, called the *instruction-execution ratio*, is 14. Let us now analyse this ratio for quicksort. It is known that for the basic version of quicksort the expected number of element comparisons performed is about $2n \ln n \approx 1.39 n \log_2 n$ and the expected number of element exchanges is one sixth of this number [16]. Combining this with the number of instructions executed in different parts of the performance-critical loop, the expected instruction-execution ratio is

$$(4 + (1/6) \times 4) \times 1.39 \approx 6.48\,.$$

This number is extremely low; even for our improved mergesort the instruction-execution ratio is higher (11 instructions).

The key issue is the conditional branches at the end of the innermost **while** loops; their outcome is unpredictable. The performance-critical loop can be made lean using the program transformation described in [5]. A bit more efficient code is obtained by numbering the code blocks and executing the moves inside the code blocks conditionally. We identify three code blocks in the performance-critical loop of Hoare's partitioning algorithm: the two innermost loops and the swap. By converting the **while** loops to **do-while** loops, we could avoid some code repetition. The outcome of the program transformation is given on the left below; variable `lambda` indicates the code block under execution. It turns out that the corresponding code is much shorter for Lomuto's partitioning algorithm described, for example, in [4]. Now the performance-critical loop only contains one **if** statement, and the swap inside it can be executed conditionally. The code obtained by applying the program transformation is shown on the right below.

```
int lambda = 2;
--p;
do {
  if (lambda == 1) *p = y;
  if (lambda == 1) *r = x;
  if (lambda == 1) lambda = 2;
  if (lambda == 2) ++p;
  x = *p;
  smaller = less(x, v);
  if (lambda != 2) smaller = true;
  if (! smaller) lambda = 3;
  if (lambda == 3) --r;
  y = *r;
  smaller = less(v, y);
  if (lambda != 3) smaller = true;
  if (! smaller) lambda = 1;
} while (p < r);
return p;
```

```
while (q < r) {
  x = *q;
  condition = less(x, v);
  if (condition) ++p;
  if (condition) *q = *p;
  if (condition) *p = x;
  ++q;
}
return p;
```

Table 4. The execution time [ns], the number of conditional branches, and the number of mispredictions, each per $n \log_2 n$, for two branch-optimized variants of quicksort

Program	**Optimized quicksort (Hoare)**				**Optimized quicksort (Lomuto)**			
	Time		**Branches**	**Mispredicts**	**Time**		**Branches**	**Mispredicts**
n	**Per**	**Ares**			**Per**	**Ares**		
2^{10}	9.2	6.5	2.93	0.43	6.5	5.2	2.14	0.40
2^{15}	9.5	6.4	3.24	0.42	6.5	5.1	2.34	0.40
2^{20}	9.7	6.5	3.33	0.42	6.6	5.2	2.40	0.40
2^{25}	9.8	6.5	3.46	0.42	6.7	5.3	2.42	0.40

The resulting programs are interesting in several respects. When we rely on Hoare's partitioning algorithm, in each iteration of the loop two element comparisons are performed. Since the loop is executed $\sim 2n \ln n$ times on the average, the expected number of element comparisons increases to $\sim 2.78 n \log_2 n$. This increase does not occur for Lomuto's partitioning algorithm. Note that in the above C code we allow conditional moves between memory locations, and even allow conditional arithmetic. If we are restricted to only use conditional moves (as in pure-C), these instructions need to be substituted by pure-C instructions. Because the underlying hardware only supports conditional moves to registers, the assembly-language instruction counts were a bit higher than that indicated by the C code above; the actual counts were 20 and 11, respectively. This means that the expected instruction-execution ratio (that is a 1.39 factor of the instruction counts) is around 27.8 when Hoare's partitioning is in use and around 15.29 when Lomuto's partitioning is in use. Thus, the cost of eliminating the unpredictable branches is high in both cases!

When testing these branch-optimized versions of quicksort, we observed that the compiler was not able to handle that many conditional moves. In some architectures each such move requires more than one clock cycle, so it may be more efficient to use conditional branches. The performance of our branch-optimized quicksort programs is reported in Table 4. Compared to introsort (see Table 1), these programs are slower. To avoid branch mispredictions, it would be necessary to write the programs in assembly language.

We also tested other variants of introsort by trying different pivot-selection strategies: random element, first element, median of the first, middle and last element, and α-skewed hypothetical element. Since in our setup the elements are given in random order, the simplest pivot-selection strategy—select the first element as the pivot—was already fast, but it was slower than the median-of-three pivot-selection strategy used by introsort. On the other hand, the selection of a skewed pivot indeed improved the performance. In our test environment, for small problem instances the median pivot worked best (i.e. $\alpha = 1/2$), whereas for large problem instances $\alpha = 1/5$ turned out to be the best choice. The results for the naive and α-skewed pivot-selection strategies are given in Table 5.

From these experiments, our conclusion is that the performance of the sorting programs considered is ranked as follows: mergesort, quicksort, and in-situ mergesort. Still, quicksort is the fastest method for in-situ sorting.

Table 5. The execution time [ns], the number of conditional branches, and the number of mispredictions, each per $n \log_2 n$, for two other variants of introsort.

Program	**Introsort (naive pivot)**				**Introsort ((1/5)-skewed pivot)**			
	Time		**Branches**	**Mispredicts**	**Time**		**Branches**	**Mispredicts**
n	**Per**	**Ares**			**Per**	**Ares**		
2^{10}	7.0	5.6	1.78	0.45	6.4	5.1	1.48	0.37
2^{15}	6.6	5.3	1.78	0.43	6.1	4.8	1.53	0.36
2^{20}	6.5	5.1	1.74	0.42	6.0	4.7	1.55	0.35
2^{25}	6.4	5.1	1.72	0.42	6.0	4.7	1.56	0.34

5 Advice for Practitioners

Like sorting programs, most programs can be optimized with respect to different criteria: the number of branch mispredictions, cache misses, element comparisons, or element moves. Optimizing one of the parameters may mean that the optimality with respect to another is lost. Not even the optimal bounds are the best in practice; the best choice depends on the environment where the programs are run. The task of a programmer is difficult. As any activity involving design, good programming requires good compromises.

In this paper we were interested in reducing the cost caused by branch mispredictions. In principle, there are two ways of removing branches from programs:

1. Store the result of a comparison in a Boolean variable and use this value in normal integer arithmetic (i.e. rely on the `setcc` family of instructions available in Intel processors).
2. Move the data from one place to another conditionally (i.e. rely on the `cmovcc` family of instructions available in Intel processors).

In Intel's architecture optimization reference manual [7, Section 3.4.1], a clear guideline is given how these two types of optimizations should be used.

> Use the `setcc` and `cmov[cc]` instructions to eliminate unpredictable conditional branches where possible. Do not do this for predictable branches. Do not use these instructions to eliminate all unpredictable conditional branches (because using these instructions will incur execution overhead due to the requirement for executing both paths of a conditional branch). In addition, converting a conditional branch to `setcc` or `cmov[cc]` trades off control flow dependence for data dependence and restricts the capability of the out-of-order engine. When tuning, note that all Intel ... processors usually have very high branch prediction rates. Consistently mispredicted branches are generally rare. Use these instructions only if the increase in computation time is less than the expected cost of a mispredicted branch.

Complicated optimizations often mean complicated programs with many branches. As a result, it will be more difficult to remove the branches by hand.

Fortunately, an automatic way of eliminating all branches, except one, is known [5]. However, the performance of the obtained program is not necessarily good due to the high constant factor introduced in the running time. In our work we got the best results by starting from a simple program and reducing branches from it by hand.

Implicitly, we assumed that the elements being manipulated are small. For large elements, it may be necessary to execute each element comparison and element move in a loop. However, in order for the general $O(n \log_2 n)$ bound for sorting to be valid, element comparisons and element moves must be constant-time operations. If this was not the case, e.g. if we were sorting strings of characters, the comparison-based methods would not be optimal any more. On the other hand, if the elements were large but of fixed length, the loops involved in element comparisons and element moves could be unfolded and conditional branches could be avoided. Nonetheless, the increase in the number of element moves can become significant.

At this point we can reveal that we started this research by trying to make quicksort lean (as was done with heapsort and mergesort in [5]). However, we had big problems in forcing the compiler(s) to use conditional moves, and our hand-written assembly-language code was slower than the code produced by the compiler. So be warned; it is not always easy to eliminate unpredictable branches without a significant penalty in performance.

6 Afterword

We leave it for the reader to decide whether quicksort should still be considered the quickest sorting algorithm. It is definitely an interesting randomized algorithm. However, many of the implementation enhancements proposed for it seem to have little relevance in contemporary computers.

We are clearly in favour of mergesort instead of quicksort. If extra memory is allowed, mergesort is stable. Multi-way mergesort removes most of the problems associated with expensive element moves. The algorithm itself does not—even though our in-situ mergesort does—use random access. This would facilitate an extension to the interface of the C++ standard-library sort function: The input sequence should only support forward iterators, not random-access iterators.

In algorithm education at many universities a programming language is used that is far from a raw machine. Under such circumstances it gives little meaning to talk about the branch-prediction features of sorting programs. A cursory examination showed that in one of our test computers (Ares) the Python standard-library sort was a factor of 135 slower than the C++ standard-library sort when sorting million integers.

References

1. Biggar, P., Nash, N., Williams, K., Gregg, D.: An Experimental Study of Sorting and Branch Prediction. ACM J. Exp. Algorithmics 12, Article 1.8 (2008)
2. Brodal, G.S., Moruz, G.: Tradeoffs Between Branch Mispredictions and Comparisons for Sorting Algorithms. In: Dehne, F., López-Ortiz, A., Sack, J.-R. (eds.) WADS 2005. LNCS, vol. 3608, pp. 385–395. Springer, Heidelberg (2005)
3. Brodal, G.S., Moruz, G.: Skewed Binary Search Trees. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 708–719. Springer, Heidelberg (2006)
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. The MIT Press, Cambridge (2009)
5. Elmasry, A., Katajainen, J.: Lean Programs, Branch Mispredictions, and Sorting. In: Kranakis, E., Krizanc, D., Luccio, F. (eds.) FUN 2012. LNCS, vol. 7288, pp. 119–130. Springer, Heidelberg (2012)
6. Hoare, C.A.R.: Quicksort. Comput. J. 5(1), 10–16 (1962)
7. Intel Corporation: Intel® 64 and IA-32 Architectures Optimization Reference Manual, version 025, Santa Clara (1997–2011)
8. Kaligosi, K., Sanders, P.: How Branch Mispredictions Affect Quicksort. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 780–791. Springer, Heidelberg (2006)
9. Katajainen, J., Pasanen, T., Teuhola, J.: Practical in-place mergesort. Nordic J. Comput. 3(1), 27–40 (1996)
10. Katajainen, J., Träff, J.L.: A Meticulous Analysis of Mergesort Programs. In: Bongiovanni, G., Bovet, D.P., Di Battista, G. (eds.) CIAC 1997. LNCS, vol. 1203, pp. 217–228. Springer, Heidelberg (1997)
11. Kernighan, B.W., Ritchie, D.M.: The C Programming Language, 2nd edn. Prentice Hall, Englewood Cliffs (1988)
12. Mortensen, S.: Refining the Pure-C Cost Model. Master's Thesis, Department of Computer Science. University of Copenhagen, Copenhagen (2001)
13. Musser, D.R.: Introspective Sorting and Selection Algorithms. Software Pract. Exper. 27(8), 983–993 (1997)
14. Patterson, D.A., Hennessy, J.L.: Computer Organization and Design, The Hardware/Software Interface, 4th edn. Morgan Kaufmann Publishers, Burlington (2009)
15. Sanders, P., Winkel, S.: Super Scalar Sample Sort. In: Albers, S., Radzik, T. (eds.) ESA 2004. LNCS, vol. 3221, pp. 784–796. Springer, Heidelberg (2004)
16. Sedgewick, R.: The Analysis of Quicksort Programs. Acta Inform. 7(4), 327–355 (1977)

A Multiple Sliding Windows Approach to Speed Up String Matching Algorithms

Simone Faro[1] and Thierry Lecroq[2]

[1] Università di Catania, Viale A.Doria n.6, 95125 Catania, Italy
[2] Université de Rouen, LITIS EA 4108, 76821 Mont-Saint-Aignan Cedex, France
faro@dmi.unict.it, thierry.lecroq@univ-rouen.fr

Abstract. In this paper we present a general approach to string matching based on multiple sliding text-windows, and show how it can be applied to some among the most efficient algorithms for the problem based on nondeterministic automata and comparison of characters.

From our experimental results it turns out that the new multiple sliding windows approach leads to algorithms which obtain better results than the original ones when searching texts over relatively large alphabets. Best improvements are obtained especially for short patterns.

Keywords: string matching, bit parallelism, text processing, nondeterministic automata, design and analysis of algorithms, natural languages.

1 Introduction

Given a text t of length n and a pattern p of length m over some alphabet Σ of size σ, the *string matching problem* consists in finding *all* occurrences of the pattern p in t. This problem has been extensively studied in computer science because of its direct application to many areas. Moreover string matching algorithms are basic components in many software applications and play an important role in theoretical computer science by providing challenging problems.

Among the most efficient solutions the Boyer-Moore algorithm [1] is a comparison based algorithm which deserves a special mention since it has been particularly successful and has inspired much work. Also automata based solutions have been developed to design algorithms with efficient performance on average. This is done by using factor automata, data structures which identify all factors of a word. Among them the Extended Backward Oracle Matching algorithm [6] (EBOM for short) is one of the most efficient algorithms especially for long patterns. Another algorithm based on the bit-parallel simulation [17] of the nondeterministic factor automaton, and called Backward Nondeterministic Dawg Match algorithm [13] (BNDM), is very efficient for short patterns.

Most string matching algorithms are based on a general framework which works by scanning the text with the help of a substring of the text, called *window*, whose size is equal to m. An *attempt* of the algorithm consists in checking whenever the current window is an occurrence of the pattern. This check is generally carried out by comparing the pattern and the window character by character, or by performing transitions on some kind of automaton. After a whole

R. Klasing (Ed.): SEA 2012, LNCS 7276, pp. 172–183, 2012.

match of the pattern (or after a mismatch is detected) the window is shifted to the right by a certain number of positions, according to a shift strategy. This approach is usually called the *sliding window mechanism*.

At the beginning of the search the left end of the window is aligned with the left end of the text, then the sliding window mechanism is repeated until the right end of the window goes beyond the right end of the text.

In this paper we investigate the performance of a straightforward, yet efficient, approach which consists in sliding two or more windows of the same size along the text and in performing comparisons (or transitions) at the same time as long as possible, with the aim of speeding up the searching phase. We call this general method the *multiple sliding windows approach*.

The idea of sliding multiple windows along the text is not original. It was firstly introduced in [11] where the author presented a simple algorithm, called Two Sliding Windows, which divides the text into two parts of size $\lceil n/2 \rceil$ and searches for matches of p in t by sliding two windows of size m. The first window scans the left part of the text, proceeding from left to right, while the second window slides from right to left scanning the right part of the text. An additional test is performed for searching occurrences in the middle of the text.

More recently Cantone *et al.* proposed in [3] a similar approach for increasing the instruction level parallelism of string matching algorithms based on bit-parallelism. This technique, called *bit-(parallelism)*^{2}, is general enough to be applied to a lot of bit-parallel algorithms.

It includes two different approaches which run two copies of the same automata in parallel and process two adjacent windows simultaneously, sliding them from left to right. In both cases the two automata are encoded by using a single computer word. Thus, due to its representation, the approach turns out to be efficient only for searching very short patterns.

However the two approaches introduced by the bit-parallelism^2 technique are not general enough to be applied to all bit-parallel algorithms. For instance the structure of the BNDM algorithm does not allow it to process in parallel two adjacent (or partially overlapping) windows, since the starting position of the next window alignment depends on the the shift value performed by the leftmost one. Unfortunately such a shift has not a fixed value and it can be computed only at the end of the each attempt.

Moreover the approaches proposed in [3] can be applied only for improving the performance of automata based algorithms, thus they are not applicable for all algorithms based on comparison of characters.

The paper is organized as follows. In Section 2 we introduce some notation and the terminology used along the paper. In Section 3 we describe the new general multiple sliding windows approach and apply it to some of the most efficient algorithms based on the simulation of nondeterministic automata (Section 3.1) and on comparison of characters (Section 3.2). Finally we present an experimental evaluation in Section 4. Conclusions are drawn in Section 5.

2 Notations and Terminology

Throughout the paper we will make use of the following notations and terminology. A string p of length $m > 0$ is represented as a finite array $p[0\,..\,m-1]$ of characters from a finite alphabet Σ of size σ. Thus $p[i]$ will denote the $(i+1)$-st character of p, for $0 \leq i < m$, and $p[i\,..\,j]$ will denote the *factor* (or *substring*) of p contained between the $(i+1)$-st and the $(j+1)$-st characters of p, for $0 \leq i \leq j < m$. A factor of the form $p[0\,..\,i]$ is called a *prefix* of p and a factor of the form $p[i\,..\,m-1]$ is called a *suffix* of p for $0 \leqslant i < m$. We denote with $Fact(p)$ the set of all factors of a string p.

The factor automaton of a pattern p, also called the factor DAWG of p (for Directed Acyclic Word Graph), is a Deterministic Finite State Automaton (DFA) which recognizes all the factors of p. Formally its recognized language is defined as the set $\{u \in \Sigma^* \mid \text{ exists } v, w \in \Sigma^* \text{ such that } p = vuw\}$.

We also denote the reverse of the string p by $\bar{p}$, i.e. $\bar{p} = p[m-1]p[m-2]\cdots p[0]$.

Finally, we recall the notation of some bitwise infix operators on computer words, namely the bitwise `and` “&”, the bitwise `or` “|” and the `left shift` “$\ll$” operator (which shifts to the left its first argument by a number of bits equal to its second argument).

3 A General Multiple Sliding Windows Approach

In this section we describe a general multiple windows approach which can be used for improving the practical performances of a large class of string matching algorithms, including all comparison based algorithms. The general approach can be seen as a filtering method which consists in processing k different windows of the text at the same time, with $k \geq 2$. Then, specific occurrences of the pattern are tested only when candidate positions have been located.

Suppose p is a pattern of length m and t is a text of length n. Without loss in generality we can suppose that n can be divided by k, otherwise the rightmost $(n \mod k)$ characters of the text could be associated with the last window (as described below). Moreover we assume for simplicity that $m < n/k$ and that the value k is even. Under the above assumptions the new approach can be summarized as follows: if the algorithm searches for the pattern p in t using a text window of size m, then partition the text in $k/2$ partially overlapping substrings, $t_0, t_1, \ldots, t_{k/2-1}$, where t_i is the substring $t[2i\lceil n/k\rceil\,..\,2(i+1)n/k+m-2]$, for $i = 0, \ldots, (k-1)/2$, and $t_{k/2-1}$ (the last window) is set to $t[n-(2n/k)\,..\,n-1]$.

Then process simultaneously the k different text windows, $w_0, w_1, \ldots, w_{k-1}$, where we set $w_{2i} = t[s_{2i}-m+1\,..\,s_{2i}]$ (and call them *left windows*) and $w_{2i+1} = t[s_{2i+1}\,..\,s_{2i+1}+m-1]$ (and call them *right windows*), for $i = 0, \ldots, (k-2)/2$.

The couple of windows (w_{2i}, w_{2i+1}), for $i = 0, \ldots, (k-2)/2$, is used to process the substring of the text t_i. Specifically the window w_{2i} starts from position $s_{2i} = (2n/k)i + m - 1$ of t and slides from left to right, while window w_{2i+1} starts from position $s_{2i+1} = (2n/k)(i+1) - 1$ of t and slides from right to left (the window w_{k-1} starts from position $s_{k-1} = n - m$). For each couple

```
MultipleSlidingWinsowsMatcher($p, m, t, n, k$)
 1. for $i \leftarrow 0$ to $(k-2)/2$ do $s_{2i} \leftarrow (2n/k)i + m - 1$
 2. for $i \leftarrow 0$ to $(k-2)/2 - 1$ do $s_{2i+1} \leftarrow (2n/k)(i+1) - 1$
 3. $s_{k-1} \leftarrow n - m$
 4. while ($\exists\ i$ such that $s_{2i} \leqslant s_{2i+1} + m - 1$) do
 5.     if (checkSimultaneously($w_0, w_1, \ldots, w_{k-1}$)=true) then
 6.         for $i \leftarrow 0$ to $(k-2)/2$ do
 7.             if ($s_{2i} < s_{2i+1} - m + 1$) then
 8.                 Naively check if $p = t[s_{2i} \,..\, s_{2i} + m - 1]$
 9.             if ($s_{2i} \leqslant s_{2i+1} - m + 1$) then
10.                 Naively check if $p = t[s_{2i+1} - m + 1 \,..\, s_{2i+1}]$
11.     Performs shifts for $s_{2i}$ and $s_{2i+1}$
```

Fig. 1. A General Multiple Sliding Windows Matcher

of windows $(w_{2i},\ w_{2i+1})$ the sliding process ends when the window w_{2i} slides over the window w_{2i+1}, i.e. when $s_{2i} > s_{2i+1} + m - 1$. It is easy to prove that no candidate occurrence is left by the algorithm due to the $m-1$ overlapping characters between adjacent substrings t_i and t_{i+1}, for $i = 0, \ldots, k-2$.

Fig. 1 shows the pseudocode of a general multiple sliding windows matcher which processes k windows in parallel, while Fig. 2 presents a scheme of the search iteration of the multiple sliding windows matcher for $k = 1$, 2 and 4.

Procedure checkSimultaneously is used as a filtering method for locating candidate occurrences of the pattern in the k different text windows. Observe that, when n cannot be divided by k, the assignment of line 3, which set s_{k-1} to $n-m$, implies that the rightmost $(n \mod k)$ characters of the text are associated with the last window. Moreover it can be proved that the general matcher algorithm shown in Fig. 1 is correct if the value of checkSimultaneously($w_0, w_1, \ldots, w_{k-1}$) is true whenever $w_i = p$ for some i in the set $\{0, \ldots, k-1\}$.

If $s_{2i} = s_{2i+1} + m - 1$ (which implies $w_{2i} = w_{2i+1}$) and a candidate position is found, only the right window is checked for a real occurrence, in order to avoid to report a duplicate occurrence of the pattern. This is done in lines 7 and 9.

This general approach can be applied to all string matching algorithms, including comparison based and BMDN based algorithms. Moreover it can be noticed that the worst case time complexity of the original algorithm does not degrade with the application of the multiple sliding windows approach.

From a practical point of view, when computation of the filter can be done in parallel and the alphabet is large enough, the new approach leads to more efficient algorithms. As an additional feature, observe also that the two way sliding approach enables exploitation of the structure of the pattern in both directions, leading on average to larger shift advancements and improving further the performance of the algorithm. On the other hand, when the alphabet is small the performances of the original algorithm degrade by applying the new method, since the probability to find mixed candidate positions increases substantially.

In the following sections we present simple variants, based on the multiple sliding windows approach, of some among the most efficient algorithms based on comparison of characters and bit-parallelism.

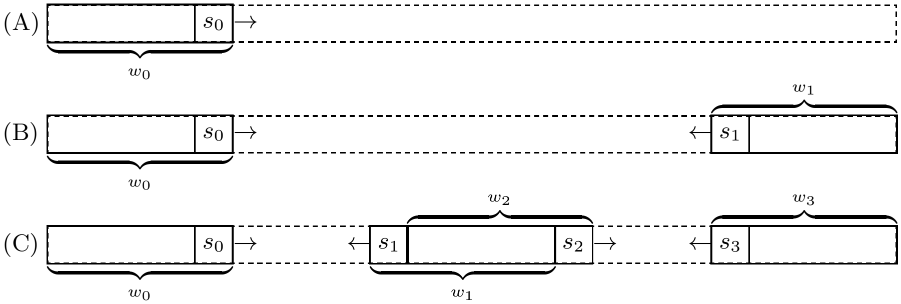


Fig. 2. A general scheme for the multiple sliding windows matcher. The scheme with (A) a single window, (B) two windows and (C) four windows

3.1 Multiple Windows Variants of Bit-Parallel Algorithms

In the context of string matching, bit-parallelism [17] is used for simulating the behavior of Nondeterministic Finite State Automata (NFA) and allows multiple transactions to perform in parallel with a constant number of operations.

In this section we show the application of the multiple sliding windows approach to a simplified version of the BNDM algorithm [13] which, among the several algorithms based on bit-parallelism, deserves a particular attention as it has inspired a lot of variants and is still considered one of the fastest algorithms. Among the various improvements of the BNDM algorithm we mention the Simplified BNDM algorithm improved with Horspool shift [9], with q-grams [4,15], and with lookahead characters [6,15]. The BNDM algorithm has been also modified in order to match long patterns [5] and binary strings [7].

Specifically, the BNDM algorithm simulates the suffix automaton of $\bar{p}$. Its bit-parallel representation of the suffix automaton uses an array, B_p, of σ bit-vectors, each of size m, where the i-th bit of $B_p[c]$ is set iff $p[i] = c$, for $c \in \Sigma$ and $0 \leqslant i < m$. The algorithm works by shifting a window of length m over the text. Specifically, for each text window alignment $t[s-m+1\,..\,s]$, it searches the pattern by scanning the current window backwards and updating the automaton configuration accordingly. In the simplified version of BNDM the bit vector D is initialized to $B_p[p[s]]$, i.e. the configuration of the automaton after the first transition. Then any subsequent transition on character c can be implemented as $D \leftarrow ((D \ll 1)\ \&\ B_p[c])$.

An attempt ends when either D becomes zero (i.e., when no further factors of p can be found) or the algorithm has performed m iterations (i.e., when a match has been found). The window is then shifted to the start position of the longest recognized proper factor.

Fig. 3 shows the code of the Simplified BNDM algorithm [14] (SBNDM for short) and the code of its multiple sliding windows variant with 4 windows.

In general the k-windows variant of a SBNDM algorithm simulates k different copies of two suffix automata, one for the reverse of the pattern and one for

```
SBNDM($p, m, t, n$)
for each $c \in \Sigma$ do $B_p(c) \leftarrow 0^m$
$F_p \leftarrow 0^{m-1}1$
for $i \leftarrow m-1$ downto 0 do
    $B_p(p[i]) \leftarrow B_p(p[i]) \mid F_p$
    $F_p \leftarrow F_p \ll 1$
$s \leftarrow m-1$
while ($s \leq n-1$) do
    $D \leftarrow B_p(t[s])$
    $j \leftarrow s-m+1$
    while ($D \neq 0$) do
        $s \leftarrow s-1$
        $D \leftarrow (D \ll 1)$ & $B_p(t[s])$
    if ($s < j$) then
        $s \leftarrow s+1$
        Output($s$)
    $s \leftarrow s+m$
```

```
SBNDM-W4($p, m, t, n$)
for each $c \in \Sigma$ do $B_p(c) \leftarrow B_{\bar{p}} \leftarrow 0^m$
$F_p \leftarrow 0^{m-1}1$
for $i \leftarrow m-1$ downto 0 do
    $B_p(p[i]) \leftarrow B_p(p[i]) \mid F_p$
    $B_{\bar{p}}(p[m-1-i]) \leftarrow B_{\bar{p}}(p[m-1-i]) \mid F_p$
    $F_p \leftarrow F_p \ll 1$
$s_0 \leftarrow m-1;\ s_1 \leftarrow n/2-1;\ s_2 \leftarrow n/2+m-1;\ s_3 \leftarrow n-m$
while ($s_0 \leq s_1+m-1$ and $s_2 \leq s_3+m-1$) do
    $D \leftarrow B_p(t[s_0]) \mid B_{\bar{p}}(t[s_1]) \mid B_p(t[s_2]) \mid B_{\bar{p}}(t[s_3])$
    $j \leftarrow s_0-m+1$
    while ($D \neq 0$) do
        $s_0 \leftarrow s_0-1;\ s_1 \leftarrow s_1+1;\ s_2 \leftarrow s_2-1;\ s_3 \leftarrow s_3+1$
        $D \leftarrow D \ll 1$
        $D \leftarrow D$ & $(B_p(t[s_0]) \mid B_{\bar{p}}(t[s_1]) \mid B_p(t[s_2]) \mid B_{\bar{p}}(t[s_3]))$
    if ($s_0 < j$) then
        $s_0 \leftarrow s_0+1;\ s_1 \leftarrow s_1-1;\ s_2 \leftarrow s_2+1;\ s_3 \leftarrow s_3-1$
        if ($s_0 < s_1+m-1$ and $p = t[s_0-m+1\,..\,s_0]$)
            then Output($s_0-m+1$)
        if ($s_0 \leq s_1+m-1$ and $p = t[s_1\,..\,s_1+m-1]$)
            then Output($s_1$)
        if ($s_2 < s_3+m-1$ and $p = t[s_2-m+1\,..\,s_2]$)
            then Output($s_2-m+1$)
        if ($s_2 \leq s_3+m-1$ and $p = t[s_3\,..\,s_3+m-1]$)
            then Output($s_3$)
    $s_0 \leftarrow s_0+m;\ s_1 \leftarrow s_1-m;\ s_2 \leftarrow s_2+m;\ s_3 \leftarrow s_3-m$
```

Fig. 3. (On the left) The SBNDM algorithm and (on the right) the multiple sliding windows variant of the SBNDM algorithm with 4 windows.

the pattern p itself. Their bit-parallel representations use two arrays $B_{\bar{p}}$ and B_p, respectively, of σ bit-vectors, each of size m.

Specifically the i-th bit of $B_p[c]$ is set iff $p[i] = c$, while the i-th bit of $B_{\bar{p}}[c]$ is set iff $p[m-i-1] = c$, for $c \in \Sigma$, $0 \leqslant i < m$. Then while searching, left windows use vector B_p while right windows use vector $B_{\bar{p}}$, to perform transitions.

A bit-vector D, with m bits, is used for simulating the mixed behavior of the k automata, when running in parallel. Specifically the i-th bit of D is set to 1 iff the i-th state of, at least, one of the k automata is active. For this purpose, at the beginning of each attempt, the bit-vector D is initialized with the assignment

$$D \leftarrow B_p[t[s_0]] \mid B_{\bar{p}}[t[s_1]] \mid B_p[t[s_2]] \mid \cdots \mid B_{\bar{p}}[t[s_{k-1}]].$$

Then the k windows are searched for candidate occurrences of the pattern by scanning the left windows backwards and the right windows forwards, and updating the mixed automaton configurations accordingly. The transitions are performed in parallel for all k windows by applying the following bitwise operations, for $i = 1$ to $m-1$

$$D \leftarrow D \ll 1$$
$$D \leftarrow D \;\&\; (B_p[t[s_0-i]] \mid B_{\bar{p}}[t[s_1+i]] \mid B_p[t[s_2-i]] \mid \cdots \mid B_{\bar{p}}[t[s_{k-1}+i]]).$$

An attempt ends when either D becomes zero (i.e., when no further candidate factors of p or $\bar{p}$ can be found) or the algorithm inspected m characters of the windows (i.e., when candidate matches has been found). In the last case a additional naive test is performed in order to check if any of the current windows corresponds in a whole match of the pattern.

At the end of each attempt the left and the right windows are shifted to the right and to the left, respectively, in order to be aligned with the start position of the longest recognized proper candidate factor.

3.2 Multiple Windows Variants of Comparison Based Algorithms

Most efficient comparison based algorithms are variants of the well-known Boyer-Moore algorithm [1]. It compares characters of the pattern and the current window of the text by scanning the pattern p from right to left and, at the end of the matching phase, computes the shift increment as the maximum value suggested by the *good suffix rule* and the *bad character rule*, provided that both of them are applicable (see [1] for more details).

Many variants of the Boyer-Moore algorithm have been proposed over the years, mostly focusing on the bad character rule. The first practical variant was introduced by Horspool in [10], which proposed a simple modification of the original rule and used it as a simple filtering method for locating candidate occurrences of the pattern.

Specifically the Horspool algorithm labels the current window of the text, $t[s-m+1\,..\,s]$, as a candidate occurrence if $p[m-1]=t[s]$. Failing this the shift advancement is computed in such a way that the rightmost character of the current window, $t[s]$, is aligned with its rightmost occurrence in $p[0\,..\,m-2]$, if present; otherwise the pattern is advanced just past the window. This corresponds to advance the shift by $hbc_p(t[s])$ positions where, for all $c \in \Sigma$, $hbc_p(c) = \min(\{0 < k < m \mid p[m-1-k] = c\} \cup \{m\})$.

Otherwise, when $t[s]=p[m-1]$ a candidate occurrence of the pattern has been located and a naive test is used for checking the whole occurrence (there is no need to test again if $p[m-1]=t[s]$). After an occurrence is found the pattern is advanced of an amount equal to $\min(\{0 < k < m \mid p[m-1-k] = t[s]\} \cup \{m\})$.

Fig. 4 (on the left) shows the code of the Horspool algorithm and (on the right) the code of its multiple sliding windows variant with 4 windows.

In general the k-windows variant of the Horspool algorithm searches for candidate positions by checking if the rightmost (leftmost) character of the pattern is equal to the rightmost (leftmost) character of any of the left (right) windows (line 11 in our example code).

A lot of variants of the Horspool algorithm have been proposed over the years. In this paper we propose the application of the new approach to the Fast-Search algorithm [2] and to the TVSBS algorithm [16], since they turn out to be among the most efficient in practice.

In particular the Fast-Search algorithm computes its shift increments by applying the Horspool bad-character rule when $p[m-1] \neq t[s]$, otherwise, if a candidate occurrence is found, it uses the good-suffix rule for shifting.

Differently the TVSBS algorithm discards the good suffix rule and uses the first and the last characters of the current window of the text, $(t[s-m+1],$

```
Horspool($p, m, t, n$)
for each $c \in \Sigma$ do $hbc_p(c) \leftarrow m$
for $i \leftarrow 0$ to $m - 2$ do
    $hbc_p(p[i]) \leftarrow m - 1 - i$
$s \leftarrow m - 1$
$c_l \leftarrow p[m - 1]$
while ($s \leq n - 1$) do
    if ($c_l = t[s]$) then
        if $p = t[s - m + 1 \,..\, s]$
            then Output($s - m + 1$)
    $s \leftarrow s + hbc_p(t[s])$
```

```
Horspool-W4($p, m, t, n$)
for each $c \in \Sigma$ do $hbc_p(c) \leftarrow hbc_{\bar{p}}(c) \leftarrow m$
for $i \leftarrow 0$ to $m - 2$ do
    $hbc_p(p[i]) \leftarrow m - 1 - i$
    $hbc_{\bar{p}}(p[m - 1 - i]) \leftarrow i$
$s_0 \leftarrow m - 1$, $s_1 \leftarrow n/2 - 1$, $s_2 \leftarrow n/2 + m - 1$, $s_3 \leftarrow n - m$
$c_f = p[0]$; $c_l \leftarrow p[m - 1]$
while ($s_0 \leq s_1 + m - 1$ or $s_2 \leq s_3 + m - 1$) do
    if ($c_l = t[s_0]$ or $c_f = t[s_1]$ or $c_l = t[s_2]$ or $c_f = t[s_3]$) then
        if ($s_0 < s_1 + m - 1$ and $p = t[s_0 - m + 1 \,..\, s_0]$)
            then Output($s_0 - m + 1$)
        if ($s_0 \leq s_1 + m - 1$ and $p = t[s_1 \,..\, s_1 + m - 1]$)
            then Output($s_1$)
        if ($s_2 < s_3 + m - 1$ and $p = t[s_2 - m + 1 \,..\, s_2]$)
            then Output($s_2 - m + 1$)
        if ($s_2 \leq s_3 + m - 1$ and $p = t[s_3 \,..\, s_3 + m - 1]$)
            then Output($s_3$)
    $s_0 \leftarrow s_0 + hbc_p(t[s_0])$, $s_1 \leftarrow s_1 - hbc_{\bar{p}}(t[s_1])$
    $s_2 \leftarrow s_2 + hbc_p(t[s_2])$, $s_3 \leftarrow s_3 - hbc_{\bar{p}}(t[s_3])$
```

Fig. 4. (On the left) The Horspool algorithm and (on the right) the multiple sliding windows variant of the Horspool algorithm with 4 windows.

$t[s]$), for locating a candidate occurrence of the pattern, while using the couple of characters ($t[s + 1]$, $t[s + 2]$) for computing the shift advancement.

4 Experimental Results

In this section we compare, in terms of running times and under various conditions, the performances of the multiple sliding windows variants of some among the most efficient string matching algorithms. Specifically we tested the k-windows variants of the following algorithms:

- Fast-Search (FS-W(k)), with $k \in \{1, 2, 4, 6, 8\}$
- TVSBS (TVSBS-W(k)), with $k \in \{1, 2, 4, 6, 8\}$
- Simplified BNDM (SBNDM-W(k)), with $k \in \{1, 2, 4, 6\}$
- Forward Simplified BNDM (FSBNDM-W(k)), with $k \in \{1, 2, 4, 6\}$

Observe that when k is equal to 1 we refer to the original algorithms.

All algorithms have been implemented in the C programming language and tested using the smart research tool [8]. The code used in our evaluation can be downloaded from the smart tool.

The experiments have been conducted on a MacBook Pro with a 2 GHz Intel Core i7 processor, a 4 GB 1333 MHz DDR3 memory and a 64 bit word size. In particular, all algorithms have been tested on seven 4MB random text buffers (randσ in smart) over alphabets of size σ with a uniform character distribution, where σ ranges in the set $\{16, 32, 64, 128\}$. For each input file, we have searched sets of 1000 patterns of fixed length m randomly extracted from the text, for m ranging from 2 to 512. Then, the mean of the running times has been reported.

Table 1 shows experimental results obtained by comparing multiple sliding windows variants of the above algorithms. Running times are expressed in hundredths of seconds and best results have been boldfaced and underlined.

Table 1. Running times of multiple sliding windows variants of the **(A)** Fast-Search, **(B)** TVSBS, **(C)** SBNDM and **(D)** FSBNDM algorithms

(A) m	2	4	8	16	32	64
$\sigma = 16$						
FS-W(1)	15.83	10.79	8.32	7.26	6.97	6.91
FS-W(2)	11.31	8.12	6.56	5.89	5.72	5.69
FS-W(4)	**9.41**	**7.11**	**5.93**	**5.48**	**5.37**	**5.36**
FS-W(6)	9.59	7.20	5.99	5.51	5.41	5.38
FS-W(8)	9.98	7.41	6.14	5.63	5.48	5.48
$\sigma = 32$						
FS-W(1)	15.02	10.18	7.79	6.69	6.44	6.39
FS-W(2)	10.21	7.45	6.06	5.45	5.39	5.41
FS-W(4)	**8.13**	**6.31**	5.46	5.08	5.07	5.14
FS-W(6)	8.17	6.34	**5.45**	**5.07**	**5.06**	**5.10**
FS-W(8)	8.22	6.35	5.47	5.12	5.10	5.13
$\sigma = 128$						
FS-W(1)	14.29	9.69	7.43	6.34	5.88	5.92
FS-W(2)	9.43	6.97	5.76	5.21	5.04	5.10
FS-W(4)	7.16	5.79	5.17	4.90	4.88	4.95
FS-W(6)	7.18	5.79	5.15	**4.87**	**4.84**	**4.90**
FS-W(8)	**7.04**	**5.73**	**5.12**	4.91	4.87	4.91

(B) m	2	4	8	16	32	64
$\sigma = 16$						
TVSBS-W(1)	12.89	10.32	8.30	7.01	6.28	5.99
TVSBS-W(2)	10.00	8.31	6.98	6.04	5.62	5.49
TVSBS-W(4)	**9.77**	**8.13**	6.84	5.94	5.51	5.37
TVSBS-W(6)	10.30	8.22	**6.81**	**5.92**	**5.48**	**5.33**
TVSBS-W(8)	12.31	8.76	7.02	6.08	5.64	5.52
$\sigma = 32$						
TVSBS-W(1)	12.18	9.90	8.10	6.93	6.21	5.94
TVSBS-W(2)	9.64	8.05	6.83	5.95	5.54	5.42
TVSBS-W(4)	**8.65**	**7.38**	6.36	5.68	5.36	5.29
TVSBS-W(6)	8.80	**7.38**	**6.32**	**5.65**	**5.33**	**5.24**
TVSBS-W(8)	10.26	8.00	6.77	5.92	5.53	5.38
$\sigma = 128$						
TVSBS-W(1)	13.88	10.99	8.76	7.29	6.46	6.07
TVSBS-W(2)	10.15	8.53	7.18	6.27	5.77	5.54
TVSBS-W(4)	8.51	7.38	6.40	**5.78**	**5.53**	**5.44**
TVSBS-W(6)	**8.26**	**7.19**	**6.28**	5.84	5.63	**5.44**
TVSBS-W(8)	9.44	8.07	7.04	6.38	6.02	5.69

(C) m	2	4	8	16	32	64
$\sigma = 16$						
SBNDM-W(1)	12.9	9.73	8.20	6.59	**5.61**	**5.61**
SBNDM-W(2)	**11.2**	8.57	6.66	**5.52**	6.85	6.90
SBNDM-W(4)	11.7	**7.97**	**6.41**	5.70	9.21	9.36
SBNDM-W(6)	12.5	8.75	6.99	5.80	16.8	17.1
$\sigma = 32$						
SBNDM-W(1)	11.4	8.56	7.22	6.62	**5.78**	**5.79**
SBNDM-W(2)	9.03	7.18	6.31	5.46	5.97	5.96
SBNDM-W(4)	**8.48**	**6.88**	**5.84**	**5.22**	6.08	6.08
SBNDM-W(6)	9.41	7.22	5.90	5.31	7.53	7.58
$\sigma = 128$						
SBNDM-W(1)	10.6	7.86	6.57	5.97	5.89	5.85
SBNDM-W(2)	7.76	6.23	5.49	5.19	5.20	5.18
SBNDM-W(4)	**6.54**	**5.61**	**5.22**	**5.06**	**5.11**	**5.09**
SBNDM-W(6)	7.08	5.76	5.32	5.11	5.26	5.28

(D) m	2	4	8	16	32	64
$\sigma = 16$						
FSBNDM-W(1)	11.5	8.26	6.72	5.95	5.53	5.54
FSBNDM-W(2)	**10.4**	**7.34**	6.05	5.71	5.28	5.30
FSBNDM-W(4)	10.9	7.42	**6.01**	**5.43**	**5.08**	**5.11**
FSBNDM-W(6)	12.6	8.31	6.39	5.63	5.27	5.27
$\sigma = 32$						
FSBNDM-W(1)	9.75	7.36	6.22	5.62	5.35	5.34
FSBNDM-W(2)	8.91	6.69	5.60	5.40	5.19	5.20
FSBNDM-W(4)	**8.44**	**6.28**	**5.40**	**5.08**	**5.05**	**5.04**
FSBNDM-W(6)	9.42	6.78	5.62	5.21	5.10	5.10
$\sigma = 128$						
FSBNDM-W(1)	8.84	6.72	5.86	5.45	5.18	5.18
FSBNDM-W(2)	7.94	6.18	5.36	5.00	4.98	4.97
FSBNDM-W(4)	**7.32**	**5.71**	**5.12**	**4.90**	**4.89**	**4.88**
FSBNDM-W(6)	7.45	5.92	5.19	**4.90**	**4.89**	**4.88**

Best running times are obtained when a good compromise between the values of k, m and σ is reached. In the case of comparison based algorithms it turns out that the best results are obtained for $k = 4$ and $k = 6$ while, for large alphabets and short patterns, the Fast-Search algorithm performs better with $k = 8$. In this latter case the variant is up to 50% faster than the original algorithm.

In the case of bit parallel algorithm we obtain in most cases best results for multiple windows variants with $k = 4$. For small alphabets and long patterns the better results are obtained with $k = 2$. In the case of the SBNDM algorithm, when the value of m is small enough and the alphabet is large, we obtain results up to 40% better than that obtained by the original algorithm. However the performance of the new variants degrades when the length of the pattern increases, especially for small alphabets. In all cases best improvements are obtained in the case of short patterns.

Table 2. Experimental results on random text buffers over alphabets of size 16, 32, 64 and 128, respectively. Best results are boldfaced and underlined

$(\sigma = 16)$ / m	2	4	8	16	32	64	128	256	512
EBOM	**9.14**	**6.77**	6.07	5.75	5.61	5.58	5.54	5.53	5.49
HASH(q)	17.4$^{(1)}$	11.52$^{(1)}$	8.46$^{(2)}$	6.72$^{(2)}$	5.81$^{(5)}$	5.41$^{(5)}$	5.32$^{(5)}$	5.30$^{(4)}$	5.26$^{(5)}$
FSBNDM(q, f)	10.6$^{(2,1)}$	7.6$^{(2,0)}$	6.29$^{(3,1)}$	5.63$^{(3,1)}$	5.37$^{(3,1)}$	5.40$^{(3,1)}$	5.38$^{(3,1)}$	5.39$^{(3,1)}$	5.39$^{(3,1)}$
QF(q, s)	-	7.5$^{(2,4)}$	6.16$^{(2,4)}$	5.63$^{(3,4)}$	5.37$^{(3,4)}$	5.24$^{(3,4)}$	5.18$^{(3,4)}$	**4.97**$^{(3,4)}$	**4.70**$^{(3,4)}$
FSBNDM-W(k)	10.4$^{(2)}$	7.34$^{(2)}$	6.01$^{(4)}$	**5.43**$^{(4)}$	**5.08**$^{(4)}$	**5.11**$^{(4)}$	**5.11**$^{(4)}$	5.10$^{(4)}$	5.11$^{(4)}$
SBNDM-W(k)	11.2$^{(2)}$	7.97$^{(4)}$	6.41$^{(4)}$	5.52$^{(2)}$	5.38$^{(1)}$	5.37$^{(1)}$	5.37$^{(1)}$	5.37$^{(1)}$	5.38$^{(1)}$
FS-W(k)	9.41$^{(4)}$	7.11$^{(4)}$	**5.93**$^{(4)}$	5.48$^{(4)}$	5.37$^{(4)}$	5.36$^{(4)}$	5.36$^{(4)}$	5.33$^{(4)}$	5.31$^{(4)}$
TVSBS-W(k)	9.77$^{(4)}$	8.13$^{(4)}$	6.81$^{(6)}$	5.92$^{(6)}$	5.48$^{(6)}$	5.33$^{(6)}$	5.19$^{(6)}$	5.06$^{(6)}$	4.94$^{(6)}$

$(\sigma = 32)$ / m	2	4	8	16	32	64	128	256	512
EBOM	9.05	6.61	5.90	5.64	5.51	5.48	5.43	5.32	5.37
HASH(q)	15.88$^{(1)}$	10.79$^{(1)}$	7.95$^{(1)}$	6.54$^{(2)}$	5.80$^{(3)}$	5.45$^{(5)}$	5.36$^{(5)}$	5.30$^{(4)}$	5.26$^{(4)}$
FSBNDM(q, f)	9.28$^{(2,1)}$	7.09$^{(2,1)}$	6.06$^{(2,0)}$	5.54$^{(2,0)}$	5.32$^{(2,0)}$	5.33$^{(2,0)}$	5.33$^{(2,0)}$	5.33$^{(2,0)}$	5.32$^{(2,0)}$
QF(q, s)	-	7.21$^{(2,6)}$	5.97$^{(2,6)}$	5.50$^{(2,6)}$	5.31$^{(2,6)}$	5.22$^{(2,6)}$	5.12$^{(2,6)}$	4.92$^{(2,6)}$	4.71$^{(4,3)}$
FS-W(k)	**8.13**$^{(4)}$	6.31$^{(4)}$	5.45$^{(6)}$	**5.07**$^{(6)}$	5.06$^{(6)}$	5.10$^{(6)}$	5.06$^{(6)}$	5.06$^{(6)}$	5.05$^{(6)}$
FSBNDM-W(k)	8.44$^{(4)}$	**6.28**$^{(2)}$	**5.40**$^{(4)}$	5.08$^{(4)}$	**5.05**$^{(4)}$	**5.04**$^{(4)}$	**5.04**$^{(4)}$	5.03$^{(4)}$	5.04$^{(4)}$
SBNDM-W(k)	9.03$^{(2)}$	6.88$^{(4)}$	5.84$^{(4)}$	5.22$^{(4)}$	5.57$^{(1)}$	5.55$^{(1)}$	5.57$^{(1)}$	5.57$^{(1)}$	5.56$^{(1)}$
TVSBS-W(k)	8.65$^{(4)}$	7.38$^{(4)}$	6.32$^{(6)}$	5.65$^{(6)}$	5.33$^{(6)}$	5.24$^{(6)}$	5.07$^{(6)}$	**4.84**$^{(6)}$	**4.66**$^{(6)}$

$(\sigma = 64)$ / m	2	4	8	16	32	64	128	256	512
EBOM	9.75	6.81	5.95	5.65	5.51	5.47	5.25	5.29	5.29
HASH(q)	15.18$^{(1)}$	10.11$^{(1)}$	7.60$^{(1)}$	6.46$^{(2)}$	5.70$^{(2)}$	5.44$^{(5)}$	5.35$^{(4)}$	5.31$^{(4)}$	5.27$^{(4)}$
QF(q, s)	-	7.13$^{(2,6)}$	5.92$^{(2,6)}$	5.47$^{(2,6)}$	5.28$^{(2,6)}$	5.03$^{(2,6)}$	5.11$^{(2,6)}$	4.85$^{(2,6)}$	4.61$^{(2,6)}$
FSBNDM(q, f)	8.59$^{(2,1)}$	6.80$^{(2,1)}$	5.89$^{(2,1)}$	5.44$^{(2,1)}$	5.25$^{(2,1)}$	5.27$^{(2,1)}$	5.10$^{(2,1)}$	5.27$^{(2,0)}$	5.28$^{(2,0)}$
FS-W(k)	7.37$^{(8)}$	5.91$^{(8)}$	5.20$^{(8)}$	**4.92**$^{(6)}$	**4.90**$^{(6)}$	4.94$^{(8)}$	4.97$^{(6)}$	4.92$^{(6)}$	4.91$^{(6)}$
FSBNDM-W(k)	8.02$^{(6)}$	**5.87**$^{(4)}$	**5.17**$^{(4)}$	**4.92**$^{(4)}$	4.92$^{(4)}$	**4.92**$^{(4)}$	**4.95**$^{(4)}$	4.93$^{(4)}$	4.92$^{(4)}$
SBNDM-W(k)	**7.11**$^{(4)}$	6.01$^{(4)}$	5.50$^{(4)}$	5.09$^{(6)}$	5.33$^{(2)}$	5.32$^{(2)}$	5.34$^{(2)}$	5.32$^{(2)}$	5.32$^{(2)}$
TVSBS-W(k)	8.19$^{(6)}$	7.07$^{(6)}$	6.14$^{(6)}$	5.57$^{(6)}$	5.28$^{(6)}$	5.12$^{(6)}$	5.03$^{(6)}$	**4.75**$^{(6)}$	**4.54**$^{(6)}$

$(\sigma = 128)$ / m	2	4	8	16	32	64	128	256	512
EBOM	9.99	6.93	6.02	5.74	5.57	5.52	5.42	5.28	5.30
HASH(q)	14.83$^{(1)}$	9.88$^{(1)}$	7.42$^{(1)}$	6.27$^{(1)}$	5.63$^{(2)}$	5.43$^{(5)}$	5.33$^{(3)}$	5.28$^{(4)}$	5.24$^{(4)}$
QF(q, s)	-	7.15$^{(2,6)}$	5.95$^{(2,6)}$	5.50$^{(2,6)}$	5.29$^{(2,6)}$	5.19$^{(2,6)}$	5.10$^{(2,6)}$	4.87$^{(2,6)}$	4.61$^{(2,6)}$
FSBNDM(q, f)	8.28$^{(2,1)}$	6.65$^{(2,1)}$	5.83$^{(2,1)}$	5.43$^{(2,1)}$	5.25$^{(2,1)}$	5.25$^{(2,1)}$	5.25$^{(2,1)}$	5.26$^{(2,0)}$	5.25$^{(2,1)}$
FS-W(k)	7.04$^{(8)}$	5.73$^{(8)}$	**5.12**$^{(8)}$	**4.90**$^{(4)}$	**4.84**$^{(6)}$	4.90$^{(6)}$	**4.81**$^{(8)}$	**4.66**$^{(8)}$	4.57$^{(8)}$
FSBNDM-W(k)	7.32$^{(4)}$	5.71$^{(4)}$	**5.12**$^{(4)}$	**4.90**$^{(4)}$	4.89$^{(4)}$	**4.88**$^{(4)}$	4.88$^{(4)}$	4.89$^{(4)}$	4.89$^{(6)}$
SBNDM-W(k)	**6.54**$^{(4)}$	**5.61**$^{(4)}$	5.22$^{(4)}$	5.06$^{(4)}$	5.11$^{(4)}$	5.09$^{(4)}$	5.09$^{(4)}$	5.09$^{(4)}$	5.10$^{(4)}$
TVSBS-W(k)	8.26$^{(6)}$	7.19$^{(6)}$	6.28$^{(6)}$	5.78$^{(4)}$	5.53$^{(4)}$	5.44$^{(4)}$	4.99$^{(6)}$	4.70$^{(6)}$	**4.53**$^{(6)}$

We performed an additional comparison where we tested our proposed variants against the following efficient algorithms, which turn out to be very efficient in practical cases:

- the HASHq algorithm [12] (HASH(q)), with $q \in \{1, \ldots, 8\}$;
- the EBOM algorithm [6] (EBOM);
- the Forward SBNDM algorithm [6] enhanced with q-grams and f forward characters [15] (FSBNDM(q, f)), with $q \in \{2, \ldots, 8\}$ and $f \in \{0, \ldots, 6\}$;
- the Q-gram Filtering algorithm [5] (QF(q, s)), with $q \in \{2, \ldots, 6\}$, $s \in \{2, \ldots, 8\}$.

Table 3. Experimental results on a protein sequence (on the top) and on a natural language text buffer (on the bottom). Best results are boldfaced and underlined.

protein / m	2	4	8	16	32	64	128	256	512
EBOM	**9.21**	**6.83**	6.05	5.76	5.61	5.57	5.54	5.51	5.47
HASH(q)	17.29$^{(1)}$	11.45$^{(1)}$	8.35$^{(2)}$	6.68$^{(2)}$	5.90$^{(3)}$	5.49$^{(5)}$	5.41$^{(5)}$	5.36$^{(4)}$	5.31$^{(5)}$
QF(q, s)	-	7.65$^{(2,6)}$	6.21$^{(2,6)}$	5.64$^{(3,4)}$	5.39$^{(3,4)}$	5.26$^{(3,4)}$	5.19$^{(4,3)}$	**4.9**$^{(4,3)}$	**4.73**$^{(4,3)}$
FSBNDM(q, f)	10.5$^{(2,1)}$	7.61$^{(2,0)}$	6.32$^{(3,1)}$	5.63$^{(3,1)}$	5.38$^{(3,1)}$	5.39$^{(3,1)}$	5.37$^{(3,1)}$	5.37$^{(3,1)}$	5.37$^{(3,1)}$
FS-W(k)	9.41$^{(4)}$	7.12$^{(4)}$	**5.92**$^{(4)}$	5.47$^{(4)}$	5.35$^{(4)}$	5.30$^{(6)}$	5.24$^{(4)}$	5.21$^{(6)}$	5.24$^{(4)}$
FSBNDM-W(k)	10.4$^{(2)}$	7.35$^{(4)}$	5.96$^{(4)}$	**5.44**$^{(4)}$	**5.13**$^{(4)}$	**5.15**$^{(4)}$	**5.16**$^{(4)}$	5.13$^{(4)}$	5.12$^{(4)}$
SBNDM-W(k)	11.1$^{(2)}$	7.93$^{(4)}$	6.39$^{(4)}$	5.61$^{(2)}$	5.64$^{(1)}$	5.54$^{(1)}$	5.54$^{(1)}$	5.53$^{(1)}$	5.53$^{(1)}$
TVSBS-W(k)	9.85$^{(4)}$	8.13$^{(4)}$	6.79$^{(6)}$	5.94$^{(6)}$	5.50$^{(6)}$	5.36$^{(6)}$	5.21$^{(4)}$	5.02$^{(6)}$	4.89$^{(6)}$

natural lang./m	2	4	8	16	32	64	128	256	512
EBOM	**9.69**	**7.35**	6.63	6.30	6.00	5.87	5.71	5.62	5.62
HASH(q)	17.72$^{(1)}$	12.02$^{(1)}$	8.38$^{(2)}$	6.72$^{(2)}$	5.92$^{(3)}$	5.55$^{(5)}$	5.46$^{(4)}$	5.38$^{(4)}$	5.32$^{(4)}$
QF(q, s)	-	8.52$^{(2,6)}$	6.66$^{(3,4)}$	5.77$^{(4,3)}$	5.41$^{(4,3)}$	5.24$^{(6,2)}$	5.12$^{(4,3)}$	**4.97**$^{(4,3)}$	**4.7**$^{(6,2)}$
FSBNDM(q, f)	11.4$^{(2,1)}$	8.37$^{(2,0)}$	6.73$^{(3,1)}$	5.98$^{(4,1)}$	5.46$^{(4,1)}$	5.46$^{(4,1)}$	5.45$^{(4,1)}$	5.46$^{(4,1)}$	5.47$^{(4,1)}$
FS-W(k)	10.3$^{(4)}$	7.80$^{(4)}$	**6.41**$^{(4)}$	5.80$^{(4)}$	5.38$^{(4)}$	5.23$^{(4)}$	**5.08**$^{(4)}$	5.00$^{(6)}$	4.89$^{(6)}$
FSBNDM-W(k)	11.7$^{(2)}$	8.17$^{(2)}$	6.49$^{(4)}$	**5.72**$^{(4)}$	**5.21**$^{(4)}$	**5.20**$^{(4)}$	5.23$^{(4)}$	5.22$^{(4)}$	5.23$^{(4)}$
SBNDM-W(k)	12.2$^{(4)}$	8.90$^{(4)}$	7.10$^{(4)}$	6.06$^{(4)}$	6.07$^{(1)}$	6.07$^{(1)}$	6.06$^{(1)}$	6.07$^{(1)}$	6.08$^{(1)}$
TVSBS-W(k)	10.6$^{(4)}$	8.66$^{(4)}$	7.04$^{(6)}$	6.02$^{(6)}$	5.52$^{(6)}$	5.36$^{(6)}$	5.18$^{(4)}$	5.02$^{(6)}$	4.82$^{(6)}$

Table 2 shows the experimental results obtained on random text buffers over alphabets of size 16, 32, 64 and 128, while Table 3 shows experimental results performed on two real data problems and in particular a protein sequence and on a natural language text buffer. All text buffers are available on smart.

For each different algorithm we have reported only the best result obtained by its variants. Parameters of the variants which obtained the best running times are reported as apices. The reader can find a more detailed discussion about the performances of the different variants in the original papers [12,15,5].

From experimental results it tuns out that the new variants obtain very good results, especially in the cases of large alphabets ($\sigma \geq 32$). Specifically, best results are obtained by the FS-W(4) algorithm, in the case of small patterns, and by the FSBNDM-W(2) algorithm, in the case of moderately length patterns ($16 \leq m \leq 128$). Moreover for very long patterns ($m \geq 256$) multiple windows variants of the TVSBS algorithm turn out to be the fastest, even in the case of moderate large alphabets. In the case of small alphabets the EBOM and the QF algorithms still obtain the best results for short and long patterns, respectively.

5 Conclusions

We presented a general multiple sliding windows approach which could be applied to a wide family of comparison based and automata based algorithms. The new approach turns out to be simple to implement and leads to very fast algorithms in practical cases, especially in the case of large alphabets and natural language texts, as shown in our experimental results. It would be interesting to investigate further the application of this approach to other more effective solutions for the string matching problem, or to apply it to other related problems.

References

1. Boyer, R.S., Moore, J.S.: A fast string searching algorithm. Commun. ACM 20(10), 762–772 (1977)
2. Cantone, D., Faro, S.: Fast-Search Algorithms: New Efficient Variants of the Boyer-Moore Pattern-Matching Algorithm. J. Autom. Lang. Comb. 10(5/6), 589–608 (2005)
3. Cantone, D., Faro, S., Giaquinta, E.: Bit-(Parallelism)^2: Getting to the next level of parallelism. In: Boldi, P. (ed.) FUN 2010. LNCS, vol. 6099, pp. 166–177. Springer, Heidelberg (2010)
4. Durian, B., Holub, J., Peltola, H., Tarhio, J.: Tuning BNDM with q-grams. In: Finocchi, I., Hershberger, J. (eds.) Workshop on Algorithm Engineering and Experiments, pp. 29–37 (2009)
5. Ďurian, B., Peltola, H., Salmela, L., Tarhio, J.: Bit-Parallel Search Algorithms for Long Patterns. In: Festa, P. (ed.) SEA 2010. LNCS, vol. 6049, pp. 129–140. Springer, Heidelberg (2010)
6. Faro, S., Lecroq, T.: Efficient variants of the Backward-Oracle-Matching algorithm. In: Holub, J., Žďárek, J. (eds.) Prague Stringology Conference 2008, pp. 146–160. Czech Technical University in Prague, Czech Republic (2008)
7. Faro, S., Lecroq, T.: An Efficient Matching Algorithm for Encoded DNA Sequences and Binary Strings. In: Kucherov, G., Ukkonen, E. (eds.) CPM 2009 Lille. LNCS, vol. 5577, pp. 106–115. Springer, Heidelberg (2009)
8. Faro, S., Lecroq, T.: Smart: a string matching algorithm research tool. University of Catania and University of Rouen (2011), http://www.dmi.unict.it/~faro/smart/
9. Holub, J., Durian, B.: Talk: Fast variants of bit parallel approach to suffix automata. In: The Second Haifa Annual International Stringology Research Workshop of the Israeli Science Foundation (2005), http://www.cri.haifa.ac.il/events/2005/string/presentations/Holub.pdf
10. Horspool, R.N.: Practical fast searching in strings. Softw. Pract. Exp. 10(6), 501–506 (1980)
11. Hudaib, A., Al-Khalid, R., Suleiman, D., Itriq, M., Al-Anani, A.: A fast pattern matching algorithm with two sliding windows (TSW). J. Comput. Sci. 4(5), 393–401 (2008)
12. Lecroq, T.: Fast exact string matching algorithms. Inf. Process. Lett. 102(6), 229–235 (2007)
13. Navarro, G., Raffinot, M.: A Bit-parallel Approach to Suffix Automata: Fast Extended String Matching. In: Farach-Colton, M. (ed.) CPM 1998. LNCS, vol. 1448, pp. 14–33. Springer, Heidelberg (1998)
14. Peltola, H., Tarhio, J.: Alternative Algorithms for Bit-parallel String Matching. In: Nascimento, M.A., de Moura, E.S., Oliveira, A.L. (eds.) SPIRE 2003. LNCS, vol. 2857, pp. 80–93. Springer, Heidelberg (2003)
15. Peltola, H., Tarhio, J.: Variations of forward-SBNDM. In: Holub, J., Žďárek, J. (eds.) Prague Stringology Conference 2011, pp. 3–14. Czech Technical University in Prague, Czech Republic (2011)
16. Thathoo, R., Virmani, A., Lakshmi, S.S., Balakrishnan, N., Sekar, K.: TVSBS: A fast exact pattern matching algorithm for biological sequences. J. Indian Acad. Sci., Current Sci. 91(1), 47–53 (2006)
17. Wu, S., Manber, U.: Fast text searching: allowing errors. Commun. ACM 35, 83–91 (1992)

Algorithms for Subnetwork Mining in Heterogeneous Networks

Guillaume Fertin, Hafedh Mohamed Babou, and Irena Rusu

LINA, UMR 6241, Université de Nantes, France
{Guillaume.Fertin,Hafedh.Mohamed-Babou,Irena.Rusu}@univ-nantes.fr

Abstract. Subnetwork mining is an essential issue in network analysis, with specific applications *e.g.* in biological networks, social networks, information networks and communication networks. Recent applications require the extraction of subnetworks (or *patterns*) involving several relations between the objects of interest, each such relation being given as a network. The complexity of a particular mining problem increases with the different nature of the networks, their number, their size, the topology of the requested pattern, the criteria to optimize. In this emerging field, our paper deals with two networks respectively represented as a directed acyclic graph and an undirected graph, on the same vertex set. The sought pattern is a longest path in the directed graph whose vertex set induces a connected subgraph in the undirected graph. This problem has immediate applications in biological networks, and predictable applications in social, information and communication networks. We study the complexity of the problem, thus identifying polynomial, NP-complete and APX-hard cases. In order to solve the difficult cases, we propose a heuristic and a branch-and-bound algorithm. We further perform experimental evaluation on both simulated and real data.

1 Introduction

The use of communication, social and telecommunication networks has dramatically increased recently, resulting in new prominent applications of network analysis. In addition to these real-world applications, network representations of new types of data - and particularly biological data - highlight the drastic need for a new, multi-dimensional, type of (sub)network mining in which several networks, representing several relations between the same objects, are simultaneously investigated for the extraction of a multi-dimensional pattern [5,13,15].

The study of multi-dimensional mining started several years ago, but it mainly concerns homogeneous representations of data: directed graph alignment [4], undirected graph alignment [6], relational data mining [8], social networks mining [13] are several examples. Recently, such approaches found applications in computational biology [12,14,16], but also showed their limits, due to the multiple types of biological networks that are used to describe different views of the same biological process. In such applications, a process is often represented as a path in a directed network (e.g., a metabolic network), and as a connected graph

R. Klasing (Ed.): SEA 2012, LNCS 7276, pp. 184–194, 2012.

in an undirected network (e.g., a protein-protein interaction network); the link between the two networks is then ensured by the components involved in the process, that are represented as vertices in each network. Identifying a particular biological process then requires to identify parts of the two networks (directed and undirected) that have the suited topological patterns and the same vertex set. The need for such applications to replace either manual or case-by-case studies is nowadays fundamental [3,7,15,17].

In this paper, we approach multi-dimensional mining within two heterogeneous networks, driven by the previously cited applications in biological networks. The paper is organized as follows. Section 2 presents the problem. In Section 3, we show that the problem is APX-hard in the general case, NP-complete even in restricted cases, and exhibit classes of instances for which the problem is polynomial. In Section 4, we propose a heuristic and a branch-and-bound algorithm, that we evaluate in Section 5 both on simulated and real (biological) data. Section 6 is the conclusion. Note that due to space constraints, some proofs and illustrations are omitted.

2 The Problem

All along the paper, D will denote a directed graph and G an undirected graph, built on the same set of vertices V. In general, given a graph H, $V(H)$ is its vertex set. If H is undirected (resp. directed) then its edge set (resp. its arc set) is denoted $E(H)$ (resp. $A(H)$). Given a set $S \subseteq V(H)$, we denote $H[S]$ the subgraph of H induced by S.

A (D, G)-*consistent path* is a (directed) path P in D such that $G[V(P)]$ is connected. The Skew SubGraph Mining problem (abbreviated SkewGraM) is formulated as follows:

SkewGraM
Instance : A DAG D and an undirected graph G.
Requires : Find a longest (D, G)-consistent path.

Several more general variants of the problem (*e.g.* when D has circuits, or when D and G have different, but related, vertex sets) may be useful in practice, but they should be reduced to this simpler variant, for which we provide effective solutions. See [2] for details.

3 The Complexity of SkewGraM

We study the complexity of SkewGraM considering different topological constraints on graphs D and G. In Table 1, D^* is the *underlying graph* of D, obtained by removing the arc orientations. A *star* (resp. *bi-star*) is a tree whose number of vertices with degree 2 or more is exactly one (resp. two), whereas an *outerplanar graph* is a graph admitting a planar embedding with all vertices on a circle, all edges inside the circle and such that edges do not cross each other. Recall that

Table 1. Complexity of the SkewGraM problem

G \ D^*	Tree	Outerplanar	General graph
Chordless path or cycle, (bi-)star	P [Lem. 1]		
Tree with diameter 4	P [Lem. 1]	NPC [Thm. 1]	NPC [Thm. 1]
General graph	P [Lem. 1]	NPC [Thm. 1]	NPC [Thm. 1] APX-hard [Thm. 2]

the *diameter* of a graph is the maximum length of a shortest path between any two of its vertices.

According to Table 1, SkewGraM is polynomially solvable as soon as D^* is a tree, but it becomes NP-complete even for relatively simple graph structures, *e.g.* when G is a tree with diameter 4 and D^* is an outerplanar graph. In the most difficult cases, the problem is APX-hard.

The following lemma gathers together the polynomial-time solvable cases we identified:

Lemma 1. SkewGraM *is polynomial-time solvable when at least one of the following conditions holds: a) D^* is a tree. b) G is a chordless path or cycle. c) G is a (bi-)star.*

The theorem below shows the NP-completeness of the problem in a particular configuration.

Theorem 1. SkewGraM *(in its decision version) is* NP*-complete, even when D^* is an outerplanar graph and G is a tree with diameter 4.*

Proof. The problem is clearly in NP. We propose a reduction from MAX 2Sat[11]. Let $\mathcal{C} = \{C_1, \ldots, C_p\}$ be a collection of p clauses with two literals each, over the variable set $\mathcal{X}_n = \{x_1, \ldots x_n\}$. Let D be built on $2p+2n+2$ levels, called *optional* (marked with a star) or *compulsory* (not marked):

- level 0 : a vertex s;
- level* $2i-1$, $1 \leq i \leq p$: two vertices $v_{i,1}$ and $v_{i,2}$ corresponding to the literals of clause C_i;
- level $2i$, $1 \leq i \leq p$: a vertex c_i corresponding to the clause C_i;
- level $2p+1$: two vertices $v_{p+1,1}$ and $v_{p+1,2}$ corresponding, respectively, to variables x_n and $\overline{x_n}$;
- level $2p+2$: a vertex c_{p+1};
- level $2p+2+2i-1$, $1 \leq i \leq n$: two vertices a_i and b_i;
- level $2p+2+2i$, $1 \leq i < n$: a vertex A_i.

Then add (a) all possible arcs between any two consecutive levels, (b) the arc sc_1 and (c) the arcs $c_ic_{i+1}, 1 \leq i < p$. It is clear that D is a DAG. To see that D^* is an outerplanar graph, it is sufficient to draw the vertices on a circle according to the order $s, v_{1,1}, c_1, v_{2,1}, c_2, \ldots, v_{p+1,1},\ c_{p+1},\ a_1,\ A_1, \ldots, a_{n-1}, A_{n-1}, a_n, b_n, \ldots, b_1, v_{p+1,2}, \ldots, v_{1,2}$.

Graph G is a tree with root s. There is an edge between s and each vertex in $\{a_i, b_i : 1 \leq i \leq n\} \cup \{A_i : 1 \leq i < n\} \cup \{c_i : 1 \leq i \leq p+1\}$. There is an edge between each vertex a_i (resp. b_i) and any vertex $v_{l,m}$ with $1 \leq l \leq p+1, 1 \leq m \leq 2$, such that $v_{l,m}$ corresponds to the literal x_i (resp. $\overline{x_i}$). Obviously, G has diameter 4. We claim that there is an assignment for the variables in $\mathcal{X}_n$ that satisfies at least k clauses if and only if there is a (D, G)-consistent path with length at least $p + k + 1 + 2n$. This assertion is based on the following remarks:

1. Each consistent path of length at least 1 contains s. Indeed, the connected components of $G - \{s\}$ do not allow to compute paths in D.
2. No consistent path P contains two vertices associated with literals $x_i, \overline{x_i}$, for some i. Indeed, at most one of the vertices a_i, b_i belongs to P (by construction of D), and thus in G only vertices corresponding to x_i or to $\overline{x_i}$ may be connected to s (via a_i or, respectively, b_i).
3. Every consistent path P of length at least $p+1$ necessarily contains one vertex from each compulsory level. Indeed, such a path contains s (as before) and, because of its length, at least one vertex $v_{i,j}$, $1 \leq i \leq p+1$ and $1 \leq j \leq 2$. The connectivity in G implies that a_i or b_i belongs to P and consequently, in D, we deduce that $v_{p+1,1}$ or $v_{p+1,2}$ (that correspond respectively to x_n and $\overline{x_n}$) belongs to P. Then, the connectivity in G implies that a_n or b_n belongs to P and the claim follows.

$\Rightarrow$: Given an assignment $\mathcal{A}$ of the variable set $\mathcal{X}_n$ that satisfies k' clauses of $\mathcal{C}$, s.t. $k' \geq k$, assume w.l.o.g. that variables $v_{i_1,1}, \dots, v_{i_{k'},1}$, $v_{p+1,1}$ correspond to true literals. Let $\mathcal{B}(i) = a_i$ if x_i is true, and $\mathcal{B}(i) = b_i$ otherwise. Then the path P with vertices $s, v_{i_1,1}, \dots, v_{i_{k'},1}, v_{p+1,1}, c_1, \dots,$ c_{p+1}, $\mathcal{B}(1), A_1, \mathcal{B}(2), A_2, \dots, \mathcal{B}(n-1), A_{n-1}, \mathcal{B}(n)$ is (D, G)-consistent and has length $p + k' + 1 + 2n$. Indeed, in G the vertices $v_{i_1,1}, \dots, v_{i_{k'},1}, v_{p+1,1}$ are connected to s using the corresponding vertex $\mathcal{B}(l_{i_j})$ (s. t. $v_{i_j,1}$ is $x_{l_{i_j}}$ or $\overline{x_{l_{i_j}}}$), $1 \leq j \leq k'$, and $\mathcal{B}(n)$ respectively. All the other vertices are adjacent to s.
$\Leftarrow$: Let P be a (D, G)-consistent path P s.t. $|V(P)| \geq p+k+2+2n$. Let k' be the number of vertices in P that belong to optional levels. According to Remark 2, assigning the value `true` to the literals associated to these vertices yields a correct assignment, that satisfies k' clauses. Moreover, by Remarks 1 and 3, P contains $p + 2 + 2n$ vertices on the compulsory levels, and thus $|V(P)| = p + 2 + k' + 2n$ vertices. We deduce that $k' \geq k$. □

Moreover, we can also show (proof omitted here) using a reduction from Maximum Independent Set on cubic graphs that:

Theorem 2. SkewGraM *is* APX*-hard.*

4 Two Algorithms for SkewGraM

We propose two algorithms for SkewGraM: a heuristic called AlgoH, and an exact exponential-time algorithm called AlgoBB using the branch and bound method. Both algorithms look for a longest (D, G)-consistent path going through a given arc xy of D. Then, to solve SkewGram, an execution is needed for every arc xy of D.

Let $i \rightsquigarrow^D j$ denote a path in D from vertex i to vertex j (becomes $i \rightarrow^D j$ when reduced to an edge). Given two undirected graphs $G_1(U, E_1)$ and $G_2(U, E_2)$, a *common connected component* of G_1 and G_2 is any maximal set $X \subseteq U$ such that $G_1[X]$ and $G_2[X]$ are connected. We note by $CCC(D^*, G, i \rightsquigarrow^D j)$ the common connected component of D^* and G that contains all the vertices of the path $i \rightsquigarrow^D j$, if such a common connected component exists (equal to $\emptyset$ otherwise). The notation S_i^+ stands for the set of vertices that are reachable by a path from vertex i in D, whereas S_i^- stands for the set of vertices of D reaching vertex i by a path in D. Finally, vertex $r \in V$ is called a *bridge of* $i \rightsquigarrow^D j$ *with respect to* G if there is no common connected component of $D^*[V - \{r\}]$ and $G[V - \{r\}]$ containing all the vertices of $i \rightsquigarrow^D j$ (i.e. $CCC(D^*[V - \{r\}], G[V - \{r\}], i \rightsquigarrow^D j) = \emptyset$).

The Heuristic AlgoH. We construct the (D,G)-consistent path progressively by starting with the given arc xy and extending it. AlgoH first computes the cover set of a path, which is used to reduce the graph by removing vertices not compatible with the bridges.

Definition 1. *The* cover set *of a path* $i \rightsquigarrow^D j$*, denoted* CoverSet*(*D*,*G*,* $i \rightsquigarrow^D j$*), is the set* X *satisfying:*

1. $V(i \rightsquigarrow^D j) \subseteq X \subseteq S_i^- \cup S_j^+ \cup V(i \rightsquigarrow^D j)$.
2. $D^*[X]$ *and* $G[X]$ *are connected.*
3. *If* r *is a bridge of* $i \rightsquigarrow^{D[X]} j$ *w.r.t.* $G[X]$ *then* $X \subseteq S_r^- \cup S_r^+ \cup \{r\}$.
4. X *is maximal (with respect to the inclusion order).*

If, for a path $i \rightsquigarrow^D j$*, no vertex set* X *satisfies conditions* 1., 2. *and* 3.*, then by convention* CoverSet*(*D*,*G*,* $i \rightsquigarrow^D j$*)=* $\emptyset$*.*

The cover set of a path is unique, and easily computable (see Algorithm 1 which uses Algorithm GenPartRefinement described in [10] to compute the common connected components):

Lemma 2. *The cover set of a given path* $i \rightsquigarrow^D j$ *is well-defined.*

Lemma 3. *Algorithm* GetCoverSet *correctly computes the cover set of a given path in* $\mathcal{O}(n^2 \log n + nm \log^2 n)$.

Now, to compute a (D, G)-consistent path going through xy, we use Algorithm AlgoH (see Algorithm 2) to successively increase the current path *cp* (which is initially the arc $x \rightarrow^D y$) as follows. Once Algorithm GetCoverSet is applied to reduce D and G when possible (line 7), either $V(D) = \emptyset$ (and there is no (D, G)-consistent path containing xy), or D is Hamiltonian (and the algorithm returns the best current solution), or *cp* must be extended. The possible extensions p (by adding one vertex at the beginning or at the end of the path, using function Extend) are computed (lines 12-16) as well as the resulting reduced

Algorithm 1. GetCoverSet($D, G, i \leadsto^D j$)

```
Require: A DAG $D = (V, A(D))$, an undirected graph $G(V, E(G))$, a path $i \leadsto^D j$.
Ensure: Computes the cover set of $i \leadsto^D j$.
$S := S_i^- \cup S_j^+ \cup V(i \leadsto^D j)$
$S := CCC(D^*[S], G[S], i \leadsto^{D[S]} j)$; $STOP := false$;
while (($STOP = false$) and ($S \neq \emptyset$)) do
    $S_{tmp} := S$; /* note that $i \leadsto^{D[S]} j$ is identical to $i \leadsto^D j$ */
    for each bridge $r$ of $i \leadsto^D j$ in $G$ do
        $S_{tmp} := S_{tmp} \cap (\{r\} \cup S_r^- \cup S_r^+)$
    end for
    if ($S = S_{tmp}$) then
        $STOP := true$
    else
        $S := S_{tmp}$; $S := CCC(D^*[S], G[S], i \leadsto^{D[S]} j)$;
    end if
end while
return $S$
```

Algorithm 2. AlgoH(D, G, xy)

```
Require: A DAG $D = (V, A(D))$, an undirected graph $G(V, E(G))$, an arc $xy \in A(D)$.
Ensure: $S \subseteq V$: $D[S]$ is a $(D, G)$-consistent path containing the arc $xy$ or $S = \emptyset$.
 1: /* $bcs$: best current solution; $cp$: current path */
 2: /* $L_{ext}$: the list of all paths in $D$ extending the current path by one vertex */
 3: /* $DCS$: the subgraphs of $D$ induced by the cover sets of paths in $L_{ext}$ */
 4: /* $HCS$: the Hamiltonian subgraphs induced by the cover sets of paths in $L_{ext}$ */
 5: $bcs := \emptyset$; $cp := x \rightarrow^D y$; $f := x$; $l := y$; $STOP := false$;
 6: while ($STOP = false$) do
 7:     $S :=$ GetCoverSet($D, G, cp$); $D := D[S]$; $G := G[S]$;
 8:     if ($S = \emptyset$ or $D$ is Hamiltonian) then
 9:         $STOP := true$;
10:         if $|bcs| > |S|$ then $S := bcs$ end if
11:     else
12:         $L_{ext} := \emptyset$; /* compute the extensions of $cp = f \leadsto^D l$ */
13:         for each $v$ that is a predecessor of $f$ or a successor of $l$ do
14:             $p =$ Extend($cp$, $v$); /* extends $cp$ with $v$ */
15:             $L_{ext} := L_{ext} \cup \{p\}$;
16:         end for
17:         $DCS = \{D[\text{CoverSet}(D, G, p)] : p \in L_{ext}\}$;
18:         $HCS := \{d \in DCS : d \text{ is Hamiltonian}\}$;
19:         Let $h_{max} \in HCS$ s.t. $|V(h_{max})| = max\{|V(h)| : h \in HCS\}$
20:         if $|bcs| < |V(h_{max})|$ then $bcs := V(h_{max})$ end if
21:         Let $p_{max} = f_{max} \leadsto^D l_{max}$ s.t. $value(p_{max}) = max\{value(p) : p \in L_{ext}\}$;
22:         if $|bcs| \geq value(p_{max})$ then
23:             /* No $(D, G)$-consistent path through $xy$ and longer than $|bcs|$ exists */
24:             $S := bcs$; $STOP := true$;
25:         else
26:             $cp := p_{max}$; /* continue with the most promising extension */
27:             $f := f_{max}$; $l := l_{max}$
28:         end if
29:     end if
30: end while
31: return $S$
```

graphs (line 17). The Hamiltonian graphs among them are considered for improving the best current value (lines 18-20). Then, the most promising extension is computed using the evaluation $value(p)$, equal to the length of the longest path in $D[\text{CoverSet}(D, G, p)]$ (line 21). If this extension allows to hope an improvement of bcs, then it is kept (lines 22-28). An experimental evaluation of AlgoH is given in the next section.

Complexity of Algorithm AlgoH: let Δ be the maximum total degree of a vertex in D, and L be the length of the optimal solution of SkewGraM. Then, the **while** loop in line 6 is executed at most L times. The most time consuming internal instruction is the one in line 17, which makes 2Δ calls to Algorithm GetCoverSet. Recall that the longest path is easily computed in a DAG. Then the complexity of Algorithm AlgoH is in $\mathcal{O}(\Delta L(n^2 \log n + nm \log^2 n))$.

The Exact Algorithm AlgoBB. This algorithm is based on the branch and bound method. The tree TS of sub-solutions is built as follows. The root is associated to the arc xy given as input of SkewGraM. Each vertex s of TS is associated to a path $p(s)$ extending the arc xy. At the end of the construction of TS, its leaves are associated to (D, G)-consistent paths containing xy. The solution of SkewGraM is thus a longest path $i \leadsto^D j$ such that there exists a leaf of TS associated to $i \leadsto^D j$.

Branching. We expand vertex s with $p(s) = v_l \leadsto^D v_m$ as follows. For each v_k that is a predecessor of v_l (resp. successor of v_m), we add in TS a child of s associated to the path $v_k.p(s)$ (resp. $p(s).v_k$). For a vertex $s \in TS$, recall that $value(p(s))$ is the length of the longest path in $D[\text{CoverSet}(D, G, p(s))]$. Let $BBvalue(s)$ denote the evaluation of a vertex s in TS. This function is defined as follows: (i) if s is to be expanded, then $BBvalue(s) = value(p(s))$; (ii) if s has already been expanded at some specific moment of the construction of TS, then $BBvalue(s)$ is the length of $p(s)$ if it is (D, G)-consistent. Otherwise, $BBvalue(s) = 0$. Using this evaluation function, we define the *bounding* and *pruning* rules as follows.

*Bounding (**Rule 1**).* Among vertices $\{s_1, s_2, \ldots, s_k\}$ to be expanded, we choose the vertex s^* such that $BBvalue(s^*) = \max\{BBvalue(s_i) : 1 \leq i \leq k\}$. If there are several such vertices, we arbitrarily choose one.

*Pruning (**Rule 2**).* Let s_{max} be a vertex of TS satisfying the two following conditions: (i) s_{max} was expanded, and (ii) $BBvalue(s_{max}) \geq BBvalue(s)$, for any expanded vertex s of TS. Then, delete from TS any leaf vertex s with $BBvalue(s) \leq BBvalue(s_{max})$. This deletion is applied recursively for vertices that become leaves after the deletion of all their children.

Theorem 3. *Algorithm* AlgoBB *exactly solves* SkewGraM.

5 Experimental Results

In order to show the reliability of our heuristic AlgoH, we first applied it on random (Erdös-Rényi and scale-free) graphs and we compared the obtained solutions to the optimal ones computed by our exact algorithm AlgoBB. We also applied AlgoH on different types of biological networks.

5.1 Performances of AlgoH

Let |AlgoBB| (resp. |AlgoH|) be the number of vertices of a solution found by AlgoBB (resp. found by AlgoH). We measured the performance of AlgoH by

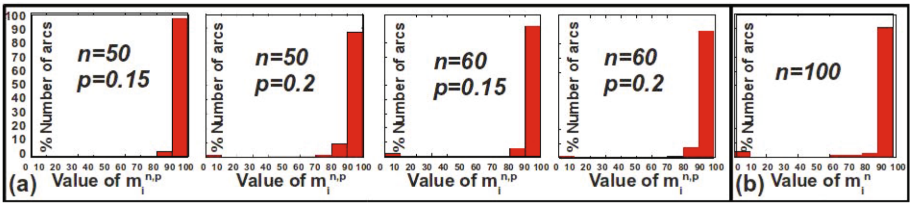


Fig. 1. Performances of heuristic AlgoH. We show the percentage of arcs whose $\frac{|\text{AlgoH}|}{|\text{AlgoBB}|} \times 100$ belongs to an interval I_i (see Section 5.2). **(a) General graphs**. For fixed n and p we generated 100 couples (D, G). **(b) Scale-free graphs**. Algorithms AlgoH and AlgoBB were run on 100 couples (D, G) of order 100.

computing the ratio $\rho = \frac{|\text{AlgoH}|}{|\text{AlgoBB}|}$ for every input instance. By convention, $\rho = 1$ whenever the exact algorithm finds no (D, G)-consistent path for a given arc.

a) General graphs. We chose to vary two parameters: the number n of vertices of D and G (in the range 20, 30, 40, 50, 60), and the probability p that an edge between any given two vertices exists (in the range 0.05, 0.1, 0.15, 0.2). Taking any combination of these two parameters thus leads to 20 runs. We generated the undirected graphs G by using the Erdös-Rényi [9] random graphs generation method. We adapted this method, in order to construct random DAGs D, by randomly orienting the edges. For fixed n and p, we generated 100 couples (D, G). For each of these couples, we applied AlgoH and AlgoBB for 5 randomly chosen arcs, and computed the ratio ρ for each of the 5 corresponding instances. We then computed the number denoted $N_i^{(D,G,p,n)}$, $0 \leq N_i^{(D,G,p,n)} \leq 5$, of arcs whose $\rho \times 100$ belongs to I_i, with $1 \leq i \leq 10$ and $I_1 = [0, 10[, I_2 = [10, 20[, \ldots, I_{10} = [90, 100]$. We obtained a global result by computing, for each I_i, the value $m_i^{(n,p)} = \sum_{(D,G)} N_i^{(D,G,n,p)}$ i.e., the sum of $N_i^{(D,G,n,p)}$ for all 100 generated couples (D,G), with fixed n, p .

b) Scale-free graphs. We also applied our method on randomly generated scale-free graphs, since recent studies have shown that a large number of real-world networks tend to be scale-free (see e.g. [1]). In this experiment, we generated 100 couples (D,G) of 100 vertices, by using the public toolkit *NGCE* (http://ngce.sourceforge.net/). We observed that consistent paths are not abundant in scale-free graphs, thus we randomly chose, for each graph D, 10 arcs rather than 5 arcs as in the previous experiment. We then computed for each couple (D, G) and for each interval I_i, the number $N_i^{(D,G,n)}$, $0 \leq N_i^{(D,G,n)} \leq 10$, of arcs whose $\rho \times 100$ belongs to I_i and the global value $m_i^n = \sum_{(D,G)} N_i^{(D,G,n)}$ (here, n is fixed to 100).

We observe a very good behaviour of our heuristic AlgoH, since more than 90% of the input instances have a $\rho \times 100$ belonging to the interval $[90, 100]$ (see Figure 1). Also, it is very important to note the speed-up obtained by our heuristic with respect to the exact algorithm AlgoBB. For example, for the 500 instances of random graphs evaluated in the case $n = 60$ and $p = 0.2$ (Figure 1.**a**), Algorithm AlgoH was 11 times faster than AlgoBB.

5.2 Applying AlgoH on Biological Networks

We have also applied our heuristic AlgoH on real biological networks, in two different contexts, in order to verify that its results corroborate biological assumptions.

Metabolic Pathway vs PPI Network. A metabolic network is usually modeled by a directed graph (called a *reaction graph*) whose vertices are the reactions, and where there is an arc between two reactions if the first uses, as substrate, a product of the second. A metabolic network is represented in the *KEGG* database as a collection of functional modules (small networks) called *metabolic pathways*. Therefore, metabolic pathways can be modeled by DAGs [16]. A protein-protein interaction network (PPI) is modeled by an undirected graph whose vertices are the proteins, and there is an edge between each pair of physically interacting proteins. We applied our heuristic to extract automatically, in a metabolic pathway, a chain of reactions (i.e., a path) that are catalyzed by interacting proteins (i.e., a connected subgraph) in a PPI network. Such paths are biologically meaningful [7]: indeed, the authors of [7] divided the PPI network for the species *S. cerevisiae* into functional clusters and observed that proteins involved in successive reactions are generally more likely to interact than other protein pairs. They provided an example of a short path (of length 6) in the metabolic pathway *"Glycolysis/Gluconeogenesis"* corresponding to a functional cluster in the PPI network. In order to compare our results with theirs, we built the PPI graph G, of the same species *S. cerevisiae*, from the BioGRID database (http://thebiogrid.org/, version (v2.0.63)).We also constructed the metabolic pathway *"Glycolysis/Gluconeogenesis"* (graph D) from *KEGG* (*pathwaysce00010.xml*). We established the correspondence between the two graphs using the names of the genes which encode proteins that (a) catalyze reactions in the metabolic pathway and (b) interact in the PPI network. Notice that G does not have the same vertex set as D. In order to circumvent this difficulty, we used an additional graph G': the new graph G' is an undirected graph whose vertices are the reactions ($V(G') = V(D)$) and there is an edge between two vertices $r_1, r_2 \in V(G')$, iff there are two interacting proteins $p_1, p_2 \in V(G)$ s.t. p_1 catalyzes r_1 and p_2 catalyzes r_2 (see also [2], where the construction of such a graph is detailed). Applying our heuristic AlgoH with the arc between vertices 1 and 23 as input, we automatically computed a (D, G')-consistent path of 12 vertices, inducing a connected subgraph of 20 proteins in the PPI network. Such a path includes those observed in [7].

Metabolic Pathway vs. Linear Genome. We also applied AlgoH to automatically extract, in a metabolic pathway, a chain of reactions that are catalyzed by the products of adjacent genes in the genome. The genome network is modeled by an undirected graph whose vertices are the genes, and in which there is an edge between each adjacent pair of genes. The species under study was the bacterium *E. coli*. We built the linear sequence of genes (graph G) from the *NCBI* database. We constructed the metabolic pathway (graph D) from *KEGG* (*pathwayeco00550.xml*). We established the correspondence between the two graphs using the names of the genes. Finally, as in the previous example, we constructed an additional undirected graph G' built on same vertex set of D. The longest (D, G')-consistent path we found by applying AlgoH for all the arcs of D is the same as the path found by Boyer et al. [3] (see Figure 3.b in [3]).

6 Conclusion

The SkewGraM problem belongs to a new type of subnetwork mining problems, arising from recent applications of biological, social or information networks: several graphs, of various types, represent different relations between objects, and a subset of objects is sought, with particular properties in each network. Due to an important set of parameters (the networks nature, the properties to fulfill, etc.), these problems are very complex. Still, they need good algorithmic solutions, since the size of the networks is often very large. In this paper, we studied the limits, in terms of graph classes (a graph representing a network), between difficult and easy cases, and we provided two algorithms, a reliable heuristic and an exact algorithm. We tested them on random data, in order to show the performances of our heuristics in terms of execution time and quality of the results. We also tested them on real data, in order to show their effectiveness on biological networks. Further studies should either investigate the complexity of SkewGraM in terms of approximability (on specific graph classes) and fixed parameter algorithms, or inexact variants of the problem obtained, for instance by allowing small differences between the vertex set of the path in D and the connected set in G.

References

1. Barabási, A.-L.: Scale-free networks: a decade and beyond. Science 325, 412–413 (2009)
2. Blin, G., Fertin, G., Mohamed-Babou, H., Rusu, I., Sikora, F., Vialette, S.: Algorithmic Aspects of Heterogeneous Biological Networks Comparison. In: Wang, W., Zhu, X., Du, D.-Z. (eds.) COCOA 2011. LNCS, vol. 6831, pp. 272–286. Springer, Heidelberg (2011)
3. Boyer, F., Morgat, A., Labarre, L., Pothier, J., Viari, A.: Syntons, metabolons and interactons: an exact graph-theoretical approach for exploring neighbourhood between genomic and functional data. Bioinformatics 21(23), 4209–4215 (2005)
4. Bunke, H.: Graph matching: theoretical foundations, algorithms and applications. In: Proc. Vision Interface, pp. 82–88 (2000)
5. Cai, D., Shao, Z., He, X., Yan, X., Han, J.: Mining hidden community in heterogeneous social networks. In: Proc. of the 3rd International Workshop on Link Discovery, pp. 58–65 (2005)

6. Conte, D., Foggia, P., Sansone, C., Vento, M.: Thirty years of graph matching in pattern recognition. International Journal of Pattern Recognition and Artificial Intelligence 18, 265–298 (2004)
7. Durek, P., Walther, D.: The integrated analysis of metabolic and protein interaction networks reveals novel molecular organizing principles. BMC Systems Biology 2(1) (2008)
8. Dzeroski, S., Lavrac, N.: Relational data mining. Springer (2001)
9. Erdös, P., Rényi, A.: On random graphs, I. Publicationes Mathematicae (Debrecen) 6, 290–297 (1959)
10. Gai, A.-T., Habib, M., Paul, C., Raffinot, M.: Identifying common connected components of graphs. Technical Report RR-LIRMM-03016, LIRMM (2003)
11. Garey, M.R., Johnson, D.S.: Computers and Intractability: a guide to the theory of NP-completeness. W.H. Freeman (1979)
12. Kelley, B.P., Yuan, B., Lewitter, F., Sharan, R., Stockwell, B.R., Ideker, T.: Pathblast: a tool for alignment of protein interaction networks. Nucleic Acids Research 32, 83–88 (2004)
13. Matsuo, Y., Hamasaki, M., Takeda, H., Mori, J., Bollegara, D., Nakamura, Y., Nishimura, T., Hasida, K., Ishizuka, M.: Spinning multiple social networks for semantic web. In: Proc. of the Twenty-First National Conference on Artificial Intelligence (2006)
14. Sharan, R., Suthram, S., Kelley, R.M., Kuhn, T., Mccuine, S., Uetz, P., Sittler, T., Karp, R.M., Ideker, T.: Conserved patterns of protein interaction in multiple species. National Academy of Sciences 102(6), 1974–1979 (2005)
15. Vicentini, R., Menossi, M.: Data mining and knowledge discovery in real life applications. Julio Ponce and Adem Karahoca edition. In-tech (2009)
16. Wernicke, S., Rasche, F.: Simple and fast alignment of metabolic pathways by exploiting local diversity. Bioinformatics 23, 1978–1985 (2007)
17. Williams, E., Bowles, D.J.: Coexpression of neighboring genes in the genome of arabidopsis thaliana. Genome Research 14, 1060–1067 (2004)

Computing Strong Articulation Points and Strong Bridges in Large Scale Graphs*

Donatella Firmani[1], Giuseppe F. Italiano[2], Luigi Laura[1],
Alessio Orlandi[3], and Federico Santaroni[2]

[1] Dept. of Computer Science and Systems, Sapienza Univ. of Rome,
via Ariosto, 25 - 00185 Roma, Italy
{firmani,laura}@dis.uniroma1.it

[2] Dept. of Computer Science, Systems and Production, Univ. of Rome "Tor Vergata",
via del Politecnico 1 - 00133 Roma, Italy
{italiano,santaroni}@disp.uniroma2.it

[3] Dept. of Computer Science, Univ. of Pisa,
Largo Bruno Pontecorvo 3, 56127 Pisa, Italy
aorlandi@di.unipi.it

Abstract. Let $G = (V, E)$ be a directed graph. A vertex $v \in V$ (respectively an edge $e \in E$) is a *strong articulation point* (respectively a *strong bridge*) if its removal increases the number of strongly connected components of G. We implement and engineer the linear-time algorithms in [9] for computing all the strong articulation points and all the strong bridges of a directed graph. Our implementations are tested against real-world graphs taken from several application domains, including social networks, communication graphs, web graphs, peer2peer networks and product co-purchase graphs. The algorithms implemented turn out to be very efficient in practice, and are able to run on large scale graphs, i.e., on graphs with ten million vertices and half billion edges. Our experiments on such graphs highlight some properties of strong articulation points, which might be of independent interest.

Keywords: graph algorithms, strong connectivity, strong articulation points, strong bridges, large scale graphs.

1 Introduction

Let $G = (V, E)$ be a directed graph. A vertex $v \in V$ is a *strong articulation point* if its removal increases the number of strongly connected components of G. Similarly, an edge $e \in E$ is a *strong bridge* if its removal increases the number of strongly connected components of G (see Figure 1). Note that strong articulation points and strong bridges are related to the notion of 2-vertex and 2-edge

* This work has been partially supported by the 7th Framework Programme of the EU (Network of Excellence "EuroNF: Anticipating the Network of the Future - From Theory to Design") and by MIUR, the Italian Ministry of Education, University and Research, under Project AlgoDEEP.

R. Klasing (Ed.): SEA 2012, LNCS 7276, pp. 195–207, 2012.

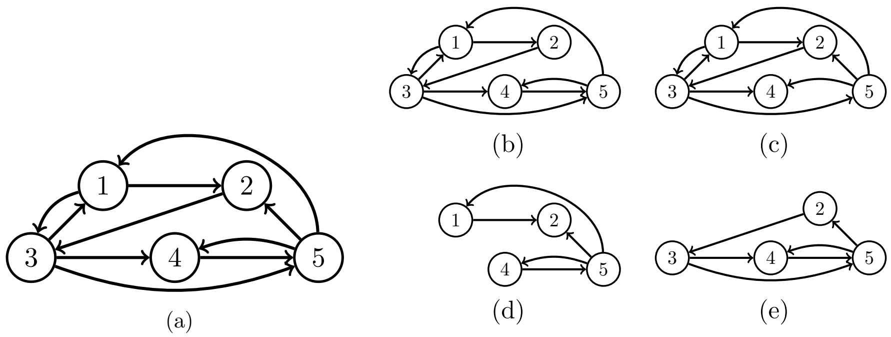


Fig. 1. (a) A strongly connected graph $G = (V, E)$. (b) Edge $(5, 2)$ is **not** a strong bridge, while (c) edge $(4, 5)$ is a strong bridge. (d) Vertex 3 is a strong articulation point, while (e) vertex 1 is **not** a strong articulation point. The strong articulation points in G are vertices 3 and 5, and the strong bridges in G are edges $(2, 3)$ and $(4, 5)$.

connectivity of directed graphs. We recall that a strongly connected graph G is said to be 2-vertex-connected if the removal of any vertex leaves G strongly connected; similarly, a strongly connected graph G is said to be 2-edge-connected if the removal of any edge leaves G strongly connected. The strong articulation points are exactly the vertex cuts for 2-vertex connectivity, while the strong bridges are exactly the edge cuts for 2-edge connectivity: G is 2-vertex-connected (respectively 2-edge-connected) if and only if G does not contain any strong articulation point (respectively strong bridge). Surprisingly, the study of 2-vertex and 2-edge connectivity in directed graphs seems to have been overlooked and it only received attention quite recently. Although there is no specific linear-time algorithm in the literature, the 2-edge connectivity of a directed graph can be tested in $O(m + n)$: one can check whether a directed graph is 2-edge-connected by using Tarjan's algorithm [14] to compute two edge-disjoint spanning trees in combination with the disjoint set-union algorithm of Gabow and Tarjan [6]. For testing 2-vertex connectivity, there is a very recent linear-time algorithm of Georgiadis [7]. Note that none of the above algorithms finds *all* the strong articulation points or *all* the strong bridges of a directed graph.

There are certain applications, however, when one is interested in computing *all* the strong articulation points or *all* the strong bridges of a directed graph. This includes the identification of cores in directed social networks [11], filtering algorithms for the tree constraint in constrained programming [2], and verifying the restricted edge connectivity of strongly connected graphs [15]. Linear-time algorithms for computing all the strong bridges and all the strong articulation points of a directed graph were recently proposed in [9]. In this paper, we implement and engineer the algorithms in [9] and test our implementations against real-world graphs taken from several application domains, including social networks, communication graphs (such as email graphs), web graphs, peer2peer networks and product co-purchase graphs. Our implementations appear to be fast and memory-efficient in practice, and this makes it possible to compute all the strong articulation points and the strong bridges of large scale graphs, namely graphs up to twenty million vertices and half billion edges, in a dozen minutes. In addition, our experiments highlight some properties of strong articulation points, which might help to further characterize the structure of large scale real-world graphs, and in particular to identify cores in social networks. We next describe briefly some of the observations that arise from our experiments.

First, strong articulation points appear frequently in real-world graphs: indeed, most of the graphs in our datasets have quite a high number of strong articulation points. As an example, between 15% and 25% of the vertices in product co-purchase graphs are strong articulation points, while in social graphs this figure ranges between 11% and 18%. The relative number of strong bridges is much smaller, with the only notable exception of email graphs, where strong bridges are about 10% of the total number of edges. Another interesting property is that in our dataset the vast majority of the strong articulation points and the strong bridges tend to be inside the largest strongly connected component. As a further indication of their importance, we mention that the indegree, outdegree and PageRank of the strong articulation points tend to be much higher than average in communication graphs and substantially higher than average in social and web graphs.

2 Graph Terminology

We assume that the reader is familiar with the standard graph terminology, as contained for instance in [5]. Let $G = (V, E)$ be a directed graph, with m edges and n vertices. A *directed path* in G is a sequence of vertices v_1, v_2, ..., v_k, such that edge $(v_i, v_{i+1}) \in E$ for $i = 1, 2, \ldots, k-1$. A directed graph G is *strongly connected* if there is a directed path from each vertex in the graph to every other vertex. The *strongly connected components* of G are its maximal strongly connected subgraphs. A directed graph G^t is said to be a *transitive reduction* of G if (i) G^t has a directed path from vertex u to vertex v if and only if G has a directed path from vertex u to vertex v, and (ii) there is no graph with fewer edges than G^t satisfying condition (i). Given a directed graph $G = (V, E)$, its *reversal graph* $G^R = (V, E^R)$ is defined by reversing all edges of G: namely, G^R has the same vertex set as G and for each edge (u, v) in G there is an edge (v, u) in G^R. We say that the edge (v, u) in G^R is the *reversal* of edge (u, v) in G.

A flowgraph $G(s) = (V, E, s)$ is a directed graph with a *start vertex* $s \in V$ such that every vertex in V is reachable from s. The *dominance relation* in $G(s)$ is defined as follows: a vertex u is a *dominator* of vertex v if every path from vertex s to vertex v contains vertex u. Let $dom(v)$ be the set of dominators of v. Clearly, $dom(s) = \{s\}$ and for any $v \neq s$ we have that $\{s, v\} \subseteq dom(v)$: we say that s and v are the *trivial dominators* of v in the flowgraph $G(s)$. The dominance relation is transitive and its transitive reduction is referred to as the *dominator tree* $DT(s)$. Note that the dominator tree $DT(s)$ is rooted at vertex s. Furthermore, vertex u dominates vertex v if and only if u is an ancestor of

v in $DT(s)$. We say that u is an *immediate dominator* of v if u is a dominator of v, and every other non-trivial dominator of v also dominates u. It is known that if a vertex v has any non-trivial dominators, then v has a unique immediate dominator: the immediate dominator of v is the parent of v in the dominator tree $DT(s)$. In the following, we denote by $D(s)$ the set of non-trivial dominators in $G(s)$. Let $G^R = (V, E^R)$ be the reversal graph of G, and let $G^R(s) = (V, E^R, s)$ be the flowgraph with start vertex s: we denote by $D^R(s)$ the set of non-trivial dominators in $G^R(s)$.

Similarly, we say that an edge (u, v) is an *edge dominator* of vertex w if every path from vertex s to vertex w contains edge (u, v). Furthermore, if (u, v) is an edge dominator of w, and every other edge dominator of u also dominates w, we say that (u, v) is an *immediate edge dominator* of w. Similar to the notion of dominators, if a vertex has any edge dominators, then it has a unique immediate edge dominator. As before, we denote by $DE(s)$ the set of edge dominators in $G(s)$ and by $DE^R(s)$ the set of edge dominators in $G^R(s)$.

3 Experimental Setup

Test Environment. All our experiments were performed on a machine with a CPU Intel Xeon X5650 with 6 cores, running at 2.67GHz, with 12MB of cache and 32GB RAM DDR3 at 1GHz. The operating system was Linux Red Hat 4.1.2-46, with kernel version 2.6.18, Java Virtual Machine version 1.6.0_16 (64-Bit) and WebGraph library version 3.0.1. Our implementations have been written in Java, in order to exploit the features offered by the WebGraph library [3], which has been designed especially to deal with large graphs. Our code is available upon request. All the running times reported in our experiments were averaged over ten different runs.

Datasets. In our experiments we considered several large-scale real-world graphs, with up to ten million vertices and half billion edges. Our graphs come from different application areas, including web graphs, communication networks, peer-to-peer networks, social networks and product co-purchase graphs. Vertices of a co-purchase graph correspond to products, while edges connect commonly co-purchased products (i.e., there is a directed edge from x to y when users who bought product x also bought product y). The graphs considered in our experiments, together with their type, information about the repository where the graph was taken, number of vertices (n) and edges (m), average vertex degree (δ_{avg}), are listed in Table 1. As already mentioned, we represent the graphs using the WebGraph file format, storing both the original and the reversal of a graph in order to access the adjacency lists in both directions. All the graphs taken from other repositories (such as the SNAP graphs) were converted into this format before performing the experiments. For lack of space, we refer the interested reader to the repositories SNAP [12] and WebGraph [16] for an explanation and more information about those graph datasets and their file formats.

Table 1. Real-world graphs in our experiments, sorted by number of edges (m)

Graph	**Type**	**Repository**	n	m	δ_{avg}
p2p-Gnutella04	Peer2peer	SNAP	11 K	40 K	3.7
wiki-Vote	Social	SNAP	7 K	103 K	14.5
enron	Communication	WebGraph	69 K	276 K	4.0
email-EuAll	Communication	SNAP	265 K	420 K	1.58
soc-Epinions1	Social	SNAP	76 K	509 K	30.5
soc-Slashdot0811	Social	SNAP	77 K	905 K	11.7
soc-Slashdot0902	Social	SNAP	82 K	948 K	11.5
amazon0302	Product co-purchase	SNAP	262 K	1.2 M	4.7
web-NotreDame	Web	WebGraph	325 K	1.4M	4.6
uk-2007-05@100K	Web	WebGraph	100 K	3 M	30.5
cnr-2000	Web	WebGraph	325 K	3.2 M	9.8
amazon0312	Product co-purchase	SNAP	400 K	3.2 M	8.0
amazon-2008	Product co-purchase	SNAP	735 K	5.1 M	7.0
wiki-Talk	Communication	SNAP	2.3 M	5.0 M	2.1
web-Google	Web	SNAP	875 K	5.1 M	5.8
web-BerkStan	Web	SNAP	685 K	7.6 M	11.0
in-2004	Web	WebGraph	1.3 M	16.9 M	12.2
eu-2005	Web	WebGraph	862 K	19.2 M	22.3
uk-2007-05@1M	Web	WebGraph	1 M	41.2 M	41.2
soc-LiveJournal1	Social	SNAP	4.8 M	68.9 M	14.2
ljournal-2008	Social	SNAP	5.3 M	79.0 M	14.7
indochina-2004	Web	WebGraph	7.4 M	194 M	26.1
uk-2002	Web	WebGraph	18.5M	298 M	16.1
arabic-2005	Web	WebGraph	22.7 M	640 M	28.1

4 Algorithms for Strong Articulation Points and Strong Bridges

In this section we describe algorithms for computing strong bridges and strong articulation points for strongly connected graphs. This is without loss of generality, since the strong bridges (respectively strong articulation points) of a directed graph G are given by the union of the strong bridges (respectively strong articulation points) of the strongly connected components of G.

There are trivial algorithms that compute all the strong articulation points and all the strong bridges of a graph G in $O(n(m+n))$ time. To compute whether a vertex v is a strong articulation point, it is enough to check whether the graph $G \setminus v$ is strongly connected. This yields an $O(n(m+n))$ time algorithm for computing all strong articulation points. To compute all strong bridges in $O(n(m+n))$ time is less immediate, and we describe next how to accomplish this task. Fix any vertex v of G. Let $T^+(v)$ be an out-branching rooted at v, i.e., a directed spanning tree rooted at v with all edges directed away from v. Similarly, let $T^-(v)$ be an in-branching rooted at v, i.e., a directed spanning tree rooted at v with all edges directed towards v. Note that $G' = T^+(v) \cup T^-(v)$ is not unique. However, as shown in [9], every graph G' produced in this way contains at most

1. Choose arbitrarily a vertex $s \in V$ in G, and test whether s is a strong articulation point in G. If s is a strong articulation point, output s.
2. Compute $D(s)$, the set of non-trivial dominators in the flowgraph $G(s) = (V, E, s)$.
3. Compute the reversal graph $G^R = (V, E^R)$.
4. Compute $D^R(s)$, the set of non-trivial dominators in the flowgraph $G^R(s) = (V, E^R, s)$.
5. Output $D(s) \cup D^R(s)$.

Fig. 2. Algorithm `ILS(SAP)` for computing all strong articulation points of G

1. Choose arbitrarily a vertex $s \in V$ in G.
2. Compute $DE(s)$, the set of edge dominators in the flowgraph $G(s) = (V, E, s)$.
3. Compute the reversal graph $G^R = (V, E^R)$.
4. Compute $DE^R(s)$, the set of edge dominators in the flowgraph $G^R(s) = (V, E^R, s)$.
5. Output $DE(s) \cup DE^R(s)$.

Fig. 3. Algorithm `ILS(SB)` for computing all strong bridges of G

$(2n-2)$ edges, can be computed in $O(m+n)$ time and includes all the strong bridges of G. This implies that all the strong bridges of G can be computed in $O(n(m+n))$ time by simply computing a graph $G' = T^+(v) \cup T^-(v)$ from G and checking for each edge e of G' whether the graph $G \setminus e$ is strongly connected.

Throughout this paper, we refer to both trivial $O(n(m+n))$ algorithms as `Naive`, and in particular we refer to the naive algorithm for computing strong articulation points (respectively strong bridges) as `Naive(SAP)` (respectively `Naive(SB)`). To obtain their linear-time algorithms, Italiano et al. [9] exploited the relationship between strong articulation points, strong bridges and dominators in flowgraphs stated in the following theorem:

Theorem 1. [9] *Let $G = (V, E)$ be a strongly connected graph, and let $s \in V$ be any vertex in G. Then vertex $v \neq s$ is a strong articulation point in G if and only if $v \in D(s) \cup D^R(s)$. Furthermore, edge (u, v) is a strong bridge in G if and only if $(u, v) \in DE(s)$ or $(v, u) \in DE^R(s)$.*

We implemented the algorithms for computing all strong articulation points and all strong bridges which stem directly from Theorem 1, as described respectively in Figures 2 and 3. We call both algorithms `ILS`, and we refer to the algorithm for computing strong articulation points (respectively strong bridges) as `ILS(SAP)` (respectively `ILS(SB)`). We next show how to compute dominators and edge dominators, which are the building blocks for the `ILS` algorithms.

Computing Dominators. Several algorithms for computing dominators in flowgraphs have been proposed in the literature. The algorithm by Lengauer and Tarjan [10], which we refer to as `LT`, runs in $O(m\alpha(m, n))$ worst-case time, where $\alpha(m, n)$ is a slowly growing inverse Ackermann function. Later on,

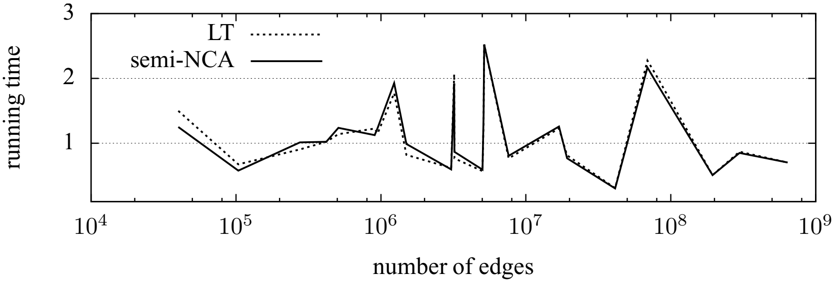


Fig. 4. Experimental comparison of `LT` and `semi-NCA`. The number of edges is shown in logarithmic scale and the running times (in microsecs) are normalized to the number of edges.

Alstrup et al. [1] designed linear-time solutions for computing dominators. As observed in the experimental study of Georgiadis et al. [8], this linear-time solution is "significantly more complex and thus unlikely to be faster than `LT` in practice". Georgiadis et al. [8] proposed a hybrid algorithm, dubbed `semi-NCA`, which despite being slower from the theoretical viewpoint (it runs in $O(n^2 + m \log n)$ time), appears to be very efficient in practice. In our study, we rewrote the implementations of `LT` and `semi-NCA` within the WebGraph framework [3] and performed some experiments in order to select the best method for computing dominators in our framework. We confirm on a larger scale the experimental findings of Georgiadis et al. [8], as the running times of `LT` and `semi-NCA` are very close in practice. The result of one such experiment on graphs in our dataset is illustrated in Figure 4. Since both algorithms appear to be comparable, and there is no clear winner, in the remainder of the paper we will only show the results of experiments where the dominator trees are computed by one of them, in particular `LT`.

Computing Edge Dominators. Our algorithm for computing edge dominators in flowgraphs hinges on the following lemma by Tarjan:

Lemma 1. [13] *Let $G = (V, E, s)$ be a flowgraph and let T be a DFS tree of G with start vertex s. Edge (v, w) is an edge dominator in G if and only if all of the following conditions are met: (v, w) is a tree edge, w has no entering forward edge or cross edge, and there is no back edge (x, w) such that w does not dominate x.*

From the algorithmic viewpoint, the only non-trivial part of this lemma is to check whether w dominates x. To do this, it is enough to test whether w is an ancestor of x in the dominator tree $DT(s)$. We can accomplish this task in constant time, once $DT(s)$ is available together with the preordering and postordering numbers produced by a depth-first search visit of $DT(s)$. Recall that the preordering number *first*(v) is given by the order in which vertex v was *first* visited, while the postordering number *last*(v) is given by the order in which vertex v was *last* visited. Now, it is easy to see that w dominates x if and only if *first*$(w) <$ *first*$(x) <$ *last*$(x) <$ *last*(w). Our algorithm for computing edge dominators requires all the extra work implied by Lemma 1 in addition to the computation of dominator trees. Thus, we expect that in practice ILS(SB) will be slightly slower than ILS(SAP). This was confirmed by our experiments.

5 Experimental Results

In this section we report the results of our experimental evaluation. To check when they become faster than the simple-minded Naive algorithms, we performed some experiments with small graphs. As it can be seen from Table 2, the ILS algorithms appear to be superior to the Naive algorithms even for very small graphs (few hundred vertices and thousand edges), and they become substantially faster for larger graphs. This shows that the ILS implementations seem to be the method of choice even in the case of very small graphs.

Table 3 illustrates the results of another experiment with large scale graphs, where the Naive algorithms are omitted due to their extremely high running times. In our experiments the ILS algorithms appeared to be very fast in practice: quite surprisingly, they were able to handle large graphs (hundred million edges) only in about a dozen minutes. As expected, ILS(SAP) was slightly faster than ILS(SB), since computing edge dominators is slightly more complicated than computing dominators (see Lemma 1).

The efficiency of our ILS implementations makes it possible to compute the strong articulation points and the strong bridges of large graphs, and allows us to try to characterize some of their main properties, as shown in Table 3. The data collected in our experiments suggest that in the graphs considered most of the strong articulation points are in the largest strongly connected component. This seems to be true especially for peer2peer, social and product co-purchase graphs. For instance in p2p-Gnutella04 all the strong articulation points and all strong bridges are inside the largest strongly connected component, even if this component contains only about 40% of the vertices and 47% of the edges. For web graphs, however, there are few instances (such as uk-2007-05@100k and web-BerkStan) where a susbstantial fraction of the strong articulation points lie

Table 2. ILS and Naive running times, measured in seconds. We use #*sap* and #*sb* to denote respectively the number of strong articulation points and of strong bridges in the graph

Graph	n	m	#*sap*	#*sb*	ILS(SAP)	Naive(SAP)	ILS(SB)	Naive(SB)
c.elegans	0.3 K	2.3 K	36	45	0.097	0.231	0.067	0.313
political_blogs1	0.6 K	2.7 K	43	84	0.049	0.214	0.064	0.296
political_blogs2	1.0 K	8.2 K	80	154	0.050	0.822	0.068	1.479
political_blogs3	1.4 K	19 K	115	216	0.101	2.049	0.099	4.238
p2p-Gnutella04	11 K	40 K	1344	1674	0.150	29.22	0.180	42.73
wiki-Vote	7 K	103 K	143	152	0.180	12.57	0.180	24.77
enron	69 K	276 K	796	4967	0.580	371.39	0.690	700.37

Table 3. Analysis of strong articulation points and bridges, and corresponding ILS running time (in seconds), for graphs from 40 K to 640 M edges. We denote by n_{scc} (resp. m_{scc}) the number of vertices (resp. edges) in the largest strongly connected component of the graph, and $\#sap_{scc}$ and $\#sb_{scc}$ to denote the number of strong articulation points and of strong bridges in that component.

Graph	n	m	$\#sap$	$\#sb$	n_{scc}	m_{scc}	$\#sap_{scc}$	$\#sb_{scc}$	ILS(SAP)	ILS(SB)
p2p-Gnutella04	11 K	40 K	1.3 K	1.6 K	4.3 K	18.7 K	1.3 K	1.6 K	0.15	0.18
wiki-Vote	7 K	103 K	143	152	1.3 K	39.4 K	143	152	0.18	0.18
enron	69 K	276 K	796	4.9 K	8.2 K	147 K	781	4.8 K	0.58	0.69
email-EuAll	265 K	420 K	962	46.0 K	34 K	151 K	960	46.0 K	1.32	1.80
soc-Epinions1	76 K	509 K	8.4 K	23.5 K	32 K	443 K	8.1 K	20.9 K	1.07	1.20
soc-Slashdot0811	77 K	905 K	14.0 K	417	70 K	888 K	13.0 K	3	1.76	2.19
soc-Slashdot0902	82 K	948 K	14.2 K	501	71 K	912 K	14.1 K	69	1.79	2.35
amazon0302	262 K	1.2 M	71.9 K	75.7 K	241 K	11.3 M	69.6 K	73.3 K	3.55	4.77
web-NotreDame	325 K	1.4 M	13.8 K	61.8 K	54 K	304 K	9.6 K	31.9 K	2.48	3.27
uk-2007-05@100K	100 K	3 M	9.4 K	47.0 K	53 K	1.6 M	2.8 K	16.8 K	3.00	3.46
cnr-2000	325 K	3.2 M	32.5 K	104 K	112 K	1.6 M	14.6 K	44.1 K	4.28	5.08
amazon0312	400 K	3.2 M	69.5 K	83.2 K	380 K	3.0 M	69.0 K	82.6 K	11.37	12.40
amazon-2008	735 K	5.1 M	103 K	159 K	627 K	4.7 M	102 K	156 K	25.81	21.89
wiki-Talk	2.3 M	5.0 M	14.8 K	86.7 K	111 K	14.7 M	14.8 K	85.5 K	19.02	18.58
web-Google	875 K	5.1 M	102 K	267 K	434 K	3.4 M	89.8 K	211 K	13.59	15.48
web-BerkStan	685 K	7.6 M	108 K	297 K	334 K	4.5 M	53.6 K	164 K	9.91	12.15
in-2004	1.3 M	16.9 M	82.0 K	421 K	480 K	7.8 M	33.5 K	216 K	32.39	39.02
eu-2005	862 K	19.2 M	104 K	160 K	752 K	17.9 M	99.3 K	146 K	23.95	27.67
uk-2007-05@1M	1 M	41.2 M	147 K	415 K	593 K	22.0 M	82.5 K	259 K	20.90	24.51
soc-LiveJournal1	4.8 M	68.9 M	654 K	1.3 M	3.8 M	65.8 M	649 K	1.3 M	260.03	273.60
ljournal-2008	5.3 M	79.0 M	734 K	1.3 M	4.1 M	74.9 M	727 K	1.3 M	275.53	299.57
indochina-2004	7.4 M	194 M	774 K	2.2 M	3.8 M	98.8 M	503 K	1.4 M	155.83	192.06
uk-2002	18.5 M	298 M	2.3 M	6.1 M	12.0 M	232 M	1.8 M	4.8 M	404.92	478.13
arabic-2005	22.7 M	640 M	2.7 M	6.7 M	15.1 M	473 M	2.2 M	5.2 M	681.47	837.89

outside of the largest strongly connected component. Similar properties hold for strong bridges, with the only exception of few instances of social graphs, namely soc-Slashdot0811 and soc-Slashdot0902, where, differently from strong articulation points, more than 90% of the strong bridges lie outside of the largest strongly connected component.

In general, our experiments show that strong articulation points appear frequently in our datasets. Their actual frequency seems to depend on the type of the graph considered: product co-purchase graphs seem to have the highest percentage of strong articulation points (between 15% and 25% of their vertices are strong articulation points), followed closely by social graphs (where between 11% and 18% of the vertices are strong articulation points). Another interesting aspect is that in our dataset only the email graphs (such as email-EuAll) seem to have a large number of strong bridges, which are again mostly contained in the largest strongly connected component.

To further check what are the bottlenecks of the ILS algorithms and whether there could be room for further improvements, we broke down their running times into smaller subtasks. In particular, Figure 5 shows the results of such an experiment with ILS(SAP), where we measured the running time of (i) loading the graph G into memory; (ii) computing the strongly connected components of G; (iii) computing the dominators with LT and (iv) testing whether the root r is a strong articulation point (by simply checking whether the strongly connected

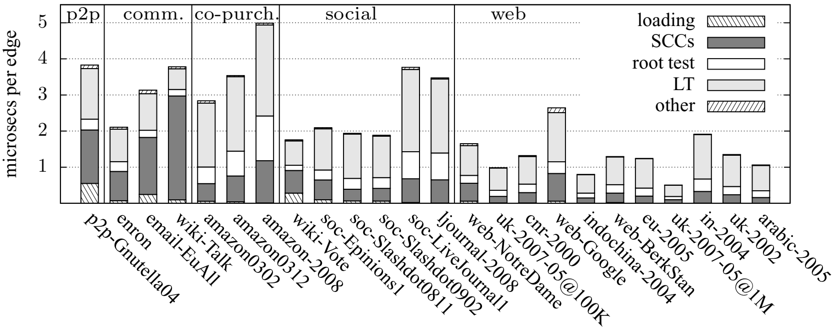


Fig. 5. Running times of ILS(SAP), broken down in the following subtasks: 1) loading the graph G into memory, 2) computing the strongly connected components of G, 3) computing the dominators with LT and 4) testing whether the root r is a strong articulation point. All the remaining running times are accounted for in a generic subtask called "other". The input graphs are partitioned according to their type (peer2peer, communication, product co-purchase, social and web graphs).

component containing r breaks down after the removal of r). Note that all the remaining running times are accounted for in a generic subtask called "other".

As shown in Figure 5, most of the graphs of the same type (peer2peer, communication, product co-purchase, social and web graphs) tend to share the same break down structure in their running times. As expected, for most of the graphs the bottleneck of ILS(SAP) appears to be the computation of dominators (via the LT algorithm), which nevertheless seems to be slower than the computation of the strongly connected components only by a small constant factor. On the other hand, checking whether the root is a strong articulation point is consistently faster than computing the strongly connected components of the entire graph. The most notable exception to this general behavior arises from the communication graphs, such as enron, email-EuAll and wiki-Talk. This happens since those graphs tend to have a much different structure from the other graphs in our dataset: indeed, they consist of one big strongly connected component, containing slightly more than 10% of the vertices, and a very large number of tiny strongly connected components (each having less than about 10 vertices). In such a case, LT and the root test run on much smaller inputs (i.e., the strongly connected components) than the entire graph.

Similar results are obtained for strong bridges. Figure 6 shows the results of such an experiment with ILS(SB), where we measured the running time of (i) loading the graph G into memory; (ii) computing the strongly connected components of G and (iii) computing the edge dominators. As with strong articulation points, all the remaining running times are accounted for in a generic subtask called "other". The main difference is that now the computation of edge dominators requires more time than the computation of dominators, as implied by Lemma 1. All those experiments seem to suggest that in order to obtain substantially faster ILS implementations, one should be able to produce faster codes for computing dominators.

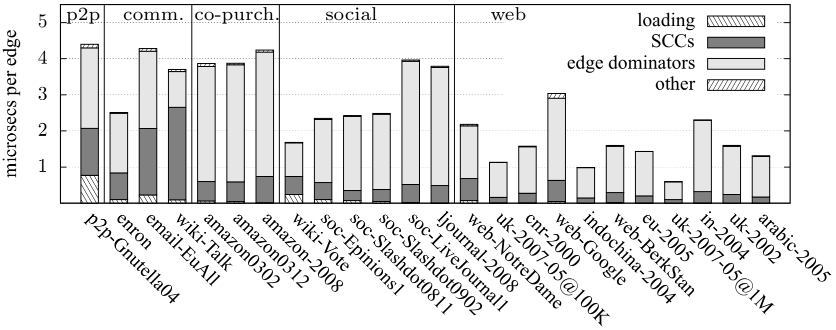


Fig. 6. Running times of `ILS(SB)`, broken down in the following subtasks: 1) loading the graph G into memory, 2) computing the strongly connected components of G and 3) computing the edge dominators. All the remaining running times are accounted for in a generic subtask called "other". The input graphs are partitioned according to their type (peer2peer, communication, product co-purchase, social and web graphs).

Table 4. For each graph, in this table we report, distinguished between the vertices (V) and the strong articulation points (sap), the average values of indegree (δ^-_{avg}), outdegree (δ^+_{avg}) and PageRank (PR_{avg}); in the last three columns we can see, respectively, the number of the strong articulation points in the main strongly connected components (sap_{scc}) and, amongst these, the one that belongs to the top 10% vertices sorted by indegree ($\delta^-(10\%)$) and by outdegree ($\delta^+(10\%)$).

	δ^-_{avg}		δ^+_{avg}		PR_{avg}		$\#sap_{scc}$	$\#sap_{scc}$ in	$\#sap_{scc}$ in
Graph	V	sap	V	sap	V	sap		$\delta^-(10\%)$	$\delta^+(10\%)$
`p2p-Gnutella04`	3.68	4.87	3.68	9.60	$9.19{\cdot}10^{-5}$	$1.12{\cdot}10^{-4}$	1.3 K	132	69
`enron`	3.99	62.79	3.99	103.48	$1.46{\cdot}10^{-5}$	$1.63{\cdot}10^{-4}$	781	24	8
`email-EuAll`	1.58	280.43	1.58	103.06	$3.80{\cdot}10^{-6}$	$4.29{\cdot}10^{-4}$	960	3	5
`wiki-Talk`	2.1	69.35	2.10	290.50	$4.18{\cdot}10^{-7}$	$2.86{\cdot}10^{-6}$	14.8 K	53	152
`amazon0302`	4.71	4.65	4.71	4.89	$3.82{\cdot}10^{-6}$	$3.42{\cdot}10^{-6}$	69.6 K	2497	4801
`amazon0312`	7.99	6.81	7.99	8.65	$2.50{\cdot}10^{-6}$	$2.06{\cdot}10^{-6}$	69.0 K	998	3628
`amazon-2008`	7.02	8.21	7.02	8.97	$1.36{\cdot}10^{-6}$	$2.05{\cdot}10^{-6}$	102 K	374	1825
`wiki-Vote`	12.5	79.29	12.50	68.91	$1.21{\cdot}10^{-4}$	$6.83{\cdot}10^{-4}$	143	3	11
`soc-Epinions1`	6.71	34.29	6.71	32.47	$1.32{\cdot}10^{-5}$	$5.83{\cdot}10^{-5}$	8.1 K	89	90
`soc-Slashdot0811`	11.7	38.73	11.70	39.90	$1.21{\cdot}10^{-5}$	$3.06{\cdot}10^{-5}$	13.0 K	1	3
`soc-Slashdot0902`	11.54	39.08	11.54	40.48	$1.16{\cdot}10^{-5}$	$2.92{\cdot}10^{-5}$	14.1 K	2	5
`soc-LiveJournal1`	14.23	35.89	14.23	35.47	$2.07{\cdot}10^{-7}$	$5.69{\cdot}10^{-7}$	649 K	3057	3113
`ljournal-2008`	14.73	38.46	14.73	37.56	$1.88{\cdot}10^{-7}$	$5.10{\cdot}10^{-7}$	727 K	3729	3666
`web-NotreDame`	4.6	14.13	4.60	13.63	$3.11{\cdot}10^{-6}$	$1.25{\cdot}10^{-5}$	9.6 K	797	478
`uk-2007-05@100K`	30.51	139.95	30.51	46.40	$1.01{\cdot}10^{-5}$	$4.32{\cdot}10^{-5}$	2.8 K	24	153
`cnr-2000`	9.88	27.59	9.88	16.79	$3.12{\cdot}10^{-6}$	$9.08{\cdot}10^{-6}$	14.6 K	1587	741
`web-BerkStan`	11.09	21.08	11.09	10.87	$1.46{\cdot}10^{-6}$	$3.36{\cdot}10^{-6}$	53.6 K	1640	4179
`web-Google`	5.57	17.80	5.57	9.24	$1.09{\cdot}10^{-6}$	$3.30{\cdot}10^{-6}$	89.8 K	5150	4037
`in-2004`	41.25	222.67	41.25	45.78	$1.01{\cdot}10^{-6}$	$5.06{\cdot}10^{-6}$	33.5 K	2641	2907
`eu-2005`	22.3	49.35	22.30	25.20	$1.16{\cdot}10^{-6}$	$2.99{\cdot}10^{-6}$	99.3 K	10441	12928
`uk-2007-05@1M`	12.23	41.71	12.23	19.61	$7.32{\cdot}10^{-7}$	$2.09{\cdot}10^{-6}$	82.5 K	5825	8360
`indochina-2004`	26.18	62.23	26.18	27.60	$1.37{\cdot}10^{-7}$	$4.09{\cdot}10^{-7}$	503 K	30252	38724
`uk-2002`	16.1	43.93	16.10	20.82	$5.48{\cdot}10^{-8}$	$1.43{\cdot}10^{-7}$	1.8 M	152773	183997
`arabic-2005`	28.14	82.68	28.14	34.96	$4.44{\cdot}10^{-8}$	$1.31{\cdot}10^{-7}$	2.2 M	139907	176214

To investigate further properties of strong articulation points, we computed average indegree, outdegree and PageRank of strong articulation points and of vertices in our graph instances. The result of such experiment is shown in Table 4. While in co-purchase graphs strong articulation points seem to have vertex degrees and PageRank close to average, there are more striking differences for the other graphs in our dataset. In particular, the indegree, outdegree and PageRank of the strong articulation points tend to be much higher than average in communication graphs and higher than average in social and web graphs.

Another application where strong articulation points may be handy is the identification of cores in directed social networks. As defined in [11], a core is a minimal set of vertices which are necessary for the connectivity of the network, i.e., removing vertices in the core breaks the remainder of the vertices into many small, disconnected strongly connected components. In recent work, Mislove et al. [11], following an approximation commonly used in web graph analysis [4], observed that after removing 10% of the highest indegree (or highest outdegree) vertices in a social graph, the largest strongly connected component will be split into smaller components. It seems thus natural to ask how many of the removed vertices are actually strong articulation points, i.e., how many of the removed vertices are really effective in splitting the largest strongly connected component. Table 4 reports the results of an experiment where we tried to answer this question. As shown in the last columns of Table 4, only few strong articulation points are selected by this process, which indeed seems to miss the vast majority of strong articulation points (roughly 95% on average). This gives some evidence that using strong articulation points in place of high degree vertices may provide a better approximation of the core of a directed graph.

Acknowledgments. We are indebted to Massimo Bernaschi for providing the computing infrastructure on which our experiments were run. We thank also Loukas Georgiadis for helpful discussions and for providing the source code of the implementation of `LT` and `semi-NCA` described in [8], and the anonymous referees for useful comments.

References

1. Alstrup, S., Harel, D., Lauridsen, P.W., Thorup, M.: Dominators in linear time. SIAM J. Comput. 28(6), 2117–2132 (1999)
2. Beldiceanu, N., Flener, P., Lorca, X.: The *tree* Constraint. In: Barták, R., Milano, M. (eds.) CPAIOR 2005. LNCS, vol. 3524, pp. 64–78. Springer, Heidelberg (2005)
3. Boldi, P., Vigna, S.: The WebGraph framework I: Compression techniques. In: Proc. 13th Int. World Wide Web Conference (WWW 2004), pp. 595–601 (2004)
4. Broder, A.Z., Kumar, R., Maghoul, F., Raghavan, P., Rajagopalan, S., Stata, R., Tomkins, A., Wiener, J.L.: Graph structure in the web. Computer Networks 33(1-6), 309–320 (2000)
5. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press (2009)
6. Gabow, H.N., Tarjan, R.E.: A linear-time algorithm for a special case of disjoint set union. Journal of Computer and System Sciences 30(2), 209–221 (1985)

7. Georgiadis, L.: Testing 2-Vertex Connectivity and Computing Pairs of Vertex-Disjoint s-t Paths in Digraphs. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010. LNCS, vol. 6198, pp. 738–749. Springer, Heidelberg (2010)
8. Georgiadis, L., Tarjan, R.E., Werneck, R.F.F.: Finding dominators in practice. J. Graph Algorithms Appl. 10(1), 69–94 (2006)
9. Italiano, G.F., Laura, L., Santaroni, F.: Finding strong bridges and strong articulation points in linear time. Theoretical Computer Science (to appear), doi: `http://dx.doi.org/10.1016/j.tcs.2011.11.011`
10. Lengauer, T., Tarjan, R.E.: A fast algorithm for finding dominators in a flowgraph. ACM Trans. Program. Lang. Syst. 1(1), 121–141 (1979)
11. Mislove, A., Marcon, M., Gummadi, K.P., Druschel, P., Bhattacharjee, B.: Measurement and analysis of online social networks. In: Proc. 7th ACM SIGCOMM Conference on Internet Measurement, IMC 2007, pp. 29–42 (2007)
12. SNAP: Stanford Network Analysis Project, `http://snap.stanford.edu/`
13. Tarjan, R.E.: Edge-disjoint spanning trees, dominators, and depth-first search. Technical report, Stanford, CA, USA (1974)
14. Tarjan, R.E.: Edge-disjoint spanning trees and depth-first search. Acta Inf. 6, 171–185 (1976)
15. Volkmann, L.: Restricted arc-connectivity of digraphs. Inf. Process. Lett. 103(6), 234–239 (2007)
16. The WebGraph Framework Home Page, `http://webgraph.dsi.unimi.it/`

Adaptive Distributed b-Matching in Overlays with Preferences[⋆]

Georgios Georgiadis and Marina Papatriantafilou

Department of Computer Science and Engineering,
Chalmers University of Technology, S-412 96 Göteborg, Sweden
{georgiog,ptrianta}@chalmers.se

Abstract. An important function of overlay networks is the facilitation of connection, interaction and resource sharing between peers. The peers may maintain some private notion of how a "desirable" peer should look like and they share their bounded resources with peers that they prefer better than others. Recent research proposed that this problem can be modeled and studied analytically as a many-to-many matching problem with preferences. The solutions suggested by the latter proposal guarantee both algorithmic convergence and stabilization, however they address static networks with specific properties, where no node joining or leaving is considered. In this paper we present an adaptive, distributed algorithm for the many-to-many matching problem with preferences that works over any network, provides a guaranteed approximation for the total satisfaction in the network and guarantees convergence. In addition, we provide a detailed experimental study of the algorithm that focuses on the levels of achieved satisfaction as well as convergence and reconvergence speed. Finally, we improve, both for static and dynamic networks, the previous known approximation ratio.

1 Introduction

Overlay networks play an increasingly important role in today's world: from social networks to ad hoc communication networks, people and machines connect, interact and share resources through novel overlay networks laid over Internet's infrastructure or other communication substrate. Unstructured overlays in particular aim at connecting peers with minimal assumptions on the protocols to be used, while addressing universal challenges: peers are willing to share both concrete resources, such as bandwidth, and abstract ones, such as attention span, but these resources are naturally limited and usually the contributors expect something in return. As an example, in a generic, fully distributed scenario each peer may rate its neighbors according to one or more individual metrics (e.g. distance, interests, available resources) but choose to connect to only a handful of them due to its own resource scarcity. The challenge in this scenario is to maintain a high level of the

[⋆] Work supported by the European Union 7th Framework Programme (FP7/2007-2013) under grant agreement No. 257007 and the Swedish Foundation for Strategic research (SSF), grant nr. IMF11-0051.

R. Klasing (Ed.): SEA 2012, LNCS 7276, pp. 208–223, 2012.

implemented service on a network scale, while at the same time adapt to and tolerate the high dynamicity commonly found in these networks, with peers leaving, joining or changing ratings about their neighbors at any time.

This kind of connection problem with limited resources and reciprocal relations between contributors gives rise to a natural modeling using undirected graphs where nodes have limited (but different) connection capacities. It is essentially a form of a *matching problem* in a graph, where nodes must be matched one to one with some neighbor of theirs in a maximal way on the graph level. The particular form of matching which is relevant here is *many-to-many matching with preferences* (commonly referred to as *stable fixtures* [1] or *b-matching with preferences* [2] problem), where each node maintains a preference list of its neighbors (rated from most to least preferable) and a total quota of desired connections b_i for each node i. The goal for each node in this setting is to be able to form the desired amount of connections with the highest quality (most preferred) neighbors.

Although current literature includes efficient algorithms for many-to-many matching with preferences that can produce a stable configuration if one exists, recent research showed [2] that the problem does not always admit stable solutions. Furthermore, most suitable algorithms from literature are centralized and cannot be deployed in a distributed setting such as the unstructured overlay networks considered here. In addition to the above, none of the currently available distributed algorithms for the b-matching problem can handle the dynamic aspect of overlay networks (e.g. node arrivals/departures and preference changes). Concrete examples of previous work, along with their relation to the present study, can be found in the subsection below.

In this paper we focus on the problem of adaptive, distributed many-to-many matching problem with preferences. Using the metric of node *satisfaction* [2] that can be used for measuring the quality of a node's connections, we build on earlier work that modeled the problem from an optimization perspective. The contribution is threefold:

(i) We show an improved approximation ratio, that can also be applied to existing algorithms and imply improved bounds for their satisfaction guarantees.

(ii) We propose an adaptive, distributed algorithm for the problem, which guarantees that the calculated solution maximizes the total satisfaction in the network within the newly shown bound: an approximation of $\frac{1}{4}\left(1 + b_{\max}^{-1}\left(1 + \frac{1-b_{\max}^{-1}}{2s+b_{\max}^{-1}}\right)\right)$ in every case, where $b_{\max}$ is the maximum connection quota in the graph and $s = \frac{L_{\max}}{L_{\min}}$ is the ratio of the maximum and minimum neighbor list sizes in the graph, $L_{\max}$ and $L_{\min}$ respectively. To the extend of our knowledge it is the first algorithm that can handle dynamicity while solving the many-to-many matching problem, with node joining, leaving and preference changes fully supported. Its key features include the use of only local information, a given approximation bound and a guaranteed convergence once the changes complete.

(iii) We also provide a extensive experimental study of the behavior of the algorithm under a variety of scenarios, including normal operation but also operation under high stress. Under normal operation, we focus on the levels of achieved satisfaction as well as convergence and reconvergence speed. Specifically, we show that the resulting satisfaction is high but also remains on high levels during and after reconvergence, while reconvergence is achieved in an efficient way under a variety of changes. Besides, motivated by [3,4] we conducted experiments that focus on the stability of the network under join/leave attacks, by exposing it to high churn rates, and observed that it withstands the attacks while maintaining graceful satisfaction values throughout them.

1.1 Related Work

Matching problems are well studied in their centralized form and an extensive literature exists, including solutions for the many-to-many variants (cf for example [1,5,6,7]). However, it has been shown [8] that exact solutions of even simple matching problems cannot be derived locally in a distributed manner, leading to a significant research interest for approximation distributed algorithms [9,10,11]. Prominent examples of this research area are the one-to-one weighted matching algorithms of Manne et al. [12] and Lotker et al. [10,13], with the former having proven self-stabilization properties and the latter having variants that can handle joins and leavings of nodes. However, whether it is possible to extend these techniques to many-to-many matchings remains an open research question. On the other hand, Koufogiannakis et al. [14] proposed a randomized δ-approximation distributed algorithm for maximum weighted b-matching in hypergraphs (with $\delta = 2$ for simple graphs) but it addresses only static graphs and its elaborate nature makes its extension to support a dynamic setting non-trivial.

Additionally to the approaches above, there is an extensive research focus on many-to-many matchings with preferences [2,15,16,17]. First Gai et al. in [16] proved that in the case of an acyclic preference system there is always a stable configuration, and also supplied examples of preference systems based on global or symmetric metrics. Mathieu in [2] introduced the measure of node satisfaction as a metric aimed to describe the quality of the proposed solutions. Georgiadis et al. in [18] modeled the b-matching with preferences problem as an optimization problem that uses satisfaction to achieve convergence. The authors showed an approximation is possible through the reduction of the original problem to a many-to-many weighted matching problem. However, the proposed algorithm is only suitable for static networks, since it cannot adapt to and guarantee convergence after changes in the topology of the network (joins/leavings) or nodes' preferences. Previously Lee [17] had used a similar *credit* metric in order to optimize the proposed solutions from their heuristic algorithms.

2 Problem Definition and System Model

In this paper we use standard terms and notions from the literature [1,2,13,18]) which we briefly describe here for self-containment. We represent an overlay network as an undirected graph $G(V, E)$ with $|V| = n$, $|E| = m$, where V is the set of overlay peers and E the set of potential connections. Each node i has degree d_i and keeps a preference list L_i of all nodes in its neighborhood Γ_i[1]. Let $R_i(j)$ denote the rank of node j in node i's preference list, with $R_i(\cdot) \in \{0, 1, \dots, |L_i| - 1\}$, attributing 0 to its most desirable neighbor. Each node i wants to maintain at most b_i connections to the best possible nodes according to its preference list and rank function, and at no point it can exceed this number. In the following sections we will refer to two nodes as *neighboring nodes* when they are connected by an edge in graph G and *connected* or *matched nodes* when they are matched by a matching algorithm. The problem of trying to find a many-to-many matching that respects the individual preferences and connection quotas b_i is a form of a generalized stable roommates problem called the *stable fixtures problem* [1] or *b-matching* [2]. We call *adaptive b-matching* the dynamic form of b-matching, where nodes can join, leave or change preferences at any time. In the remaining of this paper we will refer to these events simply as *changes.* We will also consider an asynchronous model for messages and will not consider link or node failures, i.e. messages arrive asynchronously but do not get lost and nodes depart gracefully or their absence can be detected by other means (for example special periodic "alive" messages).

In order to measure the success of a node i's efforts in establishing its b_i connections, we make use of the notion of *satisfaction* S_i (defined in [2] and analyzed in [18]) to be equivalent to the following:

$$S_i = \frac{c_i}{b_i} - \frac{\sum\limits_{j \in C_i} \left(R_i\left(j\right) - Q_i\left(j\right)\right)}{b_i L_i} \tag{1}$$

where C_i (with $|C_i| = c_i \leq b_i$) is an ordered list of node i's established connections in decreasing preference and $Q_i\left(j\right)$ is the rank of node j in the connection list C_i of node i. According to the above formula, S_i takes values between 0 and 1, depending on how many and which connections a node has formed. A satisfaction value 1 is achieved by a node that has formed all b_i desired connections with its b_i most preferable neighbors, while a penalty is inserted for each non-optimal connection that it forms. Note that we can write formula 1 as:

$$S_i = \sum_{j \in C_i} \left(\frac{L_i - R_i\left(j\right)}{b_i L_i} \right) + \sum_{j \in C_i} \left(\frac{Q_i\left(j\right)}{b_i L_i} \right) = \sum_{j \in C_i} S^s_{i,j} + \sum_{j \in C_i} S^d_{i,j} = S^s_i + S^d_i \tag{2}$$

We refer to the quantities S^s_i and S^d_i as the *a priori part* and the *a posteriori part* of node i's satisfaction respectively, as the former summation terms are computable for each j regardless of whether it is matched with i or not, while

[1] In the rest of the paper and when it is clear from context, we will use notation L_i to denote both the list and its length.

the latter summation's terms are computable only for those j that are matched with i by the solution.

The Truncateds Maximizing Satisfaction b-Matching Problem. Georgiadis et al. in [18] modeled the b-matching problem as an optimization problem that uses satisfaction to achieve convergence, and defined the problem of maximizing the total sum of node satisfaction as the *maximizing satisfaction b-matching* problem. That paper showed that an approximation to the original problem is possible by forming edge weights $w(i,j)$ using only both endpoints' marginal a priori part of satisfaction $S^s_{i,j}$ and $S^s_{j,i}$,

$$w\left(i,j\right) = S^s_{i,j} + S^s_{j,i} = \left(\frac{L_i - R_i\left(j\right)}{b_i L_i}\right) + \left(\frac{L_j - R_j\left(i\right)}{b_j L_j}\right). \tag{3}$$

When the above approximation is used for satisfaction calculations, the satisfaction for each node i is essentially computed by using only the a priori part of satisfaction in formula 2. The resulting problem, called here *truncatedS maximizing satisfaction b-matching* problem, has been proven equivalent to a *many-to-many weighted matching* problem with edge weights as defined in formula 3 [18]. Note here that a simple weighted matching problem is defined as the problem of finding a set of edges whose weight sum is maximized and which have no common endpoints between them. The many-to-many variant used here replaces the constraint on no common endpoints with node capacities that need to be respected, in this case the connection quotas b_i per node i.

In the analysis of the algorithm we will also use the notion of a *locally heaviest edge* [19]. Let E_{ij} be the set of edges having either of nodes i and j as an endpoint (but not both):

$$E_{ij} = \left\{\left(i,n_i\right)|n_i \in \Gamma_i \backslash j\right\} \cup \left\{\left(j,n_j\right)|n_j \in \Gamma_j \backslash i\right\} \tag{4}$$

An edge (i,j) is called locally heaviest if it has the greatest weight among all edges $e \in E_{ij}$:

$$w\left(i,j\right) > w\left(e\right), e \in E_{ij} \tag{5}$$

3 Improved Approximation Ratio

Before proceeding to the presentation and analysis of the algorithm, we show an improved approximation ratio for the problem under study. As shown in the proof (due to space limitations, it can be found in [20]), the new ratio applies both for the static case of [18] and the dynamic case studied here.

Theorem 1. *The truncatedS maximizing satisfaction b-matching problem is a* $\frac{1}{2}\left(1 + b^{-1}_{\max}\left(1 + \frac{1-b^{-1}_{\max}}{2s+b^{-1}_{\max}}\right)\right)$*-approximation of the maximizing satisfaction b-matching problem, where* $b_{\max}$ *is the maximum connection quota in the graph and* $s = \frac{L_{\max}}{L_{\min}}$ *is the ratio of the maximum and minimum neighbor list sizes in the graph,* $L_{\max}$ *and* $L_{\min}$ *respectively.*

The above theorem does not make any assumptions on whether the problem/-graph is static or dynamic. Hence it leads to an improved approximation bound for the Local Information-based Distributed (LID) algorithm in [18] that solves the b-matching with preferences problem.

Corollary 1. *The* LID *algorithm solves the b-matching with preferences problem with* $\frac{1}{4}\left(1 + b_{\max}^{-1}\left(1 + \frac{1-b_{\max}^{-1}}{2s+b_{\max}^{-1}}\right)\right)$*-approximation.*

4 Adaptive Matching Algorithm

The adaptive matching algorithm builds on the modeling of the Local Information-based Distributed (LID) algorithm [18] and utilizes the notion of node satisfaction to optimize the matching; for self-containment we summarize it here. The core idea of the LID algorithm is that every node maintains a preference list of all its neighbors and regards every potential connection as able to give a fraction of satisfaction, amounting to 1 for a full connection quota with top choices or less in the case of sub-optimal choices (0 for no connections). The network optimization goal we are considering is to maximize the total sum of individual node satisfaction while respecting individual node preferences and connection quotas. During initialization of the algorithm every node i exchanges approximated marginal a priori parts of satisfaction scores $S_{i,j}^{s}$ with its neighbors j and forms edge weights $w(i, j)$ using the scores it receives. Already at this point, the first approximation over the original maximizing satisfaction many-to-many matching is being employed by using the approximated form of marginal satisfaction. Using the resulting weights for the matching effectively converts the original problem into a maximum weight many-to-many matching problem, which the nodes proceed to solve by choosing greedily only locally heaviest edges. This second approximation (i.e. choosing locally heaviest edges instead of globally heaviest ones) along with the first one jointly lead to a $\frac{1}{4}\left(1 + b_{\max}^{-1}\left(1 + \frac{1-b_{\max}^{-1}}{2s+b_{\max}^{-1}}\right)\right)$-approximation solution of the original problem but also lead to valuable properties for the algorithm: on the one hand to a simple, fully distributed scheme and on the other to provable termination even when cycles are present among node preferences (cf lemma 5 in [18]).

In a dynamic setting, where nodes join/leave the network or change preferences about their neighbors at any time, there is a partial or full solution that is disturbed by a specific operation. In this case it is desirable to "repair" the solution instead of recomputing it from the beginning. It would also be advantageous to limit the repairs to the neighborhood of the operation, so that far enough nodes would remain unaffected. Note that the locally-heaviest-edge property that we are using here seems ideal for this purpose: it only makes sense to preserve and use it further to support dynamicity.

In the adaptive algorithm AdaptiveLID presented here, all three cases of dynamicity mentioned above (join/leave/change) are supported. In the case of a joining (resp. departing) node, neighboring nodes add (resp. delete) it to (resp. from) their preference lists. On the other hand, when a node changes preferences no change occurs to the neighboring nodes' preference lists but edge weights may change radically. A common thread between these cases is that the nodes directly involved in the operations must recalculate their marginal satisfactions for their neighbors and exchange them so that their adjacent edges have the correct weights. Afterwards, they must re-evaluate their connections: if they are not locally heaviest any more, the nodes abandon the least weighted ones and try to get matched with the locally heaviest ones. Note here that this method avoids the recalculation of the solution over the whole network, instead limiting it to a neighborhood around the network area where the dynamic operation took place. An additional benefit is that the involved nodes maintain their current connections unless proven to be non-optimal, i.e. they change them only if necessary.

In literature, many of the algorithms for matching with preferences are inspired by the proposal-refusal algorithm of Gale and Shapley [6]. This also the case with AdaptiveLID but it addresses a different problem with unique characteristics: while the Gale-Shapley algorithm is focused on absolute stability, AdaptiveLID solves an optimization problem and aims for the maximum possible satisfaction. So, for example, it is important to guarantee that no cycles exist in the case of Gale-Shapley algorithm since, given the distributed nature of the algorithm, a reply may not be possible to be given immediately by a node to another node's proposal. This is not necessary in the case of AdaptiveLID since any cycles in preference orders are broken by reducing the original problem to an acyclic weighted many-to-many matching, upon which the algorithm operates (cf also lemma 3). On the other hand, it is important for the AdaptiveLID algorithm to tolerate and work under changes in the underlying network and by focusing on optimization it achieves exactly that (cf lemma 5).

The AdaptiveLID algorithm uses at each node i five sets $(P_i, K_i, A_i, R_i, B_i)$ and an incoming message queue $queue_i$, and sends three kinds of messages (PROP, REJ and WAKE):

- A node i sends PROP messages to propose to its heaviest-weight neighbors the establishment of a connection. If an asked node also sends a PROP message to node i then the connection is established (*locked*): note that this will happen in both endpoints. Set P_i stores the neighbors to which node i proposed with a PROP message, A_i stores the neighbors which approached node i with a PROP message, K_i stores the locked neighbors, B_i stores the neighbors that rejected node i and R_i the neighbors that node i rejected. Sets B_i^* and A_i^* are copies of sets B_i and A_i respectively that do not contain neighbors of edges heavier than the edge of the worst connected neighbor.
- A node sends a REJ message when it has locked as many neighbors as it could. Nodes can send additional PROP messages to available neighbors if they receive a REJ message. PROP messages are sent to neighbors in decreasing ranking order and there are at most b_i such unanswered messages originated from i at any time.
- Node i is constantly checking if its PROP messages are addressed to heaviest-weight neighbors, as ranking can change due to a change in the network. If it detects a better available node than the currently proposed ones, it sends a REJ message to the worst connected neighbor and a PROP to the better candidate. However, if the better candidate has simultaneously rejected and been rejected by node i, node i sends only a WAKE message.

Algorithm 1. AdaptiveLID()

```
ReceiveMsgs()
SendMsgs()
BookkeepingUpdates()
```

Procedure 1. ReceiveMsgs()

```
for $msg \in queue_i$ do
    if $msg.type = PROP$ then
        $A_i \leftarrow A_i \cup msg.sender$
        $B_i \leftarrow B_i - msg.sender$
    if $msg.type = REJ$ then
        $B_i \leftarrow B_i \cup msg.sender$
        $A_i \leftarrow A_i - msg.sender$
        $K_i \leftarrow K_i - msg.sender$
        $P_i \leftarrow P_i - msg.sender$
    if $msg.type = WAKE$ then
        $B_i \leftarrow B_i - msg.sender$
```

Procedure 2. SendMsgs()

```
while $(|\Gamma_i - P_i - (B_i - R_i)| \neq 0) \wedge (|P_i| < b_i)$ do
    find heaviest edge neighbor $c$ that belongs
    in $(\Gamma_i - P_i - (B_i - R_i))$
    if $c \neq null$ then
        if $c \in B_i$ then
            send a WAKE msg to $c$
            $R_i \leftarrow R_i - c$
        else
            send a PROP msg to $c$
            $P_i \leftarrow P_i \cup c$
            $R_i \leftarrow R_i - c$
```

Function 1. GetWorstNode(node i)

```
return $\left\{ l : w(l,i) = \min_{j \in P_i} w(j,i) \right\}$
```

Function 2. GetBestNode(node i)

```
return $\left\{ h : w(h,i) = \max_{j \in (\Gamma_i - P_i - (B_i - R_i))} w(j,i) \right\}$
```

Procedure 3. BookkeepingUpdates()

```
$T_i \leftarrow (P_i - K_i) \cap A_i$
if $|T_i| \neq 0$ then
    $A_i \leftarrow A_i - T_i$
    $K_i \leftarrow K_i \cup T_i$
    match node $i$ to all nodes in $T_i$
if $(|\Gamma_i - P_i - (B_i - R_i)| \neq 0) \wedge (|P_i| \neq 0) \wedge (|K_i| \neq 0)$
then
    $l \leftarrow$ GetWorstNode($i$)
    $h \leftarrow$ GetBestNode($i$)
    while $(l \neq null) \wedge (h \neq null)$ do
        if $w(h,i) > w(l,i)$ then
            if $h \in B_i$ then
                send a WAKE msg to $h$
                $R_i \leftarrow R_i - h$
            else
                send a REJ msg to $l$
                $A_i \leftarrow A_i - l$
                $R_i \leftarrow R_i \cup l$
                $P_i \leftarrow P_i - l$
                $K_i \leftarrow K_i - l$
                send a PROP msg to $h$
                $P_i \leftarrow P_i \cup h$
                $R_i \leftarrow R_i - h$
            $l \leftarrow$ GetWorstNode($i$)
            $h \leftarrow$ GetBestNode($i$)
        else if $P_i = K_i$ then
            for $j \in (\Gamma_i - R_i - B_i^* + A_i^* - P_i)$ do
                send a REJ msg to $j$
                $A_i \leftarrow A_i - j$
                $R_i \leftarrow R_i \cup j$
            break
        else
            break
unmatch node $i$ from all nodes in $B_i$
```

5 Analysis

The following lemmas (omitted proofs can be found in [20]) prove that the algorithm always converges after a finite amount of steps or, in the case of changes in the network, in a finite amount of steps after the changes stop. Although implied by the distributed nature of the algorithm, it is useful to note that the algorithm continues to run at all nodes regardless of any changes that are happening in the network. In fact, as we show in the experimental section, it manages to maintain a reduced but steady level of service while under extremely heavy stress or possibly a network attack. However, convergence can be guaranteed after all changes complete since any changes that might occur require appropriate readjustment by the distributed algorithm.

Lemma 1. *In a failure free execution, edge weight updates that are caused by node or preference changes complete in a finite amount of time.*

We define as *available* with respect to node i, a node j in the neighborhood of node i that has neither been proposed by node i nor rejected node i.

A node j is a *locally heaviest node* in the neighborhood of node i at some point in time if there are no available nodes that are endpoints of heavier edges. Note that when the endpoints of an edge consider simultaneously each other locally heaviest, the edge between them is a locally heaviest edge.

Lemma 2. *In a finite amount of time after a node or preference change, every node cancels all proposals towards neighbors that are no longer locally heaviest and issues an equal amount towards available neighbors that are locally heaviest.*

Lemma 3. *The* AdaptiveLID *algorithm terminates for every node $i \in V$ after changes complete.*

Lemma 4. *For every node i, algorithm* AdaptiveLID *chooses all locally heaviest edges that are adjacent to it, if there is enough quota b_i available, or otherwise chooses b_i of them that are heavier than any unchosen one.*

Lemma 5. *The* AdaptiveLID *algorithm when run on a network with changes produces the same matching with the* LID *algorithm that is run on the same network after the changes complete.*

From lemma 5 and theorem 1, as well as lemma 2 and theorem 2 of [18], we get the following theorem about the approximation ratio of AdaptiveLID:

Theorem 2. *The* AdaptiveLID *algorithm solves the adaptive b-matching with preferences problem with $\frac{1}{4}\left(1 + b_{\max}^{-1}\left(1 + \frac{1-b_{\max}^{-1}}{2s+b_{\max}^{-1}}\right)\right)$-approximation.*

Observation 1. *Following the main argument of the classic Chandy and Misra [21] Drinking Philosophers algorithm, we observe that the convergence complexity is bounded by the longest increasing edge weight path in the network.*

6 Experimental Study

The following extensive experimental study complements the preceding analytical part with useful observations and conclusions about the behavior of the AdaptiveLID algorithm in a variety of scenarios. Focus was given on the performance of the algorithm in regard to the following points:

- Behavior on different types of networks
- Behavior during different operations (joins, leaves, preference changes and churn)
- Convergence and reconvergence times

- Satisfaction levels, both in normal operation and under heavy stress (i.e. during a network-level attack)
- Fairness properties of satisfaction-based optimization

The following experiments were conducted using the PeerSim [22] platform in a synchronous way, i.e. execution proceeded in rounds, where each node in each round made a receive-respond-process step, unless it had nothing to execute. This synchronous execution mode is not necessary for the algorithm but it is used here to measure the time needed by the AdaptiveLID algorithm to converge. For every network instance used, a matching was calculated and the following operations were performed on a varying amount of nodes (1% to 50% of the network size, in increments of 1%): *join/leave*, where nodes enter/exit the network simultaneously, *preference change*, where existing nodes change the ranking of their neighbors in their preference lists simultaneously, and *churn*, where existing nodes exit and an equal amount of new nodes enter the network simultaneously. For the first three cases the network was left to reconverge after one operation, while in the case of churn the operation was repeated for several rounds before the network was left to reconverge. Each of these operations was conducted on networks of size $n = 100, 250, 500, 750$ and 1000 nodes, considering 30 network instances for each size, and the results presented here are mean values over these instances.

6.1 Network Types

The networks used during the experiments were power-law and random networks, created with the Barabási-Albert (*BA*) [23] and Erdős-Rényi (*ER*) [24] procedures respectively. These networks were selected for their different node degree distributions: in power-law networks the vast majority of nodes has very low degree and few nodes have very high degree (power-law distribution), while in ER random networks all nodes have comparable degrees (binomial distribution). In fact, high degree nodes in BA networks are connected mostly with many low degree ones, which leads to the creation of very different neighborhoods around individual nodes for these two network types. This difference, coupled with the algorithm's ability to perform local repairing operations, leads to the varying behaviors that can be seen in the experiments below.

On the other hand, both network types had node preferences formed uniformly at random since previous research [2,16] showed that (a) a strict matching solution can not always be found when they are used and (b) the measured satisfaction of unconverged instances can be relatively low. These characteristics make random preferences the challenging test case to evaluate the performance of the algorithm.

6.2 Convergence and Reconvergence

The mean value and standard deviation of convergence speed for a variety of network sizes can be found in figure 1. It is easy to see that the convergence speed depends on the type of the network. For example, BA networks of size 1000 take almost twice the amount of time to converge than networks of size 100, while ER networks of size 1000 need less than twice the amount of time needed by networks of size 100 and only slightly higher amount of time than the networks of size 500 and 750.

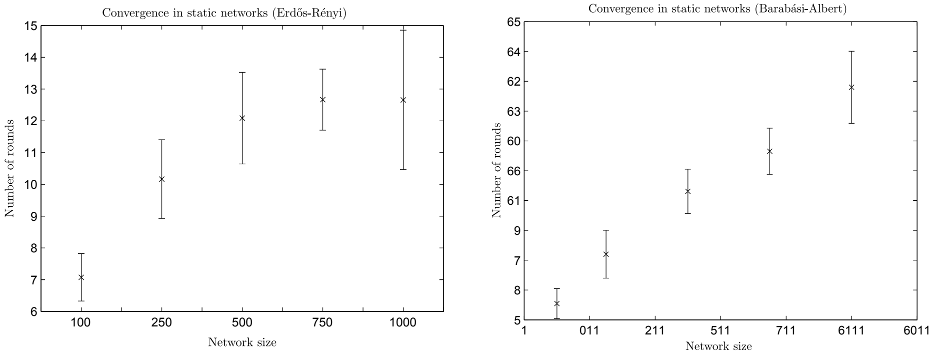


Fig. 1. Convergence speed per network size

Likewise, the time needed for reconvergence can be seen in figures 2 and 3, for networks of size 1000 of both types and for the four types of operations under consideration. By focusing on low percentages of affected nodes (i.e. up to 20% of the network size, which is a high volume of change), it is easy to see that reconvergence is obtained in most cases for a *fraction* of the rounds needed for initial convergence. For join and leave operations reconvergence is expressed not in rounds but in relation to the convergence time, since network sizes change significantly. Note here that this extreme change in network sizes leads in some cases to percentages greater than 100%, i.e. more rounds are needed for the reconvergence than for the initial convergence. For the preference change and churn operations this is not the case: the network size remains the same either because no node joins or leaves (preference change case) or the amount of nodes joining and leaving is the same (churn case) and the reconvergence time is expressed is rounds.

In the case of join operations, nodes arrive at the network and want to join the already established equilibrium of connections by being more attractive choices for some of the nodes they are neighbors with. This creates cascade effects of nodes rejecting old connections in favor of the newcomers, the rejected nodes trying to repair their lost connections and so on. Naturally, the more nodes wanting to join the network the bigger upheaval is created. A similar effect is generated during leave operations, where previously rejected nodes suddenly become attractive choices for nodes that were left behind by departing nodes. Note here that the BA networks reconverge much faster than the corresponding ER networks in the case of join operations. This happens because new nodes (being of low degree) connect preferentially to relatively few high degree nodes, limiting the extension of the upheaval in the network. In the case of leave operations the same behavior poses a challenge since departing nodes may happen to be of high degree themselves, leaving behind a lot of low degree nodes to repair their connections. Notice though that in both cases (ER or BA networks), when a substantial percentage of the network departs (i.e. above 35%) the remaining nodes repair their connections much easier since they have more unformed connections than established ones.

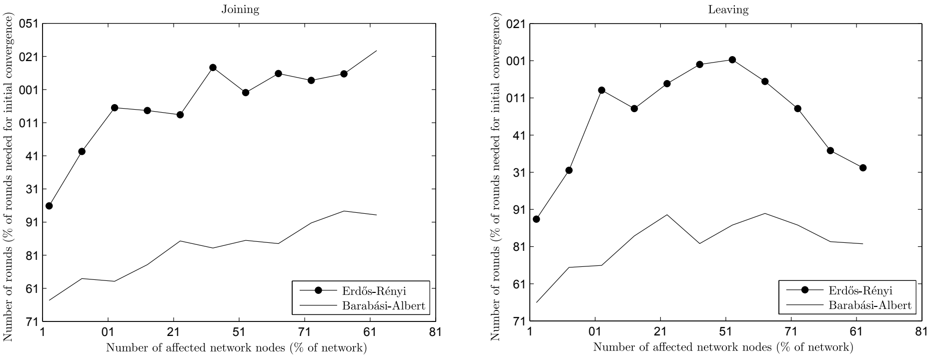


Fig. 2. Reconvergence speed per operation, joins/leaves, $n = 1000$

Preference change affects both network types in the same way: a node that changes preferences destroys some connections, creating waves of changes in its neighborhood. For the case of BA networks, it may happen that a node changing preferences is a high degree one, causing a lot of nodes to repair their connections. However, this effect dies off quickly, since most of its neighbors are of low degree, leading to an overall performance similar to the ER case.

The two network types show their differences more prominently under churn (figure 3). For the ER networks, a joining node under churn can be seen as a “reincarnation” of a leaving node with all its previous connections dropped and its preferences changed, since both of them have comparable node degrees. However, the churn operation is detrimental for the BA network since the joining nodes are of low degree and the departing ones of potentially much higher degree. As a result the degree distribution itself is changing, leading to higher reconvergence times (cf join operation).

In both cases though, by comparing the churn and preference change graphs in figure 3 it is easy to see that, somewhat counterintuitively, it takes progressively more time for the algorithm to reconverge when more nodes change preferences but the reconvergence time stays more or less the same even for high values of churn or it is consistently lower than preference change, as is the case in BA networks.

The reason behind this behavior is that in churn situations there are more parallel events taking place: a new, joining node that replaces a leaving one starts as an empty slate and sends an amount of PROP messages equal to the desired number of connections. On the other hand, a node that changes preferences might need to repair only some of its connections (which are now suboptimal) by

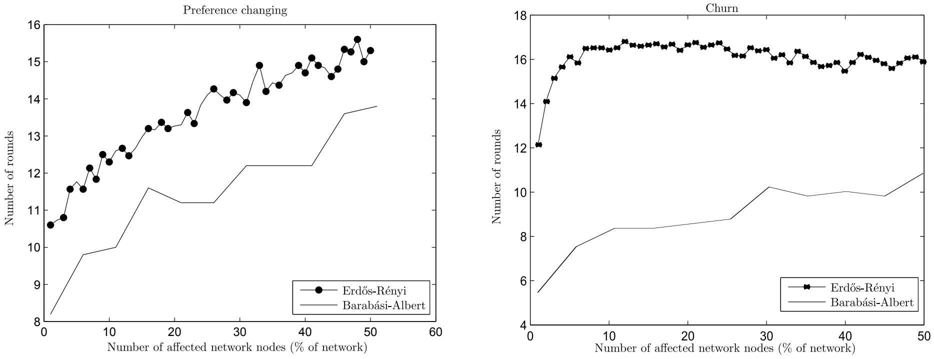


Fig. 3. Reconvergence speed per operation, preference change/churn, $n = 1000$

sending appropriate PROP messages. However, in both cases some responding nodes might decline, which will lead to additional PROP messages to be send and so on, until the issuing nodes are satisfied or no available nodes are left.

One may even wish to compare the two situations from the point of view of what is a desirable action by a node who changes preferences: to improve existing connections or to perform a leave and come back (thus contributing to churn). This is especially meaningful in the case of ER networks, since for BA networks churn is consistently cheaper in any case due to their special structure and the parallelism mentioned above. In the case of ER networks, comparing the reconvergence times of the two situations, churn has the advantage over preference change in high values. This is only natural since in that case more nodes start with no connections and all possibilities are explored in parallel. In contrast, having high values of preference change means that more nodes want to repair their connections but other nodes have already connections that they want to maintain, leading to longer times of reconvergence. It could be useful in practical terms if there was a mechanism able to detect high volume of preference changes in the network and enforce a policy of pseudo-churn, with nodes dropping all connections when changing preferences. However, as it is shown below, the amount of satisfaction under churn is far less than the satisfaction under preference change before reconvergence, which is a significant argument in favor of improving connections instead of dropping them and starting again.

6.3 Satisfaction

The mean satisfaction in the network achieved by the AdaptiveLID algorithm for a variety of network sizes can be found in figure 4, along with the values of minimum and maximum satisfaction in the network. Note that satisfaction is slightly lower in the case of BA networks due to differences in topology (i.e. minimum satisfaction is lower due to the large amount of low degree nodes) but it follows the same behavior as in the ER case. It is easy to see that the AdaptiveLID algorithm achieves consistently high satisfaction values, which are also increasing as network sizes increase. Of particular interest is that (a) the minimum satisfaction in the network is being increased also, meaning that individual nodes enjoy high levels of satisfaction too, implying asymptotically improved *fairness* properties as well and (b) the minimum satisfaction does not affect significantly the mean satisfaction, which implies that the number of nodes having low satisfaction is consistently very low compared to the size of the network.

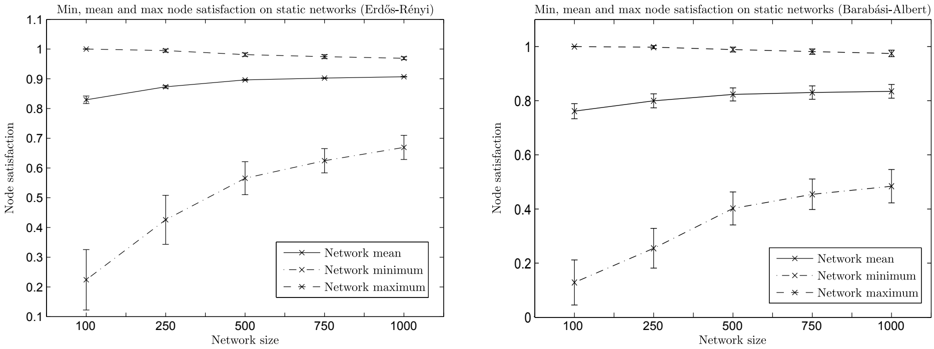


Fig. 4. Satisfaction per network size

Even though the reconvergence results showed that the algorithm can efficiently repair its solution once churn stops, it is interesting to see the levels of achieved satisfaction while churn is in progress. The relative satisfaction for ER networks under churn (to the one achieved before churn starts) can be found in figure 5: the different graphs from top to bottom correspond to the relative satisfaction when churn affects 5% to 50% of the network's nodes (in steps of 5%), for

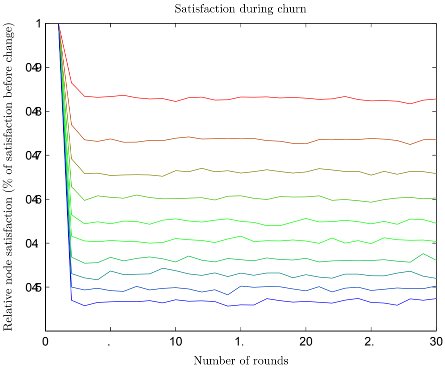


Fig. 5. Satisfaction while churn is in progress, affecting 5% to 50% of the network's nodes (in steps of 5%, top to bottom)

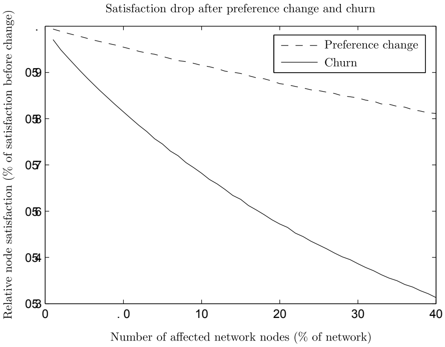


Fig. 6. Satisfaction right after preference change or churn per amount of affected nodes

a network of 100 nodes. It is obvious that the amount of satisfaction achieved remains fairly constant during churn and depends greatly on the amount of churn. However, even though churn is an intense operation, it is possible to retain a significant percentage of the original satisfaction, even for churn as high as 50% (i.e. when half of the network is changing at every round).

On the other hand the satisfaction drop is significant compared to the one caused by preference change. Figure 6 shows the relative satisfaction for both preference change and churn on ER networks, right after the change happens, for various amounts of affected nodes, supporting our argument in favor of improving connections instead of rebuilding them from the beginning.

7 Conclusions

The adaptive algorithm AdaptiveLID for distributed matching with preferences proposed in this paper provides a method to form overlays with preferences with guaranteed satisfaction and convergence, as shown in the analysis. The paper also shows an improved approximation ratio for the maximizing satisfaction problem, which holds both for static and dynamic networks.

Besides, an extensive experimental study of the proposed algorithm encompasses a variety of scenarios, including ones that put the algorithm under heavy stress and that have been previously used in literature to simulate network attacks. In these scenarios the algorithm succeeds in maintaining a reduced but steady level of network service while under attack, and resumes to normal service levels after the attack stops. Furthermore, the algorithm shows attractive properties with respect to the satisfaction it can achieve and the convergence time (and hence overhead) it needs. In particular, the experiments clearly strengthen the argument that it is preferable to improve connections and adapt to changes instead of rebuilding all the connections from the ground up.

To the best of our knowledge it is the first adaptive method with satisfaction and convergence guarantees for this problem. We expect that this contribution will be helpful for future work in the area, since the method can facilitate overlay construction with guarantees in a wide range of applications, from peer-to-peer resource sharing, to overlays in intelligent transportation systems and adaptive power grid environments.

References

1. Irving, R.W., Scott, S.: The stable fixtures problem - a many-to-many extension of stable roommates. Discrete Appl. Math. 155(16), 2118–2129 (2007)
2. Mathieu, F.: Self-stabilization in preference-based systems. Peer-to-Peer Networking and Applications 1(2), 104–121 (2008)
3. Jacob, R., Richa, A., Scheideler, C., Schmid, S., Täubig, H.: A distributed polylogarithmic time algorithm for self-stabilizing skip graphs. In: Proceedings of the 28th ACM Symposium on Principles of Distributed Computing, PODC 2009, pp. 131–140. ACM (2009)

4. Awerbuch, B., Scheideler, C.: Towards a scalable and robust dht. Theory of Computing Systems 45, 234–260 (2009), doi:10.1007/s00224-008-9099-9
5. Edmonds, J.: Paths, trees and flowers. Canadian Journal of Mathematics 17, 449–467 (1965)
6. Gale, D., Shapley, L.S.: College admissions and the stability of marriage. American Mathematical Monthly 69, 9–15 (1962)
7. Gusfield, D., Irving, R.W.: The stable marriage problem: structure and algorithms. MIT Press, Cambridge (1989)
8. Kuhn, F., Moscibroda, T., Wattenhofer, R.: The price of being near-sighted. In: SODA 2006: Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 980–989. ACM (2006)
9. Hoepman, J.H.: Simple distributed weighted matchings. CoRR cs.DC/0410047 (2004)
10. Lotker, Z., Patt-Shamir, B., Rosen, A.: Distributed approximate matching. In: PODC 2007: Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing, pp. 167–174. ACM (2007)
11. Wattenhofer, M., Wattenhofer, R.: Distributed Weighted Matching. In: Guerraoui, R. (ed.) DISC 2004. LNCS, vol. 3274, pp. 335–348. Springer, Heidelberg (2004)
12. Manne, F., Mjelde, M.: A Self-stabilizing Weighted Matching Algorithm. In: Masuzawa, T., Tixeuil, S. (eds.) SSS 2007. LNCS, vol. 4838, pp. 383–393. Springer, Heidelberg (2007)
13. Lotker, Z., Patt-Shamir, B., Pettie, S.: Improved distributed approximate matching. In: SPAA 2008: Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures, pp. 129–136. ACM (2008)
14. Koufogiannakis, C., Young, N.E.: Distributed Fractional Packing and Maximum Weighted b-Matching via Tail-Recursive Duality. In: Keidar, I. (ed.) DISC 2009. LNCS, vol. 5805, pp. 221–238. Springer, Heidelberg (2009)
15. Cechlárová, K., Fleiner, T.: On a generalization of the stable roommates problem. ACM Trans. Algorithms 1(1), 143–156 (2005)
16. Gai, A.-T., Lebedev, D., Mathieu, F., de Montgolfier, F., Reynier, J., Viennot, L.: Acyclic Preference Systems in P2P Networks. In: Kermarrec, A.-M., Bougé, L., Priol, T. (eds.) Euro-Par 2007. LNCS, vol. 4641, pp. 825–834. Springer, Heidelberg (2007)
17. Lee, H.: Online stable matching as a means of allocating distributed resources. Journal of Systems Architecture 45(15), 1345–1355 (1999)
18. Georgiadis, G., Papatriantafilou, M.: Overlays with preferences: Approximation algorithms for matching with preference lists. In: Proceedings of 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010). IEEE Computer Society Press (April 2010)
19. Preis, R.: Linear Time $\frac{1}{2}$-Approximation Algorithm for Maximum Weighted Matching in General Graphs. In: Meinel, C., Tison, S. (eds.) STACS 1999. LNCS, vol. 1563, pp. 259–269. Springer, Heidelberg (1999)
20. Georgiadis, G., Papatriantafilou, M.: Adaptive distributed b-matching in overlays with preferences. Technical report, Chalmers University of Technology (March 2012)
21. Chandy, K.M., Misra, J.: The drinking philosophers problem. ACM Transactions on Programming Languages and Systems 6(4), 632–646 (1984)
22. Jelasity, M., Montresor, A., Jesi, G.P., Voulgaris, S.: The Peersim simulator, `http://peersim.sf.net`
23. Barabási, A.L., Albert, R.: Emergence of scaling in random networks. Science 286, 509–512 (1999)
24. Bollobás, B.: Random Graphs. Academic Press, New York (1985)

Dynamizing Succinct Tree Representations

Stelios Joannou and Rajeev Raman

University of Leicester, Department of Computer Science, University of Leicester,
University Road, Leicester, LE1 7RH

Abstract. We consider *succinct*, or space-efficient, representations of *ordinal* trees. Representations exist that take $2n + o(n)$ bits to represent a *static* n-node ordinal tree – close to the information-theoretic minimum – and support navigational operations in $O(1)$ time on a RAM model; and some implementations have good practical performance.

The situation is different for *dynamic* ordinal trees. Although there is theoretical work on succinct dynamic ordinal trees, there is little work on the practical performance of these data structures. Motivated by applications to representing XML documents, in this paper, we report on a preliminary study on dynamic succinct data structures. Our implementation is based on representing the tree structure as a sequence of balanced parentheses, with navigation done using the *min-max* tree of Sadakane and Navarro (SODA '10). Our implementation shows promising performance for update and navigation, and our findings highlight two issues that we believe will be important to future implementations: the difference between the *finger model* of (say) Farzan and Munro (ICALP '09) and the *parenthesis model* of Sadakane and Navarro, and the choice of the balanced tree used to represent the min-max tree.

1 Introduction

A number of applications that involve indexing and processing textual or semi-structured data now need to deal with increasingly large volumes of data. Typically, these applications do not have satisfactory external-memory solutions and so the data has to be held and processed in main memory; examples include the various applications of suffix trees and a number of XML processing tasks (including XQuery search, XSLT transformation). An important bottleneck in many such applications is the space required to represent some kind of tree-structured object: in such applications, *succinct*, or highly space-efficient, representations of trees[18] are having increasing impact. The focus of this paper is on *ordinal trees*, which are arbitrary rooted trees where the children of each node are ordered. As there are $\frac{1}{n}\binom{2n-2}{n-1}$ ordinal trees on n nodes, storing an ordinal tree requires $2n - O(\lg n)$ bits, as opposed to the standard representation that takes asymptotically $\Theta(n)$ words, or $\Theta(n \lg n)$ bits, of memory. In recent years, a number of representations of *static* ordinal trees have been developed [14,15,5,13,20] that use $2n + o(n)$ bits of memory, and support a wide range of navigational operations in $O(1)$ time assuming the RAM model of computation with word size $\Theta(\log n)$ (the default theoretical model that we will assume in

R. Klasing (Ed.): SEA 2012, LNCS 7276, pp. 224–235, 2012.

this paper). The excellent practical performance of succinct ordinal tree representations has been shown in many papers including [8,12,2]. For example, when representing XML documents, which are essentially ordinal trees, a standard pointer-based representation [1] has *five* pointers per node[1] (320 bits per node on a 64-bit machine) to represent the tree structure and support fast navigation; thus, the attractiveness of a representation that takes just a few bits per node but supports operations quickly in practice, is clear. Indeed, succinct ordinal trees have been successfully applied to several XML applications [25,9,3,7].

Despite the success of static succinct data structures, more needs to be done. For example, in the XML context, efficient support for updates to documents is fundamental: the W3C standard DOM API specifies a number of methods for modifying XML documents [24]. In discussions with industry contacts we have found that there are few “purely static” real-world XML applications that deal with large documents. While static succinct trees have received a lot of attention, there has been much less work on dynamizing them, in theory or in practice. Farzan and Munro [10] and Sadakane and Navarro [20] studied the theory of dynamic succinct ordinal trees. Practical studies of dynamic succinct data structures are few, and we are only aware of one work that discusses the implementation of a dynamic succinct ordinal tree [25]; however, they implement a theoretically non-optimal tree, and also their (good) perfomance results are for their entire system, rather than the tree in isolation.

In this paper, we consider practical performance issues in implementing dynamic succinct trees. The operations we consider are:

- Basic navigation: `first-child`, `last-child`, `parent`, `next-sibling`, `prev-sibling`.
- Updates: insertion and deletions of leaves.

Furthermore, we are concerned not just with the performance of individual operations, but also focus on *traversals*, or relatively long sequences of navigational operations. We give a brief introduction to the approach we take to dynamization, before summarizing our main contributions.

Our approach is to represent the current n-node ordinal tree as a *balanced parenthesis (BP)* sequence of length $2n$ (see Fig. 1). For specificity, assume that a node is represented by the opening parenthesis of the pair representing it. We obtain the BP sequence representing a tree by going through a tree depth-first, outputting an opening parenthesis when a node is first encountered and a closing one when every node of its sub-tree has been encountered. The BP sequence is divided into *blocks* of size $\Theta(B)$ for some parameter B–in theory $B = \Theta((\log n)^2)$–and a *min-max* excess tree, a balanced binary tree, is stored over the blocks to perform *excess search* [17,20] (see Section 2 and 4 for details). So far, we are following the approach proposed by [20], who show that if some details are handled carefully, this approach yields an implementation with space bound $2n + o(n)$ bits, and time bounds $O(\log n)$ for both navigation and update operations; the time bound can further be reduced to $O(\log n/ \log\log n)$ while

[1] Parent, first child, last child, previous sibling and next sibling.

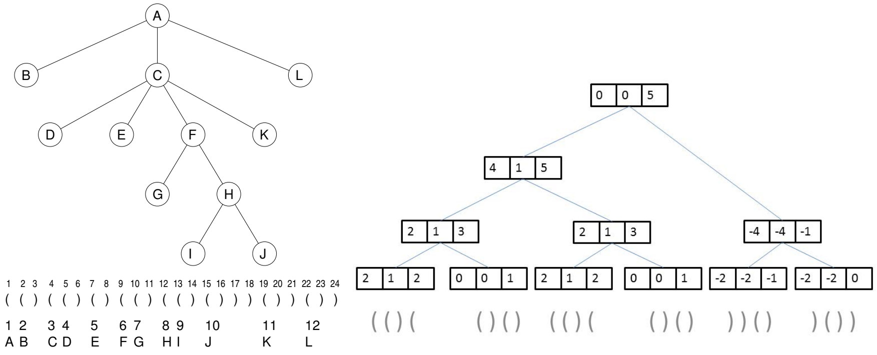


Fig. 1. Example ordinal tree and its BP representation (left), the min-max tree (right)

maintaining the space bound. However, more work needs to be done to obtain a satisfactory practical data structure from this idea, as we now discuss.

Parentheses versus Fingers. The first question that arises is what the precise interface through which the data structure implements the navigation operations should be. Navigation in the BP is usually understood [15] in terms of the following two operations:

- `findclose`(i): if the i-th parenthesis is an opening parenthesis, then find the position of the matching closing parenthesis (`findopen` is similar).
- `enclose`(i): find the opening parenthesis that corresponds to the parenthesis pair that most closely encloses position i.

In this scenario, we may, for example, number the nodes $1, \ldots, n$ in depth-first order, and the operation `next-sibling`(i) may take a node number as an argument[2]. In this case, we need to proceed as follows:

1. Find the position of the opening parenthesis corresponding to i, say p.
2. Let $q =$ `findclose`(p).
3. Inspect the $q+1$st parenthesis of the sequence, and if it is a closing parenthesis, then return null. Otherwise, the next sibling is the node whose opening parenthesis is in at position $q + 1$.
4. Determine the number j such that the parenthesis at position $q + 1$ is the j-th opening parenthesis; return j.

Unfortunately, *all four* of the above steps require $\Omega(\log n/ \log\log n)$ time if we wish to support updates to the BP string in at most poly-log n time, by reductions [6,10] to the well-known list-ranking and subset-sum problems [11]. This appears to be a high price to pay for dynamizing, considering that operations take $O(1)$ time in the static case.

An alternative is the *finger* model [10,19,16], where the key object is a finger `f`, on which the following operations may be supported: (a) initialize the finger

[2] This is the approach taken by the implementation of [2].

to the root of the tree (b) perform operations such as `f.op` where `op` is one of the navigation operations mentioned above. The result of an operation `f.op` is either `true`, in which case the required node (parent, next sibling etc.) exists, and the finger is moved to that node, or `false`, i.e. the node does not exist, and the finger does not move. Updates are limited to occurring in the vicinity of the finger. Interestingly, in the finger model, navigation operations can in fact be performed in $O(1)$ time [10]. While the approach proposed by [10] appears to be complex and unsuitable for a practical implementation, the simpler and practical approach of [20] that we are following appears inherently to take time $\Omega(\log n)$ for an individual operation. This raises the question: in an implementation of dynamic succinct ordinal trees based on [20], is there any practical difference between the finger model and the parenthesis-position model?

We show that the answer is "yes". For traversals, this is partly because we can largely omit steps (1), (3) and (4) above in the finger model, but also because (2) turns out to be significantly cheaper than "logarithmic" for many practical traversal seqeuences. For updates, we show that a simple strategy of "buffering" updates (only possible in the finger model) greatly improves the speed of updates for some update patterns (in our implementations we assume that we have only one finger, but the principle easily extends to multiple fingers).

Traversals vs. Individual Operations. An implementation of static succinct ordinal trees, based on the approach of [20], was reported in [2]. As with the dynamic approach of [20], the implementation of [2] also has $\Theta(\log n)$ worst-case time for individual operations. On the other hand, the implementation of [12] is in principle $O(1)$ time per operation. The starting point of our investigation was that the implementation of [2], on traversals of ordinal trees derived from some typical "benchmark" XML files, apparently had linear—rather than the expected $\Theta(n \log n)$—behaviour. The basic cause of this is that if the answer to a `findclose` or `enclose` operation is at distance d from the argument, then the answer is usually found in $O(\log d)$ rather than $O(\log n)$ time. This led us to investigate some "worst-case" trees for the implementation of [2]. While this investigation (see Section 3) produced some interesting results, the real insight was to try and directly exploit the "locality" of typical traversal sequences. As noted above, in the data structure of [20], the min-max tree built over the blocks needs to be a balanced binary tree, but the choice of balanced binary tree is not specified (however, a kind of (a, b)-tree is needed to obtain the optimal $O(\log n/ \log \log n)$ time bound). The desire to exploit locality led us to consider using a *splay* tree [22], which has a number of interesting locality properties [4,22], some conjectured, others proven. In the traversals we considered, using splay trees allowed our example files to be traversed in linear observed time in some cases. The time spent on splay tree operations (measured by number of nodes accessed) appears sub-logarithmic in all the cases we considered. We do not yet have a theoretical understanding of this phenomenon.

Structure of Paper. The rest of the paper is structured as follows. Section 2 summarizes the approach of [20,2], Section 3 summarizes our experiments on static trees, Section 4 summarizes our dynamic implementation, which is followed by an empirical evaluation in Section 5.

2 Preliminaries

Consider the BP bit string of length $2n$ where 1 represents '(' and 0 ')'. The *excess* at any position i is the number of 1's minus the number of 0's prior to position i. The excess of an opening parenthesis is also the depth of the node in the ordinal tree. We use the terms *global excess* for the excess as defined above, and *local excess* relative to some sub-string of the BP bit string for the excess of a position, measured from the start of the sub-string. We also use the term *sum* (relative to a sub-string again) for the excess at the end of the substring (which is 0 for the entire BP). We use the terms *min excess* and *max excess* to denote the minimum and maximum excess reached in a sub-string of the BP bit string. A key step in [2,17,20] is *excess search*, which is as follows:

- `fwd_excess`(i, rel): starts at position i going forward in the bit string searching for the leftmost node after i that has relative excess to i equal to rel;
- `bwd_excess`(i, rel): starts at position i going backward in the bit string searching for the rightmost node before i that has relative excess to i equal to rel.

Using the operations `fwd_excess`(i, rel) and `bwd_excess`(i, rel), the operations `findclose`(i), `findopen`(i) and `enclose`(i) can be implemented. For example, `findclose`(i) = `fwd_excess`$(i, -1)$.

Excess Search and the Min-Max Tree. As noted above, the BP sequence is partitioned into *blocks* of size B each. These blocks are placed at the leaves of a tree such that each node contains the minimum, maximum (local) excess and sum of the concatenation of blocks under it. In the static case, this min-max tree is implemented as a binary tree stored in an array using the "heap-like" numbering; in the dynamic case, an unspecified balanced tree is recommended. An excess search starting at a block p and ending at a block $q \neq p$ will navigate up to the lowest common ancestor of p and q in the min-max tree from p, and down again to q, see [20] or Section 4 for details.

3 Traversals on Static Trees

Input trees and traversals. Although a number of real-life "benchmark" XML files are available [9], we did not use them extensively, for a variety of reasons. Firstly, the files were relatively small – the largest, although 600MB in size, had only about 25 million nodes: if stored in the information-theoretic minimum amount of space, the tree would almost completely fit into a 6MB cache. Clearly, experiments on such trees would not give a complete account of the performance of our data structures for very large trees, particularly since cache misses are an important cause of poor performance in succinct data structures. Furthermore, our aim was to detect patterns of performance and to find “worst-case” instances. (Although not reported here, the results we got from the larger real-life benchmark XML files were in line with those we have reported.)

The experiments we performed were on four kinds of trees, of sizes approximately 64, 128, 256 and 512 million nodes. These were:

- Trees of about the above sizes obtained from XML files generated by the XMark synthetic benchmark generator (this is a standard generator for testing XML systems) [21].
- Regular k-ary trees of height h: we looked at the case $k = 2$ and $h = 25, 26, 27$ and 28 (referred to as *binary trees* henceforth), and $h = 2$ and $k = 8000, 11314, 16000$ and 22618 (henceforth *k-ary trees*).
- A “centipede” tree, where a tree with n nodes (n odd) has a path of $(n+1)/2$ nodes, with each non-root node on this path having a leaf as a right sibling.

We considered two kinds of traversals: a non-recursive depth-first traversal (DFS) and the *all root-leaf (ALR)* traversal [9], where we do DFS, but whenever the DFS encounters a leaf we trace the path back to the root.

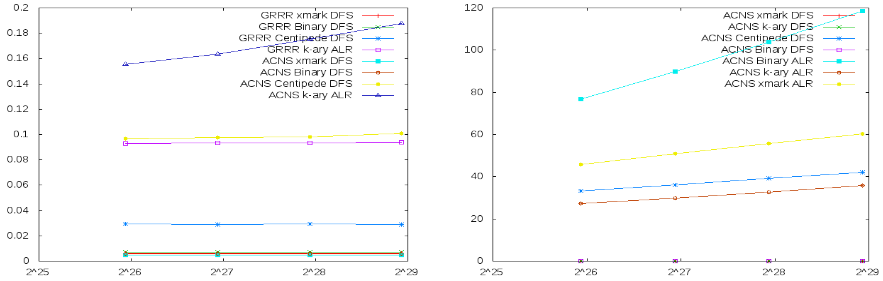


Fig. 2. Performance of the *static* implementations of Geary et al. (GRRR) and Arroyuelo et al. X-axis is the number of nodes (log-scale). (Left) Y-axis is the time for traversal divided by the number of nodes. (Right) number of tree nodes visited by Arroyuelo et al.'s implementation (ACNS) divided by the number of nodes.

Input Trees and Traversals. We timed the static implementations of [2,12], code obtained from Sadakane, on the above sets of trees (see Figure 2[3]). Timings were only taken for the following pairs of inputs: XMark, Binary and Centipede with DFS, and k-ary with ALR.[4] It should be emphasized that we cannot really compare the two data structures against each other: the code of [2] is a 32-bit C implementation that takes about 2.3 bits/node with the default parameter settings, while the code of [12] is a 64-bit C++ implementation that takes just over twice as much space with the default parameter settings. We note that the

[3] Some plotted data sets have y-values close to 0, hence they are not clearly visible

[4] This is mainly for succinctness, though clearly it is infeasible to run ALR on Centipede for our data sizes.

time for DFS (per node) varies greatly with the tree for both implementations: Centipede is an order of magnitude slower. Also, for DFS on Xmark, Centipede and the binary tree, the implementation of [2] shows linear behaviour, but for ALR traversal on k-ary trees it clearly shows $\Theta(n \log n)$ behaviour. The logarithmic growth of numbers of tree nodes visited per input node is clearly visible in both the (Centipede, DFS) and (k-ary, ALR) pairs in the graph on the right. As expected in Geary et al.'s implementation, the traversal time per node is constant for all traversals.

We explain this by looking at the *traversal distance* of a traversal on a BP sequence: if the ith step of the DFS moves from a node whose parenthesis is at position p to one at position q, we set $d_i = |p - q|$, and the traversal distance is simply $\sum_i d_i$. It is easy to see that (i) the traversal distances of DFS and ARL on a k-ary tree are $\Theta(n)$ and $\Theta(n^2)$ respectively, and (ii) the traversal distance of DFS on a binary tree and Centipede is respectively $\Theta(n \log n)$ and $\Theta(n^2)$. In general it is easy to show (proof omitted):

Lemma 1. *The traversal distance of non-recursive DFS on an ordinal tree with n nodes of height h is $O(nh)$.*

Since the XMark files have a small (fixed) depth, the traversal distance for DFS on XMark files is also linear.

Using the heuristic that a navigation operation with traversal distance d usually takes $O(1 + \log\lceil d/B \rceil)$ time (since the start and end points will be $\lceil d/B \rceil$ blocks apart), we see that the time spent in the min-max tree for a traversal over a tree with n nodes with overall traversal distance D would be $O(n(1 + \log\lceil D/(nB) \rceil))$. This provides some explanation of the observed data (albeit partially) – for example, we do not see a logarithmic growth in time for Arroyuelo et al.'s implementation when running DFS on Centipede. However, the traversal distance argument does suggest that in the dynamic case, a normal balanced tree (e.g. red-black) will also have relatively poor performance in cases such as ALR traversal on k-ary trees; one may get better performance by exploiting locality directly (e.g. successive leaf-to-root traversals in ALR traversals will tend to visit many min-max tree nodes in common, and have high temporal locality).

4 Engineering a Dynamic Succinct Implementation

Our *base implementation* divides the BP sequence into blocks of size B, which are leaf nodes in a binary min-max tree. Each node of the min-max tree contains the data mentioned in Section 2, see Figure 1 (right) for example. Similarly to [20,2] we use the excess to navigate around the succinct tree and find the matches of our parentheses.

Forward excess search using the min-max tree is shown below (backward excess is similar). The function $E(i)$ returns the global excess at position i. In Step 1 we use the length information in the min-max tree to locate the block in which parenthesis i is located, starting from the root of the tree and descending (this is needed as the number of parentheses in the blocks is not equal). We initialize the base implementation by *bulk loading* it. Bulk loading splits the full BP string of a tree in equal sized blocks (except the last) and builds the min-max tree on top of them, which results in a complete balanced binary tree.

```
forwardExcess ($i$, $d$)
1. Use the min-max tree to locate the block in which $i$ resides keeping track of global excess at beginning of block
2. Scan the block of $i$ for the next parenthesis $j$ where $i < j$ and $E(j) = E(i) + d$
     – If found return $j$
3. Search min-max tree for lowest common ancestor of block containing $i$ and block containing $E(i)+d$ using minimum and maximum excess adjusting global excess at start of block while moving between nodes
4. Search min-max tree for a leaf where min excess $\leq (E(i) + d) \leq$ max excess starting from right child of lowest common ancestor, by moving to the right child of current node when $E(i) + d$ not within range of excesses of left child
5. Scan the current node and find position $j$ such that $E(j) = E(i) + d$
```

Finger Model. We add the finger model to our base implementation. A finger sits at a node (the *finger node*) and contains the block in which the parenthesis representing the finger node lies, its local position in the block, its current local excess and the excess at the beginning of the block. Clearly, Step 1 above is not needed in the finger model.

In addition to bulk loading, we also provide the following dynamic operations for modifying the finger model:

- `insert-first-child`(): insert a leaf as the first child of the finger node
- `insert-next-sibling`(): insert a leaf as the next sibling of the finger node

Operations `delete-first-child`() and `delete-next-sibling`() are analogous. This API can also be used to create an ordinal tree.

Implementation of Updates. To insert a new leaf we shift all the parenthesis in the block to the right of the finger to make room for the leaf node. Observe that the sum of a block does not change by adding a leaf, and neither does the minimum excess. The maximum excess increases by 1, but only if a leaf is inserted at a position where the excess is already maximum. Thus, we do not need to scan the block after an insertion. Deleting a node is similar, but if the excess at the node to be deleted is the same as the current maximum excess in the block we will need to rescan the block to discover if there is another node that has the same excess, or if the block maximum excess has changed. In all cases though, the length values (if needed the excess values) of all ancestors of the block in the min-max tree have to be updated.

When the block is full, it is then split into two new blocks, each containing half of the parentheses of the previous block. If we use fixed-size blocks then we run the risk of having each block half empty. To improve the space usage of blocks (as low as 50% in the above), we implemented *incremental* copying of the blocks. We start off with a block of size B. When the block gets full we increase its capacity by a word, with an upper bound of $2B$. When the block size is $2B$ we split the block as above. By incremental copying, the wasted space is at most one word per block. Furthermore, since $\Theta(\log n)$ updates happen before the block's capacity is extended again, and extension costs $O(\log n)$ time – each block has only $O(\log n)$ words – the amortized cost of extension is only $O(1)$ per update. Moreover, we buffer all updates that occur in a block so long as the finger does not move to a different block. When the finger moves we *flush* the excess and length changes to all the ancestors of the block before leaving the block.

The min-max tree was implemented as a splay tree [23]. When the finger moves to a new block, then the parent of that block (which is an internal node) is splayed to the top of the tree.

5 Experimental Evaluation

The data structure as well as the tests were written in C++. The machine that was used to run these tests was an Intel Pentium 64-bit machine with 8GB of main memory and a G6950 CPU clocked at 2.80GHz with 3MB L2 cache, running Ubuntu 10.04.1 LTS Linux and g++ 4.4.3 with optimization level 3. The Xerces-C++ 2.80 was used. Furthermore, we use the code from [2], henceforth referred to in this section as ACNS.

We first aim to justify the use of the finger model in a dynamic succinct data structures. As noted previously, the base implementation is based on the parenthesis model, and therefore each navigation operation is based upon the `findopen`(i), `findclose`(i) and `enclose`(i) operations, where i is the position of a parenthesis in the BP bit string. The first step in these operations is to start from the root of the min-max tree to locate the block in which the ith parenthesis lies. This step is unnecessary if no updates are made to a bulk-loaded tree, as in this case all blocks are equal-sized; we augment the base implementation with an array containing pointers to each block and find the block containing the ith parenthesis by indexing into this array. This is the *base + pointer array* implementation. Using the trees in Section 3 we perform traversals on a bulk-loaded base implementation and the base+pointer array implementation, and also on the ACNS implementation (as a "control" test).

Table 1 presents the time it took to do a DFS traversal on XMark files of size 64, 128 and 256 million nodes. We observe from the the results of the test that ACNS and Base + Pointer implementation are showing $O(n)$ behaviour for XMark files. However, the base implementation shows $O(n \log n)$ increase and is much slower. Since the only difference of base implementation with base + pointer is that to access any block we need to descend the tree, it is clearly shown that this has a significant impact on the performance of the navigational operations.

For our second test, we test the speed of the insertions. Since there is no existing succinct dynamic tree implementation we compare with Xerces-C++. This is accomplished by using a SAX parser. A SAX parser will go through an XML file and raise an event when an opening/closing XML tag was encountered. We use that with our finger implementation to create an ordinal tree using

Table 1. (Left) DFS Traversal per-node time of base implementation, base + pointer array implementation, ACNS in μs (Right) Parsing time of XMark file for DOM and Splay tree implementation. Results measure CPU time in seconds.

Nodes	Base Implementation	Base + Pointer Array Implementation	ACNS
64M	0.336	0.060	0.049
128M	0.361	0.060	0.049
256M	0.383	0.060	0.049
512M	0.407	0.060	0.049

Nodes	DOM	Splay Trees
16M	46.17	31.95
32M	90.83	65.05
64M	204.04	128.37

`insert-first-child`, `insert-next-sibling` and we compare the parsing time by creating a Xerces-C++ DOM tree using an analogous method. For the DOM case all nodes are named "a". In these tests we used XMark files up to 64M nodes. Larger sizes were not attempted due to the memory usage of DOM.

As shown in Table 1, the creation time for our finger model is faster. This can be partially explained by the cache effects. Due to the small number of nodes, most of the tree will easily fit to cache in the finger model case. It was 40% faster in the case of 32 million nodes and 60% faster in the case of 64 million nodes. In that specific case the fact that the percentage was so high might have been due to the excessive memory usage of a Xerces-C++ DOM tree (for 64M nodes virtual memory exceeded 12GB). Also the speed difference seems to diverge slightly with the last test. The gap seems to be widening due to thrashing.

As a third test, we compare the traversal times for a splay tree against a bulk loaded tree. To do these we check both against 64, 128 and 256 million nodes generated by XMark, as well as with k-ary trees with similar number of nodes.

Table 2. DFS and ALR traversal comparison with using balanced tree and splay tree. Results are in seconds.

Nodes	XMark						k-ary tree					
	ACNS		Balanced		Splay		ACNS		Balanced		Splay	
	DFS	ALR	DFS	ALR	DFS	ALR	DFS	ALR	DFS	ALR	DFS	ALR
64M	3.17	130.15	1.77	155.83	2.06	144.22	2.03	99.38	0.86	163.21	0.7	145.71
128M	6.35	261.25	3.54	310.76	4.15	281.45	4.05	209.60	1.74	330.36	1.41	294.29
256M	12.70	550.00	7.10	628.82	8.31	573.43	8.11	448.65	3.45	667.01	2.76	591.84

From Table 2 it is clear that for DFS our data structure, both with the balanced tree and splay tree on top is faster than the ACNS data structure. This is possibly a result that was influenced from the different block sizes used between ACNS and the finger model (balanced tree and splay tree versions use the same block size). Using a balanced tree appears to be faster for DFS, but when for the ALR traversal the splay tree version appears to be faster when compared with the balance tree. Figure 3 shows that for our tests the splay tree accesses fewer nodes of the min-max tree, hence proving that previously

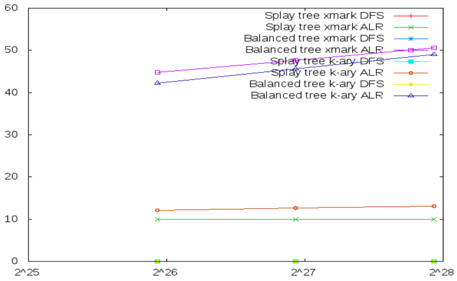


Fig. 3. Comparison of Splay trees and balanced trees. X-axis is number of nodes (log-scale). Y-axis is nodes of min-max tree visited divided by number of nodes

traversed nodes are closer to the top of the min-max tree. The fewer number of accessed blocks explains the speed difference for ALR traversals, since the specific traversal makes uses of previously accessed blocks to reach the root of the ordinal tree, after a leaf is encountered.

6 Conclusions

We have performed an empirical evaluation of a first implementation of dynamic succinct trees. We observe that the performance of static succinct tree implementations, particularly those based on the min-max tree which have (near-)logarithmic worst-case per-operation time complexity, is very dependent on the tree and the sequence of operations performed, more specifically, on the locality properties of the sequence of operations. By using a self-adjusting tree (the splay tree) as a basis for the min-max tree, we obtain good performance, which is arguably superior to any other balanced search tree scheme. The dynamic implementation compares well with static succinct implementations for navigation operations and with pointer-based ones for update operations. However, further work is required to expand the functionality, to better understand the effects of the splay tree on traversal performance and to investigate implementations of $O(1)$-time dynamic succinct trees.

References

1. Apache: Xerces-c++ xml parser (January 2012), http://xerces.apache.org/xerces-c/
2. Arroyuelo, D., Cánovas, R., Navarro, G., Sadakane, K.: Succinct trees in practice. In: Blelloch, G.E., Halperin, D. (eds.) ALENEX, pp. 84–97. SIAM (2010)
3. Arroyuelo, D., Claude, F., Maneth, S., Mäkinen, V., Navarro, G., Nguyen, K., Sirén, J., Välimäki, N.: Fast in-memory xpath search using compressed indexes. In: ICDE, pp. 417–428 (2010)
4. Badoiu, M., Cole, R., Demaine, E.D., Iacono, J.: A unified access bound on comparison-based dynamic dictionaries. Theor. Comput. Sci. 382(2), 86–96 (2007)

5. Benoit, D., Demaine, E.D., Munro, J.I., Raman, R., Raman, V., Rao, S.S.: Representing trees of higher degree. Algorithmica 43(4), 275–292 (2005)
6. Chan, H.L., Hon, W.K., Lam, T.W., Sadakane, K.: Compressed indexes for dynamic text collections. ACM Transactions on Algorithms 3(2) (2007)
7. Delpratt, O.: The sixml project (March 2010), http://www.cs.le.ac.uk/SiXML/
8. Delpratt, O., Rahman, N., Raman, R.: Engineering the LOUDS Succinct Tree Representation. In: Àlvarez, C., Serna, M. (eds.) WEA 2006. LNCS, vol. 4007, pp. 134–145. Springer, Heidelberg (2006)
9. Delpratt, O., Raman, R., Rahman, N.: Engineering succinct dom. In: EDBT, pp. 49–60 (2008)
10. Farzan, A., Munro, J.I.: Succinct representation of dynamic trees. Theor. Comput. Sci. 412(24), 2668–2678 (2011); prelim. version ICALP 2009
11. Fredman, M.L., Saks, M.E.: The cell probe complexity of dynamic data structures. In: Johnson, D.S. (ed.) STOC, pp. 345–354. ACM (1989)
12. Geary, R.F., Rahman, N., Raman, R., Raman, V.: A simple optimal representation for balanced parentheses. Theor. Comput. Sci. 368(3), 231–246 (2006)
13. Geary, R.F., Raman, R., Raman, V.: Succinct ordinal trees with level-ancestor queries. ACM Transactions on Algorithms 2(4), 510–534 (2006)
14. Jacobson, G.: Space-efficient static trees and graphs. In: FOCS, pp. 549–554. IEEE Computer Society (1989)
15. Munro, J.I., Raman, V.: Succinct representation of balanced parentheses and static trees. SIAM J. Comput. 31(3), 762–776 (2001)
16. Munro, J.I., Raman, V., Storm, A.J.: Representing dynamic binary trees succinctly. In: SODA, pp. 529–536 (2001)
17. Munro, J.I., Rao, S.S.: Succinct Representations of Functions. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 1006–1015. Springer, Heidelberg (2004)
18. Munro, J.I., Rao, S.S.: Succinct Representation of Data Structures. In: Handbook of Data Structures and Applications, ch. 37. Chapman & Hall/CRC (2004)
19. Raman, R., Rao, S.S.: Succinct Dynamic Dictionaries and Trees. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 357–368. Springer, Heidelberg (2003)
20. Sadakane, K., Navarro, G.: Fully-functional succinct trees. In: Charikar, M. (ed.) SODA, pp. 134–149. SIAM (2010)
21. Schmidt, A., Waas, F., Kersten, M.L., Carey, M.J., Manolescu, I., Busse, R.: Xmark: A benchmark for xml data management. In: VLDB, pp. 974–985. Morgan Kaufmann (2002)
22. Sleator, D.D., Tarjan, R.E.: Self-adjusting binary search trees. J. ACM 32(3), 652–686 (1985)
23. Tarjan, R.E.: Data Structures and Network Algorithms. SIAM (1987)
24. W3C: Document object model (January 2009), http://www.w3.org/DOM/
25. Wong, R.K., Lam, F., Shui, W.M.: Querying and maintaining a compact xml storage. In: Williamson, C.L., Zurko, M.E., Patel-Schneider, P.F., Shenoy, P.J. (eds.) WWW, pp. 1073–1082. ACM (2007)

A Label Correcting Algorithm for the Shortest Path Problem on a Multi-modal Route Network

Dominik Kirchler[1,2,3], Leo Liberti[1], and Roberto Wolfler Calvo[2]

[1] LIX, Ecole Polytechnique
{kirchler,liberti}@lix.polytechnique.fr
[2] LIPN, Université Paris 13
roberto.wolfler@lipn.univ-paris13.fr
[3] Mediamobile

Abstract. We consider shortest paths on a realistic multi-modal transportation network where restrictions or preferences on the use of certain modes of transportation or types of roads arise. The regular language constraint shortest path problem deals with this kind of problem. It models constraints by using regular languages. The problem can be solved efficiently by using a generalization of Dijkstra's algorithm (D_{RegLC}). In this paper we present a new label correcting algorithm (lcSDALT) with a speed-up, in our setting, of a factor 3 to 30 in respect to D_{RegLC} and up to a factor 2 in respect to a similar algorithm.

1 Introduction

Multi-modal transportation networks include roads, public transportation, bicycle lanes, etc. Shortest paths in such networks must satisfy some additional constraints: passengers may want to exclude some modes of transportation, they may wish to pass by some grocery shop on the way or to limit the number of changes between public transportation vehicles. Feasibility has also to be assured: whereas walking is permitted at any point of the itinerary, private cars or bicycles can only be used when they are available.

The *regular language constrained shortest path problem* (RegLCSP) deals with this kind of problem. It uses an appropriately labeled graph and a regular language to model constraints. A valid shortest path minimizes some cost function (distance, time, etc.) and, in addition, the word produced by concatenating the labels on the arcs along the shortest path must form an element of the regular language. In [3], a systematic theoretical study of the more general *formal language constrained shortest path problem* can be found. The authors prove that the problem is solvable in deterministic polynomial time when regular languages are used and they provide a generalization of Dijkstra's algorithm (D_{RegLC}) to solve RegLCSP.

In recent years many scholars worked on speed-up techniques for Dijkstra's algorithm and shortest paths on continental sized road networks can now be found in a few milliseconds; see [5] for a comprehensive overview. The authors identify three basic ingredients to most modern speed-up techniques: bi-directional search, goal-directed search, and contraction.

R. Klasing (Ed.): SEA 2012, LNCS 7276, pp. 236–247, 2012.

D_{RegLC} has received less attention. In [1], various speed-up techniques including bi-directional and goal-directed search have been applied to D_{RegLC} on rail and road networks. The performance of the algorithm depends on the network properties and on how restrictive the regular language is.

The authors of [4] propose Access Node Routing to isolate the public transportation network from road networks so that they can be treated individually. A similar approach has been followed in [7] where contraction has been applied only to arcs belonging to the road network of a multi-modal transportation network including roads, public transport, and flight data.

The authors of [17] use contraction on a large road network where roads are labeled according to their road type. A subclass of the regular languages, the Kleene languages, is used to constrain the shortest path. It can be used to exclude certain road types. Kleene languages are less expressive than regular languages but contraction proves to be very effective in such a scenario. The authors of [7] and [17] report on speed-ups of over 3 orders of magnitude compared to D_{RegLC}.

A recent work [14] proposes the algorithm SDALT (State Dependent uniALT) which succeeds in accelerating D_{RegLC} by anticipating information on the additional constraints, modeled by the regular language, during a pre-processing phase. SDALT is based on ALT which is a bi-directional, goal directed search technique based on the A^* search algorithm [11] and which has been first discussed in [9]. It uses lower bounds on the distance to the target to guide Dijkstra's algorithm. UniALT is the uni-directional version of the ALT algorithm. Efficient implementations of uniALT and ALT as well as experimental data on continental size road networks with time-dependent edges cost are given in [15].

Our Contribution. SDALT is based on the uniALT algorithm, which works correctly only if the node potential function is *feasible*. The potential function is used to determine lower bounds to guide the algorithm faster to the target. We propose the algorithm lcSDALT which overcomes this limitation. It introduces more flexibility in the choice of the potential function. It therefore runs faster than SDALT by up to a factor 2 in our setting. Furthermore, it reduces memory requirements. We provide experimental results on a realistic multi-modal transportation network including time-dependent cost functions on arcs. It is composed of the road and public transportation network of the French region Ile-de-France, private and rental bicycles, walking, car (including changing traffic conditions), and public transportation. The experiments show that our algorithm performs well in networks where some modes of transportation tend to be faster than others or the constraints cause a major detour on the non-constrained shortest path. We observed speed-ups of a factor 3 to 30, in respect to plain D_{RegLC}.

Overview. Section 2 will first define the graph we are using to model the transportation network and give more details about RegLCSP and SDALT. Section 3 presents our new algorithm lcSDALT and its implementation. Its application to a realistic multi-modal transportation network and computational results, as well as some conclusive remarks are presented in section 4.

2 Preliminaries

Consider a *labeled*, directed graph $G = (V, A, \Sigma)$ consisting of a set of nodes $v \in V$, a set of labels $l \in \Sigma$, and a set of labeled arcs $(i, j, l) \in A$ which are triplets in $V \times V \times \Sigma$, $i, j \in V$, $l \in \Sigma$. The triple (i, j, l) represents an arc from node i to node j having label l. The labels are used to mark arcs as, e.g., foot paths (label f), bicycle lanes (label b), etc. Arc costs represent travel times. They are positive and time-dependent: $c : A \to (\mathbb{R}_+ \to \mathbb{R}_+)$, i.e. $c_{ijl}(\tau)$ gives the traveling times from node i to node j using transportation mode l at time $\tau \geq 0$. We only use functions which satisfy the FIFO property as the time-dependent shortest path problem in FIFO networks is polynomially solvable [13]. FIFO means that $c_{ijl}(x) + x \leq c_{ijl}(y) + y$ for all $x, y \in \mathbb{R}_+$, $x \leq y, (i, j, l) \in A$ or, in other words, that for any arc (i, j, l), leaving node i earlier guarantees that one will not arrive later at node j.

2.1 Solving the RegLCSP

The *regular language constrained shortest path problem* (RegLCSP) consists in finding a shortest path from a source node r to a target node t with starting time τ_{start} on the graph G by minimizing some cost function and, in addition, the concatenated labels along the shortest path must form a word of a given regular language L_0. The regular language is used to model the constraints on the sequence of labels (e.g., exclusion of labels, predefined order of labels, etc.). Any regular language L_0 can be described by a non-deterministic finite state automaton $\mathcal{A}_0 = (S_0, \Sigma_0, \delta_0, s_0, F_0)$, consisting of a set of states S_0, a set of labels $\Sigma_0 \subseteq \Sigma$, a transition function $\delta_0 : \Sigma_0 \times S_0 \to 2^{S_0}$, an initial state s_0, and a set of final states F_0 (for an example see Figure 4a). Note that $\overleftrightarrow{S}(s, \mathcal{A})$ and $\overleftrightarrow{\Sigma}(s, \mathcal{A})$ return all states and labels reachable by multiple transitions on an automaton $\mathcal{A}$ by starting at state s, backward and forward, respectively. E.g., in Figure 4a, $\overleftarrow{S}(s_1, \mathcal{A}_0) = \{s_0, s_1, s_3\}$, $\overrightarrow{\Sigma}(s_4, \mathcal{A}_0) = \{c_{\mathsf{fast}}, c_{\mathsf{pav}}, f, t, v\}$.

To efficiently solve RegLCSP, a generalization of Dijkstra's algorithm (which we denote D_{RegLC} throughout this paper) has first been proposed in [3]. The D_{RegLC} algorithm can be seen as the application of Dijkstra's algorithm [8] to the product graph $P = G \times S_0$ with tuples (v, s) as nodes for each $v \in V$ and $s \in S_0$ such that there is an arc $((v, s)(w, s'))$ between (v, s) and (w, s') if there is an arc $(i, j, l) \in A$ and a transition such that $s' \in \delta_0(l, s)$. To reduce storage space D_{RegLC} works on the *implicit* product graph P by generating all the neighbors which have to be explored only when necessary.

2.2 SDALT

The SDALT algorithm [14] is a speed-up technique for D_{RegLC} based on A^* [11] and *landmarks* [9]. Different from D_{RegLC}, SDALT uses an estimated lower bound of the distance to the target to guide the search more directly toward the target. More precisely, it employs a key $k(v, s) = d_r(v, s) + \pi(v, s)$ where $d_r(v, s)$

is the tentative distance label from (r, s_0) to (v, s) and the *potential function* $\pi : (V \times S) \to \mathbb{R}$ gives an under-estimation of the distance from (v, s) to (t, s), $s \in F_0$. At every iteration, the algorithm selects the node (v, s) with the smallest key $k(v, s)$. Intuitively, nodes which are close to the shortest estimated path from the source to the target node are explored first. So the closer $\pi(v, s)$ is to the actual remaining distance, the faster the algorithm will find the target.

It can be shown [12] that SDALT is equivalent to D_{RegLC} on a product graph with *reduced arc costs* $c^{\pi}_{(v,s_v)(w,s_w)l} = c_{vwl} - \pi(v, s_v) + \pi(w, s_w)$. D_{RegLC} is based on Dijkstra's algorithm which works only for non-negative arc costs, so not all potential functions can be used. A potential function π is *feasible*, if $c^{\pi}_{(v,s_v)(w,s_w)l}$ is non-negative for all $(v, w, l) \in A$. Note that if π' and π'' are feasible potential functions, then $\max(\pi', \pi'')$ is a feasible potential function [9].

Good bounds can be produced by using landmarks and the triangle inequality [9]. The main idea is to select a small set of nodes $\ell \in \mathcal{L} \subset V$ (also called *landmarks*), appropriately spread over the network, and pre-compute all distances of shortest paths between the landmarks and any other node. By using these *landmark distances* and the triangle inequality, lower bounds on the distances between any two nodes can be derived. Finding good landmarks is difficult and several heuristics exist [9,10]. SDALT applies these concepts to speed-up D_{RegLC} by using the following potential function (see also Figure 1):

$$\pi(v, s) = \max \bigcup_{\ell \in \mathcal{L}} \{d'(\ell, t, s) - d'(\ell, v, s), d'(v, \ell, s) - d'(t, \ell, s), 0\} \qquad (1)$$

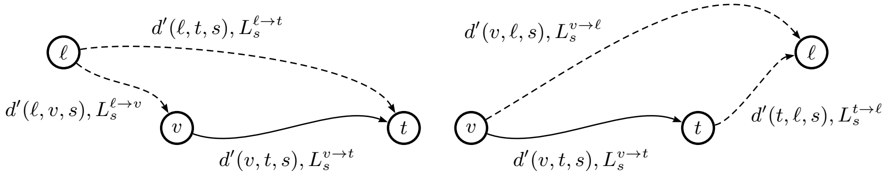

$d'(\ell, t, s), L_s^{\ell \to t}$
ℓ
$d'(\ell, v, s), L_s^{\ell \to v}$
v
t
$d'(v, t, s), L_s^{v \to t}$
$d'(v, \ell, s), L_s^{v \to \ell}$
ℓ
v
t
$d'(t, \ell, s), L_s^{t \to \ell}$
$d'(v, t, s), L_s^{v \to t}$

Fig. 1. Landmark distances for SDALT

$d'(i, j, s)$ denotes the *constrained landmark distance*, which is the travel time on the shortest path from (i, s) to (j, s') for some $s' \in F_0$ *constrained* by a regular language $L_s^{i \to j}$. Here lies the major conceptual difference between SDALT and uniALT. Different from uniALT, SDALT does not use Dijkstra's algorithm to determine landmark distances, but uses the D_{RegLC} algorithm instead. In this way, it is possible to constrain the landmark distance calculation by a regular languages $L_s^{i \to j}$ which will be derived from L_0. Thus SDALT is able to already consider the constraints given by L_0 during the preprocessing phase. Note that $d'(v, t, s)$ is constrained by $L_s^{v \to t} = L_0^s$. L_0^s is equal to L_0 except that the initial state of $\mathcal{A}_0$ is replaced by s. Intuitively L_0^s represents the constraints of L_0 to be considered for the remaining portion of the shortest path from an arbitrary node (v, s) to the target.

In [14] three methods, (bas), (adv), and (spe) have been presented on how to choose $L_s^{\ell\to t}$, $L_s^{\ell\to v}$, $L_s^{v\to\ell}$, $L_s^{t\to\ell}$ used to constrain the calculation of $d'(\ell,t,s)$, $d'(\ell,v,s)$, $d'(v,\ell,s)$, $d'(t,\ell,s)$, respectively, in such a way that the resulting potential function (Equation 1) results feasible and provides correct lower bounds. They are based on Proposition 1. Note that the concatenation of two regular languages L_1 and L_2 is the regular language $L_3 = L_1 \circ L_2 = \{v \circ w | (v,w) \in L_1 \times L_2\}$. E.g., if $L_1 = \{a,b\}$ and $L_1 = \{c,d\}$ then $L_1 \circ L_2 = L_3 = \{ac, ad, bc, bd\}$.

Proposition 1 ([14]). *For all $s \in S$, if $L_s^{\ell\to v} \circ L_s^{v\to t} \subseteq L_s^{\ell\to t}$, then $d'(\ell,t,s) - d'(\ell,v,s)$ is a lower bound for the distance $d'(v,t,s)$. Equally, if $L_s^{v\to t} \circ L_s^{t\to\ell} \subseteq L_s^{v\to\ell}$ then $d'(v,\ell,s) - d'(t,\ell,s)$ is a lower bound for $d'(v,t,s)$.*

3 Label Correcting State Dependent uniALT: lcSDALT

SDALT works correctly only if reduced arc costs are non-negative. It turns out, however, that by violating this condition often tighter lower bounds can be produced. This compensates, at least in our scenario, the additional computational effort required to remedy the disturbing effects of negative reduced costs on the underlying Dijkstra's algorithm and in addition results in shorter query times and lower memory requirements. This is why we propose a variation of SDALT, which can handle negative reduced costs. The major impact of this is that *settled* nodes may be re-inserted into the priority queue for re-examination (*correction*). In our setting, the number of arcs with non-negative reduced arc costs is limited and we can prove that the algorithm may stop once the target node is extracted from the priority queue. We name the new algorithm Label Correcting State Dependent uniALT (lcSDALT). Note that here *label* refers to the distance labels and not to the labels on arcs.

3.1 Query

The algorithm lcSDALT is similar to D_{RegLC} and uniALT. As priority queue Q we use a binary heap. The pseudo-code in figure 2 works as follows: the algorithm maintains, for every visited node (v,s) in the product graph P, a tentative distance label $d_r(v,s)$ and a parent node $p(v,s)$. It starts by computing the potential for the start node (r,s_0) and by inserting it into Q (line 3). At every iteration, the algorithm extracts the node (v,s) in Q with the smallest key (the node is *settled*) and *relaxes* all outgoing arcs (line 9), i.e. checking and possibly updating the key and distance label for every node (w,s') where $s' \in \delta(l,s)$. More precisely, a new temporary distance label $d_{\mathsf{tmp}} = d_r(v,s) + c_{vwl}(\tau_{start} + d_r(v,s))$ is compared to the currently assigned distance label (lines 10). If it is smaller, it either inserts or re-inserts node (w,s') into the priority queue or decreases its key (line 14, 18, 21). Note that it is necessary to calculate the potential of a node (w,s') only the first time it is visited. The cost of arc (v,w,l) might be time-dependent and thus has to be evaluated for time $\tau_{start} + d_r(v,s)$. The algorithm terminates when a node (t,s) with $s \in F_0$ is settled.

```
function lcSDALT($G$,$r$,$t$,$\tau_{start}$,$L_0$)
  $d_r(v,s)$:= $\infty$, $p(v,s)$:=-1, $\pi_{v,s} := 0$, $\forall(v,s) \in V \times S$
  path_found:= false, $d_r(r,s_0)$:= 0, $k(r,s_0)$:= $\pi(r,s_0)$, $p(r,s_0)$:= -1
  insert $(r,s_0)$ in priority queue $Q$
  while $Q$ is not empty:
    extract $(v,s)$ with smallest key $k$ from $Q$
    if $v$ == $t$ and $s \in F_0$:
      path_found:= true, break
    for each $(w,s')$ s.t. $(v,w,l) \in A \wedge s' \in \delta(l,s)$:
      d$_{tmp}$:= $d_r(v,s)$ + $c_{vwl}(\tau_{start}$ + $d_r(v,s))$   //time-dependency
      if d$_{tmp}$ < $d_r(w,s')$:
        $p(w,s')$:= $(v,s)$, $d_r(w,s')$:= d$_{tmp}$
        if $(w,s')$ not in $Q$ and never visited:   //insert
          $\pi_{w,s'}$:= $\pi(w,s')$
          $k(w,s')$:= $d_r(w,s')$ + $\pi_{w,s'}$
          insert $(w,s')$ in $Q$
        elif $(w,s')$ not in $Q$:                   //re-insert
          $k(w,s')$:= $d_r(w,s')$ + $\pi_{w,s'}$
          insert $(w,s')$ in $Q$
        else:                                       //decrease
          $k(w,s')$:= $d_r(w,s')$ +  $\pi_{w,s'}$
          decreaseKey $w,s'$ in $Q$
    end for
  end while
```

Fig. 2. Pseudo-code lcSDALT

Correctness. The algorithm lcSDALT is based on D_{RegLC} and uniALT. It suffices to prove that when the target node (t,s'), $s' \in F_0$ is extracted from the priority queue, the algorithm can stop (see Lemma 1 and Proposition 2). Note that $\pi(t,s') = 0$, that $d_r^*(v,s)$ is the distance of the shortest path from (r,s_0) to (v,s), and that there are no negative cycles as arc costs and potentials are always non-negative.

Lemma 1. *The priority queue always contains a node (i,s') with key $k(i,s') = d_r^*(i,s') + \pi(i,s')$ and which belongs to the shortest path from (r,s_0) to (t,s'') where $s'' \in F_0$, $s' \in S$.*

Proposition 2. *If solutions exist, lcSDALT finds a shortest path.*

Proof. Let us suppose that a node (t,s), where $s \in F_0$, is extracted from the priority queue but its distance label is not optimal, so $d_r(t,s) \neq d_r^*(t,s)$. Node (t,s) has key $k(t,s_f) = d_r(t,s_f) + \pi(t,s) \neq d_r^*(t,s)$. By Lemma 1, this means that there exists some node (i,s') in the priority queue on the shortest path from (r,s_0) to (t,s) which has not been settled because its key $k(i,s') > k(t,s)$. This means $k(i,s') = d_r^*(i,s') + \pi(i,s') > d_r(t,s) + \pi(t,s) = k(t,s)$, which is a contradiction. □

Complexity Complexity of D_{RegLC}, SDALT as well as lcSDALT when a feasible potential function is used is equal to the complexity of Dijkstra's algorithm on the product graph P which is $O(m \log n)$ where $m = |A||S|^2$ and $n = |V||S|$ are the number of arcs and nodes of P. If the potential function is non-feasible the key of a node extracted from the priority queue could not be minimal, hence already extracted nodes might have to be *re-inserted* into the priority queue at a later point (line 18-20) and re-examined (*corrected*). In this case the complexity of lcSDALT is similar to the complexity of the Bellman-Ford algorithm (plus the time needed to manage the priority queue): $O(mn \log n)$.

3.2 Constrained Landmark Distances

For calculating the distance bounds for a generic node (v, s) of P, we give three procedures to determine $L_s^{\ell \to t}$, $L_s^{\ell \to v}$, $L_s^{v \to \ell}$, $L_s^{t \to \ell}$ (see Table 1). Remember, $L_s^{i \to j}$ denotes the language used to constrain the landmark distance calculation from (i, s) to (j, s'), $s' \in F_0$. The languages produced by Procedure 1 allow every combination of labels in Σ_0. The languages produced by Procedure 2 are similar but depend on the state s of the node (v, s). They allow every combination of labels in Σ_0 except those labels for which there is no longer any transition between states which are reachable from s. Procedure 3 is the trickiest and produces four distinct languages for a node (v, s). $L_s^{\ell \to t}$ is determined in such a way that all constraints given by $\mathcal{A}_0$ on shortest paths which pass by a node (v, s) on $\mathcal{A}_0$ will be considered. $L_s^{v \to \ell}$ considers only those constraints that apply to paths starting from (v, s) to a final node. $L_s^{\ell \to v}$ and $L_s^{t \to \ell}$ are derived in such a way that they put as few constraints as possible on the distance calculation but assure that Proposition 1 is valid. Consider a transportation network. The procedures are based on the intuition that fast modes of transportation which are excluded by L_0 should not be used to calculate the bounds. Also, if constraints infer a major detour from the unconstrained shortest path, this detour should also be considered by the landmark distance calculation.

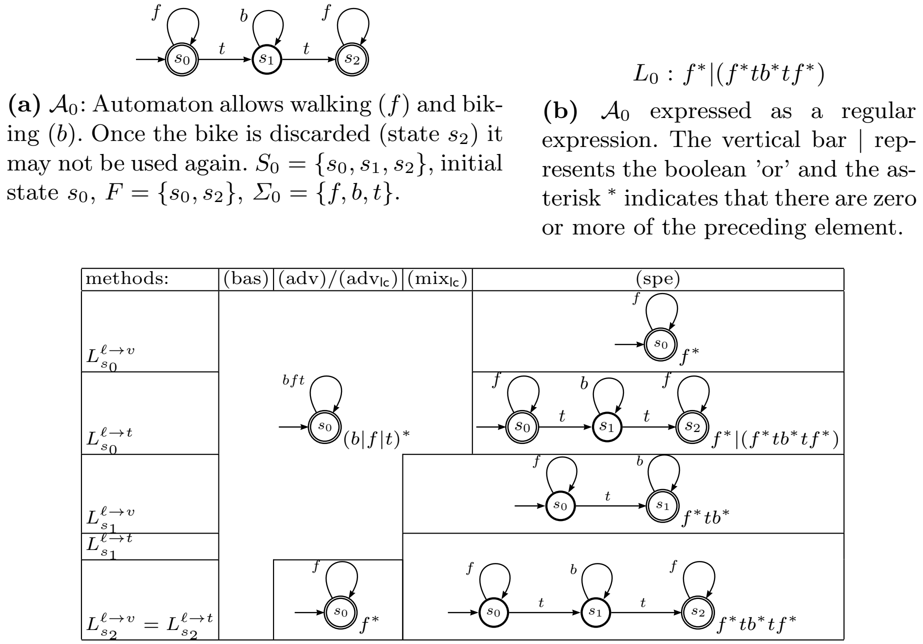
In [14] these procedures have been used with SDALT to produce three methods which assure that reduced costs are always non-negative: a basic method (bas) which applies Procedure 1 to all nodes (v, s); an advanced method (adv) which applies Procedure 2 to all nodes and uses a slightly modified potential function; a specific method (spe) which applies Procedure 3 to all nodes. These methods can be used with lcSDALT. Now we will present two new methods which can only be used with lcSDALT, as reduced costs may be negative:

($\mathbf{adv}_{\mathsf{lc}}$). Equal to (adv), this method applies Procedure 2 to all nodes (v, s) of P. Different to (adv) it uses Equation 1 as potential function and thereby considerably reduces the number of potentials to be calculated. (adv) applies the feasable potential function $\pi_{\mathsf{adv}}(v, s) = \max\{\pi(v, s_x) | s_x \in \overleftarrow{S}(s, \mathcal{A}_0)\}$.

($\mathbf{mix}_{\mathsf{lc}}$). The method (spe) applies the regular languages constructed by applying Procedure 3 for *each* state of L_0. This is space-consuming and bounds for nodes with certain states may be worse than those produced by Procedure 2. This is why we introduce a more flexible new method (mix_{lc}) which provides the possibility to freely choose for each state between the application of Procedure 2 or 3. This provides a trade-off between memory requirements and performance improvement. The right calibration for a given L_0 and the choice of whether to use Procedure 2 or 3 is determined experimentally.

Table 1. With reference to a generic RegLCSP where the SP is constrained by a regular language L_0 where $\mathcal{A}_0 = (S_0, \Sigma_0, \delta_0, s_0, F_0)$, the table shows three procedures to determine the regular language to constrain the distance calculation for a generic node (n, s) of the product graph P.

procedure	regular language and/or NFA
1	$L_s^{v\to\ell} = L_s^{t\to\ell} = L_s^{\ell\to v} = L_s^{\ell\to t} = L_{\mathsf{proc1}} = \{\Sigma_0^*\}$
	$L_{\mathsf{proc1}} : \mathcal{A}_{\mathsf{proc1}} = (\{s\}, \Sigma_0, \delta : \{s\} \times \Sigma_0 \to \{s\}, s, \{s\})$
2	$L_s^{v\to\ell} = L_s^{t\to\ell} = L_s^{\ell\to v} = L_s^{\ell\to t} = L_{\mathsf{proc2,s}} = \{\overrightarrow{\Sigma}(s, \mathcal{A}_0)^*\}$
	$L_{\mathsf{proc2,s}} : \mathcal{A}_{\mathsf{proc2,s}} = (\{s\}, \overrightarrow{\Sigma}(s, \mathcal{A}_0), \delta : \{s\} \times \overrightarrow{\Sigma}(s, \mathcal{A}_0) \to \{s\}, s, \{s\})$
3	$L_s^{\ell\to v} : \mathcal{A}_s^{\ell\to v} = (S = \overleftarrow{S}(s, \mathcal{A}_0), \Sigma = \overleftarrow{\Sigma}(s, \mathcal{A}_0), \delta_0\vert_{S\times\Sigma}, s_0, \{s\})$
	$L_s^{\ell\to t} : \mathcal{A}_s^{\ell\to t} = (S = \overleftarrow{S}(s, \mathcal{A}_0) \cup \overrightarrow{S}(s, \mathcal{A}_0),$
	$\Sigma = \overleftarrow{\Sigma}(s, \mathcal{A}_0) \cup \overrightarrow{\Sigma}(s, \mathcal{A}_0), \delta_0\vert_{S\times\Sigma}, s_0, F_0 \cap \overrightarrow{S}(s, \mathcal{A}_0)$
	$L_s^{v\to\ell} : \mathcal{A}_s^{v\to\ell} = (S = \overrightarrow{S}(s, \mathcal{A}_0), \Sigma = \overrightarrow{\Sigma}(s, \mathcal{A}_0), \delta_0\vert_{S\times\Sigma}, s, F_0 \cap S)$
	$L_s^{t\to\ell} : \mathcal{A}_s^{t\to\ell} = (\{s\}, \Sigma = \bigcap_{s\in F_0\cap\overrightarrow{S}(s,\mathcal{A}_0)}\{l\vert l \in \Sigma_0 \wedge \exists\delta_0(s,l) = \{s\}\},$
	$\delta : s \times \Sigma = \{s\}, s, \{s\})$



(a) $\mathcal{A}_0$: Automaton allows walking (f) and biking (b). Once the bike is discarded (state s_2) it may not be used again. $S_0 = \{s_0, s_1, s_2\}$, initial state s_0, $F = \{s_0, s_2\}$, $\Sigma_0 = \{f, b, t\}$.

(b) $\mathcal{A}_0$ expressed as a regular expression. The vertical bar | represents the boolean 'or' and the asterisk * indicates that there are zero or more of the preceding element.

methods:	(bas)	(adv)/(adv$_{\mathsf{lc}}$)	(mix$_{\mathsf{lc}}$)	(spe)
$L_{s_0}^{\ell\to v}$	$(b\vert f\vert t)^*$			f^*
$L_{s_0}^{\ell\to t}$				$f^*\vert(f^*tb^*tf^*)$
$L_{s_1}^{\ell\to v}$			f^*tb^*	
$L_{s_1}^{\ell\to t}$			$f^*tb^*tf^*$	
$L_{s_2}^{\ell\to v} = L_{s_2}^{\ell\to t}$		f^*		

Fig. 3. Example of a regular language L_0 (Scenario A) and its representation as an automaton (3a) and regular expression (3b). The table lists the languages used to constrain the landmark distance calculation for the different methods.. E.g., for (adv): $L_{s_0}^{\ell\to v} = L_{s_0}^{\ell\to t} = L_{s_1}^{\ell\to v} = L_{s_1}^{\ell\to t} : (b|f|t)^*$, $L_{s_2}^{\ell\to v} = L_{s_2}^{\ell\to t} : f^*$.

Memory Requirements and Time-Dependency. Memory requirements to hold preprocessing data for (adv_{lc}) grow linearly in respect to $|\mathcal{L}| \times |V| \times |S|$. ($\text{mix}_{\mathsf{lc}}$) may require in the worst case up to four times more space than (adv_{lc}). Similar to D, D_{RegLC} and lcSDALT can easily be adapted to time-dependent scenarios by selecting landmarks and calculating landmark distances by using the *minimum weight cost function* $\overline{c}_{ijl} = \min_\tau c_{ijl}(\tau)$. Potentials stay valid as long as arc weights only increase and do not drop below a minimal value [2,6].

4 Experimental Results

The algorithm is implemented in C++. A binary heap is used as priority queue. Similar to uniALT, dynamic additions of landmarks (max 6 landmarks) take place [15]. Experiments are run on a Intel Xeon, 2.6 Ghz, 16 GB RAM.

The multi-modal transportation network is based on road and public transportation data of the French region Ile-de-France. It consists of five layers: private bike, rental bike, walking, car and public transportation. Each layer is connected to the walking layer through transfer arcs which model the time needed to transfer from one layer to another. Each arc has exactly one associated label, e.g. f for arcs representing foot paths, p_{r} for rail tracks, c_{toll} for toll roads. The graph consists of approximately 4.1mil arcs and 1.2mil nodes (see Table 2).

Data of the public transportation network have been provided by STIF[1]. It includes geographical information, as well as timetable data on bus lines, tramways, subways and regional trains. Our model is similar to the one presented in [16]. Data for the car layer is based on road and traffic information provided by Mediamobile[2]. Arc labels and travel times are set according to the road type. Approximately 15% of the road arcs have a time-dependent cost function to represent changing traffic conditions throughout the day. The walking as well as the private and rental bike layers are based on road data (walking paths, cycle paths, etc.) extracted from geographical data freely available from OpenStreetMap. Arc cost equals walking or biking time. Arcs are replicated and inserted in each of the layers if both walking and biking are possible. The rental bike layer includes only arcs in the area around the city of Paris, where the rental bike service[3] is available. Bike rental stations serve as connection points between the walking layer and the rental bike layer as rental bikes have to be picked up and returned at bike rental stations. In addition, we introduced twenty arcs with label z between nodes of the foot layer. They represent foot paths close to locations of interest, and are used to simulate the problem of reaching a target and in addition passing by any pharmacy, grocery shop, etc.

We ran 500 test instances for five RegLCSP scenarios, see Figures 3a and 4. They have been chosen with the intention to represent real-world queries, which may arise when looking for constraint shortest paths on a multi-modal transportation network. Source node r, target node t, and start time τ_{start} are picked at

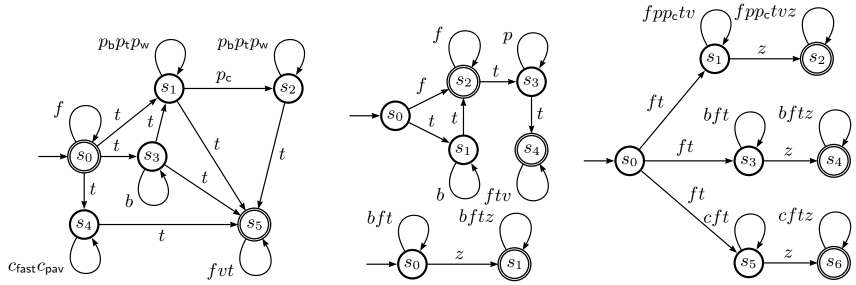
[1] Syndicat des Transports IdF, www.stif.info, data for scientific use (01/12/2010)

[2] www.v-trafic.fr, www.mediamobile.fr

[3] Vélib', www.velib.paris.fr

Table 2. Dimensions of the graph

layer	nodes	arcs	labels
walking	275 606	751 144	f (all arcs except 20 arcs with label z)
public transportation	109 922	292 113	p_{b} (bus, 72 512 arcs), p_{m} (metro, 1 746), p_{r} (tram, 1 746), p_{t} (train, 8 309), p_{c} (connection between stations, 32 490), p_{w} (walking station intern, 176 790), time-dependent 82 833
private bike	250 206	583 186	b
rental bike	38 097	83 928	v
car	613 972	1 273 170	c_{toll} (toll roads, 3 784), c_{fast} (fast roads, 16 502), c_{pav} (paved roads except toll and fast roads, 1 212 957), c_{unpav} (unpaved roads, 27 979), time-dependent 188 197
transfers	-	1 107 457	t (walking↔private bike 493 601, walking↔rental bike 2 404, walking↔car 572 604, walking↔public transportation 38 848)
tot	1 287 803	4 095 971	time-dependent arcs 271 030 (7 687 204 time points)



(a) Scenario B **(b)** Scenarios C and E **(c)** Scenario D

Fig. 4. Scenarios used for experimental evaluation. Note that labels $p_{\mathsf{b}}p_{\mathsf{m}}p_{\mathsf{r}}p_{\mathsf{t}}p_{\mathsf{w}}$ have been substituted by p and $c_{\mathsf{toll}}c_{\mathsf{fast}}c_{\mathsf{pav}}$ by c.

random. r and t always belong to the walking layer. We use 32 landmarks which are placed exclusively on the walking layer. Preprocessing for the different scenarios takes less than a minute. For each instance we compare running times of the different methods with D_{RegLC}. See Table 3 for experimental results. Column *time* gives the average running time in msec of the algorithm over 500 test instances. *SettNo*, *reInsNo* and *nCalcPot* give the average of the number of settled nodes, re-inserted nodes and calculated potentials. *MaxSett* gives the maximum number of settled nodes and *size* gives the size in MB of the file holding the landmark distances determined during the preprocessing phase. For (mix)$_{\mathsf{lc}}$ we specifiy all states for which Procedure 2 has been applied. For all other states Procedure 3 has been used.

4.1 Discussion of Experimental Results and Conclusive Remarks

The algorithm lcSDALT, by adopting the five methods (bas), (adv), (adv$_{\mathsf{lc}}$), (spe) and (mix$_{\mathsf{lc}}$), in comparison to D_{RegLC}, succeeds in directing the constrained shortest path search faster toward the target. (bas) works well in situations

Table 3. Experimental results

scenario	method	time(ms)	speed-up	settNo	maxSett	reInsNo	nCalPot	size(MB)	proc 2
A	D_{RegLC}	238	1.0	296 656	986 292	-	-	0	
	(bas)	19	12.4	21 476	950 992	0	76 144	311	
	(adv)	18	13.6	10 930	684 296	0	129 020	622	
	(adv)$_{\mathsf{lc}}$	13	18.1	10 927	682 981	11	68 820	622	
	(spe)	62	3.9	45 548	202 382	0	272 174	1 244	
	(mix)$_{\mathsf{lc}}$	13	*18.9	11 034	254 259	19	74 544	933	s_1
B	D_{RegLC}	624	1.0	610 375	1 487 766	-	-	0	
	(bas)	219	2.9	156 828	973 345	0	676 969	311	
	(adv)	190	3.3	51 258	346 549	0	2 295 030	1 866	
	(adv)$_{\mathsf{lc}}$	174	3.6	145 535	857 619	30	623 198	1 866	
	(spe)	237	2.6	176 766	667 125	0	543 388	2 177	
	(mix)$_{\mathsf{lc}}$	89	*7.0	60 053	350 237	87	343 585	1 244	s_0, s_3, s_4
C	D_{RegLC}	630	1.0	658 738	1 785 747	-	-	0	
	(bas)	515	1.2	414 553	1 721 989	0	1 981 520	311	
	(adv)	338	1.9	158 193	977 793	0	2 437 770	933	
	(adv)$_{\mathsf{lc}}$	263	2.4	223 070	2 217 083	64 034	919 554	933	
	(spe)	238	2.7	156 739	929 544	0	633 859	1 866	
	(mix)$_{\mathsf{lc}}$	149	*4.2	91 917	499 646	1 064	674 115	1 244	s_0, s_1, s_2, s_4
	(mix)$_{\mathsf{lc}}$	161	3.9	97 119	620 991	2 564	686 516	933	s_0, s_1, s_2
D	D_{RegLC}	1 722	1.0	1 248 060	3 085 428	-	-	0	
	(bas)	565	3.0	373 695	1 830 094	0	1 731 400	311	
	(adv)	695	2.5	352 969	1 557 331	0	3 297 090	1 244	
	(adv)$_{\mathsf{lc}}$	558	3.1	356 679	1 575 981	40	1 640 950	1 244	
	(spe)	476	3.6	337 849	1 791 504	0	734 068	2 488	
	(mix)$_{\mathsf{lc}}$	364	*4.7	221 631	1 278 208	48	953 802	2 177	s_0
E	D_{RegLC}	764	1.0	795 822	1 458 519	-	-	0	
	(bas)	143	5.3	115 941	598 273	0	407 108	311	
	(adv)	119	6.4	115 941	598 273	0	411 869	311	
	(adv)$_{\mathsf{lc}}$	120	6.4	116 042	598 273	32	406 877	311	
	(spe)	25	30.3	27 532	216 389	0	109 125	622	
	(mix)$_{\mathsf{lc}}$	22	*34.4	23 607	145 308	608	85 776	933	s_0

where L_0 excludes a priori fast transportation modes (Scenario A). (adv) gives a supplementary speed-up in cases where initially allowed fast transportation modes are excluded from a later state on A_0 onward. (spe) has a positive impact on running times for scenarios where the visit of some infrequent labels is imposed by L_0 (Scenario E) or the use of fast transportation modes is somehow limited (Scenario B). Finally, the new methods (adv$_{\mathsf{lc}}$) and (mix$_{\mathsf{lc}}$) prove to be very efficient. (adv$_{\mathsf{lc}}$) runs up to 20% faster than (adv) as it substantially reduces the number of calculated potentials, especially for larger automata. The extra flexibility provided by (mix$_{\mathsf{lc}}$) often results in a reduction of running time (up to 50%) and a reduction of memory space.

Recent works on finding constrained shortest paths on multi-modal networks reported speed-ups of different orders of magnitude. They succeed in doing this by using contraction hierarchies and by either limiting the constraints which can be imposed on the shortest paths [17] or by identifying homogenous regions (arcs with the same label) of the network and by applying contractions only to those regions [7]. lcSDALT does not provide such speed-ups but can handle more difficult constraints than the one considered in [17] and works on dis-homogenous graphs, such as the one considered in this paper. Also, time-dependent cost functions on arcs can be easily incorporated. Therefore, lcSDALT seems a good ingredient to future more advanced algorithms on multi-modal networks.

References

1. Barrett, C., Bisset, K., Holzer, M., Konjevod, G., Marathe, M., Wagner, D.: Engineering Label-Constrained Shortest-Path Algorithms. In: Fleischer, R., Xu, J. (eds.) AAIM 2008. LNCS, vol. 5034, pp. 27–37. Springer, Heidelberg (2008)
2. Barrett, C., Bisset, K., Jacob, R., Konjevod, G., Marathe, M.V.: Classical and Contemporary Shortest Path Problems in Road Networks: Implementation and Experimental Analysis of the TRANSIMS Router. In: Möhring, R.H., Raman, R. (eds.) ESA 2002. LNCS, vol. 2461, pp. 126–138. Springer, Heidelberg (2002)
3. Barrett, C., Jacob, R., Marathe, M.: Formal-Language-Constrained Path Problems. SIAM Journal on Computing 30(3), 809 (2000)
4. Delling, D., Pajor, T., Wagner, D.: Accelerating Multi-modal Route Planning by Access-Nodes. In: Fiat, A., Sanders, P. (eds.) ESA 2009. LNCS, vol. 5757, pp. 587–598. Springer, Heidelberg (2009)
5. Delling, D., Sanders, P., Schultes, D., Wagner, D.: Engineering route planning algorithms. Algorithmics of Large and Complex Networks 2, 117–139 (2009)
6. Delling, D., Wagner, D.: Time-Dependent Route Planning. Online 2, 207–230 (2009)
7. Dibbelt, J., Pajor, T., Wagner, D.: User-Constrained Multi-Modal Route Planning. In: Proceedings of the 14th Workshop on Algorithm Engineering and Experiments (ALENEX 2012). SIAM (2012)
8. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik 1(1), 269–271 (1959)
9. Goldberg, A.V., Harrelson, C.: Computing the shortest path: A search meets graph theory. In: Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 156–165. SIAM (2005)
10. Goldberg, A.V., Werneck, R.: Computing point-to-point shortest paths from external memory. US Patent App. 11/115,558 (2005)
11. Hart, P., Nilsson, N., Raphael, B.: A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Transactions on Systems Science and Cybernetics 4(2), 100–107 (1968)
12. Ikeda, T., Imai, H., Nishimura, S., Shimoura, H., Hashimoto, T., Tenmoku, K., Mitoh, K.: A fast algorithm for finding better routes by AI search techniques. IEEE (1994)
13. Kaufman, D., Smith, R.: Fastest paths in time-dependent networks for intelligent vehicle-highway systems applications. Journal of Intelligent Transportation Systems 1(1), 1–11 (1993)
14. Kirchler, D., Liberti, L., Pajor, T., Wolfler Calvo, R.: UniALT for Regular Language Constrained Shortest Paths on a Multi-Modal Transportation Network. In: Kontogiannis, A.C., Spyros (eds.) 11th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems, pp. 64–75. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2011)
15. Nannicini, G., Delling, D., Liberti, L., Schultes, D.: Bidirectional A search for time-dependent fast paths. Experimental Algorithms 2(2), 334–346 (2008)
16. Pyrga, E., Schulz, F., Wagner, D., Zaroliagis, C.: Efficient models for timetable information in public transportation systems. Journal of Experimental Algorithmics 12(2), 1–39 (2007)
17. Rice, M., Tsotras, V.J.: Graph indexing of road networks for shortest path queries with label restrictions. In: Proceedings of the VLDB Endowment, pp. 69–80 (2010)

Efficient Enumeration of the Directed Binary Perfect Phylogenies from Incomplete Data*

Masashi Kiyomi[1], Yoshio Okamoto[2], and Toshiki Saitoh[3]

[1] School of Information Science,
Japan Advanced Institute of Science and Technology, Nomi, Japan
[2] Center for Graduate Education Initiative,
Japan Advanced Institute of Science and Technology, Nomi, Japan
[3] ERATO Minato Discrete Structure Manipulation System Project,
Japan Technology and Science Agency, Sapporo, Japan

Abstract. We study a character-based phylogeny reconstruction problem when an incomplete set of data is given. More specifically, we consider the situation under the directed perfect phylogeny assumption with binary characters in which for some species the states of some characters are missing. Our main object is to give an efficient algorithm to enumerate (or list) all perfect phylogenies that can be obtained when the missing entries are completed. While a simple branch-and-bound algorithm (B&B) shows a theoretically good performance, we propose another approach based on a zero-suppressed binary decision diagram (ZDD). Experimental results on randomly generated data exhibit that the ZDD approach outperforms B&B. We also prove that counting the number of phylogenetic trees consistent with a given data is #P-complete, thus providing an evidence that an efficient random sampling seems hard.

1 Introduction

One of the most important problems in phylogenetics is reconstruction of phylogenetic trees. In this paper, we focus on the character-based approach. Namely, each species is described by their characters, and a mutation corresponds to a change of characters. However, in the real-world data not all states of all characters are observable or reliable, which makes the data incomplete. Thus, we need a methodology that can cope with such incompleteness.

Following Pe'er et al. [1], we work with the *perfect phylogeny* assumption, which means that the set of all nodes with the same character state induces a connected subtree. All characters are *binary*, namely take only two values. Without loss of generality, assume that these two values are encoded by 0 and 1. Then, the phylogeny is *directed* in a sense that for each character a mutation from 0 to 1 is possible only once, but a mutation from 1 to 0 is impossible (this is also called the

* Partially supported by Grant-in-Aid for Scientific Research from Ministry of Education, Science and Culture, Japan, and Japan Society for the Promotion of Science, and by Exploratory Research for Advanced Technology (ERATO) from Japan Science and Technology Agency.

R. Klasing (Ed.): SEA 2012, LNCS 7276, pp. 248–259, 2012.

Camin–Sokal parsimony [2]). We consider the situation where for some species the states of some characters are unknown. Under this setting, Pe'er et al. [1] provided a polynomial-time algorithm to reconstruct a phylogenetic tree that can be obtained when the unknown states are completed, if it exists.

Although their algorithm can find a phylogenetic tree efficiently, it does not take the likelihood into account. This motivates people to look at optimization problems; namely we may introduce an objective function (or an evaluation function) and try to find a perfect phylogeny that maximizes the value of the function. For example, Gusfield et al. [3] looked at such an optimization problem and formulated it as an integer linear program. One big issue here is that these optimization problems tend to be NP-hard, and thus we cannot expect to obtain polynomial-time algorithms. Therefore, we need some compromise. If we insist on efficiency, then we need to sacrifice the quality of an obtained solution. This approach leads us to approximation algorithms. If we insist on optimality, then we need to sacrifice the running time. This approach leads us to exponential-time exact algorithms. However, techniques in the literature as Gusfield et al. [3] with these approaches use specific structures of the form of objective functions.

Our Results. The focus of this paper is the exact approach. However, unlike the previous work, we aim at *enumeration algorithms*, which give a more flexible framework for scientific discovery independent of the form of objective functions. The use of enumeration algorithms is highlighted in data mining and artificial intelligence. For example, the apriori algorithm by Agrawal and Srikant [4] enumerates all maximal frequent itemsets in a transaction database. It is not expected that such enumeration algorithms run faster than non-enumeration algorithms. Therefore, the goal of this paper is to examine a possibility and a limitation of enumerative approaches.

One of the difficulties in designing efficient enumeration algorithms is to avoid duplication. Suppose that we are to output an object, and need to check if this object was already output or not. If we store all objects that we output so far, then we can check it by going through them. However, storing them may take too much space, and going through them may take too much time. The number of obejcts is typically exponentially large. Our algorithm cleverly avoids such checks, but still ensures exhaustive enumeration without duplication.

It is rather straightforward to give an algorithm with theoretical guarantee such as polynomiality. Namely, a simple branch-and-bound idea gives an algorithm that has a running time polynomial in the input size and linear in the output size. Notice that an enumeration algorithm outputs all the objects, and thus the running time needs to be at least as high as the number of output objects. Thus, the linearity in the output size cannot be avoided in any enumeration algorithms.

However, such a theoretically-guaranteed algorithm does not necessarily run fast in practice. Thus, we propose another algorithm that is based on a zero-suppressed binary decision diagram (ZDD). A ZDD was introduced by Minato [5]. It is a directed graph that has a similar structure to a binary decision diagram (BDD). While a BDD is used to represent a boolean function in a compressed way, a ZDD only represents the satisfying assignments of the function in a compressed way (a formal definition will be given in Section 3). Furthermore, we may employ a lot of operations on ZDDs, called the family algebra, which can be used for efficient filtering and optimization with respect to some objective functions. A book of Knuth [6] devotes one section to ZDDs, and gives numerous applications as exercises.

Although the size of a constructed ZDD is bounded by a polynomial of the number of output objects, we cannot guarantee that the size of a ZDD that is created at the intermediate steps in the course of our algorithm is bounded. This means that we cannot guarantee a polynomial-time running time (in the input size and the output size) for our ZDD algorithm. However, the crux here is that the size of a constructed ZDD can be much smaller than the number of output objects. We exhibit this phenomenon in two ways. First, we give an example in which the number of phylogenetic trees is exponential in the input size, but the size of the constructed ZDD is polynomial in the input size. Second, we perform experiments on randomly generated data, and the result shows that our ZDD algorithm can solve more instances than a branch-and-bound algorithm. This suggests that the ZDD approach is quite promising.

Having enumeration algorithms, we can also count the number of phylogenetic trees. In particular, the branch-and-bound algorithm can count them in polynomial time in the input size and the output size. This naturally raises the following question: Is it possible to count them in polynomial time only in the input size? Note that since we only compute the number, we do not have to output each object one by one, and thus the linearity of the running time in the output size could be avoided. Such a polynomial-time counting algorithm could be combined with a branch-and-bound enumeration algorithm to design a random sampling algorithm. Namely, when we branch, we count the number of outputs in each subinstance in polynomial time, and choose one subinstance at random according to the computed numbers. For more on the connection of counting and sampling, we refer to a book by Sinclair [7].

We prove that this is unlikely. Namely, counting the number of phylogenetic trees for the incomplete directed binary perfect phylogeny is #P-complete. The complexity class #P contains all counting problems in which a counted object has a polynomial-time verifiable certificate. Since no #P-complete problem is known to be solved in polynomial time, the #P-completeness suggests the unlikeliness for the problem to be solved in polynomial time.

Graph Sandwich. Pe'er et al. [1] rephrased the incomplete directed binary perfect phylogeny problem as a bipartite graph sandwich problem. The graph sandwich problem, in general, was introduced by Golumbic et al. [8]. In the graph sandwich problem, we fix a class C of graphs, and we are given two graphs $G_1 = (V, E_1), G_2 = (V, E_2)$ such that $E_1 \subseteq E_2$. Then, we are asked to find a graph $G = (V, E) \in C$ such that $E_1 \subseteq E \subseteq E_2$. Golumbic et al. [8] proved that even for some restricted classes of graphs, the problem is

NP-complete. The subsequent results by various researchers also show that for a lot of cases the problem is NP-complete, even though the recognition problem for those classes can be solved in polynomial time (we will not include here a long list of literature). Thus, the result by Pe'er et al. [1] gives a rare example for which the graph sandwich problem can be solved in polynomial time.

Recently, the graph sandwich enumeration problem has been studied. Kijima et al. [9] studied the graph sandwich enumeration problem for chordal graphs. They provided efficient algorithms when G_1 or G_2 is chordal, where "efficient" means that it runs in polynomial time in the input size and linear time in the output size. Their approach was generalized by Heggernes et al. [10] to all sandwich-monotone graph classes. In this respect, this paper gives another example of efficient graph sandwich enumeration algorithms.

Organization. In Section 2, we introduce the problem more formally. In Section 3, we provide the algorithm based on ZDDs, and give an example in which the compression really works. In Section 4, we prove that the counting version is intractable. Section 5 gives experimental results. We conclude in the final section.

For missing details, we refer to the arXiv version [11].

2 Preliminaries

Due to the pairwise compatibility lemma (see, e.g., [12]), we may define our problem in terms of laminars. We adapt this view throughout the paper.

A sequence $\mathcal{S} = (S_1, \ldots, S_m)$ of subsets of a finite set S is a *laminar* if for every two $i, j \in \{1, \ldots, m\}$ the intersection $S_i \cap S_j$ is either S_i, S_j, or $\emptyset$.[1] In the *incomplete directed binary perfect phylogeny problem* (IDBPP), we are given two sequences $\mathcal{L} = (L_1, \ldots, L_m)$, $\mathcal{U} = (U_1, \ldots, U_m)$ of m subsets of S such that $L_i \subseteq U_i \subseteq S$ for all $i \in \{1, \ldots, m\}$, and the question is to determine whether there exists a laminar $\mathcal{S} = (S_1, \ldots, S_m)$ such that $L_i \subseteq S_i \subseteq U_i$ for all $i \in \{1, \ldots, m\}$. We call such a laminar a *directed binary perfect phylogeny* for $(S, \mathcal{L}, \mathcal{U})$. The IDBPP can be solved in polynomial time [1].

Let us briefly describe the correspondence to phylogenetic trees. The set S represents the set of species, and the indices $1, \ldots, m$ represent the characters. Then, S_i represents the set of species that has the character i. The species in L_i are recognized as those we know having the character i, and the species in $S \setminus U_i$ are recognized as those we know not having i.

In this paper, we consider the following variants that take the same input as the IDBPP. In the *counting version* of IDBPP, the objective is to output the number of directed binary perfect phylogenies. In the *enumeration version* of IDBPP, the objective is to output all the directed binary perfect phylogenies. Note that enumeration should be exhaustive, and also should not output the same object twice or more.

[1] Usually, a laminar is defined as a family of subsets, but for our purpose it is convenient to define as a sequence of subsets.

3 ZDD Approach

3.1 Introduction to ZDDs

Let $f\colon \{0,1\}^N \to \{0,1\}$ be an N-variate boolean function with boolean variables $x_1, \ldots, x_N$. We assume a linear order on the variables $\{x_1, \ldots, x_N\}$ as x_i precedes x_j if and only if $i < j$. A *binary decision diagram* (BDD) for f, denoted by $B(f)$, is a vertex-labeled directed graph with the following properties.

- There is only one vertex with indegree 0, called the *root* of $B(f)$.
- There are only two vertices with outdegree 0, called the *terminals* of $B(f)$.
- Each vertex of $B(f)$, except for the terminals, is labeled by a variable from $\{x_1, \ldots, x_N\}$.
- One terminal is labeled by 0 (called the 0-*terminal*), and the other terminal is labeled by 1 (called the 1-*terminal*).
- Each edge of $B(f)$ is labeled by 0 or 1. An edge labeled by 0 is called a 0-*edge*, and an edge labeled by 1 is called a 1-*edge*.
- Each vertex of $B(f)$, except for the terminals, has exactly one outgoing 0-edge and exactly one outgoing 1-edge.
- If there is a path from a vertex v to a non-terminal vertex u in $B(f)$, then the label of v is smaller than the label of u.
- A boolean assignment $\alpha\colon \{x_1, \ldots, x_N\} \to \{0,1\}$ satisfies f (i.e., $f(\alpha(x_1), \ldots, \alpha(x_N)) = 1$) if and only if there exists a path P from the root to the 1-terminal in $B(f)$ that satisfies the following condition: $\alpha(x_i) = 1$ if and only if there exists a vertex v on P labeled by x_i such that P traverses the 1-edge leaving v.

A BDD for a function f is not unique, and may contain redundant information. However, the following reduction rules turn a BDD into a smaller equivalent BDD. A *zero-suppressed binary decision diagram* (ZDD) for a function f is a BDD, denoted by $Z(f)$, for which the reduction rules cannot be applied.

1. If the outgoing 1-edge of a vertex v points to the 0-terminal and the outgoing 0-edge of a vertex v points to a vertex u, then we remove v and its outgoing edges, and reconnect the incoming edges to v to the vertex u.
2. If two vertices v, v' have the same label x_i, their outgoing 1-edges point to the same vertex u_1, and their outgoing 0-edges point to the same vertex u_0, then replace v, v' with a single vertex w with label x_i. The incoming edges to w are those to v, v', the outgoing 1-edge from w points to u_1, and the outgoing 0-edge from w points to u_0.

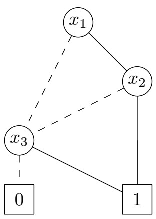


Fig. 1. A ZDD for the function $f(x_1, x_2, x_3) = (x_1 \wedge x_2 \wedge \overline{x_3}) \vee (\overline{x_2} \wedge x_3)$

Fig. 1 shows an example of a ZDD. The edges are assumed to be directed downward. A dashed line represents a 0-edge, and a solid line represents a 1-edge.

The *size* of a ZDD $Z(f)$ is defined as the number of vertices, and denoted by $|Z(f)|$. It is easy to observe that the size of ZDD $Z(f)$ is $O(NA)$ where A

is the number of satisfying assignments of f. However, this is merely an upper bound, and in practice the size can be much smaller. Thus, a ZDD for f gives a compressed representation of the family of all satisfying assignments of f. Especially, if we have a family $\mathcal{F}$ of subsets of a finite set S and consider a boolean function $f\colon \{0,1\}^S \to \{0,1\}$ such that $f(x) = 1$ if and only if $\{e \in S \mid x_e = 1\} \in \mathcal{F}$, then a ZDD for f compactly encodes the family $\mathcal{F}$.

There are a family of operations that can be performed on ZDDs. Here, we list those which we use in our algorithm. Let $f, f'\colon \{0,1\}^N \to \{0,1\}$ be boolean functions with variables $x_1, \ldots, x_N$, and ZDDs $Z(f), Z(f')$ be given. Then, a ZDD $Z(f \vee f')$ of the disjunction (logical OR) can be obtained in $O(|Z(f)||Z(f')|)$ time. Let $f^{[x_i=0]}\colon \{0,1\}^{N-1} \to \{0,1\}$ be a boolean function with variables $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_N$ obtained from f by $f^{[x_i=0]}(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_N) = f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_N)$. Then, a ZDD $Z(f^{[x_i=0]})$ can be found in $O(|Z(f)|)$ time. Similarly, we may define $f^{[x_i=1]}$, and a ZDD $Z(f^{[x_i=1]})$ can be found in $O(|Z(f)|)$ time.

3.2 ZDD-Based Enumeration Algorithm

We introduce a boolean variable $x_{i,e}$ for each pair (i, e) of an index $i \in \{1, \ldots, m\}$ and an element $e \in S$. Then, we consider the conjunction (logical AND) of the following conditions, which gives rise to a boolean function $f\colon \{0,1\}^{\{1,\ldots,m\}\times S} \to \{0,1\}$.

1. For every $i \in \{1, \ldots, m\}$, if $e \in L_i$, then $x_{i,e} = 1$.
2. For every $i \in \{1, \ldots, m\}$, if $e \in S \setminus U_i$, then $x_{i,e} = 0$.
3. For every distinct $i, j \in \{1, \ldots, m\}$, exactly one of the following three is satisfied.
   (a) For all $e \in S$, if $x_{i,e} = 1$, then $x_{j,e} = 1$.
   (b) For all $e \in S$, if $x_{i,e} = 0$, then $x_{j,e} = 0$.
   (c) For all $e \in S$, if $x_{i,e} = 1$, then $x_{j,e} = 0$.

We can easily see that if we set $S_i = \{e \in S \mid x_{i,e} = 1\}$ for every $i \in \{1, \ldots, m\}$, then $\mathcal{S} = (S_1, \ldots, S_m)$ is a directed binary perfect phylogeny for $(S, \mathcal{L}, \mathcal{U})$. Namely, the condition 1 translates to $L_i \subseteq S_i$; the condition 2 translates to $S_i \subseteq U_i$; the condition 3(a) translates to $S_i \cap S_j = S_i$; the condition 3(b) translates to $S_i \cap S_j = S_j$; the condition 3(c) translates to $S_i \cap S_j = \emptyset$.

These conditions naturally induce the following algorithm.

Algorithm: $\mathsf{ZDD}(S, \mathcal{L}, \mathcal{U})$
Precondition: S is a finite set, $\mathcal{L} = (L_1, \ldots, L_m)$, $\mathcal{U} = (U_1, \ldots, U_m)$, each member of $\mathcal{L}$ and $\mathcal{U}$ is a subset of S, and $L_i \subseteq U_i$ for every $i \in \{1, \ldots, m\}$.
Postcondition: Output a ZDD $Z(f)$ for the boolean function f over the variables $\{x_{i,e} \mid i \in \{1, \ldots, m\}, e \in S\}$ defined above, which encodes all the directed binary perfect phylogenies for $(S, \mathcal{L}, \mathcal{U})$.
Step 0: Let $g = \mathbf{1}$ be the constant-one function. Construct a ZDD $Z(g)$.
Step 1: For each $i \in \{1, \ldots, m\}$ and each $e \in S$, if $e \in L_i$, then construct $Z(g^{[x_{i,e}=1]})$ from $Z(g)$ and reset $g := g^{[x_{i,e}=1]}$.

Step 2: For each $i \in \{1, \ldots, m\}$ and each $e \in S$, if $e \in S \setminus U_i$, then construct $Z(g^{[x_{i,e}=0]})$ from $Z(g)$ and reset $g := g^{[x_{i,e}=0]}$.

Step 3: For each distinct $i, j \in \{1, \ldots, n\}$ and each $e \in S$, we perform the following.

Step 3-a: Let $g_1 := g^{[x_{i,e}=1, x_{j,e}=1]} \vee g^{[x_{i,e}=0]}$. Construct $Z(g_1)$ from $Z(g)$.

Step 3-b: Let $g_2 := g^{[x_{i,e}=0, x_{j,e}=0]} \vee g^{[x_{i,e}=1]}$. Construct $Z(g_2)$ from $Z(g)$.

Step 3-c: Let $g_3 := g^{[x_{i,e}=1, x_{j,e}=0]} \vee g^{[x_{i,e}=0]}$. Construct $Z(g_3)$ from $Z(g)$.

Step 3-d: Construct $Z(g_1 \vee g_2 \vee g_3)$ from $Z(g_1), Z(g_2), Z(g_3)$, and reset $g := g_1 \vee g_2 \vee g_3$.

Step 4: Output $Z(g)$ and halt.

Although the output size $|Z(f)|$ is bounded by $O(mnh)$ where $n = |S|$ and h is the number of directed binary perfect phylogenies for $(S, \mathcal{L}, \mathcal{U})$, we cannot guarantee that ZDDs that appear in the course of execution have such a bounded size. Thus, the algorithm could be quite slow or could stop due to memory shortage.

3.3 Example with Huge Compression

We exhibit an example for which the size of a ZDD is exponentially smaller than the number of directed binary perfect phylogenies. While the example is artificial, this indicates a possibility that our ZDD-based algorithm outperforms the branch-and-bound algorithm.

Consider the following example. Let $S = \{(i, j) \mid i \in \{1, \ldots, n\}, j \in \{0, 1, \ldots, k\}\}$. Then $|S| = (k+1)n$. For each $i \in \{1, \ldots, n\}$, let $L_i = \{(i, 0)\}$ and $U_i = \{(i, 0), (i, 1), \ldots, (i, k)\}$. As before, let $\mathcal{L} = (L_1, \ldots, L_n)$ and $\mathcal{U} = (U_1, \ldots, U_n)$. Then, the number of directed binary perfect phylogenies for $(S, \mathcal{L}, \mathcal{U})$ is 2^{kn}. On the other hand the size of a ZDD is $O(kn)$.

4 Hardness of Counting

Theorem 1. *The counting version of the IDBPP is #P-complete.*

Proof. We reduce the problem of counting the number of matchings in a (simple) bipartite graph, which is known to be #P-complete [13].

Let $G = (V, E)$ be a (simple) bipartite graph with a bipartition $V = A \cup B$ of the vertex set. For each vertex $v \in V$, we set up an element s_v, and let $S = (s_v \mid v \in V)$. Then, for each edge $e = \{a, b\} \in E$, where $a \in A$ and $b \in B$, let $L_e = \{s_a\}$ and $U_e = \{s_a, s_b\}$. Then, we set up $\mathcal{L} = (L_e \mid e \in E)$ and $\mathcal{U} = (U_e \mid e \in E)$. Note that for each $e \in E$, it holds that $L_e \subseteq U_e$. Thus, S, $\mathcal{L}$, and $\mathcal{U}$ form an instance of the IDBPP. We can show that the number of matchings in G equals the number of directed binary perfect phylogenies for $\mathcal{L}$ and $\mathcal{U}$. □

Table 1. The number of solved instances by B&B and ZDD out of 100 for each case

	B&B				ZDD			
(m, n)	$(50, 50)$	$(50, 100)$	$(100, 50)$	$(100, 100)$	$(50, 50)$	$(50, 100)$	$(100, 50)$	$(100, 100)$
$p = 0.1$	52	17	0	0	99	99	93	90
$p = 0.2$	0	0	0	0	57	33	6	4

5 Experiments

Data. We have used the program ms by Hudson [14] to generate a random data set without incompleteness that admits a directed binary perfect phylogeny $\mathcal{S} = (S_1, \ldots, S_m)$. Then, we have constructed L_i from S_i by removing each element of S_i independently with probability p, and constructed U_i from S_i by adding each element of $S \setminus S_i$ independently with probability p.

We have created 100 instances independently at random for each triple of values $(m, n, p) \in \{50, 100\} \times \{50, 100\} \times \{0.1, 0.2, 0.3, 0.4, 0.5\}$.

Implementation and Experiment Environment. We have implemented the algorithm ZDD described in Section 3 and another algorithm based on the branch-and-bound idea, which we call B&B. We have implemented both algorithms in C++. For the implementation of ZDD, we have used the library BDD+ developed by Minato. We introduced some heuristic methods to gain a practical performance. All programs have run on the machine with the following specification; OS: SuSE Linux Enterprise Server 10 (x86_64); CPU: Quad-Core AMD Opteron(tm) Processor 8393 SE (#CPUs 16, #Processors 32, Clock Freq. 3092MHz); Memory: 512GB.

The Number of Solved Instances. We have counted the number of instances that were solved by our implementation within two minutes for $p = 0.1, 0.2$. Here, "solved" means that the algorithm successfully halts. Table 1 shows the result. As we can see from the table, B&B was not able to solve most of the instances, even if they are small. On the other hand, ZDD was able to solve almost all instances when $p = 0.1$. However, when $p = 0.2$, the number of solved instances rapidly decreases.

Fig. 2 shows the accumulated number of solved instances by ZDD. Note that the horizontal axis is in log-scale. For $(m, n, p) = (50, 50, 0.1)$, ZDD solved each of the 99 instances within one second. For $(m, n, p) = (50, 100, 0.1)$, it solved each of the 99 instances within five seconds. This shows high effectiveness of the algorithm ZDD.

The Running Time of ZDD *and the Size of ZDDs.* Fig. 3 shows a scatter plot in which each point represents an instance solved by ZDD for $p = 0.1, 0.2$ with the running time (the horizontal coordinate) and the size of the ZDD constructed by ZDD (the vertical coordinate). Note that this is a log-log plot. We can see a tendency that the algorithm spends more time for instances with larger ZDDs. A simple ℓ_2-regression reveals that the spent time is dependent on the size almost linearly.

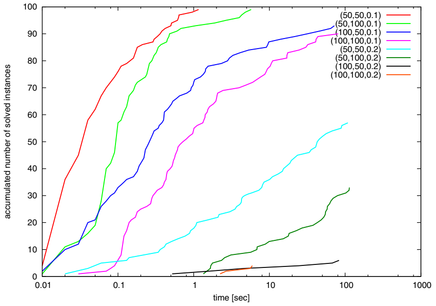


Fig. 2. The accumulated number of instances solved by ZDD for each case

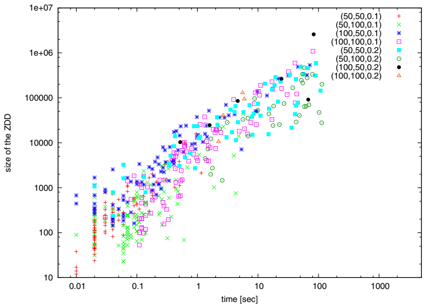


Fig. 3. The size of ZDDs and the running time of ZDD

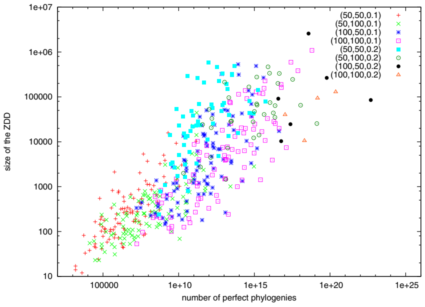


Fig. 4. The number of perfect phylogenies and the size of ZDDs

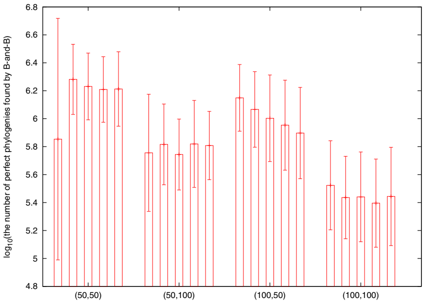


Fig. 5. The number of directed binary perfect phylogenies found by B&B for each case

The Number of Perfect Phylogenies and the Size of ZDDs. Fig. 4 shows a log-log scatter plot in which each point represents an instance solved by ZDD for $p = 0.1, 0.2$ with the number of perfect phylogenies (the horizontal coordinate) and the size of the ZDD constructed by ZDD (the vertical coordinate). The plot exhibits high compression rate of ZDDs. If we define the *logarithmic compression ratio* of ZDD by the logarithm (with base 10) of the size of ZDD divided by the number of perfect phylogenies, then Table 2 presents the means and the standard deviations of the logarithmic compression ratio of the instances solved by ZDD categorized by the choice of parameters. It shows the high-rate compression by ZDDs, and for larger values of parameters the compression ratios get larger. Among the solved instances, the logarithmic compression ratios range from -17.77 to -1.82. Namely, for the most extreme case, the size of ZDD is approximately $10^{17.77}$ times smaller than the number of perfect phylogenies.

The Number of Solutions Found by B&B*.* Unlike ZDD, the algorithm B&B can output some directed binary perfect phylogenies even if the execution is interrupted. Fig. 5 shows the averages of the logarithm of the numbers of directed binary perfect phylogenies (together with standard deviations) found by B&B within two minutes for each case: Four groups correspond to

Table 2. The means and the standard deviations of logarithmic compression ratios

p	0.1				0.2			
(m, n)	$(50, 50)$	$(50, 100)$	$(100, 50)$	$(100, 100)$	$(50, 50)$	$(50, 100)$	$(100, 50)$	$(100, 100)$
mean	-4.13	-7.25	-5.62	-10.00	-8.06	-13.61	-9.24	-14.04
s.d.	1.22	1.35	1.74	1.79	1.48	2.04	1.86	1.02

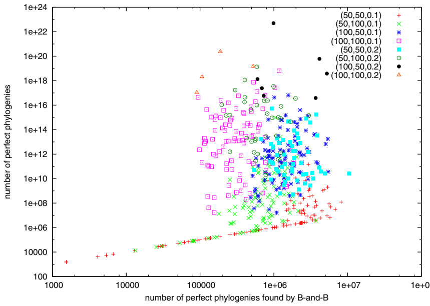


Fig. 6. The number of directed binary perfect phylogenies found by B&B for each case

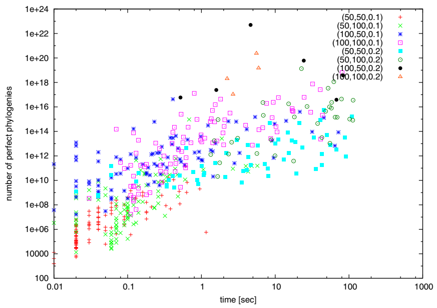


Fig. 7. The number of directed binary perfect phylogenies in the instances solved by ZDD for each case

$(m, n) = (50, 50), (50, 100), (100, 50), (100, 100)$ from left to right, and in each group there are five bars corresponding to $p = 0.1, 0.2, 0.3, 0.4, 0.5$ from left to right. When $(m, n, p) = (50, 50, 0.1)$, the standard deviation is high since about a half of the instances were solved within two minutes. Even for the seemingly difficult case $(m, n, p) = (100, 100, 0.5)$, B&B was able to find around $10^{5.4}$ perfect phylogenies. This suggests that B&B can be useful even if ZDD does not finish the computation.

The Number of Solutions Found by ZDD *and* B&B. Fig. 6 is a scatter plot in which each point represents an instance solved by ZDD with the number of directed binary perfect phylogenies found by B&B within two minutes (the horizontal coordinate) and the number of directed binary perfect phylogenies in the instance (the vertical coordinate). This shows the percentage of the directed binary perfect phylogenies that were found by B&B. Since this is a log-log plot, we can see that this percentage is quite low. There is one instance for $(m, n, p) = (100, 50, 0.2)$ with 49,614,003,829,608,756,019,200 perfect phylogenies for which B&B could only find 991,232. Thus the percentage is around 10^{-17} %. This really shows the power of ZDDs.

Running Time of ZDD *and the Number of Solutions.* Fig. 7 shows a scatter plot in which each point represents an instance solved by ZDD for $p = 0.1, 0.2$ with the running time (the horizontal coordinate) and the number of directed binary perfect phylogenies in the instance (the vertical coordinate). Note that this is a log-log plot. There is a weak tendency that the algorithm spends more time for instances with more directed binary perfect phylogenies. We can see that the algorithm is able to solve an instance with more than 10^{17} perfect phylogenies within one second.

6 Conclusion

We have presented the algorithm ZDD to enumerate all directed binary perfect phylogenies from incomplete data, and compare it with the algorithm B&B based on a simple branch-and-bound idea. Theoretically, B&B runs in polynomial time, but ZDD has no such guarantee. In experiments, ZDD solved more instances than B&B. This shows some gap between theory and practice, and it is desirable to have some theoretical justification why ZDD can outperform. We have theoretically exhibited an example for which the compression by a ZDD is effective. However, that example was artificial. The experiments also show ZDD can compress very well on random instances. It is desirable to obtain a more natural theoretical evidence why such a good compression is achieved.

The approach by ZDDs looks quite promising, and there must be more problems in bioinformatics that can get benefits from them.

Acknowledgments. We thank Jesper Jansson for bringing the problem into our attention, and Jun Kawahara and Yusuke Kobayashi for a fruitful discussion. We also thank the anonymous referees for detailed comments.

References

1. Pe'er, I., Pupko, T., Shamir, R., Sharan, R.: Incomplete directed perfect phylogeny. SIAM J. Comput. 33, 590–607 (2004)
2. Camin, J.H., Sokal, R.R.: A method for deducing branching sequences in phylogeny. Evolution 19, 311–326 (1965)
3. Gusfield, D., Frid, Y., Brown, D.: Integer Programming Formulations and Computations Solving Phylogenetic and Population Genetic Problems with Missing or Genotypic Data. In: Lin, G. (ed.) COCOON 2007. LNCS, vol. 4598, pp. 51–64. Springer, Heidelberg (2007)
4. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: Bocca, J.B., Jarke, M., Zaniolo, C. (eds.) VLDB, pp. 487–499. Morgan Kaufmann (1994)
5. Minato, S.: Zero-suppressed BDDs for set manipulation in combinatorial problems. In: DAC, pp. 272–277. ACM Press (1993)
6. Knuth, D.E.: The Art of Computer Programming Volume 4, Fascicle 1, Bitwise Tricks & Techniques, Binary Decision Diagrams. Pearson Education, Inc., Boston (2009)
7. Sinclair, A.: Algorithms for Random Generation & Counting: A Markov Chain Approach. Birkhäuser Boston, Boston Basel Berlin (1993)
8. Golumbic, M.C., Kaplan, H., Shamir, R.: Graph sandwich problems. J. Algorithms 19, 449–473 (1995)

9. Kijima, S., Kiyomi, M., Okamoto, Y., Uno, T.: On listing, sampling, and counting the chordal graphs with edge constraints. Theor. Comput. Sci. 411, 2591–2601 (2010)
10. Heggernes, P., Mancini, F., Papadopoulos, C., Sritharan, R.: Strongly chordal and chordal bipartite graphs are sandwich monotone. J. Comb. Optim. 22, 438–456 (2011)
11. Kiyomi, M., Okamoto, Y., Saitoh, T.: Efficient enumeration of the directed binary perfect phylogenies from incomplete data, arXiv:1203.3284 (2012)
12. Jansson, J.: Directed perfect phylogeny (binary characters). In: Kao, M.Y. (ed.) Encyclopedia of Algorithms, pp. 246–248. Springer, Heidelberg (2008)
13. Valiant, L.G.: The complexity of enumeration and reliability problems. SIAM J. Comput. 8, 410–421 (1979)
14. Hudson, R.R.: Generating samples under a Wright-Fisher neutral model of genetic variation. Bioinformatics 18, 337–338 (2002), http://home.uchicago.edu/~rhudson1/source/mksamples.html

Candidate Sets for Alternative Routes in Road Networks*

Dennis Luxen and Dennis Schieferdecker

Karlsruhe Institute of Technology,
Karlsruhe, Germany
{luxen,schieferdecker}@kit.edu

Abstract. We present a fast algorithm with preprocessing for computing multiple good alternative routes in road networks. Our approach is based on single via node routing on top of Contraction Hierarchies and achieves superior quality and efficiency compared to previous methods. The algorithm has neglectable memory overhead.

1 Introduction and Related Work

Today's requirements for routing services, be it in-car or as a web-service, ask for more than just computing the shortest or quickest paths. Thus it is desirable to not only present a single path to a user, but instead a set of paths which are perceived as reasonable alternatives.

We show how to engineer previous algorithms to provide reasonable alternative paths with better efficiency. Then, we build on the results and introduce the notion of *candidate via nodes* to further speed up the computation by an order of magnitude. We show how to perform query variants and how to conduct the preprocessing efficiently. Finally, we conduct an experimental study on the performance and quality of our method.

The shortest path problem can be solved by Dijkstra's seminal algorithm [1]. Unfortunately, it does not scale to large-scale instances. Heuristics to prune the search space like *that provide goal direction* [2,3] ease the problem. An early optimal and perfomant technique that provides substantial speedups is *arc flags*; originally conceived by Lauther [4,5]; later by Möhring et al. [6] and Köhler et al. [7]. The road network is partitioned into regions and each edge stores a flag to indicate if there is a shortest path into a region. Techniques exploiting the *hierarchy of a road network* follow the notion that sufficiently long routes will enter the arterial network at some point, e.g. enter a highway or national road. *Contraction Hierarchies (CH)* [8] have a convenient trade-off between preprocessing and query time. Road networks of continental size can be preprocessed within minutes and queries run in the order of about one hundred microseconds. CH heuristically order the nodes by some measure of importance and shortcut

* Partially supported by the German Research Foundation (DFG) within the Research Training Group GRK 1194 "Self-organizing Sensor-Actuator-Networks" and by DFG grant SA 933/5-1.

R. Klasing (Ed.): SEA 2012, LNCS 7276, pp. 260–270, 2012.

them in this order. This means that a node is removed from the graph and as few edges as possible are inserted to preserve shortest path distances. The original edges are augmented by the shortcut edges to build the search data structure. A query (a bidirected Dijkstra) only needs to follow edges that lead to more important nodes. Hence, the data structure forms a directed acyclic graph. Albeit the length of any shortest path is optimal, it may consist of shortcut edges that need to be recursively unpacked. The fastest CH variant is *CHASE* [9] that combines CH with arc flags. Its queries run in the order of ten microseconds.

Recently, Abraham et al. [10,11] give analyses of the performance of speedup techniques to Dijkstra's algorithm. Also, Abraham et al. [12] give an efficient implementation of the theoretical algorithm, which achieves distance query times below a single microsecond. Please note that we refer to various papers when speaking of Abraham et al. [10,11,12,13].

Alternative paths that combine two shortest paths over a *via node* are used by *Choice Routing* [14], also referred to as *plateau method*. The road network is modelled as a graph $G = (V, E)$ and shortest path trees are grown from nodes s and t. *Plateaus* $\langle u, \ldots, v\rangle$ running from node u to v are maximal paths that appear in both trees. They give candidates for natural alternative paths, i.e. follow the forward tree from s to u, then the plateau, and then the reverse tree from v to t. Although not entirely published, the plateau method provides alternatives of good quality in practice. Further discussion on this can be found in [13].

2 The Baseline Algorithm

Abraham et al. [13] define a class of *admissible* alternative paths. For a given s–t-pair and *via node* v the (via) path P_v is a concatenation of the two shortest paths s–v and v–t. The shortest path between s and t is called P_{opt} and the length of a path P_v is denoted by $l(P_v)$. Via path P_v has to be reasonable to be considered as a viable alternative and thus must obey three heuristic, but natural properties:

First, P_v has to be significantly different from P_{opt}. This states that the total length of the edges both paths share must only be a fraction of the length of the optimal path. Second, P_v has to be *T-locally optimal (T-LO)*, which means that every sufficiently short subpath P' of P_v must be a shortest path. In other words, every local decision along the alternative path must be reasonable. This is formalized by two properties. Every sufficiently short subpath $P' \subseteq P_v$ with $l(P') \leq T$ has to be a shortest path. If P' is a subpath of P_v and P'' is obtained by removing endpoints of P' then P' must also be a shortest path if $l(P') > T \wedge l(P'') < T$ holds. Third, the alternative path needs to have limited stretch. A path P_v is said to have $(1+\varepsilon)$ *uniformly bound stretch* (UBS) if every subpath $P' \subseteq P_v$ has stretch of at most $(1+\varepsilon)$. As such, every alternative should only be a fraction longer than a shortest path.

Given parameters $0 < \alpha < 1$, $0 \leq \gamma \leq 1$, and $\varepsilon \geq 0$ as well as the above properties, we formalize

Definition 1 (Admissible path). *A path P_v between s and t is an admissible alternative if*

a) $l(P_{opt} \cap P_v) \leq \gamma \cdot l(P_{opt})$ *(limited sharing),*
b) P_v *is* T*-locally optimal for* $T = \alpha \cdot l(P_{opt})$ *(local optimality), and*
c) P_v *has* $(1+\varepsilon)$*-UBS (uniformly bounded stretch).*

These measures require a quadratic number of shortest path queries to be verified, which is not feasible for a real-time setting. Thus, more practical algorithms are needed that have a narrower focus on easy computability. There exists a quick 2-approximation *(T-test)* for T-local optimality. Given a via path P_v and a parameter T, let x be the closest node on s–v that is at least T away from v or s. Likewise, y is the closest node on v–t that is also at least T away or s. A path P_v is said to *pass the T-test* if the portion of P_v between x and y is a shortest path.

Abraham et al. [13] give a practical solution based on a bidirectional Dijkstra (BD), called *X-BDV*, to compute single via paths that are reasonable and good alternatives. The algorithm incorporates ideas from the plateau method. An *Exploration Dijkstra* identifies potential alternative paths: A (forward) shortest path tree is grown from s, and another (backward) tree from t, until all nodes are settled that are not farther than $(1+\varepsilon) \cdot l(P_{opt})$ away from the root of their respective tree. Note that no admissible path can be any longer. Each node v that is settled in both search trees becomes a via node candidate and three measurements are computed in linear time: $l(P_v)$, the length of via path P_v, $\sigma(P_v)$, the amount of sharing of P_v with the optimal route, and $pl(P_v)$, the length of a longest plateau containing v. Note that if $pl(P_v) > T$, the T-test is always successful. These more practical measures are used to sort all candidates in non-decreasing order according to the priority function $f(P_v) = 2 \cdot l(P_v) + \sigma(P_v) - pl(P_v)$. The first path P_v is returned that is approximately admissable as described below.

Definition 2 (Approximately Admissible). *A path P_v between s and t is approximately admissible if the following three conditions hold*

1. $\sigma(P_v) < \gamma \cdot l(P_{opt})$ *(limited sharing),*
2. *successful T-test for* $T = \alpha \cdot l(P_v \backslash P_{opt})$ *(local optimality), and*
3. $l(P_v \backslash P_{opt}) < (1+\varepsilon) \cdot l(P_{opt} \backslash P_v)$ *(small stretch).*

Local optimality and stretch are defined with respect to the detour of the alternative. The above method yields the algorithm *X-CHV* [13] when combined with Contraction Hierarchies. The forward and backward (CH) search spaces of nodes s and t are explored. Nodes v in the forward search space are reached with a *forward distance* $l^{\uparrow}(P_{sv})$ and nodes in the backward search space with a *backward distance* $l^{\downarrow}(P_{vt})$. For nodes v that occur in both search spaces a preselection is run. Nodes are discarded, if the sum of forward and backward distance is longer than a certain fraction of the length of the shortest path: $l^{\uparrow}(P_{sv}) + l^{\downarrow}((P_{vt}) < (1+\varepsilon) \cdot l(P_{opt})$. Note that these distances are not necessarily correct but upper bounds. It is tested if the *approximated overlap*

$\sigma^{apx}(P_v)$ is no longer than a certain fraction of the length of the shortest path: $\sigma^{apx}(P_v) < (1+\varepsilon) \cdot l(P_{opt})$. Additionally, the following condition concerning the stretch must hold: $l^{\uparrow}(P_{sv}) + l^{\downarrow}(P_{vt}) - \sigma^{apx}(P_v) < (1+\varepsilon) \cdot (l(P_{opt}) - \sigma^{apx}(P_v))$. Remaining candidates are ranked according to the priority function of X-BDV. The exact path $\langle s..v..t \rangle$ is computed for nodes v in that order. The first node for which the properties of Definition 2 hold is selected as via node.

The success rate of X-CHV is inferior to X-BDV since search spaces are much narrower. To cope with the smaller success rate, Abraham et al. [13] introduce a relaxed exploration phase: The exploration query is allowed to search more nodes than the plain CH query. Let $p_i(u)$ be the i-th ancestor of u in the search tree. The *x-relaxed X-CHV query* prunes an edge (u, v) if and only if v precedes all vertices $u, p_1(u), \ldots, p_x(u)$ in the order of the CH. Note, the x-relaxed variant of *X-CHV*, with $x \in \{0, 3\}$, is the baseline of our work. This section ends the recap of previous work.

3 Engineering the Baseline Algorithm

Recall that the baseline is a two step approach. A bidirectional Exploration (CH) Dijkstra searches for via node candidates that are then tested for admissibility using a number of point-to-point (p2p) shortest path queries, which we call *Target (CH) Dijkstras*. The obvious approach to apply engineering is to handle the Target Dijkstras by faster methods than the normal Contraction Hierarchies query algorithm. For instance, we apply *CHASE* that computes these queries by exploiting additional arc flags [9]. This does not apply to Exploration Dijkstras, because search spaces would be too narrow. Storing all shortcuts pre-unpacked speeds up path computation as well. Both optimization have equal impact and result in an algorithm with query times of less than half of plain X-CHV. We refer to this straight-forward engineered baseline algorithm by *X-CHASEV*.

The analyses of Abraham et al. [10] show that speedup-techniques to Dijkstra's algorithm work especially well on certain classes of graphs in which all shortest paths out of a region are *covered* by a small node set. This theoretical analysis leads to the following assumption:

Assumption 1 (limited number of alternative paths). *If the number of shortest paths between any two sufficiently far away regions of a road network is small [10], so is the number of plateaus for Choice Routing [14]. Likewise the number of admissible paths of the algorithm of Abraham et al. [13] is small and can be covered by a small number of nodes.*

4 Single-Level via Node Candidates

We partition the graph and apply bootstrapping to generate *via node candidate sets* for pairs of partitions. Here, bootstrapping means that the query algorithm which is used later on to actually compute an alternative path is used during preprocessing as well.

Assume that for each pair of non-neighboring partitions, we have computed a set of via node candidates. Note that since candidates are already present, we do not need to identify them during an exploration step. Computing an alternative path for a given s–t-query now becomes straight-forward. We loop over all nodes v in the via node candidate set of the pair of partitions of s and t. For each v we check whether P_v is approximately admissible using the properties of Definition 2. The first approximately admissible path found is returned as the result. If no candidate is viable or if the size of the candidate set is zero, no alternative path is returned.

In a s–t query between neighboring partitions or within a single partition we perform X-CHASEV as fallback instead. The reason for this is that the number of candidates between those pairs of partitions and within a single one can be numerous. It is faster to use the fallback algorithm than to check pregenerated node sets in most of these cases.

Precomputating via node candidates starts with a partitioning of the underlying road network. A number of such schemes have been proposed before. We do not focus on that subproblem but refer to [15,16] instead. A set of via node candidates is generated greedily for each pair of partitions. A tentative via node set that keeps track of the candidates identified so far during preprocessing for each pair. We use the above algorithm with the tentative node set. If no alternative is found, we run X-CHASEV as bootstrapping to identify one. Whenever such a fallback run results in a new via node, it is inserted into the set of tentative via nodes.

4.1 Multi-level via Node Candidates

We propose a multi-level partitioning to compute via node candidates for neighboring pairs of partitions or within a single partition. The graph is further partitioned into an order of magnitude more partitions. The finer partitioning respects the coarser one in the sense that the nodes of a fine partition belong to one and only one of the coarse partitions. We do not run full preprocessing for all pairs of fine partitions. This would induce an amount of additional preprocessing steps (quadratic in the number of partitions). Our algorithm runs fine for most coarse partition pairs and we run the same preprocessing algorithm as before only on a subset of all fine partition pairs. These are the pairs for which origin an destination were too close together, i.e. in the same coarse partition or in neighboring ones. Note, we preprocess each non-neighboring fine partition pair that either belongs to the same or to a pair of neighboring coarse partitions. This implies only a linear amount of additional preprocessing work.

A query recurses to the multi-level partitioning for nodes of two neighboring coarse partitions or between nodes within the same coarse partition. When origin and destination are within the same or in neighboring fine partitions, plain X-CHASEV is run as fallback. Fine partitions are much smaller, and origin and destination are generally close to each other.

128 partitions are used for the arc flags of X-CHASEV. The number of explored nodes during a CHASE query with 128 partitions that do not belong to

the shortest path is tiny [9]. Hence, we do not see any benefit of investing time into the generation of arc flags for 1 024 partitions.

4.2 Further Engineering

The preprocessing is easily adaptable to shared-memory parallelism by preprocessing all pairs of partitions independently. This parallelization scales almost linearly with the number of processors until the memory bandwidth is reached. Most preprocessing runs verify the existence of a via node, but do not result in a new one. Sampling effectively decreases the preprocessing time when the sample is of reasonable size. E.g. running such a preprocessing on 1/16 of all of the pairs of boundary nodes for each partition pair results in only slightly inferior query performance.

Much effort during preprocessing is spent in search space exploration. The search space of each boundary node is required repeatedly. This can be hastened by about a factor of three by storing the search spaces of boundary nodes. Another tuning parameter is the order in which the nodes are stored in the tentative sets. We order by the number of how often a node occurs as a via node during preprocessing. This order is not necessarily the best of all orders. It depends on the order in which the pairs of boundary nodes are visited. Computing a best among all possible sorting orders, independent of the visiting order, is feasible and leads to slightly superior query times, but is computationally expensive. Note that selecting a via node greedily is of course faster since the first viable node is used, while selecting the via node that yields a best quality alternative is more expensive. Queries can be further accelerated by storing (forward and backward) search spaces of the via node candidate sets and also by storing the shortcuts pre-unpacked, as mentioned before.

5 Experiments

We implement the above algorithms in C++ using GCC's compiler with full optimizations. A binary heap is used as priority queue data structure. The experiments are conducted on two separate machines. Queries run on one core of an Intel Core i7-920 CPU (4 cores), clocked at 2.66 GHz with 12 GiB main memory. It is running Linux (kernel 2.6.34, gcc version 4.5.0). Parallel preprocessing is done on 4 AMD Opteron 6168 CPUs (12 cores each), clocked at 1.90 Ghz with 256 GiB main memory. This machine is running Linux (kernel 2.6.38, GCC version 4.5.2) and has roughly half the single-core performance compared to the Core i7 machine. Timings are done using the clock cycle counter available in 64 bit x86 CPUs.

5.1 Methodology

We test our approach on a road network of Western Europe provided by PTV AG for the 9th Dimacs Challenge [17]. It consists of 18 million nodes and 42 million edges and uses the travel time metric as edge weights. We partition the graph into 128 partitions using the algorithm of Sanders and Schulz [16], yielding an average edge cut of 6 360 and 91.8 boundary nodes per partition. Note that their partitioner does not necessarily yield connected partitions. On average each partition is adjacent to 5.2 neighboring ones. Our finer partioning into 1 024 partitions has an edge cut of 25 715 with an average of 46.5 boundary nodes and 5.3 neighbors. All figures are based on 10 000 random but fixed queries, unless otherwise stated. To compare against the results of [13], we use the same quality parameter values. Minimum (detour based) local-optimality is set to $\alpha = 0.25$, maximum sharing to $\gamma = 0.8$, and maximum stretch to $\varepsilon = 0.25$.

We test the performance of our algorithm in terms of both efficiency and quality according to Definition 1.

5.2 Engineered Baseline Algorithm

We compare our engineered baseline algorithm, X-CHASEV, against X-BDV and X-CHV. The results of Table 1 report on the query performance and path quality of the engineered baseline algorithm. As described in Section 3 the engineered baseline algorithm is faster by a factor of two than the other algorithms. We reimplemented both X-BDV and X-CHV algorithms. A direct comparison against the numbers of Abraham et al. [13] is unfair, since the heuristics of the underlying CH are different. X-BDV has the highest success rate and, of course, the highest query times by several orders of magnitude. This makes X-BDV unsuitable for any practical setting in which speed is a factor. The success rates of all three algorithms drop with the number of alternatives. The average path quality measures are very similar for all algorithms and identical for X-CHV and X-CHASEV by design. This is expected behavior.

5.3 Preprocessed Candidate Sets

Table 2 reports on the performance of the preprocessing required for the single- and multi-level algorithms. Preprocessing is run in parallel for up to three alternatives with relaxation either off ($x = 0$) or set to $x = 3$. Row *multi-level*

Table 1. Query performance of algorithms for alternatives $p = 1, 2, 3$

		performance		path quality					
		time	success	UBS[%]		sharing[%]		locality[%]	
p	algorithm	[ms]	rate[%]	avg	max	avg	max	avg	min
1	X-BDV	11 451.5	94.5	9.4	52.5	42.7	79.9	77.0	26.2
	X-CHV	1.2	75.5	9.2	48.1	44.7	80.0	74.8	26.3
	X-CHASEV	0.5	75.5	9.2	48.1	44.7	80.0	74.8	26.3
2	X-BDV	12225.8	80.6	11.5	43.0	60.0	80.0	78.6	27.0
	X-CHV	1.7	40.2	10.1	39.7	59.1	80.0	79.7	27.0
	X-CHASEV	0.7	40.2	10.1	39.7	59.1	80.0	79.7	27.0
3	X-BDV	13330.9	59.5	13.2	52.9	68.1	80.0	76.2	25.9
	X-CHV	2.3	14.2	10.0	33.4	65.0	79.9	84.3	30.9
	X-CHASEV	1.0	14.2	10.0	33.4	65.0	79.9	84.3	30.9

Table 2. Preprocessing results for normal ($x = 0$) and 3-relaxed ($x = 3$) algorithms

				candidate sets					
				p=1		p=2		p=3	
		time	size	empty	avg.	empty	avg.	empty	avg.
x	preprocessing	[h]	[kiB]	[%]	size	[%]	size	[%]	size
0	single-level	1.1	859	2.6	4.4	12.7	5.1	30.5	4.4
	multi-level	1.7	3 669	6.2	6.1	17.4	5.9	36.9	4.2
3	single-level	2.3	1 742	1.4	6.7	3.0	10.2	10.8	11.5
	multi-level	4.3	8 909	1.1	12.2	4.9	15.0	11.6	14.2

denotes the results of adding a finer partitioning compared to just the single-level approach. Numbers are listed for alternative $p = 1, 2, 3$ and only pertain to candidate sets of non-neighboring, non-equal pairs of partitions.

We note that preprocessing can be done on server hardware in a few hours for all of the experiments. The relative speedup on 48 cores is only about 28 due to the memory-bandwidth bottleneck, which is about 60% of the perfect linear speedup. The space overhead is more or less neglectable. Even for relaxation with $x = 3$ and multi-level partitioning the amount of additionally data is less than 9 MiB. Multi-level preprocessing shows a higher average number of candidates per partition pair as only partition pairs close to each other are processed. Fewer candidate sets remain empty using the relaxed algorithm.

X-CHASEV without candidate sets is compared to single- and multi-level candidate sets. Table 3 gives basic performance numbers. Algorithms with preprocessed candidate sets have query times well below 0.5 ms on average even for the third alternative, which is more than practical. We see that the multi-level optimization even improves the success rate, while the path quality remains at high level. Fallback rates to the baseline are generally low, 95% of the queries are covered by preprocessed via node candidates. We tested on omitting the fallback entirely for this setting and observe that results do not degrade noticeably. A third partitioning level would not give any further improvements to the performance of the query.

Table 3. Query performance with preprocessed candidate sets

		performance		path quality						candidate sets		
		time	success	UBS[%]		sharing[%]		locality[%]		v.cand.	fallb.	avg.
p	algorithm	[ms]	rate [%]	avg	max	avg	max	avg	min	[%]	[%]	tested
1	X-CHASEV	0.5	75.5	9.2	48.1	44.7	80.0	74.8	26.3	-	-	-
	single-level	0.1	80.7	9.8	48.1	48.5	80.0	75.8	26.3	92.4	4.9	1.9
	multi-level	0.1	81.2	9.9	48.1	48.6	80.0	75.8	26.3	96.5	0.6	2.0
2	X-CHASEV	0.7	40.2	10.1	39.7	59.1	80.0	79.7	27.0	-	-	-
	single-level	0.3	50.8	10.7	40.4	57.1	80.0	80.3	26.3	91.6	2.6	2.8
	multi-level	0.3	51.2	10.7	40.4	57.0	80.0	80.4	26.3	93.8	0.3	2.9
3	X-CHASEV	1.0	14.2	10.0	33.4	65.0	79.9	84.3	30.9	-	-	-
	single-level	0.4	24.8	10.7	41.0	59.9	79.9	82.5	27.9	88.7	1.1	3.8
	multi-level	0.4	25.0	10.7	41.0	59.8	79.9	82.6	27.9	89.7	0.1	3.8

Table 4. Query performance of multiple algorithms with 3-relaxation

	p=1			p=2			p=3		
	time	success	avg.	time	success	avg.	time	success	avg.
algorithm	[ms]	rate[%]	tested	[ms]	rate[%]	tested	[ms]	rate[%]	tested
X-BDV	11 451.5	94.5	-	12 225.8	80.6	-	13 330.9	59.5	-
X-CHV	3.4	88.5	-	4.3	64.7	-	5.3	38.0	-
X-CHASEV	2.7	88.5	-	3.2	64.7	-	3.8	38.0	-
single-level	0.2	90.0	2.22	0.4	70.2	3.8	0.6	44.0	5.6
multi-level	0.1	90.0	2.3	0.3	70.4	4.0	0.5	44.2	5.8

Results of the 3-relaxed variant of the query are given in Table 4. Numbers for X-BDV and X-CHV are shown for reference. We omit path quality since it is virtually unaffected and remains high.

The success rate further improves especially for the second and third alternative. Using precomputed candidate sets is faster by an order of magnitude than X-CHASEV and naturally much faster than the original method. We identify two reasons. A) an expensive (relaxed) Exploration Dijkstra has to be done only in the rare case when a fallback is needed. B) the average number of nodes to be tested as via node candidates is small and always less than half a dozen. Our single- and multi-level approaches deliver consistently higher success rates than the (engineered) baseline with the more speedup the more relaxation is applied.

Figure 1 shows success rates with varying Dijkstra ranks to test performance for local and long range queries alike. Success rates (left) are consistently equal or better for our algorithms than for the baseline. With relaxation (right) the numbers get even closer to the rates of X-BDV. The difference is less than 10%. Success rates are compared to X-BDV as the quality "gold standard" even though its computation is prohibitively high.

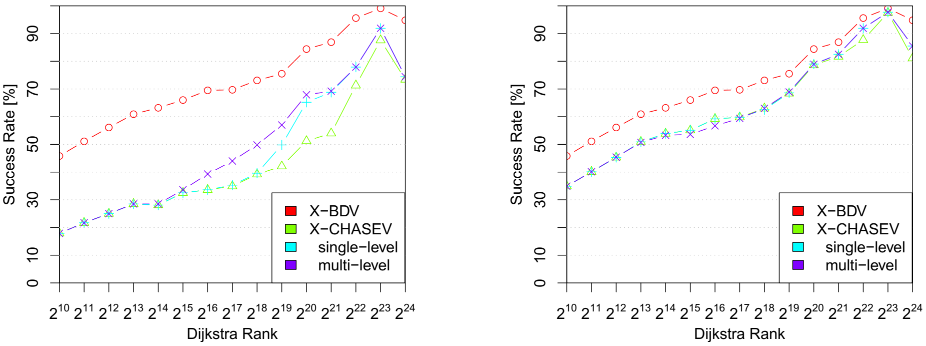


Fig. 1. Success rates according to Dijkstra rank: normal ($x = 0$, left) and 3-relaxed algorithm ($x = 3$, right). The Dijkstra rank of node v with respect to a node s is i if v is the i-th node removed from the priority queue of a unidirectional Dijkstra started at s. Each data point represents 1 000 queries.

6 Conclusion and Future Work

We introduced via node candidate sets. We showed their compact size, their efficient precomputation on large-scale networks and report one order of magnitude faster queries. We also show that success rates are higher than for previous algorithms with neglectable memory overhead. As a result of our extensive experimental evaluation, we conclude that Assumption 1 holds. There are a number of interesting directions for future work. We would like to explore the amount of preprocessing that is necessary to match the success rates of X-BDV. Also, we would like to use our method to generate alternative graphs similar to [18]. A challenging question is to extend alternative path computation to multiple via nodes. Combining the idea of *transit nodes* with via node candidates may be a great opportunity of future research. Instead of characterizing an alternative by a single via node, *via entrance nodes* for source and target partitions may provide access to an overlay network with fast lookups of alternatives.

Acknowledgments. The authors would like to thank Christian Schulz [16] for providing the partitionings and Daniel Delling [19] for providing arc flags for the partitioning with 128 cells, and Moritz Kobitzsch for great discussions.

References

1. Dijkstra, E.W.: A Note on Two Problems in Connexion with Graphs. Numerische Mathematik 1, 269–271 (1959)
2. Hart, P., Nilsson, N., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Transact. on Syst. Sci. and Cybernetics 4 (1968)
3. Goldberg, A.V., Harrelson, C.: Computing the Shortest Path: A* Search Meets Graph Theory. In: Proceedings of the 16th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA 2005). SIAM (2005)
4. Lauther, U.: Slow preprocessing of graphs for extremely fast shortest path calculations. In: Workshop on Computational Integer Programming at ZIB (1997)
5. Lauther, U.: An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background. Geoinformation und Mobilität—von der Forschung zur praktischen Anwendung 22, 219–230 (2004)
6. Möhring, R.H., Schilling, H., Schütz, B., Wagner, D., Willhalm, T.: Partitioning graphs to speedup dijkstra's algorithm. J. Exp. Algorithmics 11 (2007)
7. Köhler, E., Möhring, R.H., Schilling, H.: Acceleration of Shortest Path and Constrained Shortest Path Computation. In: Nikoletseas, S.E. (ed.) WEA 2005. LNCS, vol. 3503, pp. 126–138. Springer, Heidelberg (2005)
8. Geisberger, R., Sanders, P., Schultes, D., Delling, D.: Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In: McGeoch, C.C. (ed.) WEA 2008. LNCS, vol. 5038, pp. 319–333. Springer, Heidelberg (2008)
9. Bauer, R., Delling, D., Sanders, P., Schieferdecker, D., Schultes, D., Wagner, D.: Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm. ACM Journ. of Exp. Algorithmics 15, 1–31 (2010)
10. Abraham, I., Fiat, A., Goldberg, A.V., Werneck, R.F.: Highway Dimension, Shortest Paths, and Provably Efficient Algorithms. In: Proc. of the 21st Annual ACM–SIAM Symposium on Discrete Algorithms, SODA 2010 (2010)

11. Abraham, I., Delling, D., Fiat, A., Goldberg, A.V., Werneck, R.F.: VC-Dimension and Shortest Path Algorithms. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011. LNCS, vol. 6755, pp. 690–699. Springer, Heidelberg (2011)
12. Abraham, I., Delling, D., Goldberg, A.V., Werneck, R.F.: A Hub-Based Labeling Algorithm for Shortest Paths in Road Networks. In: Pardalos, P.M., Rebennack, S. (eds.) SEA 2011. LNCS, vol. 6630, pp. 230–241. Springer, Heidelberg (2011)
13. Abraham, I., Delling, D., Goldberg, A.V., Werneck, R.F.: Alternative Routes in Road Networks (2011), http://88.198.59.15/~delling/tmp/alternativesJEA.pdf
14. Cambridge Vehicle Information Tech. Ltd: Choice Routing, http://camvit.com
15. Delling, D., Goldberg, A.V., Razenshteyn, I., Werneck, R.F.: Graph Partitioning with Natural Cuts. In: 25th International Parallel and Distributed Processing Symposium (IPDPS 2011). IEEE Computer Society (2011)
16. Sanders, P., Schulz, C.: Engineering Multilevel Graph Partitioning Algorithms. In: Demetrescu, C., Halldórsson, M.M. (eds.) ESA 2011. LNCS, vol. 6942, pp. 469–480. Springer, Heidelberg (2011)
17. Demetrescu, C., Goldberg, A.V., Johnson, D.S. (eds.): The 9th DIMACS Implementation Challenge – Shortest Paths. American Mathematical Society (2006)
18. Bader, R., Dees, J., Geisberger, R., Sanders, P.: Alternative Route Graphs in Road Networks. In: Marchetti-Spaccamela, A., Segal, M. (eds.) TAPAS 2011. LNCS, vol. 6595, pp. 21–32. Springer, Heidelberg (2011)
19. Delling, D., Goldberg, A.V., Nowatzyk, A., Werneck, R.F.: PHAST: Hardware-Accelerated Shortest Path Trees. In: 25th International Parallel and Distributed Processing Symposium (IPDPS 2011). IEEE (2011)

Paired and Altruistic Kidney Donation in the UK: Algorithms and Experimentation*

David F. Manlove and Gregg O'Malley

School of Computing Science, University of Glasgow, Glasgow, UK
{david.manlove,gregg.omalley}@glasgow.ac.uk

Abstract. We study the computational problem of identifying optimal sets of kidney exchanges in the UK. We show how to expand an integer programming-based formulation [1,19] in order to model the criteria that constitute the UK definition of optimality. The software arising from this work has been used by the National Health Service Blood and Transplant to find optimal sets of kidney exchanges for their National Living Donor Kidney Sharing Schemes since July 2008. We report on the characteristics of the solutions that have been obtained in matching runs of the scheme since this time. We then present empirical results arising from the real datasets that stem from these matching runs, with the aim of establishing the extent to which the particular optimality criteria that are present in the UK influence the structure of the solutions that are ultimately computed. A key observation is that allowing 4-way exchanges would be likely to lead to a significant number of additional transplants.

1 Introduction

It is understood that transplantation is the most effective treatment that is currently known for kidney failure. In the UK alone, as of 31 March 2011 there were 6871 patients waiting on the transplant list for a donor kidney, with the median waiting time being 1153 days for an adult and 307 days for a child. Kidneys used for transplantation can come from both deceased and living donors. In the UK, around 38% of all kidney transplants are from living donors [14].

It is often the case that a patient requiring a kidney transplant has a willing donor, but due to blood- and/or tissue-type incompatibilities, the transplant cannot take place. However, in the UK, the Human Tissue Act 2004 and the Human Tissue (Scotland) Act 2006 (HTA) introduced, among other things, the

* Research financially supported by the National Health Service Blood and Transplant (NHSBT) between 1 April 2010 – 30 June 2010, and between 1 January 2012 - 30 June 2013 (project 10-11-ODT). We gratefully acknowledge collaboration with Rachel Johnson, Joanne Allen and with other staff at NHSBT. The views expressed here are those of the authors and not necessarily those of NHSBT. Research also financially supported by a Knowledge Transfer Account from the Engineering and Physical Sciences Research Council between 1 July 2011 - 31 December 2011. Finally, we would like to thank Péter Biró, Rob Irving, Kirstin MacDonald and Ana Viana for valuable input into this work.

R. Klasing (Ed.): SEA 2012, LNCS 7276, pp. 271–282, 2012.

legal framework required to allow the transplantation of organs between donors and patients with no genetic or emotional connection.

With the introduction of the HTA, a patient with an incompatible donor can now "swap" their donor with that of another patient in a similar position, via "kidney exchanges" that involve two or more incompatible patient–donor pairs. For example, a *pairwise (kidney) exchange* involves two incompatible patient–donor pairs (p_1, d_1) and (p_2, d_2), where d_1 is compatible with p_2, and d_2 is compatible with p_1: d_1 donates a kidney to p_2 in exchange for d_2 donating a kidney to p_1. *3-way exchanges* extend this concept to three pairs in a cyclic manner.

In a number of countries, centralised programmes (also known as *kidney exchange matching schemes*) have been introduced to help optimise the search for kidney exchanges. These include the USA [13,2,15], the Netherlands [9,10] and South Korea [17,16].

Following the introduction of the HTA, in early 2007 the UK established what has now become the National Living Donor Kidney Sharing Schemes (NLDKSS), administered by the National Health Service Blood and Transplant (NHSBT) (formerly UK Transplant) [8]. The purpose of the NLDKSS is two-fold: firstly to identify those pairs that are compatible with one another and then subsequently to optimise the selected set of kidney exchanges subject to certain criteria. It is the responsibility of NHSBT (and in particular its Kidney Advisory Group) to supply the scoring system that is used to measure the benefit of potential transplants, and the optimality criteria for the selection of kidney exchanges.

In general, it is seen as logistically challenging to carry out the transplants involved in a kidney exchange when the number of pairs involved increases. This is because all operations have to be performed simultaneously due to the risk of a donor reneging on his/her commitment to donate a kidney after their loved one has received a kidney. Mainly for this reason, at the present time the NLDKSS does not allow exchanges involving more than three pairs.

A kidney exchange matching scheme may also include *altruistic donors*, who do not have an associated patient and who are willing to donate a kidney to a stranger. An altruistic donor d_0 can either donate directly to a patient (without a donor) on the Deceased Donor Waiting List (DDWL), or else trigger a *domino paired chain* (DPC) [3] involving one or more incompatible patient–donor pairs: here d_0 donates to a patient p_1 in exchange for p_1's donor donating to the patient p_2 in the next pair in the chain, with the final donor donating to the DDWL. A DPC is *short* (resp. *long*) if it consists of one (resp. two) incompatible patient–donor pairs). At present the NLDKSS allows short but not long chains.

Kidney exchange has received considerable attention in the computer science, economics and medical literature in recent years [1,3,4,5,6,7,18,19,20,21]. It has been observed that when only pairwise exchanges are permitted, an optimal solution can usually (depending of course on the definition of optimality) be found in polynomial time using maximum weight matching in a general graph (see e.g., [5] for more details). However when pairwise and 3-way exchanges are allowed, the problem of finding a set of exchanges that maximises the number of transplants is NP-hard [1] and indeed APX-hard [5].

Abraham *et al.* [1], and independently Roth *et al.* [19], described two integer programming (IP)-based formulations of the problem of finding a maximum weight set of kidney exchanges, when both pairwise and 3-way exchanges are permitted (here, the weights can measure the benefit of potential transplants). Abraham *et al.* [1] showed that, due to scaling issues with the first of these models (the so-called *edge formulation*), the second model (the so-called *cycle formulation*) is the preferred way to model the problem using an IP.

In this paper we present an application-driven case study, showing how the cycle formulation can be extended in order to handle kidney exchange in the UK. In particular, we show how to model a complex set of criteria (given in Section 2) that form the definition of an *optimal* set of kidney exchanges. Although most of the criteria have not been explicitly mentioned elsewhere in the literature, they are natural rather than idiosyncratic. We have implemented the technique and it has been used by NHSBT to find optimal sets of kidney exchanges for the NLDKSS since July 2008. Our contribution in this paper is as follows:

1. We describe the IP constraints that are required in order to enforce the NLDKSS optimality definition (Section 3). The description could help to inform decision makers in other countries who are in the early stages of setting up a kidney exchange matching scheme.
2. We report on our practical experience over a 3-year period of using the technique to find optimal solutions for matching runs of the NLDKSS, which are carried out approximately every quarter (Section 4).
3. We present empirical results arising from a web application that is capable of automating the experimental comparison of solutions according to a range of different optimality criteria (Section 5). Again, these results arise from real datasets and indicate the extent to which the particular optimality criteria that are present in the UK influence the structure of the solutions that are ultimately computed. A key observation is that allowing 4-way exchanges would be likely to lead to a significant number of additional transplants.

2 The NLDKSS Optimality Criteria

The problem of finding an optimal set of kidney exchanges essentially corresponds to computing optimal cycle packings in weighted directed graphs. Suppose we have n incompatible patient–donor pairs $\{(p_i, d_i) : 1 \leq i \leq n\}$ and k altruistic donors $\{d_{n+i} : 1 \leq i \leq k\}$. We associate with each altruistic donor d_{n+i} a *dummy patient* p_{n+i} who is compatible with every donor d_j where $1 \leq j \leq n$.

We model the kidney exchange problem by forming a weighted directed graph $D = (V, A)$, where $V = \{v_1, \ldots, v_{n+k}\}$ and v_i corresponds to (p_i, d_i) $(1 \leq i \leq n+k)$. Moreover $(v_i, v_j) \in A$ if and only if d_i is compatible with p_j. In this way, 2-cycles and 3-cycles in D not involving an altruistic donor correspond to pairwise and 3-way exchanges respectively, whilst 2-cycles and 3-cycles in D involving an altruistic donor d_{n+i} correspond to short and long chains respectively, where in practice the final donor in the chain donates a kidney to the DDWL. (Note that our model handles both short and long chains.)

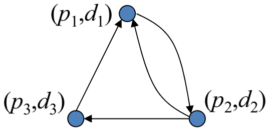


Fig. 1. Example of a 3-cycle containing a back-arc and an embedded 2-cycle

An arc (v_i, v_j) has a real-valued weight $w(v_i, v_j) > 0$ that arises from a scoring system employed by NHSBT to measure the potential benefit of a transplant from d_i to p_j. Factors involved in computing this weight include *waiting time* for p_j (based on the number of previous matching runs that p_j has been unsuccessfully involved in), p_j's *sensitisation* (based on calculated HLA antibody reaction frequency), *HLA mismatch* levels between d_i and p_j (which roughly speaking corresponds to levels of tissue-type incompatibility) and points relating to the difference in ages between d_i and d_j (see [8] for more details). The *weight* of a cycle c in D is the sum of the weights of the individual arcs in c.

A *set of exchanges* in D is a permutation π of V such that (i) for each $v_i \in V$, if $\pi(v_i) \neq v_i$ then $(v_i, \pi(v_i)) \in A$, and (ii) no cycle in π has length > 3. If $\pi(v_i) \neq v_i$ then v_i is said to be *matched*, otherwise v_i is *unmatched*. Suppose some $v_i \in V$ is unmatched. If $1 \leq i \leq n$, then neither d_i nor p_i will participate in a kidney exchange. However if $i > n$, d_i will donate directly to the DDWL. For this reason, we define the *size* of π (corresponding to the number of transplants yielded by this set of exchanges) to be the number of vertices matched by π plus the number of unmatched vertices corresponding to altruistic donors.

Given a 3-cycle c in D with arcs $(v_i, v_j), (v_j, v_k), (v_k, v_i)$, we say that c contains a *back-arc* if without loss of generality $(v_j, v_i) \in A$. In such a case we say that c contains an *embedded 2-cycle* involving arcs $(v_i, v_j), (v_j, v_i)$. A 3-cycle with a back-arc and an embedded 2-cycle is illustrated in Figure 1. An *effective 2-cycle* is either a 2-cycle or a 3-cycle with a back-arc.

A back-arc can be seen as a form of fault-tolerance in a 3-cycle. To understand why, consider the 3-cycle in Figure 1. If either p_3 or d_3 drops out (for example due to illness), then the pairwise exchange involving (p_1, d_1) and (p_2, d_2) might still be able to proceed. On the other hand, if either of the pairs (p_1, d_1) or (p_2, d_2) were to withdraw, then this pairwise exchange would have failed anyway. Thus the risk involved with a 3-way exchange, due to the greater likelihood (as compared to a pairwise exchange) of the cycle breaking down before transplants can be scheduled, is mitigated with the inclusion of a back-arc.

We now present the definition of an *optimal* set of exchanges for the NLDKSS, as determined by the Kidney Advisory Group of NHSBT.

Definition 1. *A set of exchanges π is* optimal *if:*

1. *the number of effective 2-cycles in π is maximised;*
2. *subject to (1), π has maximum size;*
3. *subject to (1)-(2), the number of 3-cycles in π is minimised;*
4. *subject to (1)-(3), the number of back-arcs in the 3-cycles in π is maximised;*
5. *subject to (1)-(4), the overall weight of the cycles in π is maximised.*

We give some intuition for Definition 1 as follows. The first priority is to ensure that there are at least as many 2-cycles and embedded 2-cycles as there would be in an optimal solution containing only 2-cycles. This is to ensure that the introduction of 3-way exchanges is not detrimental to the maximum number of pairwise exchanges that could possibly take place. Subject to this we maximise the total number of transplants (this is the number of unmatched altruistic donors, plus twice the number of pairwise exchanges and short chains, plus 3 times the number of 3-way exchanges and long chains). Subject to this we minimise the number of 3-way exchanges. Despite Criterion 1, this is still required: for example an optimal solution could either comprise three 3-way exchanges, each with a back-arc, or three pairwise exchanges and one 3-way exchange (both solutions have size 9 and contain three effective 2-cycles) – see Appendix A in [11] for an illustration. Clearly there is less risk of cycles breaking down with the second solution. Next the number of back-arcs in 3-way exchanges is maximised (note that a 3-way exchange could contain more than one back-arc). Finally we maximise the sum of the cycle weights.

3 Finding an Optimal Solution

In this section we describe an algorithm that uses a sequence of IP formulations to find an optimal set of kidney exchanges with respect to Definition 1. After each run of the IP solver, we use the optimal value calculated at that iteration to enforce a constraint that must be satisfied in subsequent iterations. This ensures that once Criteria $1..r$ in Definition 1 have been satisfied by an intermediate solution, they continue to hold when we additionally enforce Criterion $r + 1$ $(1 \leq r \leq 4)$. At the outset, an IP formulation, called the *basic IP model*, is created. This extends the cycle formulation of [1,19] in order to enable unmatched altruistic donors to be quantified. Recall that n is the number of incompatible patient–donor pairs and k is the number of altruistic donors. The basic IP model is then constructed as follows:

1. list all the possible cycles of length 2 and 3 in the directed graph D as $C_1, C_2, \ldots, C_m$, where, without loss of generality, the 2-cycles are $C_1, \ldots,$ C_{n_2}, the 3-cycles are $C_{n_2+1}, \ldots, C_{n_2+n_3}$, and the 3-cycles with back-arcs are $C_{n_2+1}, \ldots, C_{n_2+n_3^b}$ (so $m = n_2 + n_3$);
2. let x be an $(m + k) \times 1$ vector of binary variables $x_1, x_2, \ldots, x_{m+k}$, where for $1 \leq i \leq m$, $x_i = 1$ if and only if C_i belongs to an optimal solution, and for $1 \leq i \leq k$, $x_{m+i} = 1$ if and only if altruistic donor d_{n+i} is unmatched;
3. let A be an $(n + 2k) \times (m + k)$ $\{-1, 0, 1\}$-valued matrix, whose entries are all 0 apart from the following:
   (a) for $1 \leq i \leq n$ and $1 \leq j \leq m$, $A_{i,j} = 1$ if and only if C_j contains d_i;
   (b) for each i $(1 \leq i \leq k)$, in rows $n + 2i - 1$ and $n + 2i$:
      i. for $1 \leq j \leq m$, $A_{n+2i-1,j} = 1$ if and only if cycle C_j contains d_{n+i}, and for $1 \leq j \leq k$, $A_{n+2i-1,m+j} = 1$ if and only if $i = j$;
      ii. for $1 \leq j \leq m$, $A_{n+2i,j} = -1$ if and only if cycle C_j contains d_{n+i}, and for $1 \leq j \leq k$, $A_{n+2i,m+j} = -1$ if and only if $i = j$;

4. let b be an $(n + 2k) \times 1$ vector where:
   (a) for each i $(1 \le i \le n)$, $b_i = 1$;
   (b) for each i $(1 \le i \le k)$ $b_{n+2i-1} = 1$ and $b_{n+2i} = -1$;
5. let c be a $1 \times (m + k)$ vector of values corresponding to the coefficients of current objective criterion, e.g., c_j could be the length of C_j;
6. solve $\max cx$ such that $Ax \le b$.

We now provide some intuition for the model above. Part 3(a) (in combination with 4(a)) ensures that each patient–donor pair is involved in at most one cycle in any solution. Similarly 3(b)(i) (with 4(b)) ensures that each altruistic donor is involved in at most one cycle. 3(b)(i) (with 4(b)) also ensures that if a cycle involving an altruistic donor d_{n+i} is chosen then v_{n+i} must be matched. Similarly, 3(b)(ii) (with 4(b)) ensures that if no cycle involving an altruistic donor d_{n+i} is chosen then v_{n+i} must be unmatched.

We now describe the sequence of steps that is used in order to compute an optimal set of exchanges in D according to Definition 1. Item r in the following list corresponds to the step in the algorithm that enforces Criterion r (together with Criteria $1..r-1$) in the optimality definition. At each iteration we indicate the additional constraints that are added to the basic IP model and also the objective function used at each iteration (where appropriate).

1. *The number of effective 2-cycles is maximised.*
   Construct an undirected graph $G = (V, E)$ corresponding to the underlying digraph D, where the vertices in G and D are identical, and an edge in G corresponds to a 2-cycle in D (i.e., $\{v_i, v_j\} \in E$ if and only if $(v_i, v_j) \in A$ and $(v_j, v_i) \in A$). Compute N_2, the size of a maximum cardinality matching in G using Edmonds' algorithm [12]. Then add the following constraint:
$$x_1 + x_2 + \ldots + x_{n_2+n_3^b} \ge N_2. \quad (1)$$
2. *Subject to (1), the size is maximised.*
   Consider the basic IP model, together with (1), and with the objective $\max cx$, where $c_i = 2$ $(1 \le i \le n_2)$, $c_i = 3$ $(n_2 + 1 \le i \le n_2 + n_3)$ and $c_i = 1$ $(n_2 + n_3 + 1 \le i \le n_2 + n_3 + k)$. That is, for $r \in \{2, 3\}$, each variable representing an r-cycle has coefficient r, and each variable representing an altruistic donor has coefficient 1, where the objective is to maximise. After calculating the optimal value N, add the following constraint:
$$2x_1+\ldots+2x_{n_2}+3x_{n_2+1}+\ldots+3x_{n_2+n_3}+x_{n_2+n_3+1}+\ldots+x_{n_2+n_3+k} \ge N. \quad (2)$$
3. *Subject to (1)-(2), the number of 3-cycles is minimised.*
   Consider the basic IP model, together with (1)-(2), and with the objective $\min cx$, where $c_i = 0$ $(1 \le i \le n_2)$, $c_i = 1$ $(n_2 + 1 \le i \le n_2 + n_3)$ and $c_i = 0$ $(n_2 + n_3 + 1 \le i \le n_2 + n_3 + k)$. That is, each variable representing a 3-cycle has coefficient 1, whilst all others have coefficient 0. After calculating the optimal value N_3, add the following constraint:
$$x_{n_2+1} + \ldots + x_{n_2+n_3} \le N_3. \quad (3)$$

4. *Subject to (1)-(3), the number of back-arcs in the 3-cycles is maximised.* Let k_i be the number of back-arcs in cycle C_i $(n_2+1 \leq i \leq n_2+n_3)$. Consider the basic IP, together with (1)-(3), and with the objective $\max cx$, where $c_i = 0$ $(1 \leq i \leq n_2)$, $c_i = k_i$ $(n_2+1 \leq i \leq n_2+n_3)$ and $c_i = 0$ $(n_2+n_3+1 \leq i \leq n_2+n_3+k)$. That is, each variable corresponding to a 2-cycle or to an altruistic donor has coefficient 0, and each variable x_i representing a 3-cycle has coefficient k_i. Suppose that an optimal solution has value N_B. Add the following constraint:

$$k_{n_2+1}x_{n_2+1} + \ldots + k_{n_2+n_3^b}x_{n_2+n_3^b} \geq N_B. \tag{4}$$

5. *Subject to (1)-(4), the overall weight is maximised.* For each i $(1 \leq i \leq n_2+n_3)$, let w_i be the weight of cycle C_i. Consider the basic IP model, together with (1)-(4), and with the objective $\max cx$, where $c_i = w_i$ $(1 \leq i \leq n_2+n_3)$ and $c_i = 0$ $(n_2+n_3+1 \leq i \leq n_2+n_3+k)$. That is, each variable corresponding to a cycle has coefficient equal to the weight of that cycle, whilst each variable corresponding to an altruistic donor has coefficient 0. A solution to this final IP is an optimal set of exchanges relative to Definition 1.

We remark that an alternative to solving a series of IP formulations would be to solve a single IP relative to a weight function that captures the various criteria in the optimality definition (together with their priority levels) by assigning weights of successively decreasing orders of magnitude starting from Criterion 1 downwards. This is however impractical: due to the size of the datasets in practice, it would be computationally infeasible to work with such weights.

Another approach would be to assign smaller weights that somehow prioritise cycles with "good" characteristics, such as 3-cycles with back-arcs. However it is not clear how such weights should be defined, especially as theoretically there is no upper bound on the score of an arc as provided by NHSBT. Any attempt along these lines could never result in a concrete definition of exactly what is being optimised in an optimal solution, as we have obtained here.

4 NLDKSS in Practice

Prior to our involvement, NHSBT used an in-house algorithm that identified only pairwise exchanges. With the need to find both pairwise and 3-way exchanges, a new software application was developed based on the algorithm outlined in Section 3. At its heart the application uses the COIN-Cbc IP solver to solve each of the IP problems involved. COIN-Cbc was chosen due to its open licence agreement and the need to deploy the application commercially. Speed improvements using IBM ILOG CPLEX and Gurobi Optimizer were minimal with the current size of the datasets.

The application can be extended via a plugin architecture that allows constraints to be created, added or removed in a straightforward manner. This added flexibility allows our software to be easily adapted for use in other kidney

Table 1. Results arising from matching runs from July 2008 to October 2011

Matching run		2008		2009				2010				2011			
		Jul	Oct	Jan	Apr	Jul	Oct	Jan	Apr	Jun	Oct	Jan	Apr	Jun	Oct
Properties of D	#vertices	85	123	126	128	141	147	150	158	141	178	186	163	176	180
	#arcs	236	734	617	771	1248	901	832	876	533	939	1263	750	992	919
	#2-cycles	2	14	17	20	55	4	17	23	4	20	19	9	34	18
	#3-cycles	0	116	72	71	166	4	33	77	1	39	145	27	101	73
Identified solution	#2-cycles	1	6	5	5	4	0	3	2	3	3	3	0	5	7
	#3-cycles	0	3	1	2	7	2	1	6	0	2	10	4	4	5
	size	2	21	13	16	29	6	9	22	6	12	36	12	22	29
Actual transplants	#pairwise	1	4	5	2	3	0	2	4	0	3	2	0	2	6
	#3-way	0	0	0	0	2	2	0	3	0	1	5	2	4	3
	total	2	8	10	4	12	6	4	17	0	9	19	6	16	21

exchange matching schemes, whether that involves simply changing the order of constraints or adding completely new ones.

The application can either be accessed programatically through a web API or alternatively manually via a web interface[1]. The former version (along with several prototypes) has been used by NHSBT to find an optimal solution in each of the matching runs (occurring at roughly quarterly intervals), since July 2008.

Table 1 summarises the input to, and output from, each matching run between July 2008 and October 2011. In each case an optimal solution[2] was returned within a second (on a Linux Centos 5.5 machine with a Pentium 4 3GHz single core processor with 2Gb RAM) despite a gradually increasing pool of donors. In total 235 potential transplants were identified (47 pairwise and 47 3-way exchanges), which have resulted in 134 *actual* transplants[3] (34 pairwise and 22 3-way exchanges). Together with the 4 pairwise exchanges that were identified as part of the NLDKSS prior to our involvement, there have been a total of 142 actual transplants to date. Note that altruistic donors were not introduced into the NLDKSS until January 2012, and hence in Table 1, the number of vertices corresponds to the number of patient–donor pairs in each matching run.

The table shows that the matching run in January 2011 had the largest number of vertices and arcs in the underlying digraph, and the largest number of potential transplants of any matching run were identified (36). Even so, the digraph underlying the July 2009 dataset had a larger number of 2-cycles and 3-cycles. It is expected that the digraphs will become much denser once altruistic donors are introduced, and larger as awareness of the scheme grows over time.

[1] http://kidney.optimalmatching.com

[2] Note that the optimality criteria were slightly different from July 2008 to July 2009. See Appendix B in [11] for a more detailed discussion of this issue.

[3] In general not all transplants identified by the software will lead to operations in practice: one reason is that more detailed cross-matching between each donor and patient identified for transplant takes place after the matching run, which may lead to new incompatibilities being identified; also a donor or patient may become ill between the date of the matching run and the date of the operation.

5 Data Analysis Software and Empirical Results

Due to the complex nature of the optimality criteria used by the NLDKSS, it became obvious that there was a need to analyse the effect of each constraint. Furthermore, as the NLDKSS evolves it is likely that the maximum length of a DPC and/or the maximum length of cycle allowed in a solution will increase. In turn, these developments might lead to additional constraints being required. The effect of such changes is often difficult to quantify, as carrying out experimental comparisons can be time-consuming due to the significant development work required, and the execution of simulations.

To this end a web application[4] (referred to as the *toolkit*) was developed that allows NHSBT staff to examine the impact of adding/removing constraints, allowing longer altruistic chains, and increasing the maximum cycle length. The output from the application can determine information such as the size and weight of an optimal set of exchanges, the number of each type of exchange (i.e. pairwise, 3-way, etc.), and the number of DPCs. This information can be downloaded in the form of a spreadsheet.

In this section we report on an empirical analysis, using the toolkit, of the 14 matching runs that have taken place between July 2008 and October 2011. The aim is to determine the effect (in terms of the overall size or weight) of (i) prioritising pairwise exchanges, (ii) minimising the number of 3-way exchanges and maximising the number of back-arcs, and (iii) allowing 4-way exchanges in the optimality definition. Again, a Linux Centos 5.5 machine with a Pentium 4 3GHz single core processor with 2Gb RAM was used, and every optimal solution was computed in under two seconds.

First we examine the effect on the size of an optimal set of exchanges π in three cases concerning whether to prioritise 2-cycles or effective 2-cycles:

(A) when Definition 1 is unchanged;
(B) when Criterion 1 is omitted from Definition 1;
(C) when Criterion 1 is replaced by "maximise the number of 2-cycles".

Figure 2 shows the size of an optimal solution in each case, over the 14 matching runs. It reveals that on average if we relax the need to first maximise the number of 2-cycles or effective 2-cycles (case B from the above list) we would obtain only a single extra transplant per matching run. In contrast, if we require the number of pairwise exchanges alone to be maximised as first priority, then we would see a reduction in the number of transplants by, on average, 3 per matching run. In many cases obtaining a single extra transplant could make it worth changing the criteria, however in this case, given the desirable properties of embedded 2-cycles, the extra risk involved for the single extra transplant is unlikely to be justified.

We now analyse the effect on an optimal solution when we first apply Criteria 1 and 2 from Definition 1, then decide whether or not to apply Criteria 3 and 4 (i.e., minimise the number of 3-cycles and maximise the number of back-arcs

[4] `http://toolkit.optimalmatching.com`

respectively), and subsequently maximise the total weight. This gives four cases that correspond to the combinations of including / excluding Criteria 3 and 4.

It turns out that in each of these four cases, the solution output in each of the 14 matching runs is exactly the same, i.e., posting constraints to minimise the number of 3-ways exchanges or maximise the number of back-arcs has no effect. It appears that enforcing Criterion 1 (maximise the number of effective 2-cycles) results in a very small set of candidates for a solution that is optimal overall. If we no longer insist that Criterion 1 is enforced, then variations on the weight of an optimal solution are observed in the four cases. The additional time required to find a solution that satisfies Criteria 3 and 4 (as opposed to satisfying only Criteria 1, 2 and 5) is minimal (a solution is found in both cases in under two seconds for each dataset). Hence Criteria 3 and 4 should be retained as they may well have an impact for larger / denser datasets that are likely to feature in matching runs in the short / medium term.

We next determine the effect of increasing the maximum cycle size. Initially the NLDKSS allowed only pairwise exchanges in an optimal solution, but 3-way exchanges were permitted from April 2008 (subject to the condition that the number of effective 2-cycles is first maximised). Clearly extending the solution to allow for 4-way exchanges ought to increase further the number of transplants, but this must be set against the greater risk of such exchanges not proceeding.

In Figure 3 we show the total number of transplants at each of the 14 matching runs if an optimal set of exchanges π is defined as follows:

(A) maximise the size of π, allowing only 2-cycles;
(B) first maximise the number of effective 2-cycles, then subject to that maximise the total number of transplants, allowing only 2-cycles and 3-cycles;
(C) first maximise the number of effective 2-cycles, then subject to this maximise the number of *effective 3-cycles* (defined to be the number of 3-cycles plus the number of 4-cycles with embedded 3-cycles), then subject to this maximise the size of π, allowing 2-cycles, 3-cycles and 4-cycles.

As expected, allowing 4-way exchanges leads to an increased number of transplants: on average, an additional 4 transplants per matching run (compared to allowing only pairwise and 3-way exchanges). This number is smaller than the increase observed when allowing both pairwise and 3-way exchanges (compared to allowing only pairwise exchanges) where on average there are 7 additional transplants per matching run.

Finally we observe the effects of including altruistic donors in the dataset. Altruistic donors are set to be included in the NLDKSS from January 2012. In order to understand their impact in terms of increased numbers of transplants, the data from the January 2009 matching run was augmented by NHSBT staff with six altruistic donors known at that time. Of particular interest was to determine the benefits of including only short chains or both short and long chains (subject to the optimality criteria in Definition 1).

The test results indicated that, in the absence of altruistic donors, 15 transplants were obtained. When only short chains are permitted, 27 transplants were identified. Finally, if we allow both short and long chains, 31 transplants were

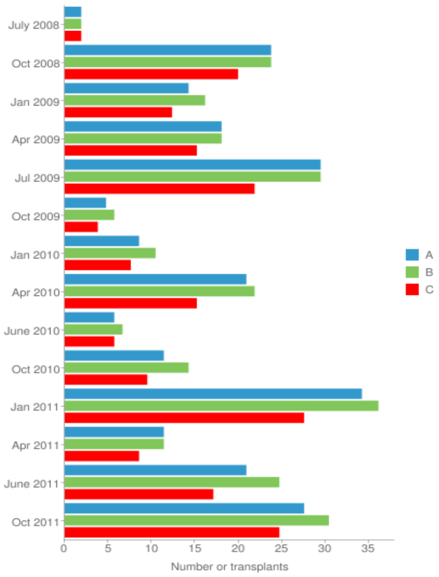


Fig. 2. Effect of prioritising pairwise exchanges

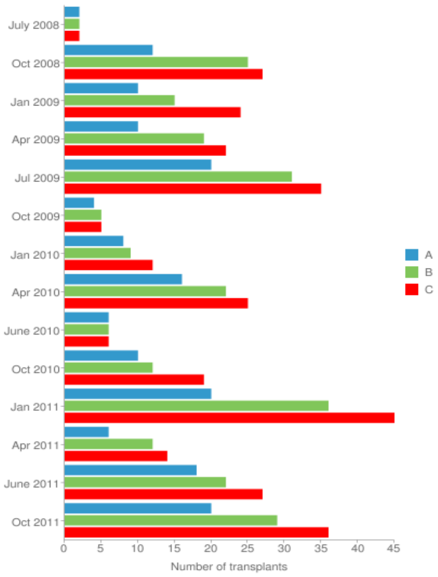


Fig. 3. Effect of increasing the maximum cycle size

identified. This shows that the difference between including only short chains, as opposed to both short and long chains, is of lesser importance than the benefit obtained by allowing short chains, as compared to not includng altruistic donors. However, given that any long chain must have at least one embedded 2-cycle, the risk of including long chains should be seen as minimal.

6 Future Work

Our case study has been driven by a particular practical application, and as such the empirical evaluation in Section 5 was based on real datasets (spanning a period of 42 months). However further experiments are required on artificially generated data which will facilitate both a larger number of trials and bigger datasets. This will provide important information on how far the software, in its current form, is likely to scale. Furthermore, using these datasets may provide greater insight into the effect a particular constraint has on the system.

Future work must also ensure that the algorithms described in this paper can scale as participation in the NLDKSS increases. It is anticipated that column generation techniques, along the lines of those described by Abraham *et al.* [1], will be required to ensure that we can meet the needs of the NLDKSS in the future, given the likelihood of larger datasets and the potential introduction of long chains and 4-way exchanges.

References

1. Abraham, D.J., Blum, A., Sandholm, T.: Clearing algorithms for barter exchange markets: enabling nationwide kidney exchanges. In: Proc. EC 2007, pp. 295–304. ACM (2007)
2. Alliance for Paired Donation, http://www.paireddonation.org
3. Ashlagi, I., Gilchrist, D.S., Roth, A.E., Rees, M.A.: Nonsimultaneous chains and dominos in kidney paired donation – revisited. American J. Transplantation 11(5), 984–994 (2011)
4. Ashlagi, I., Roth, A.: Individual rationality and participation in large scale, multi-hospital kidney exchange. In: Proc. EC 2011, pp. 321–322. ACM (2011)
5. Biró, P., Manlove, D.F., Rizzi, R.: Maximum weight cycle packing in directed graphs, with application to kidney exchange. Discrete Mathematics, Algorithms and Applications 1(4), 499–517 (2009)
6. Chen, Y., Kalbfleisch, J., Li, Y., Song, P., Zhou, Y.: Computerized platform for optimal organ allocations in kidney exchanges. In: Proc. BIOCOMP 2011 (2011)
7. Dickerson, J., Procaccia, A.D., Sandholm, T.: Optimizing kidney exchange with transplant chains: theory and reality. In: Proc. AAMAS 2012. Springer (to apppear, 2012)
8. Johnson, R.J., Allen, J.E., Fuggle, S.V., Bradley, J.A., Rudge, C.J.: Early experience of paired living kidney donation in the United Kingdom. Transplantation 86, 1672–1677 (2008)
9. Keizer, K.M., de Klerk, M., Haase-Kromwijk, B.J.J.M., Weimar, W.: The Dutch algorithm for allocation in living donor kidney exchange. Transplantation Proc. 37, 589–591 (2005)
10. de Klerk, M., Keizer, K.M., Claas, F.H.J., Witvliet, M., Haase-Kromwijk, B.J.J.M., Weimar, W.: The Dutch national living donor kidney exchange program. American J. Transplantation 5, 2302–2305 (2005)
11. Manlove, D.F., O'Malley, G.: Paired and Altruistic Kidney Donation in the UK: Algorithms and Experimentation. Technical Report, no. TR-2012-330, University of Glasgow, School of Computing Science (2012)
12. Micali, S., Vazirani, V.V.: An $O(\sqrt{|V|}\cdot|E|)$ algorithm for finding maximum matching in general graphs. In: Proc. FOCS 1980, pp. 17–27. IEEE (1980)
13. New England Program for Kidney Exchange, http://www.nepke.org
14. NHS Blood and Transplant. Transplant. Transplant Activity Report (2010-2011), http://www.organdonation.nhs.uk/ukt/statistics/transplant_activity_report/current_activity_reports/ukt/kidney_activity.pdf
15. Paired Donation Network, http://www.paireddonationnetwork.org
16. Park, K., Lee, J.H., Kim, S.I., Kim, Y.S.: Exchange living-donor kidney transplantation: Diminution of donor organ shortage. Transplantation Proc. 36, 2949–2951 (2004)
17. Park, K., Moon II, J., Kim II, S., Kim, Y.S.: Exchange-donor program in kidney transplantation. Transplantation Proc. 31, 356–357 (1999)
18. Roth, A.E., Sönmez, T., Utku Ünver, M.: Pairwise kidney exchange. J. Economic Theory 125, 151–188 (2005)
19. Roth, A.E., Sönmez, T., Ünver, M.U.: Efficient kidney exchange: Coincidence of wants in markets with compatibility based preferences. American Economic Review 97(3), 828–851 (2007)
20. Saidman, S.L., Roth, A.E., Sönmez, T., Ünver, M.U., Delmonico, S.L.: Increasing the opportunity of live kidney donation by matching for two and three way exchanges. Transplantation 81(5), 773–782 (2006)
21. Toulis, P., Parkes, D.C.: A random graph model of kidney exchanges: efficiency, individual-rationality and incentives. In: Proc. EC 2011, pp. 323–332. ACM (2011)

An Evaluation of Community Detection Algorithms on Large-Scale Email Traffic

Farnaz Moradi, Tomas Olovsson, and Philippas Tsigas

Computer Science and Engineering,
Chalmers University of Technology, Göteborg, Sweden
{moradi,tomasol,tsigas}@chalmers.se

Abstract. Community detection algorithms are widely used to study the structural properties of real-world networks. In this paper, we experimentally evaluate the qualitative performance of several community detection algorithms using large-scale email networks. The email networks were generated from real email traffic and contain both legitimate email (ham) and unsolicited email (spam). We compare the quality of the algorithms with respect to a number of structural quality functions and a logical quality measure which assesses the ability of the algorithms to separate ham and spam emails by clustering them into distinct communities. Our study reveals that the algorithms that perform well with respect to structural quality, don't achieve high logical quality. We also show that the algorithms with similar structural quality also have similar logical quality regardless of their approach to clustering. Finally, we reveal that the algorithm that performs link community detection is more suitable for clustering email networks than the node-based approaches, and it creates more distinct communities of ham and spam edges.

Keywords: Community detection, Email networks, Quality functions.

1 Introduction

Unfolding the communities in real networks is widely used to determine the structural properties of these networks. Community detection or clustering algorithms aim at finding groups of related nodes that are densely interconnected and have fewer connections with the rest of the network. These groups of nodes are called communities or clusters and they exist in a variety of different networks [9]. The problem of how to find communities in networks has been extensively studied and a substantial amount of work has been done on developing clustering algorithms (an overview can be found in [8,21]). However, there is no consensus on which algorithm is more suitable for which type of network. Therefore, a number of studies have experimentally compared the qualitative performance of different community detection algorithms on synthetic and benchmark graphs with built-in community structure [12,5]. However, these graphs are different from real-world networks as the assumptions they make are not completely realistic [8]. Delling et al. [6] have shown that the implicit dependencies between community detection algorithms, synthetic graph generators, and quality functions

R. Klasing (Ed.): SEA 2012, LNCS 7276, pp. 283–294, 2012.

used for assessing the qualitative performance of the algorithms make meaningful benchmarking very difficult. Therefore, empirical studies of the existing algorithms on real-world networks are crucial in order to evaluate different algorithms and to find the most suitable methods for different types of networks.

Moreover, community detection in real-networks has many different applications. Community detection algorithms can be used to find users with similar interests in a social network in order to provide recommendations to them, to group the peers that are geographically close in a peer-to-peer system to improve the performance of the system, or to detect the communities generated by malicious users in order to mitigate Sybil attacks [24]. In this paper, we study the community structure of a number of large *email networks* containing both legitimate *ham* and unsolicited *spam* emails. In an email network, the nodes represent email addresses and the edges represent email communications. In addition to a qualitative comparison of the algorithms, our goal is to find the best community detection algorithm that can separate spam and ham emails by clustering them into distinct communities. Such an algorithm can potentially be deployed in spam detection mechanisms that aim at mitigating the spam problem by looking at email traffic rather than email contents.

In order to achieve our goals, we have selected a number of broadly used community detection algorithms that are known to perform well on synthetic, benchmark, and a limited number of real graphs. In this study we evaluate and compare the qualitative performance of these algorithms when they are applied to large-scale email networks. Since the true community structure of our networks is unknown, it is important to use a quality measure to compare the algorithms. It is known that there is no single perfect quality metric for the comparison of the communities detected by different algorithms [2], therefore we use a number of *structural quality* functions such as modularity [17], coverage, and conductance [11], as well as a *logical quality* measure which is determined based on how homogeneous the edges inside the communities are. We use this measure to investigate and compare the ability of the selected algorithms in separating ham and spam emails into distinct communities.

The contributions of the paper are as follows. We show that there is a trade-off between creating high structural and high logical quality communities. Therefore, hierarchical and multiresolution algorithms which allow us to select the granularity of the clustering are more suitable to create the communities with the desired quality. We reveal that different algorithms that create communities with similar size distribution achieve similar structural and logical qualities, even though they use different approaches for clustering. Finally, we show that an algorithm that clusters networks based on the similarity of edges is superior to the algorithms that perform node-based clustering.

The rest of this paper is organized as follows. Section 2 presents the quality functions which are used for evaluating and comparing the algorithms. The community detection algorithms being compared are presented in Section 3. Section 4 reviews related previous research. In Section 5, the dataset used for empirical comparison is presented and the experimental results are discussed. Finally Section 6 concludes the work.

2 Quality of Community Detection Algorithms

In this section, we present the notations and the quality functions that are used in the rest of the paper.

Preliminaries. Let $G = (V, E)$ represent a connected, undirected, and unweighted graph where V is the set of n nodes and E is the set of m edges of G. A *clustering* $\mathcal{C} = \{C_1, \ldots, C_k\}$ is a partitioning of V into k clusters C_i, by a node-based community detection algorithm. A cluster containing only a single node is called a *singleton*, and a cluster with only one internal edge is called *trivial*. If nodes can be shared between clusters, the clustering is called *overlapping*. The number of intra- and inter-cluster edges of a cluster C are denoted by $m(C)$ and $\overline{m}(C)$, respectively and $m(\mathcal{C}) := \sum_{C \in \mathcal{C}} m(C)$ is the total number of intra-cluster edges in $\mathcal{C}$.

Quality Functions. A quality function is used either as an objective function to be optimized in order to find the communities of a network, or as a measure for assessing the quality of a clustering [6]. When the true community structure of a network is not known, quality functions are necessary for evaluating the qualitative performance of clustering algorithms. Since no single best quality function exists [2], we investigate three widely used structural quality functions: coverage, modularity [17], and conductance [11].

Coverage. Coverage of a clustering, $cov(\mathcal{C}) := \frac{m(\mathcal{C})}{m}$, is the most simple quality function, however, it is biased towards coarse-grained clusterings.

Modularity. Modularity is defined as $Q(\mathcal{C}) := \frac{m(\mathcal{C})}{m} - \frac{1}{4m^2} \sum_{C \in \mathcal{C}} \left(\sum_{v \in C} deg(v)\right)^2$ and is based on the idea that a good cluster should have higher internal and lower external density of edges compared to a *null model* with similar structural properties but without a community structure [17].

Conductance. Conductance of a cut $(C, V \setminus C)$ in a graph is defined as $\phi(C) := \frac{\overline{m}(C)}{\min(\sum_{v \in C} deg(v), \sum_{v \in V \setminus C} deg(v))}$, and tends to favor clusterings with fewer number of clusters [2]. Inter-cluster conductance, $\delta(\mathcal{C}) := 1 - \max_i \phi(C_i)$, $i \in \{1, \ldots, k\}$, is usually used as a worst-case measure to assess the quality of a clustering. The average conductance $(\frac{1}{|C|} \sum_{C \in \mathcal{C}} \phi(C))$ is also a useful metric, since if an algorithm creates singletons, the inter-cluster conductance value will be dominated by the zero value for these clusters, while the average would not [4].

The above widely used structural quality functions cannot be directly calculated for assessing the quality of link community detection methods because of the community overlaps. For instance, modularity of a link community can be calculated by applying a modified modularity function on a projected and weighted transformation of the network [7]. In this paper we investigate the structural quality of link communities by using two of the quality measures introduced in [1]. *Community coverage* measures the fraction of the nodes that belong to at least one non-trivial community, and *Overlap coverage* measures the average number of times a node is clustered inside non-trivial communities. Higher values for overlap coverage mean that the algorithm has extracted more information from the network. The algorithms that don't find overlapping communities yield the same value for both overlap and community coverage.

In addition to the structural quality, we determine the *logical quality* of a clustering based on the type of the edges inside its communities. A clustering which yields only homogeneous communities, in which all of the edges are of the same type, has a perfect logical quality. For instance, a clustering with communities that contain only spam emails or only ham emails has higher logical quality compared to a clustering which yields communities containing a mixture of both ham and spam. In addition, the amount of spam and ham emails that can be separated into distinct homogeneous communities by an algorithm is used to determine its logical quality.

3 Studied Community Detection Algorithms

In this section we briefly describe the community detection algorithms we have selected and compared using our email networks.

Fast Modularity Optimization (Blondel) by Blondel et al. [3]. This algorithm, also known as *Louvain* method, is a greedy approach to modularity maximization. The algorithm starts with assigning each node to a singleton and progresses by moving nodes to neighboring clusters in order to improve modularity. This method has complexity $O(m)$ and unfolds a hierarchical community structure with increasing coarseness and meaningful intermediate communities.

Maps of Random Walks (Infomap) by Rosvall and Bergstrom [19]. This algorithm is a flow-based and information theoretic clustering approach with complexity $O(m)$. It uses a random walk as a proxy for information flow on a network and minimizes a *map equation*, which measures the description length of a random walker, over all the network clusters to reveal its community structure. Infomap aims at finding a clustering which generates the most compressed description length of the random walks on the network.

Multilevel Compression of Random Walks (InfoH) by Rosvall and Bergstrom [20]. This method generalizes the Infomap method to reveal multiple levels of a network. InfoH minimizes a *hierarchical map equation* to find the shortest multilevel description length of a random walker.

Potts Model Community Detection (RN) by Ronhovde and Nussinov [18]. This algorithm is based on minimization of the Hamiltonian of a local objective function, the absolute Potts model. The multiresolution variant of the algorithm deploys information theory-based measures to find the best communities on all scales. The complexity of this method is superlinear $O(m^{1.3})$ for the community detection algorithm and $O(m^{1.3} \log n)$ for the multiresolution algorithm.

Markov Clustering (MCL) by Dongen [23]. MCL is based on the idea that a random walk entering a dense cluster likely remains for a long time inside the cluster before switching between sparsely connected communities. The random walks are calculated deterministically and simultaneously using a matrix of transition probabilities. The MCL algorithm has a complexity of $O(nk^2)$, where k refers to the average or maximum number of allowed neighbors for the nodes.

Link Community Detection (LC) by Ahn et al. [1]. All of the above algorithms aim at clustering nodes into densely connected communities. However, Ahn et al. [1] have defined communities as a group of topologically similar edges and have introduced a link community detection algorithm for revealing them. The algorithm unfolds the hierarchical structure and overlapping communities of a network. Although the clustering is meaningful at all scales, an objective function, the *partition density*, is used to select the optimum level of hierarchy.

All of the above algorithms are known to perform well on large networks. Infomap, InfoH, and MCL are suitable for clustering networks where edges represent flows. Emails can be seen as flows of data between people, so flow-based approaches are good candidates for clustering email networks. Email communications can also be seen as pairwise relationships between people, so the other topological methods could also be suitable. LC which is based on calculating the similarity of the edges in a network can also be suitable since we are interested in grouping the same type of edges into the same clusters.

In this study, we have used the implementations of the algorithms, which were publicly available, in order to conduct the experiments. Blondel creates a hierarchy of clusterings where the best modularity is achieved at its last level. We have also looked at the clustering yield at Blondel's first level of hierarchy, which has smaller meaningful communities, and refer to it as *Blondel L1*. We have also used the basic RN algorithm instead of its multiresolution variant to be able to choose the desired clustering granularity. The granularity of the clusterings should be considered when comparing the quality of the algorithms since structural quality functions are usually in favor of coarse-grained clusterings [2].

4 Related Work

Experimental comparisons of different community detection algorithms have been conducted on both real and benchmark graphs. Lancichinetti and Fortunato [12] compared different algorithms including Blondel, Infomap, RN, and MCL, on GN and LFR benchmark graphs. They showed that Infomap, Blondel, and RN perform well, but MCL performs worse especially for large communities. They also showed that the performance of Blondel decreases for large graphs, whereas Infomap remains stable. Brandes et al. [4] conducted an experimental evaluation of three clustering methods including MCL using random clustered graphs and showed that MCL performs well with respect to some quality functions but produces more clusters than contained in the network.

Community detection algorithms have also been evaluated and compared using different real networks. Tibély et al. [22] have analyzed the community structure of a large mobile phone call graph using Blondel L1, Infomap, and an overlapping method. Leskovec et al. [14] studied a number of real networks, including the Enron email network and an email network of a large organization, to empirically compare two different clustering methods. The latter dataset was also used by Lancichinetti et al. [13], in addition to other real networks, to study the characteristics of communities in different types of complex networks. They used Infomap together with another algorithm to show that although different methods output different clusterings, the statistical properties of their communities are quite similar for similar classes of networks. Studies of the community structure of email networks have also been conducted by Guimerà et al. [10] using emails in a university.

In contrast to previous studies, the dataset used in this study is based on email traffic captured on a high speed Internet backbone link, and is not limited to a single organization. To the best of our knowledge, this is the first study of the community structure of large-scale email networks containing spam. This dataset enables us to evaluate the ability of the community detection algorithms in separating spam from legitimate email by clustering them into distinct clusters.

5 Experimental Evaluation

In this section, the email dataset and the the experimental results are presented.

5.1 Dataset

The dataset used for creating the email networks was generated by collecting SMTP packets on a 10 Gbps link of the core-backbone of SUNET[1] during a period of 14 consecutive days in March 2010. During the collection period more than 797 million SMTP packets were collected, which were sent and received by 614,601 distinct domains. Around 3.4 million emails were extracted from the collected packets after removing unusable and rejected email transmissions. These emails were then classified to be either *spam* or *ham* using a well-trained filtering tool[2]. Following that, email contents were discarded and email addresses were anonymized in order to preserve privacy in a way that no information about the senders, receivers, and content of the emails are retrievable.

In addition to a complete email network, we generated daily and weekly email networks. An email network consists of email addresses as nodes, and the email communications between them as edges. More details on the measurement location, data collection and pre-processing, and the structural and temporal properties of the email networks can be found in [15] and [16], respectively.

5.2 Comparison of the Algorithms

In this section, the experimental results regarding the qualitative performance of the clustering algorithms with respect to their structural and logical quality is presented. A summary of the results can be found at the end of the section.

Table 1 shows the total number of nodes and edges, and the number of spam edges in each studied email network, as well as the number of communities

[1] The Swedish University Network (`http://www.sunet.se/`) serves as a backbone for university traffic, student dormitories, research institutes, etc.

[2] The SpamAssassin (`http://spamassassin.apache.org`) was in use for a long time in our University mail server and it incurs high detection and low false positive rates.

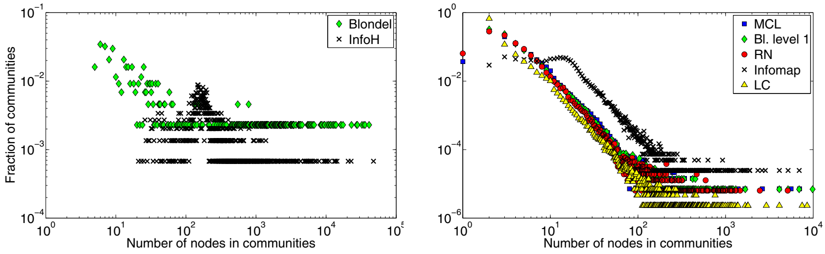
Table 1. The properties of the GCC of the generated email networks (larger networks become more connected) and the number of communities created by each algorithm.

Network	# Nodes	# Edges	# Spam	Blondel	InfoH	Infomap	Blondel L1	MCL	RN	LC
Day 1	167,329	236,673	173,640	253	546	10,505	39,477	38,775	41,215	88,028
Day 2	153,734	194,797	97,260	194	397	8,025	28,077	27,011	28,499	61,027
Day 3	123,878	168,896	108,996	218	412	8,151	29,150	28,031	30,022	64,310
Day 4	128,200	172,836	113,299	218	398	8,484	29,123	28,043	30,167	63,165
Day 5	101,643	135,195	89,119	195	311	6,664	22,212	21,593	23,935	46,928
Day 6	72,068	99,361	75,713	236	183	4,714	13,904	13,716	17,697	30,236
Day 7	73,131	103,293	85,879	199	271	4,842	17,305	16,808	18,631	37,581
Week 1	901,699	1,441,731	961,809	558	1,470	41,916	149,131	144,054	187,960	451,275
Day 8	115,232	155,919	90,299	234	379	7,745	27,661	26,514	28,409	57,931
Day 9	112,713	152,569	88,273	188	383	7,521	26,395	25,549	26,942	56,443
Day 10	140,843	195,999	121,158	255	441	8,664	31,033	30,231	39,020	67,741
Day 11	125,029	179,410	116,056	192	398	8,171	28,501	27,897	30,484	65,285
Day 12	106,816	149,407	100,595	211	380	7,319	25,314	24,328	28,040	54,317
Day 13	73,325	98,713	71,954	339	296	5,275	16,736	16,074	22,476	32,403
Day 14	68,315	100,089	76,408	179	210	4,741	14,567	14,254	17,822	31,463
Week 2	810,543	1,348,373	859,324	436	380	40,553	143,569	139,366	156,822	430,232
All	1,599,732	2,790,322	1,858,686	1,028	1,740	63,471	230,013	220,346	294,581	817,074

created by each clustering algorithm. The algorithms were applied to the giant connected component (GCC) of each email network, which is a subset of the nodes in the network where a path exists between any pair of them. The networks are also considered as unweighted and undirected.

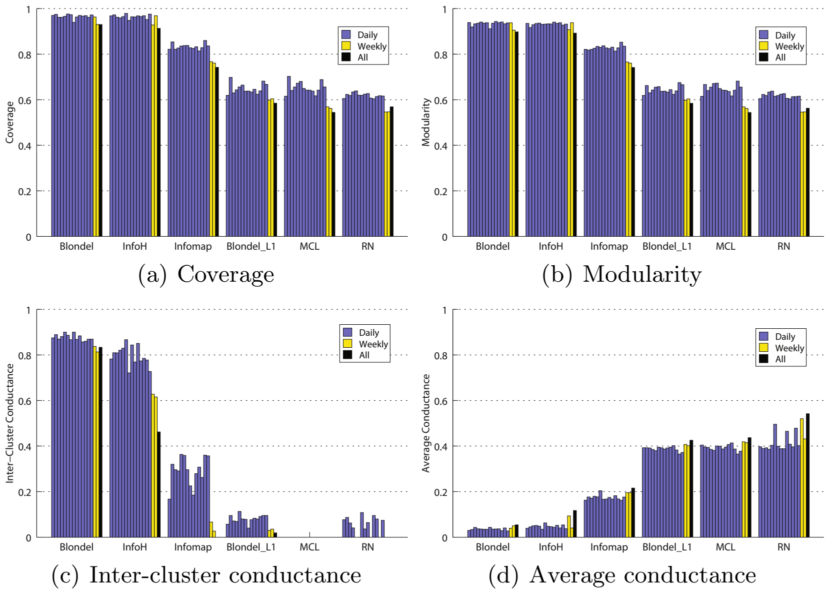
Blondel creates a coarse-grained clustering and in average achieves 46% modularity gain over Blondel L1. InfoH also creates coarse clusters and in average gains more than 15% in the compression of the description length of the random walks on the networks over the non-hierarchical version (Infomap). MCL allows us to select the granularity of the clustering by choosing an inflation parameter. It is also possible to choose the resolution parameter for RN to achieve a clustering with the desired granularity. We have selected the inflation parameter in MCL and the resolution parameter in RN so that for most of the networks they yield clusterings with a close granularity to that of Blondel L1. This allows us to further investigate and compare the effect of the granularity of the clusterings on their quality. LC is different in nature from the other algorithms as it is based on link community detection rather than a node-based approach. LC yields the finest-grained clustering for all of the networks at its best level of hierarchy.

Figure 1 summarizes the distribution of the size of the communities created by the different algorithms for the "week 2" email network. The distributions for



(a) (b)

Fig. 1. A comparison of community size distribution using "Week 2" email network. Blondel L1, MCL, and RN follow very similar distributions.



(a) Coverage (b) Modularity

(c) Inter-cluster conductance (d) Average conductance

Fig. 2. Comparison of structural quality of the algorithms on daily, weekly, and complete email networks. Blondel and InfoH yield the best structural quality.

other daily and weekly networks are quite similar. It can be seen that Blondel and InfoH, which create very coarse-grained clusters, have very different community size distributions compared to each other and the rest of the algorithms. It can also be seen in Figure 1(b) that, surprisingly, Blondel L1, MCL, and RN follow similar distributions. The main difference is that MCL and RN create a number of singletons, but Blondel L1 does not. The community size distribution of LC is also close to the other three methods, but it creates more clusters.

Structural Quality. Figure 2 shows a comparison of the structural quality of the different clusterings. Each bar corresponds to a daily network (day 1 to day 14), except the last three bars from the left for each of the algorithms, which correspond to week 1, week 2, and complete email networks, respectively. It can be seen that Blondel, which aims at maximizing modularity, have the highest structural quality with respect to all of the quality functions. Although InfoH uses a fundamentally different approach it achieves equally good structural quality, however its quality degrades for larger networks. Blondel L1, MCL, and RN, which have closer granularities, also show similar quality with respect to coverage, modularity, and average conductance. However, based on the inter-cluster conductance, MCL and RN do not perform well since they might create a number of singletons which results in an inter-cluster conductance of zero.

Our experimental results reveal that the structural quality of clusterings are roughly consistent for different daily networks. The clusterings with similar granularity and community size distribution also show similar structural quality, therefore, it is important to take the granularity of the clusterings into account when comparing the algorithms. LC creates a clustering with the finest granularity, however the studied structural quality functions cannot be directly used for assessing the quality of this algorithm due to its different nature. In this paper, we look at community coverage and overlap coverage which were introduced for assessing the quality of link-based clustering by Ahn et al.[1].

LC, Blondel, and InfoH yield full community coverage for all of the email networks. Infomap, Blondel Ll, MCL, and RN achieve community coverage of around 0.99, 0.84, 0.83, and 0.8, respectively. However, this function on its own is not enough for assessing the quality of a clustering method, it is also important to consider the overlap coverage of the clusterings. None of the algorithms, except MCL and LC, find overlapping clusters so their overlap coverage is equal to their community coverage. MCL is not an overlapping clustering method, but for some specific graphs it might find overlaps [23]. In our email networks, MCL yields very few overlapping communities so its overlap coverage is just slightly higher than its community coverage. LC yields overlap coverage of 2.6, 3.1, and 3.4 in average for the daily, weekly, and complete email networks, meaning that it unfolds more overlaps in larger networks.

Logical Quality. Our experiments show that all algorithms create a number of *spam communities* that only contain spam, *ham communities* that only contain ham, and *mix communities* with a mixture of both ham and spam edges. Figure 3 shows a comparison between the percentage of spam, ham, and mix communities created by the different algorithms. The last three bars from the left for each of the algorithms correspond to week 1, week 2, and the complete email networks, respectively. It can be seen that InfoH and Blondel perform worse, since these algorithms tend to merge smaller homogeneous communities into mix communities to achieve higher structural quality. The best results for all networks are achieved by LC.

Moreover, it is important to assess the amount of spam and ham emails that can be separated by community detection algorithms, in order to investigate the possibility of deploying clustering approaches to perform spam detection. Figure 4 shows the ratio of total spam and ham edges that are inside homogeneous spam and ham communities. In all of the networks, communities created by LC contain the highest number of spam and ham edges. Blondel and InfoH have the worst logical quality and Blondel L1, MCL, and RN have almost similar quality. For all algorithms, except LC, some of the spam and ham emails end up as inter-cluster edges and can therefore not be separated by the clustering algorithms. It can also be seen that the percentage of spam (ham) edges which are clustered inside spam (ham) communities decreases for larger networks.

Our experiments suggest that the logical quality tends to be higher for fine-grained clusterings. The granularity of the best clustering created by LC is finer than the other clusterings in our experiments. LC cuts its hierarchy of clustering at an optimum threshold which results in maximal partition density. By choosing a threshold below the optimum value, we can have a clustering with coarser granularity. Since the algorithm reveals meaningful communities at all scales, we changed the threshold so that the granularity of the clustering became more similar to that of Blondel L1, MCL, and RN. Our experiments with the new

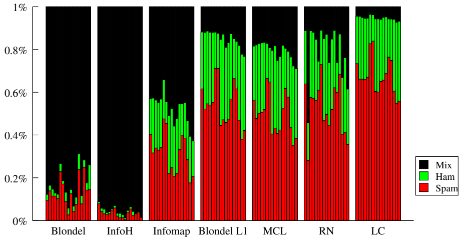
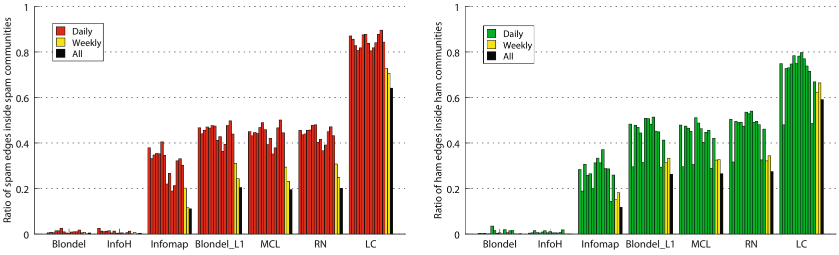


Fig. 3. Comparison of percentage of spam, ham, and mix communities created by different algorithms. LC creates the highest number of homogeneous communities.



(a) Spam communities

(b) Ham communities

Fig. 4. Ratio of spam (ham) in homogeneous spam (ham) communities. LC clusters a higher percentage of total spam (ham) edges inside the spam (ham) communities.

clusterings show that, the percentage of spam (ham) edges inside the spam (ham) communities was reduced. For instance, for the first daily network the percentage of spam (ham) edges decreased from 87% to 66% (from 76% to 56%). Although the logical quality degrades by changing the coarseness of the clustering, LC still shows higher logical quality than all of the other algorithms.

Summary of the Experimental Results

- Blondel and InfoH create coarse-grained clusters and achieve the best quality with respect to all of the structural quality functions. However, they have the worst logical quality with respect to both number of homogeneous communities and amount of spam and ham emails that are clustered inside these homogeneous communities.
- Infomap, which is the non-hierarchical version of InfoH, achieves quite good structural quality and decent logical quality. However, Blondel L1, which is based on the first level of Blondel's hierarchy of clusterings, yields much better logical quality than Infomap, but worse structural quality with respect to all of the structural quality functions.
- MCL and RN allow us to change the resolution of the clustering by modifying different parameters. When the granularity of their clusterings is set to be close to that of Blondel L1, they show almost similar community size dis-

tribution as well as similar structural and logical quality. However, Blondel L1 is superior to the other two methods due to its lower complexity.
- LC, which performs link community detection, has the best logical quality and separates the highest amount of spam and ham emails into distinct homogeneous communities.

6 Conclusions

In this study, we have performed an empirical comparison and evaluation of a number of high quality community detection algorithms using large-scale email networks. The studied email networks contain both legitimate and spam emails and are created from real email traffic. Our study reveals that yielding high structural quality by community detection algorithms is not enough to unfold the true logical communities of the email networks. Therefore, it is necessary to deploy more realistic measures for clustering real-world networks.

More specifically, our study suggests that the community detection algorithms that achieve maximum modularity, coverage, inter-cluster conductance, or minimum average conductance do not reveal the communities that coincide with the true clustering of the email networks. For instance the algorithms which yield worse, but acceptable, average conductance values actually could separate a large number of spam (ham) emails into distinct spam (ham) communities. Therefore, the value of this function can be indicative of good logical quality. However, this observation is based on our email networks, and might not be conclusive as it was shown that different classes of networks show different community structures [12,2].

Overall, our experiments reveal that link community detection is the most suitable approach for separating spam and ham emails into distinct communities compared to the other node-based algorithms.

Acknowledgments. This work was supported by .SE – The Internet Infrastructure Foundation and SUNET. The research leading to these results has also received funding from the European Union Seventh Framework Programme (FP7/ 2007-2013) under grant agreement no. 257007.

References

1. Ahn, Y.-Y., Bagrow, J.P., Lehmann, S.: Link communities reveal multiscale complexity in networks. Nature 466(7307), 761–764 (2010)
2. Almeida, H., Guedes, D., Meira Jr., W., Zaki, M.J.: Is There a Best Quality Metric for Graph Clusters? In: Gunopulos, D., Hofmann, T., Malerba, D., Vazirgiannis, M. (eds.) ECML PKDD 2011. LNCS, vol. 6911, pp. 44–59. Springer, Heidelberg (2011)
3. Blondel, V.D., Guillaume, J.-L., Lambiotte, R., Lefebvre, E.: Fast unfolding of communities in large networks. Journal of Statistical Mechanics: Theory and Experiment 2008(10), P10008 (2008)
4. Brandes, U., Gaertler, M., Wagner, D.: Experiments on Graph Clustering Algorithms. In: Di Battista, G., Zwick, U. (eds.) ESA 2003. LNCS, vol. 2832, pp. 568–579. Springer, Heidelberg (2003)

5. Danon, L., Díaz-Guilera, A., Duch, J., Arenas, A.: Comparing community structure identification. Journal of Statistical Mechanics: Theory and Experiment 2005(09), P09008 (2005)
6. Delling, D., Gaertler, M., Robert, G., Nikoloski, Z., Wagner, D.: How to Evaluate Clustering Techniques. Technical report, no. 2006-4, Universität Karlsruhe (2006)
7. Evans, T., Lambiotte, R.: Line graphs, link partitions, and overlapping communities. Physical Review E 80(1), 1–8 (2009)
8. Fortunato, S.: Community detection in graphs. Physics Reports 486(3-5), 75–174 (2010)
9. Girvan, M., Newman, M.E.J.: Community structure in social and biological networks. Proceedings of the National Academy of Sciences of the United States of America 99(12), 7821–7826 (2002)
10. Guimerà, R., Danon, L., Díaz-Guilera, A., Giralt, F., Arenas, A.: Self-similar community structure in a network of human interactions. Physical Review. E, Statistical, Nonlinear, and Soft Matter Physics 68(6 pt. 2), 065103 (2003)
11. Kannan, R., Vempala, S., Veta, A.: On clusterings-good, bad and spectral. In: Proceedings 41st Annual Symposium on Foundations of Computer Science, pp. 367–377. IEEE Comput. Soc. (2000)
12. Lancichinetti, A., Fortunato, S.: Community detection algorithms: A comparative analysis. Physical Review E 80(5), 1–11 (2009)
13. Lancichinetti, A., Kivelä, M., Saramäki, J., Fortunato, S.: Characterizing the community structure of complex networks. PloS One 5(8), e11976 (2010)
14. Leskovec, J., Lang, K.J., Mahoney, M.: Empirical comparison of algorithms for network community detection. In: Proceedings of the 19th International Conference on World Wide Web, p. 631. ACM Press, New York (2010)
15. Moradi, F., Almgren, M., John, W., Olovsson, T., Tsigas, P.: On Collection of Large-Scale Multi-Purpose Datasets on Internet Backbone Links. In: Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (2011)
16. Moradi, F., Olovsson, T., Tsigas, P.: Structural and Temporal Properties of E-mail and Spam Networks. Technical report, no. 2011-18, Chalmers University of Technology (2011)
17. Newman, M., Girvan, M.: Finding and evaluating community structure in networks. Physical Review E 69(2), 1–15 (2004)
18. Ronhovde, P., Nussinov, Z.: Multiresolution community detection for megascale networks by information-based replica correlations. Physical Review E 80(1), 1–18 (2009)
19. Rosvall, M., Bergstrom, C.T.: Maps of random walks on complex networks reveal community structure. Proceedings of the National Academy of Sciences of the United States of America 105(4), 1118–1123 (2008)
20. Rosvall, M., Bergstrom, C.T.: Multilevel compression of random walks on networks reveals hierarchical organization in large integrated systems. PloS One 6(4), e18209 (2011)
21. Schaeffer, S.E.: Graph clustering. Computer Science Review 1(1), 27–64 (2007)
22. Tibély, G., Kovanen, L., Karsai, M., Kaski, K., Kertész, J., Saramäki, J.: Communities and beyond: Mesoscopic analysis of a large social network with complementary methods. Physical Review E 83(5), 1–10 (2011)
23. Van Dongen, S.: Graph clustering by flow simulation. PhD thesis, University of Utrecht, The Netherlands (2000)
24. Viswanath, B., Post, A., Gummadi, K.P., Mislove, A.: An analysis of social network-based Sybil defenses. In: Proceedings of the ACM SIGCOMM 2010 Conference, p. 363. ACM Press, New York (2010)

Fast, Small, Simple Rank/Select on Bitmaps*

Gonzalo Navarro[1] and Eliana Providel[1,2]

[1] Department of Computer Science, University of Chile
gnavarro@dcc.uchile.cl
[2] Escuela de Ingeniería Civil Informática, Facultad de Ingeniería,
Universidad de Valparaíso, Valparaíso, Chile
eliana.providel@uv.cl

Abstract. *Rank* and *select* queries on bitmaps are fundamental for the construction of a variety of compact data structures. Both can, in theory, be answered in constant time by spending $o(n)$ extra bits on top of the original bitmap, of length n, or of a compressed version of it. However, while the solution for *rank* is indeed simple and practical, a similar result for *select* has been elusive, and practical compact data structure implementations avoid its use whenever possible. In addition, the overhead of the $o(n)$ extra bits is in many cases very significant. In this paper we bridge the gap between theory and practice by presenting two structures, one using the bitmap in plain form and another using a compressed form, that are simple to implement and combine much lower space overheads than previous work with excellent time performance for *rank* and *select* queries. In particular, our structure for plain bitmaps is far smaller and faster for *select* than any previous structure, while competitive for *rank* with the best previous structures of similar size.

1 Introduction

Compact data structures represent data within little space and can efficiently operate on it, in contrast to plain structures that do not compress, and with pure compression that needs full decompression in order to operate the data. They have become particularly interesting due to the increasing performance gap between successive levels in the memory hierarchy, in particular between main memory and disk. Since Jacobson's work [9], compact data structures for trees [9,11], graphs [9,11], strings [8,6], and texts [8,4], etc. have been proposed.

Jacobson [9] noticed that bit vectors were fundamental to support various compact data structures. In particular, he used the following two operations to simulate general trees, and since then these two operations have proved fundamental to implement many other compact structures:

$$rank_b(B,i) = \text{number of occurrences of bit } b \text{ in } B[0,i];$$
$$select_b(B,i) = \text{position of the } i\text{-th occurrence of bit } b \text{ in } B.$$

* Partially funded by Fondecyt Grant 1-110066 and Institute for Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, Chile.

R. Klasing (Ed.): SEA 2012, LNCS 7276, pp. 295–306, 2012.

Much effort has been spent on implementing *rank* and *select* efficiently. In theory, they have long been solved in $O(1)$ time using just $o(n)$ bits on top of B [2,10], an even on top of a compressed representation of B using $nH_0(B) + o(n)$ bits [15,14]. $H_0(B) = n_0 \lg(n/n_0) + n_1 \lg(n/n_1)$, where B has n_0 0s and n_1 1s, is the called the zero-order entropy of B, and is smaller when B has a few 0s or 1s. By default we assume $b = 1$ (many applications need just $rank_1$ and $select_1$).

The practical solutions for *rank* over plain bitmaps are indeed simple, and there exist implementations solving it within a few tenths of microsecond (μsec) using 5% of extra space on top of B [7]. For *select*, instead, no practical solution with guaranteed constant time exists. The original solution [2] leads to more than 60% extra space in practice [7]. Only very recently there have been better proposals running in a few tenths of μsec and requiring around 25% [12] and even 7% [16] of extra space, on a bitmap with half 0s and 1s. Even worse, those structures only solve *select*, and thus need to be coupled with another solving *rank*, and we need two copies of them to solve both $select_0$ and $select_1$ (this does not happen with *rank* because $rank_0(i) = i - rank_1(i)$). On the other hand, implementations of compressed solutions [15,3] pose in practice an overhead over 27% (of the original bitmap size) on top of the entropy of the bitmap.

This situation has noticeably slowed down the growth of the area of compact data structures in practice: While many solutions building on *select*, or relying on bitmap compression, are very attractive in theory, in practice one must try to avoid the use of *select*, and have in mind the significant space overhead associated to the $o(n)$ terms. We largely remedy both problems in this paper.

First, we introduce a new implementation of compressed bitmaps [15] that retains the time performance of the current implementation [3] while drastically reducing its space overhead by 50%–60%. The resulting structure has a space overhead of around 10% (of the original bitmap size) on top of the entropy (compare to the 27% of the current implementation [3]) and solves *rank* queries within about 0.4 μsec and *select* within 1 μsec. This is achieved by replacing the use of universal tables by on-the-fly generation of their cells. These universal tables are of size exponential on a block size t, and the space overhead of the data structure is $O(n \lg t/t)$. Removing the tables let us use t values that are 4 times larger, thereby reducing the space by a factor of $\approx \lg 4/4 = 50\%$.

Second, we present a new combined data structure that solves *rank* and *select* on plain bitmaps, instead of solving each operation separately. It integrates two samplings, one for *rank* that is regular on the bitmap positions, and one for *select* that is regular on the positions of the 1s. Each operation uses its own sampling but takes advantage of the other if possible. We show that this structure is able to solve both *rank* and *select* queries within around 0.2 μsec, using just 3% of extra space on top of the plain bitmap. This is an unprecedented result, very far from what current representations achieve, that finally puts *rank* and *select* on the map of the operations that can be used in practice without reservations.

2 Related Work

Jacobson [9] showed that attaching a dictionary of size $o(n)$ to a bit vector $B[0, n-1]$ is sufficient to support *rank* operation in constant time on the RAM model. Later, Munro [10] and Clark [2] obtained constant-time complexity for *select* on the RAM model, using also $o(n)$ extra space. Golynski [5] showed how to reduce the $o(n)$ term to $O(n \lg\lg n / \lg n)$, and that this space is optimal if B is stored explicitly. Others have studied representations that store a compressed form of B, useful when it has few or many 1s [13,15,12,14]. Some [15,14] use $nH_0(B) + o(n)$ bits and solve both operations in constant time, where $o(n)$ can be as small as $O(n/\lg^c n)$ for any constant c [14]. Others [12] solve *select* in constant time and *rank* in time $O(\lg(n/n_1))$, using $nH_0(B) + O(n_1)$ bits.

The original solution for *rank* was simple, easy to program, and required three table accesses. It has followed a simple path through practice. For example, a competitive implementation [7] solves *rank* in a few tenths of microseconds using 5% of extra space on top of B, and in a few hundredths using 37.5% of extra space. The first figure is sufficiently fast in most practical scenarios.

The solution for *select*, instead, has followed a more complicated path. The original constant-time solution [2,10] is much more complicated than that of *rank*, and in practice it is much slower and requires more space. An implementation [7] showed that it requires more than 60% of extra space. Indeed, a simple solution that works better [7] for all reasonable bitmap sizes is to solve *select* via binary searches on the *rank* directories. This, however, makes *select* significantly slower than *rank*, and less attractive in practice.

Okanohara and Sadakane [12] proposed a simplification of the original solution that, although does not guarantee constant time, it works very well in practice (a few tenths of microseconds). It requires, however, around 25% of extra space and does not solve *rank*, so their complete *rank/select* solution for their structure called *dense array* requires more than 50% extra space. Later on, Vigna [16] achieved similar times within much less space overhead (7% to 12% in our experiments), with a structure called *simple-select*. Again, this structure does not solve *rank*. He proposed other two structures, *rank9sel* and *rank9b*, that solve both operations within the same time and a space overhead of around 24%.

As for compressed bitmaps, Claude and Navarro [3] implemented the solution by Raman et al. [15], achieving around 27% space overhead (this percentage refers to the original bitmap size) on top of the entropy of the bitmap, and solved *rank* within tenths of microseconds and *select* within a microsecond. This space overhead is very significant in practice. The structure is useful for densities (n_1/n) of up to 20% (at which point the entropy is more than 80% anyway, so not much can be done). Okanohara and Sadakane [12] presented a structure called *sparse array*, that uses $n_1 \lg(n/n_1) + O(n_1)$ bits, and is very fast to solve *rank* and *select* (as fast as *rank* on plain bitmaps). However, its space overhead is very large for all but very sparse bitmaps (densities below 5%).

3 A Structure for Compressed Bitmaps

We describe our *rank*/*select* data structure for compressed bitmaps. It is based on the proposal by Raman *et. al.* [15] and its practical implementation [3]. We first describe this implementation and then our improvement.

The structure partitions the input bitmap B into blocks of length $t = \lceil \frac{\lg n}{2} \rceil$. These are assigned to *classes*: a block with k 1s belongs to class k. Class k contains $\binom{t}{k}$ elements, so $\lceil \lg \binom{t}{k} \rceil$ bits are used to refer to an element of it. A block is identified with a pair (k, r), where k is its class ($0 \leq k \leq t$, using $\lceil \lg(t+1) \rceil$ bits), and r is the index of the block within the class (using $\lceil \lg \binom{t}{k} \rceil$ bits). A global universal table for each class k translates in $O(1)$ time any index r into the corresponding t bits. The sizes of all those tables add up to $2^t\, t$ bits.

The sequence of $\lceil n/t \rceil$ class identifiers k is stored in one array, K, and that of the indexes r is stored in another, R. Note the indexes are of different width, so it is not immediate how to locate the r value of the j-th block. The blocks are grouped in superblocks of length $s = \lfloor t \lg n \rfloor$. Each superblock stores the *rank* up to its beginning, and a pointer to R where its indexes start.

To solve $rank(i)$, we first compute the superblock i belongs to, which stores the *rank* value up to its beginning. Then we scan the classes from the start of the superblock, accumulating the k values into our *rank* answer. At the same time we maintain the pointer to array R: the superblock knows the pointer value corresponding to its beginning, and then we must add $\lceil \lg \binom{t}{k} \rceil$ for each class k we process. These values are obviously preprocessed, and note we do not access R. This scanning continues up to the block i belongs to, whose index is extracted from R and its bits are recovered from the universal table. We then scan the remaining bits of the block and finish. Solving $select(j)$ is analogous, except that we first binary search for the proper superblock and then scan the blocks.

The size of R is upper bounded by $nH_0(B)$, and the main space overhead comes from K, which uses $n\lceil \lg(t+1)/t \rceil$ bits (the superblocks add $O(n/t)$ bits, which is negligible). So we wish to have a larger t to reduce the main space overhead. The size $2^t\, t$ of the universal table, plus its low access locality, however, prevents using a large t. In the practical implementation [3] they used $t = 15$, so K is read by nibbles and the universal table requires only 64 KB of memory. The price is that the space overhead is $\lceil \lg(t+1)/t \rceil = 4/15 \approx 27\%$.

Our Proposal. We propose to remove the universal table and compute its entry on the fly. Although we require $O(t)$ time to rebuild a block from its index r, we note this is done only once per query, so the impact should not be much. As a reward, we will be able to use larger block sizes, such as $t = 31$ (with a space overhead of $5/31 \approx 16\%$) and $t = 63$ (with a space overhead of $6/63 \approx 9.5\%$).[1]

The Computation of *r*. At compression time, the index r of a block of length t and k 1s is assigned as follows. We start with $r = 0$. We consider the first bit.

[1] We do not use blocks larger than 63 because we would need special multiword arithmetics in our computer, and all of our multiword approaches have been orders of magnitude slower than the native arithmetic.

We number first the blocks with this first bit in zero (there are $\binom{t-1}{k}$ of them) and then those with the first bit in one (there are $\binom{t-1}{k-1}$). Therefore, if the first bit is zero we just continue with the next bit (now the block is of length $t-1$ and still has k 1s). Otherwise, we increase r by $\binom{t-1}{k}$ and continue with the next bit (now the block is of length $t-1$ and has $k-1$ 1s).

Reconstructing a Block on the Fly. We must reverse the encoding process. By checking the range of values r belongs to, we extract the bits consecutively. If $r \geq \binom{t-1}{k}$, then the first bit of the block was a 1. In this case we decrement t and k, and decrease r by $\binom{t-1}{k}$. Otherwise, the first bit of the block was a 0. In this case we only decrement t. Now we continue extracting the next bit from a block of length $t-1$.

Note we can stop if r becomes zero, as this means that all the remaining bits are zero. Also, to solve *rank* or *select* queries, it may also be possible to stop this process before obtaining all the bits of the block. Finally, we (obviously) have all the binomial coefficients precomputed.

4 A Structure for Plain Bitmaps: Combined Sampling

We introduce a structure that answers *rank*/*select* queries on top of plain bitmaps. This structure combines two sampling schemes: the first one is a regular sampling on positions, that is, we divide the input bitmap B into blocks of fixed size S_r. Then, we store a sampling of *rank* answers from the beginning of B up to the end of each block, $\texttt{sampleRank}[j] = rank_1(B, j \cdot S_r)$. In the second sampling scheme, we create blocks containing exactly S_s 1s. Note these blocks are of variable length. We ensure that each block starts with a 1, to minimize unnecessary scanning. Thus $\texttt{sampleSelect}[j] = select_1(B, j \cdot S_s + 1)$. Both samplings are similar to the top-level samplings proposed in classical solutions [2,10] to *rank* and *select* queries. The novelty is in how we combine them to solve each query.

Answering *rank* queries. To solve $rank_1(i)$, we start from block number $j = \lfloor S_r/i \rfloor$, up to which the number of 1s in B is $\texttt{sampleRank}[j]$, and scan the bitmap from position $i' = j \cdot S_r + 1$ to position i. This scanning is done bytewise, using a universal preocomputed table that counts the number of 1s in all the 256 bytes[2]. The sampling step S_r is always a multiple of 8, so that the scanning starts at a byte boundary. On the other extreme, we may need to individually scan the bits of the last byte.

Up to this point the technique is standard, and in fact corresponds to the "small space" variant of González et al. [7]. The novelty is that we will also make use of table `sampleSelect` to speed up the scanning. Imagine that $r = \texttt{sampleRank}[j]$, and we have to complete the scanning from bit positions i' to i. It is possible that some `sampleSelect` values point between i' and i, and we can easily find them: compute $r' = \lfloor r/S_s \rfloor$. Then $\texttt{sampleSelect}[r']$ points somewhere

[2] In some architectures, a native *popcount* operation may be faster. We have not exploited this feature as it is not standard.

before i'. We scan sampleSelect$[r'+k]$ for $k = 1, 2\ldots$, until sampleSelect$[r'+k+1] > i$. At this point we call $i'' =$ sampleSelect$[r'+k] \leq i$, and know that $rank_1(i'') = (r'+k) \cdot S_s + 1$. Thus we only need to scan from $i''+1$ to i. This can give a speedup unless $k = 0$.

A final tweak is that starting from positions $i''+1$ that are not byte-aligned is cumbersome and affects the performance. For this reason we slightly change the meaning of table sampleSelect$[j]$. Instead of pointing to the precise *bit* position of the $(j \cdot S_s + 1)$-th 1, it points to its *byte* position, so the scanning is always resumed from a byte-aligned position. A problem is that now do not know which is the *rank* value up to the beginning of the byte (previously we knew that there were exactly $j \cdot S_s + 1$ bits set up to position sampleSelect$[j]$). Fortunately, pointing to bytes instead of bits frees three bits from the integers where the positions are stored. The three least significant bits are therefore used to store *correction* information, namely the number of 1s from the beginning of the byte pointed by sampleSelect$[j]$ to just before the exact position of the $(j \cdot S_s + 1)$-th 1. Therefore, the number of 1s up to the beginning of the byte pointed by sampleSelect$[j]$ is $j \cdot S_s$ minus the correction information.

Answering *select* Queries. To solve $select_1(B, i)$, we first compute $j = \lfloor S_s/i \rfloor$, and use sampleSelect$[j]$ to find a byte-aligned position p up to where we know the value $r = rank_1(B, p-1)$. Then we scan byte-wise, incrementing r using the universal *popcount* table, until we find a byte where we exceed the rank value we are looking for, $r > i$. Then we rescan the last byte bit-wise until finding the exact position of the i-th 1 in B.

Once again, the byte-wise scanning can be sped up because we may go through several sampled values in sampleRank. We compute $q = \lfloor p/S_r \rfloor$, and scan the successive values sampleRank$[q+k]$, $k = 1, 2\ldots$ until sampleRank$[q+k+1] > i$. Then we may start the byte-wise scanning from position $p' = (q+k) \cdot S_r$, with rank value $r =$ sampleRank$[q+k]$, thus speeding up the process.

5 Experimental Results

We compare our implementations with various existing solutions. All the experiments were carried out on a machine with two IntelCore2 Duo processors, with 3 Ghz and 6 MB cache each, and 8 GB of RAM. The operating system is Ubuntu 8.04.4. We used gcc compiler with full optimization. We experiment on bitmaps of length $n = 2^{28}$ and average all our values over one million repetitions.

5.1 Compressed Representations

To benchmark this structure we compared the performance of *rank* and *select* operations with the original Raman *et. al.* (RRR [15]) implementation [3], and with the sparse array of Okanohara and Sadakane (Sada sparse [12]). For our structure we considered block sizes of $b = 15$, 31, and 63, and for each block size we considered superblocks of 32, 64, and 128 blocks. (Recall that RRR can use only blocks of size 15.) We generated random bitmaps with densities 5%, 10%, and 20% (their zero-order entropies are 0.286, 0.469, and 0.772, respectively).

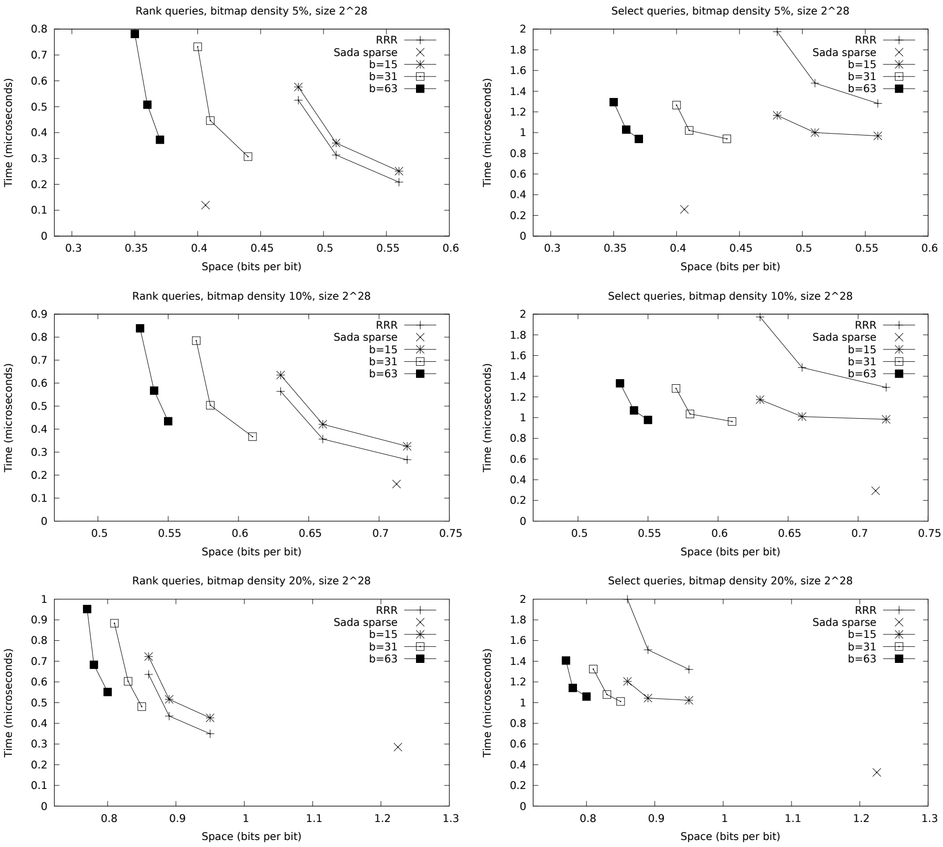


Fig. 1. Results for compressed bitmaps. Time is measured in μsec per query and space in total bits per bit of the bitmap. The x coordinates start at the zero-order entropy of the bitmap, so that the redundancy can be appreciated.

Fig. 1 gives the results. Increasing the block size from 15 to 63 greatly reduces the space overhead of the representation, to around 40%–50% of its value.

Our implementation is 30-50% faster than RRR for *select* queries, even with the same block size $b = 15$. Our times for *select* are almost insensitive to the block size. This shows that the time to decode a block is negligible compared to that of the binary search. On the other hand, smaller superblocks make the operation faster, and a superblock of 32 or 64 blocks is recommended.

RRR is slightly faster than our implementation for *rank*, and also using longer blocks has a more noticeable (but still slight) effect on the performance. Overall, the price is very low compared to the large space gain.

Finally, Sada sparse is significantly faster than all other implementations, especially for *select*. However, its space is competitive only for very low densities.

Overall, using block size $b = 63$ and superblocks of 32 or 64 blocks, our data structure computes *rank* in about half a microsecond, *select* in about one microsecond, and requires a space overhead of less than 10% on top of the entropy. Overall this is a very convenient alternative for sparse bitmaps (of density up to 20%, as thereafter the entropy becomes too close to 1.0). For very sparse bitmaps (density well below 5%), Sada sparse technique becomes the best choice. Note that our result would immediately improve on 128-bit processors.

5.2 Combined Sampling

We compare our combined sampling structure with several others. We consider, from González et al. [7], the variant that use blocks and superblocks and pose 37.5% extra space (`RG 37%`) and the one that uses one level of blocks (`RG 1Level`). From Vigna [16] we consider variants `rank9sel`, `rank9b`, and `simple-select`. From Clark [2] we consider its implementation for *select* [7] that uses 60% of extra space (`Clark`). Finally, from Okanohara and Sadakane [12] we use their dense bitmaps (`Sada dense`).

For our structure, we try different S_r and S_s combinations in the range $[2^8, 2^{13}]$. We show a curve for each S_r, with varying values of S_s. In addition, to test whether our combined approach is better than each structure separately, we add a modified Combined Sampling structure in which each kind of query only uses its respective supporting structure. In that case we use only the S_r (for *rank*) or S_s (for *select*) sampling, without using the other sampling.

We made two experiments: in the first, we evaluate *rank* and *select* queries separately, to study their performance. In the second, we emulate a real application that requires both kinds of queries, and test combined sequences with different proportions of *rank*/*select* queries.

Independent Queries We execute either *rank* or *select* queries over bitmaps of densities 10%, 50%, and 90%. The results are shown in Fig. 2.[3]

Note that our techniques provide a continuum of space/time tradeoffs by varying S_r and S_s. The most interesting zone is the one where the extra space is below 5% and the times are below 0.2 μsec (for *rank*) and 0.3 μsec (for *select*). In particular, using $S_r = 1024$ and $S_s = 8192$ achieves extra space around 3% and times in the range 0.1–0.2 μsec for *rank* and 0.2–0.25 for *select*. This space/time tradeoff had never been achieved before.

By regarding the performance without using S_s, it can be seen that the use of the sampling of *select* does not make a big difference for *rank* (so we could use the basic technique that does not use `sampleSelect`). However, the performance without using S_r shows that *select* is sped up considerably by using the sampling of *rank*. As a result, our combined structure is more than two separate structures put together, as most of the structures in the literature.

[3] We are aware that most of our curves are visually indistinguishable. However, our aim is to show that, using *some* parameterization (for which we will give explicit recommendations), our curves improve upon previous work, which is clearly distinguishable. Thus our curves can be regarded as a single, thick curve. We have not replaced them by a single one formed by the dominant points because we want to emphasize that the results are not much sensitive to a careful parameterization.

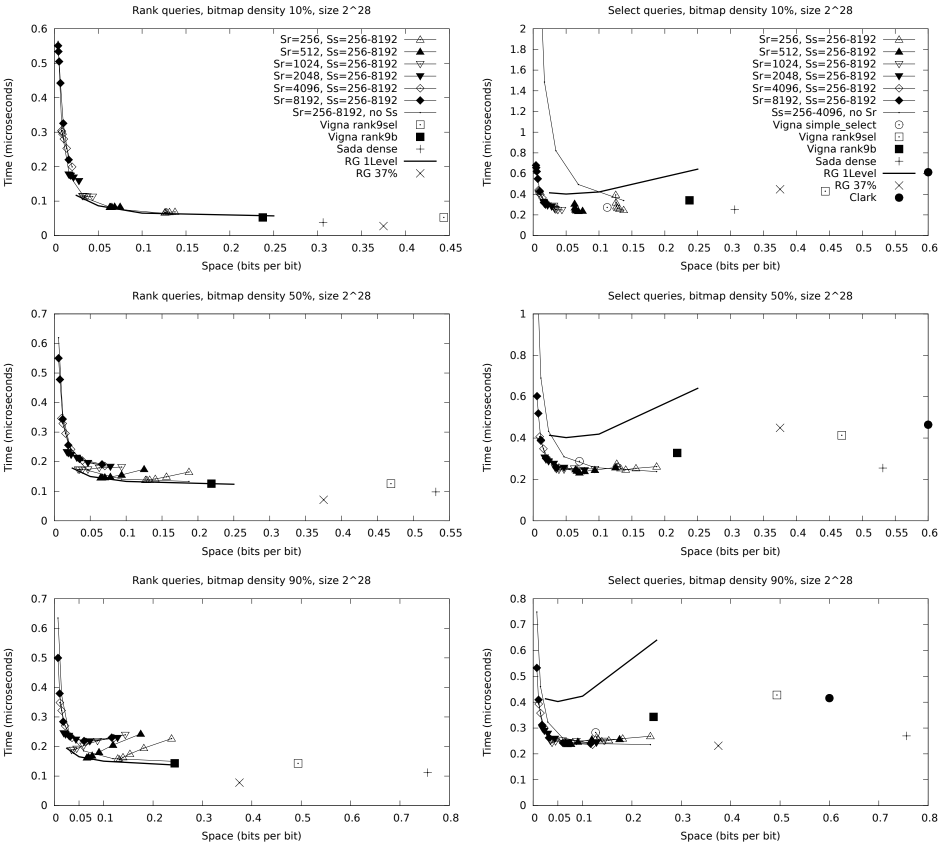


Fig. 2. Rank/Select queries for plain bitmaps. Time is measured in μsec per query and space in extra bits per bit of the bitmap.

For *rank*, structures like rank9b, rank9sel, and Sada dense, occupy significantly more space and are not noticeably faster than our 3%-space combination. Only structure RG 37% is considerably faster, yet it requires 10 times more space. Structure RG 1Level achieves a performance very similar to that of our new structure, as expected.

For *select*, however, RG 1Level performs poorly. Other structures that do not achieve better time than our 3%-space combination, while using much more space, are rank9b, rank9sel, RG37%, Sada dense, and Clark. The best performance, still not competitive with ours, is that of simple-select. This one reaches similar time, but 2–3 times more space than our 3%-space combination.

Mixed Queries In this experiment we use bitmap density 50% and execute mixed queries, which combine *rank* and *select*, to mimic applications that need both of them. We try *rank*/*select* proportions of 90%/10%, 70%/30%, 50%/50%, and 30%/70%. We include a variant of our structure with the same value for both samplings, $S_r = S_s$. The results are shown in Fig. 3 (see also Footnote 3).

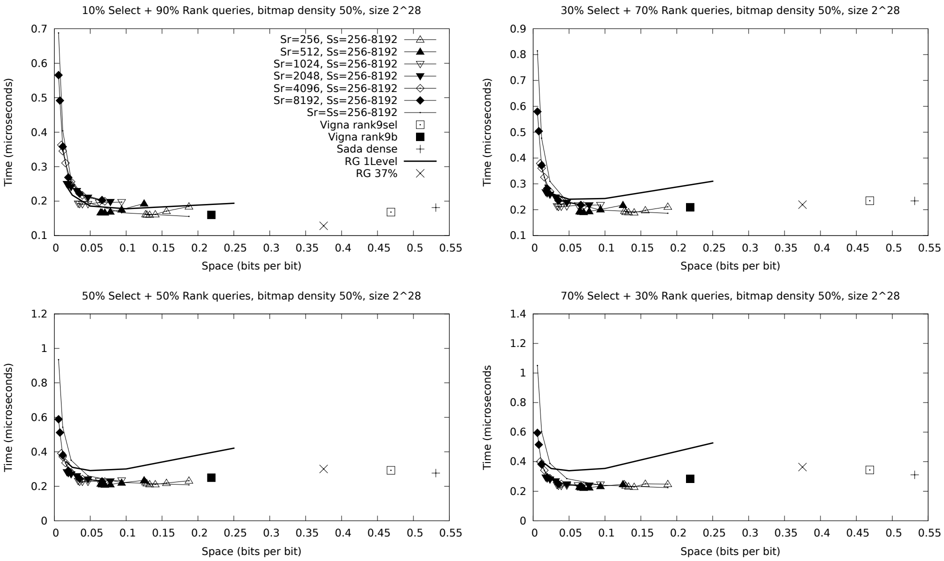


Fig. 3. Performance of mixed *rank*/*select* queries. Time is measured in μsec per query and space in extra bits per bit of the bitmap.

It can be seen that, as soon as we have more than 10% of *select* queries, our new structures become unparalleled. This is because operation *select* is, in other structures, much slower than *rank*, and therefore a small fraction of those queries affects the overall time. Our structures, on the other hand, retain a performance close to 0.2 μsec within 3% of extra space, achieved with $S_r = 1024$ and $S_s = 8192$. The variant using the same value for both samples is usually suboptimal: Spending more space on the *rank* sampling pays off.

5.3 Applications

We illustrate the many applications of *rank*/*select* data structures with two examples. In the first one, we use our compressed representation to reimplement an FM-index [4]. This is a data structure that represents a text collection in compressed form and offers search capabilities on it. Its most practical and compact implementation [3] uses a Huffman-shaped tree storing at most $n(H_0(T) + 1)$ bits, and those bits are represented in compressed form. The FM-index can count the number of occurrences of a pattern P in T via a number of *rank* operations over the bitmaps. We took this FM-index implementation from `http://libcds.recoded.cl`, and compared this implementation with one where we changed the compressed bitmaps to our new implementation. We indexed the 50 MB English text from `http://pizzachili.dcc.uchile.cl` and searched for 50,000 patterns of length 20 extracted at random from the text.

As Fig. 4 (left) shows, our index with $b = 63$ is slightly slower than the original, but still it counts in 90 μsec using 0.28 bits per bit, whereas the original one needs 0.335 bits per bit to achieve the same time. That is, we reduce the space by around 16%. Indeed, we achieve the least space ever reported for an FM-index for this text (the results on other texts were similar).

Our second experiment uses a recent benchmark on compact representations of general trees of n nodes, using $2n+o(n)$ bits [1]. They showed that LOUDS [9] was one of the best performers, if its limited functionality was sufficient. LOUDS simulates various tree operations (like going to the parent, to the i-th child, etc.) via *rank*/*select* queries on the $2n$ bits that describe the tree topology.

We consider their 72-million node "suffix tree" [1] and, following their experimental setup, choose 33,000 random nodes with 5 children or more, and carry out the operation that finds the 5th child. Fig. 4 (right) shows the effect of replacing the *rank*/*select* structure used in LOUDS with our combined sampled structure (using various samplings in the range $S_r = [512, 2048]$ and $S_s = [1024, 32768]$). As it can be seen, LOUDS is much better than the alternatives, and we improve RG-based LOUDS as soon as we use 3% of extra space or more, reaching an improvement of up to 20% when using the same space.

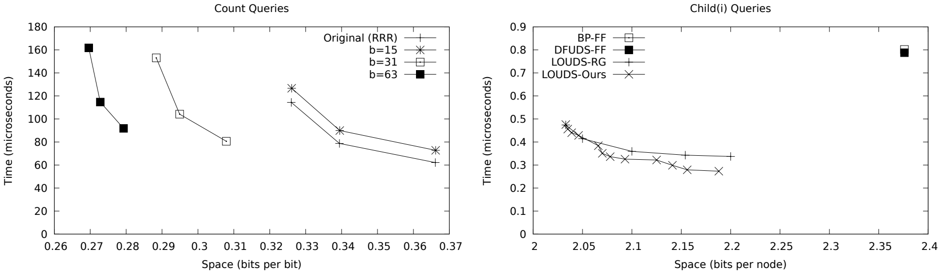


Fig. 4. On the left, counting on the FM-index. On the right, operation *child* on suffix trees. Time is measure in μsec per query, and space in overall number of bits per bit of the plain encoding of the sequence.

6 Conclusions

We have introduced two new techniques that aim at solving two recurrent problems in *rank*/*select* data structures: (1) too much space overhead, (2) bad performance for *select*. We have achieved (i) a structure for compressed bitmaps that retains the performance of existing ones, but reduces their space overhead by 50%–60%, reaching just 0.1 bit of space overhead over the entropy and solving *rank* in about 0.4 μsec and *select* in about 1 μsec; and (ii) a structure for plain bitmaps that combines *rank*/*select* data in a way that achieves just 3% of extra space and solves both operations in about 0.2 μsec, very far from the current space/time tradeoffs. In additions, our structures are rather simple.

These are two very relevant improvements that will find immediate applications. We have already illustrated one application to compressed text indexes and another to compact representations of trees.

Our data structures could perform slower for *select* if the bitmap had very long areas without 1s, even if we would traverse then fast using the S_r sampling. For such long areas we could apply the basic idea of Clark [2] and store the precomputed *select* answers inside. This requires insignificant extra space if applied on very long blocks. Also, in order to support $select_0$ we could add another sparse sampling using S_s (recall that in the recommended setup this stores just one integer every 8192 positions, so doubling this sampling impacts very little). Finally, it is natural to combine our two ideas into a single index for compressed bitmaps: right now our compressed representation carries out *select* by binary searching the superblocks. Our combined structure could speed this up to the times for *rank*, which are around 0.4 μsec on the compressed structure. This is our future work plan.

References

1. Arroyuelo, D., Cánovas, R., Navarro, G., Sadakane, K.: Succinct trees in practice. In: Proc. 11th ALENEX, pp. 84–97 (2010)
2. Clark, D.: Compact Pat Trees. PhD thesis, University of Waterloo, Canada (1996)
3. Claude, F., Navarro, G.: Practical Rank/Select Queries over Arbitrary Sequences. In: Amir, A., Turpin, A., Moffat, A. (eds.) SPIRE 2008. LNCS, vol. 5280, pp. 176–187. Springer, Heidelberg (2008)
4. Ferragina, P., Manzini, G.: Indexing compressed texts. Journal of the ACM 52(4), 552–581 (2005)
5. Golynski, A.: Optimal Lower Bounds for Rank and Select Indexes. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4051, pp. 370–381. Springer, Heidelberg (2006)
6. Golynski, A., Munro, I., Rao, S.: Rank/select operations on large alphabets: a tool for text indexing. In: Proc. 17th SODA, pp. 368–373 (2006)
7. González, R., Grabowski, S., Mäkinen, V., Navarro, G.: Practical implementation of rank and select queries. In: Proc. 4th WEA, pp. 27–38 (2005) (posters)
8. Grossi, R., Gupta, A., Vitter, J.: High-order entropy-compressed text indexes. In: Proc. 14th SODA, pp. 841–850 (2003)
9. Jacobson, G.: Space-efficient static trees and graphs. In: Proc. 30th FOCS, pp. 549–554 (1989)
10. Munro, I.: Tables. In: Chandru, V., Vinay, V. (eds.) FSTTCS 1996. LNCS, vol. 1180, pp. 37–42. Springer, Heidelberg (1996)
11. Munro, I., Raman, V.: Succinct representation of balanced parentheses, static trees and planar graphs. In: Proc. 38th FOCS, pp. 118–126 (1997)
12. Okanohara, D., Sadakane, K.: Practical entropy-compressed rank/select dictionary. In: Proc. 9th ALENEX (2007)
13. Pagh, R.: Low Redundancy in Static Dictionaries with O(1) Worst Case Lookup Time. In: Wiedermann, J., Van Emde Boas, P., Nielsen, M. (eds.) ICALP 1999. LNCS, vol. 1644, pp. 595–604. Springer, Heidelberg (1999)
14. Patrascu, M.: Succincter. In: Proc. 49th FOCS, pp. 305–313 (2008)
15. Raman, R., Raman, V., Rao, S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: Proc. 13th SODA, pp. 233–242 (2002)
16. Vigna, S.: Broadword Implementation of Rank/Select Queries. In: McGeoch, C.C. (ed.) WEA 2008. LNCS, vol. 5038, pp. 154–168. Springer, Heidelberg (2008)

Space-Efficient Top-k Document Retrieval[*]

Gonzalo Navarro and Daniel Valenzuela

Dept. of Computer Science, Univ. of Chile
{gnavarro,dvalenzu}@dcc.uchile.cl

Abstract. Supporting top-k document retrieval queries on general text databases, that is, finding the k documents where a given pattern occurs most frequently, has become a topic of interest with practical applications. While the problem has been solved in optimal time and linear space, the actual space usage is a serious concern. In this paper we study various reduced-space structures that support top-k retrieval and propose new alternatives. Our experimental results show that our novel structures and algorithms dominate almost all the space/time tradeoff.

1 Introduction

Ranked document retrieval is the basic task of most search engines. It consists in preprocessing a collection of d *documents*, $\mathcal{D} = \{D_1, D_2, \ldots, D_d\}$, so that later, given a *query pattern* P and a *threshold* k, one quickly finds the k documents where P is "most relevant".

The best known application scenario is that of documents being formed by natural language texts, that is, sequences of words, and the query patterns being words, phrases (sequences of words), or sets of words or phrases. Several relevance measures are used, which attempt to establish the significance of the query in a given document [2]. The *term frequency*, the number of times the pattern appears in the document, is the main component of most measures.

Ranked document retrieval is usually solved with some variant of a simple structure called an inverted index [2]. This structure, which is behind most search engines, handles well natural language collections. However, the term "natural language" hides several assumptions that are key to the efficiency of that solution: the text must be easily tokenized into words, there must not be too many different words, and queries must be whole words or phrases.

Those assumptions do not hold in various applications where document retrieval is of interest. The most obvious ones are documents written in Asian languages , where it is not easy to split words automatically, and search engines treat the text as a sequence of symbols, so that queries can retrieve any substring of the text. Other applications simply do not have a concept of word, yet ranked retrieval would be of interest: DNA or protein sequence databases where one seeks the sequences where a short marker appears frequently, source code repositories where one looks for functions making heavy use of an expression or

[*] Partially funded by Fondecyt Grant 1-110066, Chile.

R. Klasing (Ed.): SEA 2012, LNCS 7276, pp. 307–319, 2012.

function call, MIDI sequence databases where one seeks for pieces where a given short passage is repeated, and so on.

These problems are modeled as a text collection where the documents D_i are strings over an alphabet Σ, of size σ, and the queries are also simple strings. The most popular relevance measure is the term frequency, meaning the number of occurrences of the string P in the strings D_i (we discuss other measures in Section 6). We call $n = \sum |D_i|$ the collection size and $m{=}|P|$ the pattern length.

Muthukrishnan [17] pioneered the research on document retrieval for general strings. He solved the simpler problem of "document listing": reporting the *occ* distinct documents where P appears in optimal time $O(m + occ)$ and linear space, $O(n)$ integers (or $O(n \log n)$ bits). Muthukrishnan also considered various other document retrieval problems, but not top-k retrieval.

The first efficient solution for the top-k retrieval problem was introduced by Hon et al. [13]. They achieved $O(m + \log n \log\log n + k)$ time, yet the space was superlinear, $O(n \log^2 n)$ bits. Soon, Hon et al. [12] achieved $O(m + k \log k)$ time and linear space, $O(n \log n)$ bits. Recently, Navarro and Nekrich [18] achieved optimal time, $O(m + k)$, and reduced the space from $O(n \log n)$ to $O(n(\log \sigma + \log d))$ bits (albeit the constant is not small).

While these solutions seem to close the problem, it turns out that the space required by $O(n \log n)$-bit solutions is way excessive for practical applications. A recent space-conscious implementation of Hon et al.'s index [20] showed that it requires at least 5 times the text size.

Motivated by this challenge, there has been a parallel research track on how to reduce the space of these solutions, while retaining efficient search time [21,22,12,7,5,3,19,11]. In this work we introduce a new variant with relevant theoretical and practical properties, and show experimentally that it dominates previous work. The next section puts our contribution in context.

2 Related Work

Most of the data structures for general text searching, and in particular the classical ones for document retrieval [17,12], build on on *suffix arrays* [16] and *suffix trees* [23]. Regard the collection $\mathcal{D}$ as a single text $T[1, n] = D_1 D_2 \ldots D_d$, where each D_i is terminated by a special symbol "\$". A suffix array $A[1, n]$ is a permutation of the values $[1, n]$ that points to all the *suffixes* of T: $A[i]$ points to the suffix $T[A[i], n]$. The suffixes are *lexicographically sorted* in A: $T[A[i], n] < T[A[i{+}1], n]$ for all $1 \leq i < n$. Since the occurrences of any pattern P in T correspond to suffixes of T that are prefixed by P, the occurrences are pointed from a contiguous area in the suffix array $A[sp, ep]$. A simple binary search finds *sp* and *ep* in $O(m \log n)$ time [16]. A suffix tree is a digital tree with $O(n)$ nodes where all the suffixes of T are inserted and unary paths are compacted. Every internal node of the suffix tree corresponds to a repeated substring of T and its associated suffix array interval;, suffix tree leaves correspond to the suffixes and their corresponding suffix array cells. A top-down traversal in the suffix tree finds the internal node (called the *locus* of P) from where all the suffixes prefixed with P descend, in $O(m)$ time. Once sp and ep are known, the top-k query finds the k documents where most suffixes in $A[sp, ep]$ start.

A first step towards reducing the space in top-k solutions is to compress the suffix array. *Compressed suffix arrays (CSAs)* simulate a suffix array within as little as $nH_k(T) + o(n \log \sigma)$ bits, for any $k \leq \alpha \log_\sigma n$ and any constant $0 < \alpha < 1$. Here $H_k(T)$ is the k-th order entropy of T, a measure of its statistical compressibility. The CSA, using $|CSA|$ bits, finds sp and ep in time $\mathsf{search}(m)$, and computes any cell $A[i]$, and even $A^{-1}[i]$, in time $\mathsf{lookup}(n)$. For example, a CSA achieving the small space given above [6] achieves $\mathsf{search}(m) = O(m(1 + \frac{\log \sigma}{\log \log n}))$ and $\mathsf{lookup}(n) = O(\log^{1+\epsilon} n)$ for any constant $\epsilon > 0$. CSAs also replace the collection, as they can extract any substring of T.

In their very same foundational paper, Hon et al. [12] proposed an alternative succinct data structure to solve the top-k problem. Building on a solution by Sadakane [21] for document listing, they use a CSA for T and one smaller CSA for each document D_i, plus a little extra data, for a total space of $2|CSA|+o(n)+d\log(n/d)+O(d)$ bits. They achieve time $O(\mathsf{search}(m)+k\log^{3+\epsilon} n\cdot\mathsf{lookup}(n))$, for any constant $\epsilon > 0$. Gagie et al. [7] slightly reduced the time to $O(\mathsf{search}(m) + k \log d \log(d/k) \log^{1+\epsilon} n \cdot \mathsf{lookup}(n))$, and Belazzougui and Navarro [3] further improved it to $O(\mathsf{search}(m) + k \log k \log(d/k) \log^{\epsilon} n \cdot \mathsf{lookup}(n))$.

The essence of the succinct solution by Hon et al. [12] is to preprocess top-k answers for the lowest suffix tree nodes containing any range $A[i \cdot g, j \cdot g]$ for some sampling parameter g. Given the query interval $A[sp, ep]$, they find the highest preprocessed suffix tree node whose interval $[sp', ep']$ is contained in $[sp, ep]$. They show that $sp' - sp < g$ and $ep - ep' < g$, and then the cost of correcting the precomputed answer using the extra occurrences at $A[sp, sp'{-}1]$ and $A[ep'{+}1, ep]$ is bounded. For each such extra occurrence $A[i]$, one finds out its document, computes the number of occurrences of P within that document, and lets the document compete in the top-k precomputed list. Hon et al. use the individual CSAs and other data structures to carry out this task. The subsequent improvements [7,3] are due to small optimizations on this basic design.

Gagie et al. [7] also pointed out that in fact Hon et al.'s solution can run on any other data structure able to (1) telling which document corresponds to a given $A[i]$, and (2) count how many times the same document appears in any interval $A[sp, ep]$. A structure that is suitable for this task is the *document array* $D[1, n]$, where $D[i]$ is the document $A[i]$ belongs to [17]. While in Hon et al.'s solution this is computed from $A[i]$ using $d \log(n/d) + O(d)$ extra bits [21], we need more machinery for task (2). A good alternative was proposed by Mäkinen and Valimäki [22] in order to reduce the space of Muthukrishnan's document listing solution [17]. The structure is a *wavelet tree* [10] on D. The wavelet tree represents D using $n \log d + o(n) \log d$ bits and not only computes any $D[i]$ in $O(\log d)$ time, but it can also compute operation $rank_i(D, j)$, which is the number of occurrences of document i in $D[1, j]$, in $O(\log d)$ time too. This solves operation (2) as $rank_{D[i]}(D, ep) - rank_{D[i]}(D, sp{-}1)$. With the obvious disadvantage of the considerable extra space to represent D, this solution changes $\mathsf{lookup}(n)$ by $\log d$ in the query time. Gagie et al. show many other combinations that solve (1) and (2). One of the fastest uses Golynski et al.'s representation [9] on D and, within the same space, changes $\mathsf{lookup}(n)$ to $\log\log d$ in the time. Very recently, Hon, Shah, and Thankachan [11] presented new combinations in the line of Gagie et al., using also faster CSAs. The least space-consuming one requires $n\log d + n\,o(\log d)$ bits of extra space on top of the CSA of T, and improves the time to $O(\mathsf{search}(m) + k(\log k + (\log\log n)^{2+\epsilon}))$.

Belazzougui and Navarro [3] used an approach based on minimum perfect hash functions to replace the array D by a weaker data structure that takes $O(n\log\log\log d)$ bits of space and supports the search in time $O(\mathsf{search}(m) + k\log k\log^{1+\epsilon} n \cdot \mathsf{lookup}(n))$. This solution is intermediate between representing D or the individual CSAs and it could have practical relevance.

Culpepper et al. [5] built on an improved document listing algorithm on wavelet trees [8] to achieve two top-k algorithms, called *Quantile* and *Greedy*, that use the wavelet tree alone (i.e., without Hon et al.'s [12] extra structures). Despite their worst-case complexity being as bad as extracting the results one by one in $A[sp, ep]$, that is, $O((ep - sp + 1)\log d)$, in practice the algorithms performed very well, being Greedy superior. They implemented Sadakane's solution [21] of using individual CSAs for the documents and showed that the overheads are very high in practice. Navarro et al. [19] arrived at the same conclusion, showing that Hon et al.'s original succinct scheme is not promising in practice: both space and time were much higher in practice than Culpepper et al.'s solution. However, their preliminary experiments [19] showed that Hon et al.'s scheme could compete when running on wavelet trees.

Navarro et al. [19] also presented the first implemented alternative to reduce the space of wavelet trees, by using Re-Pair compression [15] on the bitmaps. They showed that significant reductions in space were possible in exchange for an increase in the response time of Culpepper et al.'s Greedy algorithm (half the space and twice the time is a common figure).

This review exposes interesting contrasts between the theory and the practice in this area. On one hand, the structures that are in theory larger and faster (i.e., the $n\log d$-bits wavelet tree versus a second CSA of at most $n\log\sigma$ bits) are in practice *smaller* and faster. On the other hand, algorithms with no worst-case bound (Culpepper et al.'s [5]) perform very well in practice. Yet, the space of wavelet trees is still considerably large in practice (about twice the plain size of T in several test collections [19]), especially if we consider that they represent totally redundant information that could be extracted from the CSA of T.

In this paper we study a new practical alternative. We use Hon et al.'s [12] succinct structure on top of a wavelet tree, but instead of brute force we use a variant of Culpepper et al.'s [5] method to find the extra candidate documents in $A[sp, sp'-1]$ and $A[ep'+1, ep]$. We can regard this combination either as Hon et al.'s method boosting Culpepper et al. or vice versa. Culpepper et al. boost Hon et al.'s method, while retaining its good worst-case complexities, as they find the extra occurrences more cleverly than by enumerating them all. Hon et al. boost plain Culpepper et al.'s method by having precomputed a large part of the range, and thus ensuring that only small intervals have to be handled.

We consider the plain and the compressed wavelet tree representations, and the straightforward and novel representations of Hon et al.'s succinct structure. We compare these alternatives with the original Culpepper et al.'s method (on plain and compressed wavelet trees), to test the hypothesis that adding Hon et al.'s structure is worth the extra space. Similarly, we include in the comparison the basic Hon et al.'s method (with and without compressed structure) over Golynski et al.'s [9] sequence representation, to test the hypothesis that using Culpepper et al.'s method over the wavelet tree is worth compared to the brute force method over the fastest sequence representation [9]. This brute force method is also at the core of the new proposal by Hon et al. [11].

Our experiments show that our new algorithms and data structures dominate almost all the space/time tradeoff for this problem, becoming a new practical reference point.

3 Implementing Hon et al.'s Succinct Structure

The succinct structure of Hon et al. [12] is a sparse generalized suffix tree of T (SGST; "generalized" means it indexes d strings). It is obtained by cutting $A[1,n]$ into blocks of length g and sampling the first and last cell of each block (recall that cells of A are leaves of the suffix tree). Then all the lowest common ancestors (lca) of pairs of sampled leaves are marked, and a tree τ_k is formed with those (at most) $2n/g$ marked internal nodes. The top-k answer is stored for each marked node, using $O((n/g)k\log n)$ bits. This is done for $k = 1, 2, 4, \ldots$, and parameter g is of the form $g = k \cdot g'$. The final space is $O((n/g')\log d \log n)$ bits. This is made $o(n)$ by properly choosing g'.

To answer top-k queries, they search the CSA for P, to obtain the suffix range $A[sp, ep]$ of the pattern. Then they turn to the closest higher power of two of k, $k^* = 2^{\lceil \log k \rceil}$, and let $g = k^* \cdot g'$ be the corresponding g value. They now find the locus of P in the tree τ_{k^*} by descending from the root until finding the first node v whose interval $[sp_v, ep_v]$ is contained in $[sp, ep]$. They have at v the top-k candidates for $[sp_v, ep_v]$ and have to correct the answer considering $[sp, sp_v{-}1]$ and $[ep_v{+}1, ep]$. Now we introduce two implementations of this idea.

3.1 Sparsified Generalized Suffix Tree (SGST)

Let us call $l_i = A[i]$ the i-th leaf. Given a value of k we define $g = k \cdot g'$, for a space/time tradeoff parameter g', and sample n/g leaves $l_1, l_{g+1}, l_{2g+1}, \ldots$, instead of sampling $2n/g$ leaves as in the theoretical proposal. We mark internal SGST nodes $lca(l_1, l_{g+1}), lca(l_{g+1}, l_{2g+1}), \ldots$. It is easy to prove that any $v = lca(l_{ig+1}, l_{jg+1})$ is also $v = lca(l_{rg+1}, l_{(r+1)g+1})$ for some r (precisely, r is the rightmost sampled leaf descending from the child of v that is ancestor of l_{ig+1}). Thus these n/g SGST nodes suffice and can be computed in linear time [4].

Now we note that there is a great deal of redundancy in the $\log d$ trees τ_k, since the nodes of τ_{2k} are included in those of τ_k, and the $2k$ candidates stored in the nodes of τ_{2k} contain those in the corresponding nodes of τ_k. To factor out some of this redundancy we store only one tree τ, whose nodes are the same of τ_1, and record the *class* $c(v)$ of each node $v \in \tau$. This is $c(v) = \max\{k,\ v \in \tau_k\}$ and can be stored in $\log\log d$ bits. Each node $v \in \tau$ stores the top-$c(v)$ candidates corresponding to its interval, using $c(v)\log d$ bits, and their frequencies, using $c(v)\log n$ bits, plus a pointer to the table, and the interval itself, $[sp_v, ep_v]$, using $2\log n$ bits. All the information on intervals and candidates is factored in this way, saving space. Note that the class does not necessarily decrease monotonically in a root-to-leaf path of τ, thus we store all the topologies independently to allow for efficient traversal of the τ_k trees, for $k > 1$. Apart from topology information, each node of such τ_k trees contains just a pointer to the corresponding node in τ, using $\log|\tau|$ bits.

In our first data structure, the topology of the trees τ and τ_k is represented using pointers of $\log|\tau|$ and $\log|\tau_k|$ bits, respectively. To answer top-k queries, we find the range $A[sp, ep]$ using a CSA (whose space and negligible time will not be reported because it is orthogonal to all the data structures). Now we find the locus in the appropriate tree τ_{k^*} top-down, binary searching the intervals $[sp_v, ep_v]$ of the children of the current node, and extracting those intervals using the pointer to τ. By the properties of the sampling [12] it follows that we will traverse in this descent nodes $v \in \tau_{k^*}$ such that $[sp, ep] \subseteq [sp_v, ep_v]$, until reaching a node v so that $[sp_v, ep_v] = [sp', ep'] \subseteq [sp, ep] \subseteq [sp' - g, ep' + g]$ (or reaching a leaf $u \in \tau_k$ such that $[sp, ep] \subseteq [sp_u, ep_u]$, in which case $ep - sp + 1 < 2g$). This v is the locus of P in τ_{k^*}, and we find it in time $O(m \log \sigma)$. This is negligible compared to the subsequent costs, as well as it is the CSA search.

3.2 Succinct SGST

Our second implementation uses represents the tree topologies without pointers. Although the tree operations are slightly slower than with pointers, this slowdown occurs on a less significant part of the search process, and a succinct representation allows one to reduce the sampling parameter g for the same space.

Arroyuelo et al. [1] showed that, for the functionality it provides, the most promising succinct representation of trees is the so-called LOUDS [14]. It requires $2N + o(N)$ bits of space (in practice, as little as $2.1\,N$) to represent a tree of N nodes, and it solves many operations in constant time (less than a microsecond in practice). We resort to their labeled trees [1] implementation, We encode the values sp_v and ep_v, pointers to τ (in τ_k), and pointers to the candidates in a separate array, indexed by the LOUDS rank of the node v, managing them as Arroyuelo et al. [1] manage labels. We use that implementation [1].

4 A New Top-*k* Algorithm

We run a combination of the algorithm by Hon et al. [12] and those of Culpepper et al. [5], over a wavelet tree representation of the document array $D[1, n]$. Culpepper et al. introduce, among others, a document listing method (DFS) and a Greedy top-k heuristic. We adapt these to our particular top-k subproblem.

If the search for the locus of P ends at a leaf u that still contains the interval $[sp, ep]$, Hon et al. simply scan $A[sp, ep]$ by brute force and accumulate frequencies. We use instead Culpepper et al.'s Greedy algorithm, which is always better than a brute-force scanning.

When, instead, the locus of P is a node v where $[sp_v, ep_v] = [sp', ep'] \subseteq [sp, ep]$, we start with the precomputed answer of the $k \leq k^*$ most frequent documents in $[sp', ep']$, and update it to consider the subintervals $[sp, sp'-1]$ and $[ep'+1, ep]$. We use the wavelet tree of D to solve the following problem: Given an interval $D[l, r]$, and two subintervals $[l_1, r_1]$ and $[l_2, r_2]$, enumerate all the distinct values in $[l_1, r_1] \cup [l_2, r_2]$, and their frequencies in $[l, r]$. We propose two solutions, which generalize the heuristics proposed by Culpepper et al. [5].

4.1 Restricted Depth-First Search (DFS)

Let us consider a wavelet tree [10] representation of an array D. At the root, a bitmap $B[1, n]$ stores $B[i] = 0$ if $D[i] \leq d/2$ and $B[i] = 1$ otherwise. The left child of the root is, recursively, a wavelet tree handling the subsequence of D with values $D[i] \leq d/2$, and the right child handles the subsequence of values $D[i] > d/2$. Added over the $\log d$ levels, the wavelet tree requires $n \log d$ bits of space. With $o(n \log d)$ additional bits we answer in constant time any query $rank_{0/1}(B, i)$ over any bitmap B [14].

Note that any interval $D[i, j]$ can be projected into the left child of the root as $[i_0, j_0] = [rank_0(B, i-1)+1, rank_0(B, j)]$, and into its right child as $[i_1, j_1] = [rank_1(B, i-1)+1, rank_1(B, j)]$, where B is the root bitmap. Those can then be projected recursively into other wavelet tree nodes.

Our restricted DFS algorithm begins at the root of the wavelet tree and tracks down the intervals $[l, r] = [sp, ep]$, $[l_1, r_1] = [sp, sp'-1]$, and $[l_2, r_2] = [ep'+1, ep]$. More precisely, we count the number of zeros and ones in B in ranges $[l_1, r_1] \cup [l_2, r_2]$, as well as in $[l, r]$, using a constant number of *rank* operations on B. If there are any zeros in $[l_1, r_1] \cup [l_2, r_2]$, we map all the intervals into the left child of the node and proceed recursively from this node. Similarly, if there are any ones in $[l_1, r_1] \cup [l_2, r_2]$, we continue on the right child of the node. When we reach a wavelet tree leaf we report the corresponding document, and the frequency is the length of the interval $[l, r]$ at the leaf.

When solving the problem in the context of top-k retrieval, we can prune some recursive calls. If, at some node, the size of the local interval $[l, r]$ is smaller than our current kth candidate to the answer, we stop exploring its subtree since it cannot contain competitive documents.

4.2 Restricted Greedy

Following the idea of Culpepper et al., we can not only stop the traversal when $[l, r]$ is too small, but also prioritize the traversal of the nodes by their $[l, r]$ value.

We keep a priority queue where we store the wavelet tree nodes yet to process, and their intervals $[l, r]$, $[l_1, r_1]$, and $[l_2, r_2]$. The priority queue begins with one element, the root. Iteratively, we remove the element with highest $r-l+1$ value from the queue. If it is a leaf, we report it. Otherwise, we project the intervals into its left and right children, and insert each such children containing nonempty intervals $[l_1, r_1]$ or $[l_2, r_2]$ into the queue. As soon as the $r-l+1$ value of the element we extract from the queue is not larger than the kth frequency known at the moment, we can stop.

4.3 Heaps for the *k* Most Frequent Candidates

Our two algorithms solve the query assuming that we can easily know at each moment which is the kth best candidate known up to now. We use a min-heap data structure for this purpose. It is loaded with the top-k precomputed candidates corresponding to the interval $[sp', ep']$. At each point, the top of the heap gives the kth known frequency in constant time. Given that the previous algorithms stop when they reach a wavelet tree node where $r-l+1$ is not larger than the kth known frequency, it follows that each time the algorithms report a new candidate, that candidate is more frequent than our kth known candidate. Thus we replace the top of our heap with the reported candidate and reorder the heap (which is always of size k, or less until we find k distinct elements in $D[sp, ep]$). Therefore each candidate reported costs $O(\log d+\log k)$ time (there are also steps that do not yield any result, but the overall bound is still $O(g(\log d + \log k))$).

A remaining issue is that we can find again, in our DFS or Greedy traversal, a node that was in the original top-k list, and thus possibly in the heap. This means that the document had been inserted with its frequency in $D[sp', ep']$, but since it appears more times in $D[sp, ep]$, we must now increase its frequency and restore the min-heap invariant. It is not hard to maintain a hash table with forward and backward pointers to the heap so that we can track their current positions and replace their values. However, for the small k values used in practice (say, ten or at most hundreds), it is more practical to scan the heap for each new candidate to insert than to maintain all those pointers upon all operations.

5 Experimental Results

We test our implementations of Hon et al.'s succinct structure combined with a wavelet tree (as explained, the original proposal is not competitive in practice [19]). We used three test collections of different nature: **ClueWiki**, a 141 MB sample of *ClueWeb09*, formed by 3,334 Web pages from the English Wikipedia; **KGS**, a 25 MB collection of 18,838 sgf-formatted Go game records (`http://www.u-go.net/gamerecords`); and **Proteins**, a 60 MB collection of 143,244 sequences of Human and Mouse proteins (`http://www.ebi.ac.uk/swissprot`).

Our tests were run on a 4-core 8-processors Intel Xeon, 2Ghz each, with 16GB RAM and 2MB cache. We compiled using `g++` with full optimization. For queries, we selected 1,000 substrings at random positions, of length 3 and 8, and retrieved the top-k documents for each, for $k = 1$ and 10.

Choosing Our Best Variant. Our first round of experiments compares our different implementations of SGSTs (i.e., the trees τ_k, see Section 3) over a single implementation of wavelet tree (`Alpha`, choosing the best value for α in each case [19]). We tested a pointer-based representation of the SGST (`Ptrs`, the original proposal [12]), a LOUDS-based representation (`LOUDS`), our variant of `LOUDS` that stores the topologies in a unique tree τ (`LIGHT`), and our variant of `LIGHT` that does not store frequencies of the top-k candidates (`XLIGHT`). We used sampling steps of 200 and 400 for g'. For each value of g, we obtain a curve with various sampling steps for the *rank* computations on the wavelet tree bitmaps.

We also tested different algorithms to find the top-k among the precomputed candidates and remaining leaves (see Section 4): Our modified greedy (`Greedy`), our modified depth-first-search (`DFS`), and the brute-force selection procedure of the original proposal [12] on top of the same wavelet tree (`Select`). As this is orthogonal to the data structures used, we compare these algorithms only on top of the `Ptrs` structure. The other structures will be tested using the best method.

Figure 1 shows the results. Method `Greedy` is always better than `Select` (up to 80% better) and `DFS` (up to 50%), which confirms intuition. Using `LOUDS` representation instead of `Ptr` had almost no impact on the time. This is because time needed to find the locus is usually negligible compared with that to explore the uncovered leaves. Further costless space gains are obtained with variant `LIGHT`. Variant `XLIGHT`, instead, reduces the space of `LIGHT` at a noticeable cost in time that makes it not so interesting, except on **Proteins**. In various cases the sparser sampling dominates the denser one, whereas in others the latter makes the structure faster if sufficient space is spent. To compare with other techniques, we will use variant `LIGHT` on **ClueWiki** and **KGS**, and `XLIGHT` on **Proteins**, both with $g' = 400$. This combination will be called generically `SSGST`.

Comparison with Previous Work. We now compare ours with previous work. The Greedy heuristic [5] is run over different wavelet-tree representations of the document array: a plain one (`WT-Plain`) [5], a Re-Pair compressed one (`WT-RP`), and a hybrid that at each wavelet tree level chooses between plain, Re-Pair, or entropy-based compression of the bitmaps (`WT-Alpha`) [19]. We combine these with our best implementation of Hon et al.'s structure (suffixing the previous names with `+SSGST`). We also consider variant `Goly+SSGST` [7,11], which runs the *rank*-based method (`Select`) on top of the fastest *rank*-capable sequence representation of the document array (Golynski et al.'s [9], which is faster than wavelet trees for *rank* but does not support our more sophisticated algorithms; here we used the implementation at `http://libcds.recoded.cl`).

Our new structures dominate most of the space-time map. When using little space, variant `WT-RP+SSGST` dominates, being only ocassionally and slightly superseded by `WT-RP`. When using more space, `WT-Alpha+SSGST` takes over, and finally, with even more space, `WT-Plain+SSGST` becomes the best choice. Most of the exceptions arise in **Proteins**, which due to its incompressibility [19] makes `WT-Plain+SSGST` essentially the only interesting variant. The alternative `Goly+SSGST` is no case faster than a Greedy algorithm over plain wavelet trees (`WT-Plain`), and takes more space.

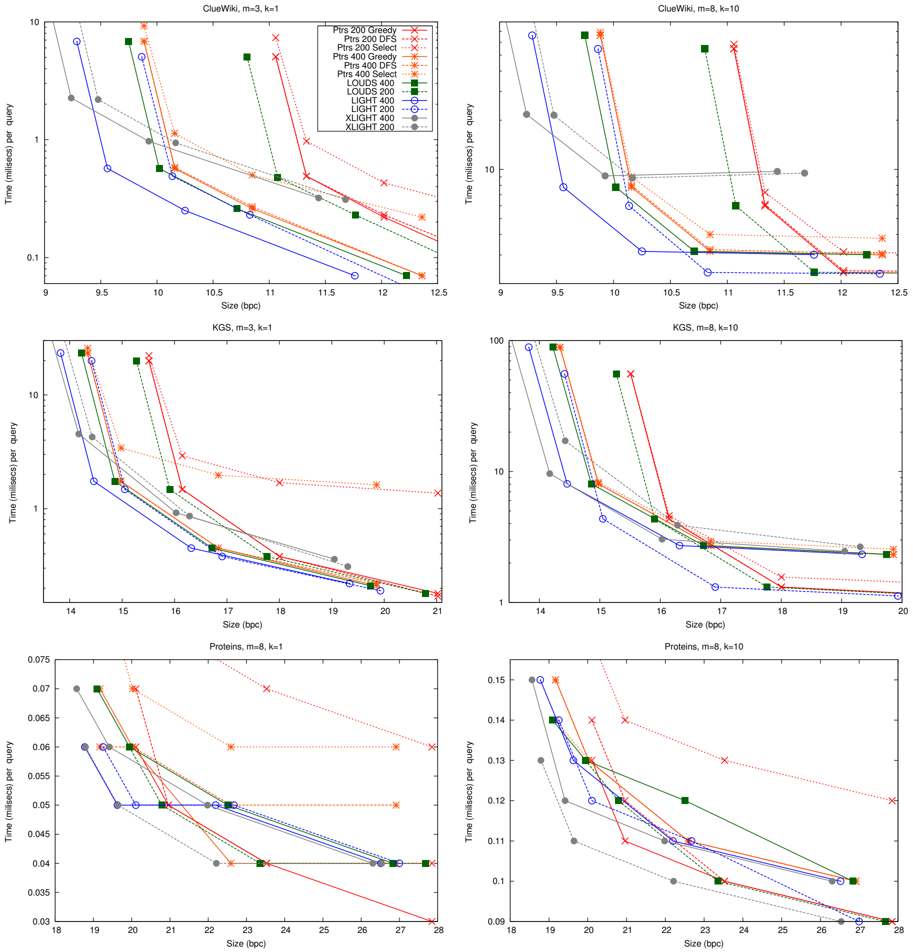


Fig. 1. Our different alternatives for top-k queries. On the left for $k = 1$ and pattern length $m = 3$; on the right for $k = 10$ and $m = 8$.

6 Final Remarks

We can further reduce the space in exchange for possibly higher times. For example the sequence of all precomputed top-k candidates can be Huffman-compressed, as there is much repetition in the sets and a zero-order compression would yield space reductions of up to 25%. The pointers to those tables could also be removed, by separating the tables by size, and computing the offset within each size using *rank* on the sequence of classes of the nodes in τ.

More in perspective, term frequency is probably the simplest relevance measure. In Information Retrieval, more sophisticated ones like BM25 are used. Such formula involves the sizes of the documents, and thus techniques like Culpepper et al.'s [5] do not immediately apply. However, Hon et al.'s [12] does,

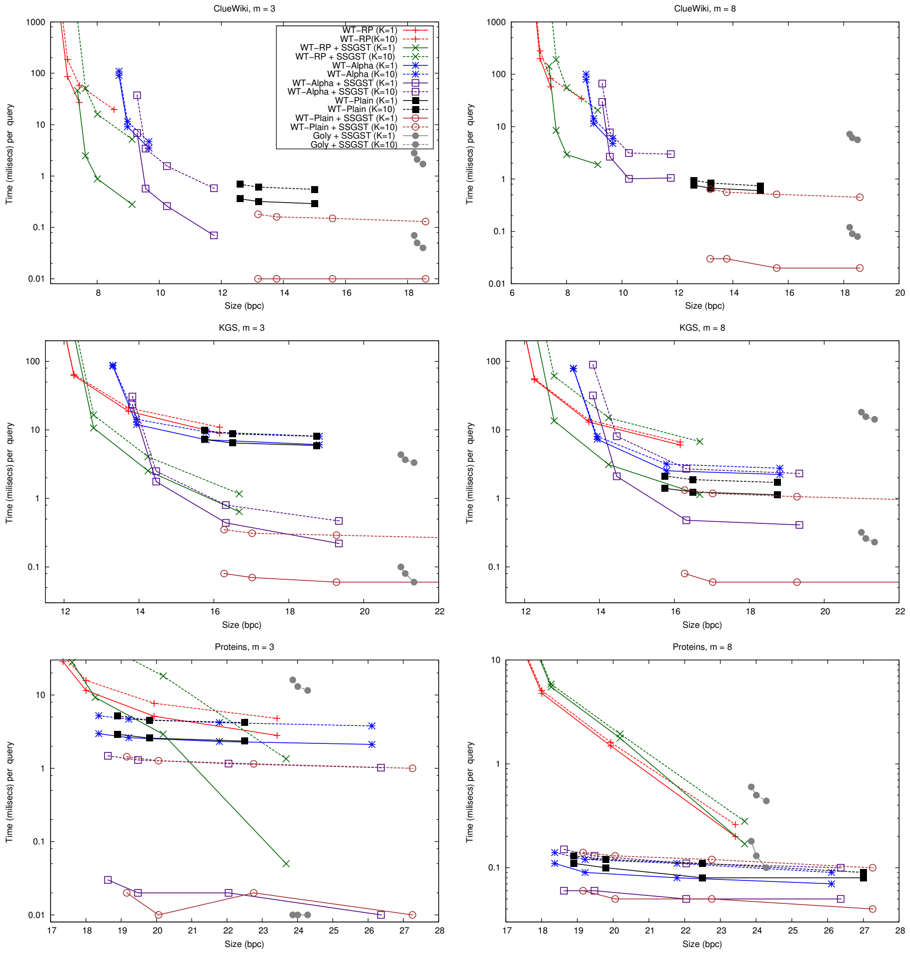


Fig. 2. Comparison with previous work, for $m = 3$ (left) and $m = 8$ (right)

by simply storing the precomputed top-k answers according to BM25 and using their brute-force traversal instead of our "restricted Greedy/DFS" methods. The times would be very similar to the variant we called `Select` in this paper.

Sadakane [21] showed how to efficiently compute *document frequencies* (i.e., in how many documents does a pattern appear), in constant time and using just $2n+o(n)$ bits. With term frequency, these two measures are sufficient to compute the popular tf-idf score. Note, however, that as long as queries are formed by a single term, the top-k ranking is the same as given by term frequency alone. Document frequency makes a difference on *bag-of-word* queries, which involve several terms. Structures like those we have explored in this paper are able to emulate a (virtual) inverted list, sorted by decreasing term frequency, for any pattern, and thus enable the implementation of any top-k algorithm for bags of words designed for inverted indexes.

References

1. Arroyuelo, D., Cánovas, R., Navarro, G., Sadakane, K.: Succinct trees in practice. In: Proc. 11th ALENEX, pp. 84–97 (2010)
2. Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval, 2nd edn. Addison-Wesley (2011)
3. Belazzougui, D., Navarro, G.: Improved Compressed Indexes for Full-Text Document Retrieval. In: Grossi, R., Sebastiani, F., Silvestri, F. (eds.) SPIRE 2011. LNCS, vol. 7024, pp. 386–397. Springer, Heidelberg (2011)
4. Bender, M., Farach-Colton, M.: The LCA Problem Revisited. In: Gonnet, G.H., Viola, A. (eds.) LATIN 2000. LNCS, vol. 1776, pp. 88–94. Springer, Heidelberg (2000)
5. Culpepper, J.S., Navarro, G., Puglisi, S.J., Turpin, A.: Top-k Ranked Document Search in General Text Databases. In: de Berg, M., Meyer, U. (eds.) ESA 2010, Part II. LNCS, vol. 6347, pp. 194–205. Springer, Heidelberg (2010)
6. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Trans. Alg. 3(2), article 20 (2007)
7. Gagie, T., Navarro, G., Puglisi, S.J.: Colored Range Queries and Document Retrieval. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 67–81. Springer, Heidelberg (2010)
8. Gagie, T., Puglisi, S.J., Turpin, A.: Range Quantile Queries: Another Virtue of Wavelet Trees. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 1–6. Springer, Heidelberg (2009)
9. Golynski, A., Munro, I., Rao, S.: Rank/select operations on large alphabets: a tool for text indexing. In: Proc. 17th SODA, pp. 368–373 (2006)
10. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proc. 14th SODA, pp. 636–645 (2003)
11. Hon, W.-K., Shah, R., Thankachan, S.: Towards an optimal space-and-query-time index for top-k document retrieval. CoRR, arXiv:1108.0554 (2011)
12. Hon, W.-K., Shah, R., Vitter, J.: Space-efficient framework for top-k string retrieval problems. In: Proc. 50th FOCS, pp. 713–722 (2009)
13. Hon, W.-K., Shah, R., Wu, S.-B.: Efficient Index for Retrieving Top-k Most Frequent Documents. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 182–193. Springer, Heidelberg (2009)
14. Jacobson, G.: Space-efficient static trees and graphs. In: Proc. 30th FOCS, pp. 549–554 (1989)
15. Larsson, J., Moffat, A.: Off-line dictionary-based compression. Proc. of the IEEE 88(11), 1722–1732 (2000)
16. Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches. SIAM J. Comp. 22(5), 935–948 (1993)
17. Muthukrishnan, S.: Efficient algorithms for document retrieval problems. In: Proc. 13th SODA, pp. 657–666 (2002)
18. Navarro, G., Nekrich, Y.: Top-k document retrieval in optimal time and linear space. In: Proc. 22nd SODA, pp. 1066–1078 (2012)
19. Navarro, G., Puglisi, S.J., Valenzuela, D.: Practical Compressed Document Retrieval. In: Pardalos, P.M., Rebennack, S. (eds.) SEA 2011. LNCS, vol. 6630, pp. 193–205. Springer, Heidelberg (2011)

20. Patil, M., Thankachan, S., Shah, R., Hon, W.-K., Vitter, J., Chandrasekaran, S.: Inverted indexes for phrases and strings. In: Proc. SIGIR, pp. 555–564 (2011)
21. Sadakane, K.: Succinct data structures for flexible text retrieval systems. J. Discr. Alg. 5(1), 12–22 (2007)
22. Välimäki, N., Mäkinen, V.: Space-Efficient Algorithms for Document Retrieval. In: Ma, B., Zhang, K. (eds.) CPM 2007. LNCS, vol. 4580, pp. 205–215. Springer, Heidelberg (2007)
23. Weiner, P.: Linear pattern matching algorithm. In: Proc. 14th Annual IEEE Symposium on Switching and Automata Theory, pp. 1–11 (1973)

Engineering Efficient Paging Algorithms[⋆]

Gabriel Moruz, Andrei Negoescu, Christian Neumann, and Volker Weichert

Goethe University Frankfurt am Main. Robert-Mayer-Str. 11-15,
60325 Frankfurt am Main, Germany
{gabi,negoescu,cneumann,weichert}@cs.uni-frankfurt.de

Abstract. In the field of online algorithms paging is a well studied problem. LRU is a simple paging algorithm which incurs few cache misses and supports efficient implementations. Algorithms outperforming LRU in terms of cache misses exist, but are in general more complex and thus not automatically better, since their increased runtime might annihilate the gains in cache misses. In this paper we focus on efficient implementations for the OnOPT class described in [13], particularly on an algorithm in this class, denoted RDM, that was shown to typically incur fewer misses than LRU. We provide experimental evidence on a wide range of cache traces showing that our implementation of RDM is competitive to LRU with respect to runtime. In a scenario incurring realistic time penalties for cache misses, we show that our implementation consistently outperforms LRU, even if the runtime of LRU is set to zero.

1 Introduction

Paging is a prominent, well studied problem in the field of online algorithms. It also has significant practical importance, since the paging strategy is an essential efficiency issue in the field of operating systems. Formally, the problem is defined as follows. Given a cache of size k and a memory of infinite size, the algorithm must process pages *online*, i.e. make decisions based on the input sequence seen so far. If the page requested is not in the cache, a *cache miss* occurs and the page must be loaded in the cache; additionally, if the cache was full, some page must be evicted to accommodate the new one. The goal is to minimize the number of cache misses.

Traditionally, when evaluating the performance of paging algorithms, most work focuses exclusively on the number of misses incurred. However, in practice, apart from cache misses, factors such as runtime and space usage have a major impact in deciding on which algorithms to use [15, Section 3.4]. In particular, the fact that LRU (Least Recently Used) and its variants are widely popular stems not only from the fact that they incur few cache misses (typically no more

[⋆] Partially supported by the DFG grants ME 3250/1-3 and MO 2057/1-1, and by MADALGO (Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation). A full version of this paper containing all experimental results is available online at www.ae.cs.uni-frankfurt.de/sea12/.

R. Klasing (Ed.): SEA 2012, LNCS 7276, pp. 320–331, 2012.

than a factor of four more than the optimal cost [16]), but also because they have efficient implementations with low overhead in terms of space and runtime.

Typically, online algorithms in general and paging algorithms in particular are analyzed using *competitive analysis* [10,14], where the online algorithm is compared against an optimal offline algorithm. An algorithm is c-competitive if the number of misses incurred is up to a factor of c away from an optimal offline solution. Any deterministic paging algorithm has a competitive ratio of at least k [14], and several k-competitive algorithms are known. Examples include LRU, FIFO, and FWF (Flush When Full); furthermore, all these algorithms can be implemented efficiently in terms of space and runtime. For randomized algorithms, in [7] a lower bound of H_k on the competitive ratio was shown[1], and a $2H_k$-competitive algorithm, denoted Mark, was proposed. Subsequently, several H_k-competitive paging algorithms were proposed, namely Partition [12], Equitable and Equitable2 [1,2], and OnMIN [6].

Based on the layer partition in [11], we proposed in [13] a measure quantifying the "evilness" of the adversary that we denoted *attack rate*. For inputs having attack rate r, we introduced a class of r-competitive algorithms, denoted OnOPT, and we showed that these algorithms achieve a small fault rate on many practical inputs. Finally, we singled out an algorithm in this class, denoted Recency Duration Mix (in short RDM), which we showed to consistently outperform LRU and some of its variants with respect to cache misses on most inputs and cache sizes considered, at times by more than a factor of two.

Our Contributions. In this work we focus on the runtime of paging algorithms that, together with the cache misses, is an important factor in practice. We propose a compressed representation of the layer partitioning in [6,11]. Based on this and on the fact that typically most requests are to so-called *revealed pages* (pages that are for sure in the cache of an optimal algorithm), we engineer speed-up techniques for implementating the OnOPT class. If the fraction of revealed requests is $1 - O(1/k)$ these yield an RDM implementation with an amortized runtime of $O(1)$ per request with very small constant factors. We show on real-world input traces[2] that the new implementation outperforms the tree based approach in [6]. Moreover, we compare the runtime of RDM with LRU and FIFO and show that the runtimes of RDM and LRU are comparable, albeit slower than FIFO. Finally, we use a more general performance measure for paging algorithms, namely the sum of runtime and cache miss penalties. Assuming a realistic cache miss penalty of 9ms, the fact that RDM typically incurs fewer misses than both LRU and FIFO ensures that it achieves better performance for many traces and cache sizes, even if we charge LRU and FIFO a runtime of zero. This shows that OnOPT algorithms in general and RDM in particular may be of practical value.

[1] $H_k = \sum_{i=1}^{k} 1/i$ is the k^{th} harmonic number.

[2] We used all the available original reference traces from http://www.cs.amherst.edu/~sfkaplan/research/trace-reduction/index.html.

Related Work. Although competitive analysis seems too pessimistic, some of its refinements have lead to paging algorithms with low fault rates on traces extracted during the execution of real-world programs. In [8], heuristics motivated by the access graph model from [4] outperformed LRU. These perform an online approximation of the access graph, which models the page access pattern. Another algorithm, RLRU (Retrospective LRU), was proposed in [5], where it was proven to be better than LRU with respect to the relative worst order ratio. RLRU uses information about the optimal offline solution for its decisions. EELRU (Early Eviction LRU) [9] is an adaptive paging algorithm from a less theoretical direction, which simulates a large collection of about 256 parametrized instances of an algorithm which is a mix of LRU and MRU (Most Recently Used). All of these algorithms, including the OnOPT class, have in common that they are more complex than classical algorithms like LRU and FIFO. Because of this it is not obvious whether there exist fast implementations such that the savings in cache misses compensate for the higher runtime overhead.

1.1 Preliminaries

Layer partitioning. Given the request sequence σ seen so far, in an online scenario it is of interest to know the actual cache content C_{OPT} of the optimal offline algorithm LFD (Longest Forward Distance), which evicts, upon a cache miss, the page in cache which is re-requested farthest in the future [3]. Although in general C_{OPT} is not known since it depends also on the future request sequence τ, we are provided with partial information (from σ) about the structure of C_{OPT}, e.g. it contains for sure the most recently requested page, and pages not requested in σ are not in C_{OPT}. We say that immediately after processing σ a set C of k pages is a *valid* configuration iff there exists a future request sequence τ such that LFDs cache content equals C. A precise mathematical characterization of all possible valid configurations was given by Koutsoupias and Papadimitriou [11] and an equivalent variant of this characterization is used by the OnOPT algorithm class [13]. It consists of a partition $L = (L_0|\dots|L_k)$ of the pageset in $k+1$ disjoint sets, denoted layers. Initially, each layer in $L_1, \dots, L_k$ contains one of the first k pairwise distinct pages and L_0 contains all the remaining pages. If L is the layer partition for input σ, let L^p denote the layer partition for σp, the sequence resulting by the request of page p. The layers are updated as follows:

$$L^p = \begin{cases} (L_0 \setminus \{p\}|L_1|\dots|L_{k-2}|L_{k-1} \cup L_k|\{p\}), & \text{if } p \in L_0 \\ (L_0|\dots|L_{i-2}|L_{i-1} \cup L_i \setminus \{p\}|L_{i+1}|\dots|L_k|\{p\}), & \text{if } p \in L_i, i > 0 \end{cases}$$

In [11] it has been shown that a cache configuration C is valid iff it holds that for each $i \in \{1, \dots, k\}$ we have $|C \cap (\cup_{j=1}^{i} L_j)| \le i$.

The *support* of L is defined as $L_1 \cup \dots \cup L_k$. Denoting by singleton a layer with one element, let r be the smallest index such that $L_r, \dots, L_k$ are singletons; the pages in $L_r \cup \dots \cup L_k$ are denoted *revealed.* We denote by *Opt-miss* pages the pages in L_0, while the remaining pages, i.e. pages in support that are not revealed, are *unrevealed* pages. A valid configuration contains all revealed pages and no page from L_0. Note that by the layer update rule all layers are non-empty.

OnOPT *Algorithms.* Algorithms from the OnOPT class use the layer partition as a subroutine. The currently requested page is assigned a priority which shall reflect the rank of its next request among the other pages. Given a priority based future prediction the cache update rule ensures that their cache content is always identical to LFD's, if the prediction is correct. The pseudo-code is given in Algorithm 1. The fact that no cache misses are performed on revealed requests guarantees a reasonable performance, due to the high percentage of revealed requests in the input. In OnOPT we singled out RDM, which combines two priority policies, one based on recency and the other on the time-frame that pages spent in support. RDM achieves good results, outperforming LRU on many real-world traces and cache sizes [13].

Algorithm 1. OnOPT framework

```
procedure OnOPT(Page $p$, Cache $M$)                          ▷ Processes page $p$
    Assign $p$ its priority
    if $p \notin M$ and $p \in L_0$ then                         ▷ Update cache
        Evict page in $M$ with smallest priority
    else if $p \notin M$ and $p \in L_i$, $i > 0$ then
        Identify minimal $j$, with $j \geq i$, satisfying $|(L_1 \cup \cdots \cup L_j) \cap M| = j$
        Evict page from $(L_1 \cup \cdots \cup L_j) \cap M$ having smallest priority
    end if
    Update the layers                                            ▷ Layers update
end procedure
```

1.2 Revealed Requests

We give experimental evidence that a very high percentage of requests are to revealed pages, which is the main motivation for the OnOPT implementations we propose in this paper. For the remainder of the paper we use a collection of cache traces extracted from various applications for our experiments; due to space limitations, details about these traces are given in the full version.

The charts in Figure 1 show that if enough pages fit in memory (usually about 10%), almost all of the requests are to revealed pages. In these cases the ratio of revealed requests in the input is about $(k-1)/k$, which we approximate by $1-O(1/k)$. For the remainder of the paper we will focus on how to process these requests as fast as possible at the expense of increasing the worst case time for processing requests to Opt-miss and unrevealed pages.

2 Compressed Layers

We simplify the layer partition with the main purpose of reducing the runtime for layer updates. The layer partition can be seen as a sequence of conditions that

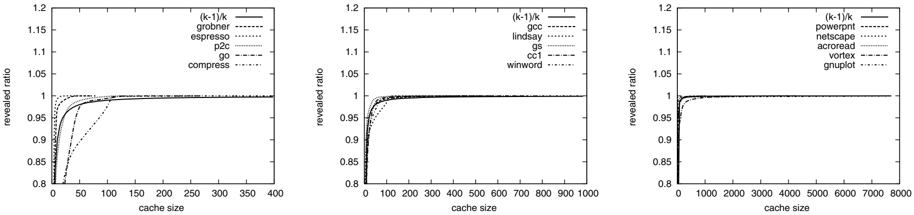


Fig. 1. The ratio of revealed requests against for all cache traces. For decently large cache sizes the ratio of revealed requests is about $(k-1)/k$ for all traces

a valid configuration must fulfill. Consider the initial partition, where each L_i contains exactly one page p_i. The partition implies the constraints that a valid configuration contains at most one element from $\{p_1\}$, two elements from $\{p_1, p_2\}$ and so on. Since each layer has only one page, these k conditions can be reduced to one, namely at most k pages from $\{p_1, \ldots, p_k\}$. We generalize this example as follows. Given the original layer partition L, we define a compressed partition $\mathcal{L}$ which groups all consecutive singletons of L into the first non-singleton layer to the right. An algorithmic description of this process is given in Algorithm 2, an example for $k = 7$ is provided below.

$$L = (10, 3|2|7, 5|4|1, 11|8|9|6),\ \mathcal{L} = (10, 3|\emptyset|2, 7, 5|\emptyset|4, 1, 11|\emptyset|\emptyset|8, 9, 6)$$

Algorithm 2. Layer compression

```
procedure LAYER COMPRESSION(Partition $L = (L_0, \ldots, L_i)$)        ▷ Compress $L$
    $T = \emptyset$;
    for $i = 1$ to $k - 1$ do
        if $|L_i| = 1$ then                                            ▷ $L_i$ is singleton
            $\mathcal{L}_i = \emptyset$; $T = T \cup L_i$;
        else                                                           ▷ $L_i$ is not singleton
            $\mathcal{L}_i = L_i \cup T$; $T = \emptyset$;
        end if
    end for
    $\mathcal{L}_k = L_k \cup T$;
end procedure
```

The compressed partition $\mathcal{L}$ may contain empty sets and describes the same valid configurations as L. For $\mathcal{L}$ we provide a corresponding update rule, which has the advantage that upon revealed requests nothing changes, leading to significant runtime improvements of OnOPT algorithms. Another advantage is that on the cache traces considered the number of non-empty layers is much smaller than k, which allows for more efficient implementations.

Denoting $S_i = L_1 \cup \cdots \cup L_i$, a set of k pages is a valid configuration iff $|C \cap S_i| \leq i$ for all i. Similarly, let $\mathcal{S}_i = \mathcal{L}_1 \cup \cdots \cup \mathcal{L}_i$.

Lemma 1. *The compressed partition $\mathcal{L}$ describes the same valid configurations as L, more precisely it holds for all i: $|C \cap S_i| \leq i$ iff $|C \cap \mathcal{S}_i| \leq i$.*

Proof. Let x and y, $x < y$, be two indices such that $|L_x| > 1$, $|L_y| > 1$, and $L_{x+1}, \ldots, L_{y-1}$ are singletons. Further let L' be the partially compressed layer partition up to the iteration step $i = x$. We assume that L' and L describe the same valid configurations and show that this also holds for L'', the latter resulting from iterating up to $i = y$. For $j \leq x$ or $j \geq y$ it holds $S'_j = S''_j$ and thus $|C \cap S'_j| \leq j$ iff $|C \cap S''_j| \leq j$. It remains to prove the equivalence for $x < j < y$. Assume that C is a valid configuration in L'. This means $|C \cap S'_x| \leq x < j$ and $S''_j = S''_{j-1} = \cdots = S''_x = S'_x$ resulting in $|C \cap S''_j| < j$.

Now let C be a valid configuration in L'' implying $|C \cap S''_x| \leq x$. We have $|C \cap S'_j| = |C \cap (S'_x \cup L_{x+1} \cup \cdots \cup L_j)| \leq x + (j - x) = j$. The last inequality results from $S'_x = S''_x$ and the fact that $L_{x+1}, \ldots, L_j$ are singletons. □

Given the compressing mechanism which shows how to construct $\mathcal{L}$ from L we adapt the update rule of L for $\mathcal{L}$. Let $p_1, \ldots, p_k$ be the first k pairwise distinct pages. We initially set $\mathcal{L}_k$ to the set of these k pages, $\mathcal{L}_0$ contains all other pages and the remaining layers are empty. The update rule of $\mathcal{L}$ is given in Theorem 1.

Theorem 1. *Let $\mathcal{L}$ and $\mathcal{L}^p$ be the compressed partition of L and L^p respectively. $\mathcal{L}^p$ can be obtained directly from $\mathcal{L}$ as follows:*

$$\mathcal{L}^p = \begin{cases} (\mathcal{L}_0 \setminus \{p\} | \mathcal{L}_1 | \ldots | \mathcal{L}_{k-2} | \mathcal{L}_{k-1} \cup \mathcal{L}_k | \{p\}), & \textit{if } p \in \mathcal{L}_0 \\ (\mathcal{L}_0 | \ldots | \mathcal{L}_{i-2} | \mathcal{L}_{i-1} \cup \mathcal{L}_i \setminus \{p\} | \mathcal{L}_{i+1} | \ldots | \emptyset | \mathcal{L}_k \cup \{p\}), & \textit{if } p \in \mathcal{L}_i, 0 < i < k \\ (\mathcal{L}_0 | \mathcal{L}_1 | \ldots | \mathcal{L}_{k-1} | \mathcal{L}_k), & \textit{if } p \in \mathcal{L}_k \end{cases}$$

Proof. Due to space limitations, the proof is available only in the full version.

3 Engineering an Implementation for RDM

In this section we first engineer a novel implementation for OnOPT algorithms in general and RDM in particular, with the goal of obtaining runtimes as fast as possible. We then give experimental results which support that our improved implementation not only significantly outperforms the original approaches from [6], but also is competitive with LRU and FIFO.

3.1 Implementation

Given the overwhelming amount of requests to revealed pages in practical inputs, our implementation mainly focuses on processing these as fast as possible. We first recall that RDM is an OnOPT algorithm which assigns to each requested page the priority $0.8t + 0.1(t - t_0)$, where t is the current timestamp and t_0 is the timestamp when the page lastly entered the support. Moreover, t is not increased upon revealed requests.

Throughout this section we denote by n the input size (the number of requests), by l the number of non-empty layers (at the current request time), and by m the page-set size (the number of pairwise distinct pages in the input).

Structure. We require that for each page p in the support the following information is stored: $p.t$ – the timestamp of the last request, $p.prio$ – the priority of the page, and any additional fields that might be required for computing the priority (e.g., in the case of RDM $p.t_0$ – the timestamp when p entered the support). To do so we use direct addressing, i.e. an array of page-set size where for each page the associated information is accessed by a look-up at the corresponding element. We note that an alternative implementation using a hash table has the advantage of using space proportional to the support size, but this increases the runtime via higher constant factors.

We first note that new layers are created only upon requests to Opt-miss pages, i.e. pages in $\mathcal{L}_0$. When this happens, we assign to the newly created layer a timestamp t equal to the current timestamp. This value is not modified while the layer is in support, i.e. until it is merged with $\mathcal{L}_0$.

Fact 1 *For each layer $\mathcal{L}_i$ having timestamp t and for any pages $p \in \mathcal{L}_i$ and $q \in \mathcal{L}_j$ with $j < i$, it holds that $t \leq p.t$ and $t > q.t$.*

Proof. By construction, for each i with $0 \leq i < k$ we have that the last request for any page in $\mathcal{L}_i$ is smaller than the last request of any page in $\mathcal{L}_{i+1}$. □

We store in a *layer structure* information only about the non-empty layers in the support. We do not store the empty layers – it suffices for each non-empty one to keep the number of empty layers preceding it. For each non-empty layer $\mathcal{L}_i$, we keep the following: the timestamp t, a value v which is at all times equal to $1+e_i$ where e_i is the number of consecutive empty layers preceding $\mathcal{L}_i$, and *mem* which stores the amount of pages in $\mathcal{L}_i$ that are in the cache, see e.g. Figure 2. We store these layers in an array $(l_1, \ldots, l_l)$ where l_i corresponds to the ith non-empty layer. By Fact 1, we have that $l_1.t < l_2.t < \cdots < l_l.t$. Therefore, identifying the layer that a certain page belongs to can be done using binary search with its last request as key. Also, layers can be inserted and deleted in $O(l)$ time. Finally, it supports a *find-layer-j* operation, which, given a layer index i, returns some layer l_j, with $j \geq i$ such that $|M \cap \mathcal{S}_j| = j$. This layer is identified as the first l_j with $j > i$, satisfying $\sum_{i=1}^{j} l_i.v = \sum_{i=1}^{j} l_i.mem$.

We note that, asymptotically, a search tree augmented with fields for prefix sum computations is much more efficient than an array. Nonetheless, we chose the array structure because of the particular characteristics of the layers: insertions are actually appends and take $O(1)$ time, few non-empty layers, and the constants involved are small.

Finally, we store the pages contained in the cache in an (unsorted) array of size k, where page replacements are done by overwriting.

Implementing OnOPT. We implement OnOPT algorithms using the structures described above.

If a page is revealed, no replacement is done because it is in cache. Moreover, no layer changes are required. A page is revealed iff its last request is greater than or equal to $l_l.t$. Therefore, processing a revealed page takes $O(1)$ time.

page	2	3	5	10	1	4	7	11	6	8	9
v	3				2				3		
mem	2				3				3		
	$\mathcal{L}_3$				$\mathcal{L}_5$				$\mathcal{L}_8$		

Fig. 2. Example for $\mathcal{L} = (\emptyset|\emptyset|2, 3, 5, 10|\emptyset|1, 4, 7, 11|\emptyset|\emptyset|6, 8, 9)$, emphasized pages are in cache $M = (2, 10, 1, 4, 11, 6, 8, 9)$. Pages are not stored in the layer structure.

If the requested page is an Opt-miss page, it is not in the cache and we first evict the page having the smallest priority. We identify the victim page by scanning the cache array for the minimum priority. Finally, we replace the selected page with the requested one. To update the layers, we first merge $\mathcal{L}_{k-1}$ and $\mathcal{L}_k$ as follows: if $l_l.v > 1$ then set $l_l.v = l_l.v - 1$ as $\mathcal{L}_{k-1}$ was empty; otherwise, i.e. $l_l.v = 1$, delete this layer. Afterwards, we simply append a new layer l_l with $l_l.t$ set to the current timestamp, $l_l.v = 1$, and $l_l.mem = 1$: there are no empty sets before the last layer and the new $\mathcal{L}_k$ has one element which is in memory. Altogether, processing an Opt-miss page takes $O(k)$ time.

It remains to deal with requests to unrevealed pages. If a cache miss occurs we first identify a page to evict as follows. We look the page's layer l_i up in the layer structure. Using the operation *find-layer-j*, we identify the layer l_j, and then by scanning the cache array find and evict the page with the smallest priority among the pages having last request strictly less than $l_{j+1}.t$. This ensures that the selected page is in the first j layers. To update the layers, we set $l_i.v = l_i.v - 1$ if $l_i.v > 1$ and delete l_i otherwise. This not only sets $\mathcal{L}_{i-1} = \mathcal{L}_{i-1} \cup \mathcal{L}_i$, but also ensures the necessary left shifts of the layers to the right. Finally, we set $l_l.v = l_l.v + 1$ to reflect a new empty layer before $\mathcal{L}_k$. After updating the last request for the requested page, it becomes revealed since this value is greater than $l_l.t$. Thus, processing an unrevealed page takes $O(l)$ time for a cache hit and $O(k)$ time for a cache miss.

Theorem 2. *Assuming m pairwise distinct pages are requested, a cache of size k, and l non-empty layers, our implementation uses $O(m)$ space and processes a revealed page in $O(1)$ time and an Opt-miss page in $O(k)$ time. Unrevealed pages take $O(l)$ time for cache hits and $O(k)$ time for cache misses.*

Corollary 1. *Assuming that a ratio of $1 - O(1/k)$ requests are to revealed pages, our implementation processes a request in $O(1)$ amortized time.*

3.2 Experimental Results

In this section we conduct experiments which demonstrate empirically that our implementation for OnOPT algorithms is competitive with both LRU and FIFO, which leads us to believe that algorithms in this class, and RDM in particular, are of practical interest.

Experimental Setup. For OnOPT algorithms, apart from the engineered version previously introduced we implemented the two versions described in [6]. The first one uses linked lists and processes a page in $O(|S|)$ time and the second uses a binary search tree which takes $O(\log|S|)$ time per page, where $|S|$ is the support size. Furthermore, for each of these implementations we also developed versions using the compressed layer partition. We also consider two implementations for LRU and one for FIFO. Similarly to the OnOPT implementations, we assume that for each page we associate $O(1)$ information which can be accessed in $O(1)$ time. This is done by direct addressing, i.e. we store an m-sized array where the ith entry stores data about page i. For LRU, the first implementation, denoted LRUList, uses a linked list storing the pages in cache sorted by their last request. Keeping for each cached page a pointer to the corresponding list element, a page request takes $O(1)$ time. The second implementation, LRULinear, uses an array of size k to store the cache contents. On a cache miss, the array is scanned to identify the page to evict. The first implementation treats a cache miss much faster than the second one but pays more time per cache hit to update the recency list. For FIFO, a circular array stores the FIFO queue.

All the experiments were conducted on all cache traces on a regular Linux computer having an Intel i7 hex-core CPU at 3.20 GHz, 10 GB of RAM, kernel version 3.1, and the sources were compiled using gcc version 4.5.3 with optimization -O3 enabled. For each data set and each cache size the runtimes were obtained as the median of five runs. Due to space limitations we show experimental results for only two cache traces, namely `go` and `winword`; the results for all traces are available in the full version.

Amount of Non-empty Layers. We first compare the amount of non-empty layers that we use in our implementation against the k layers used in the non-compressed one. In Figure 3 it is shown that typically both the maximum and the average number of layers are much smaller than k. As an extreme example, the `gnuplot` trace has a page-set of nearly 8000 pages, yet the maximum number of layers never exceeds 7, and the average is mostly between 2 and 3. This greatly reduces the runtime for updating the layers.

OnOPT Implementations. We compare the five OnOPT implementations using the RDM priority assignment, namely the one previously described and the two

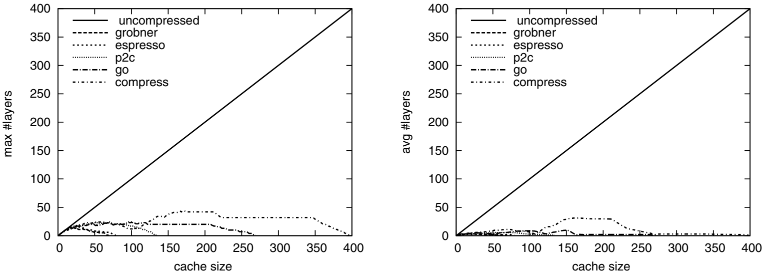


Fig. 3. Maximum (left) and average (right) number of non-empty layers in the compressed layer partition against the number of layers k in the uncompressed partition

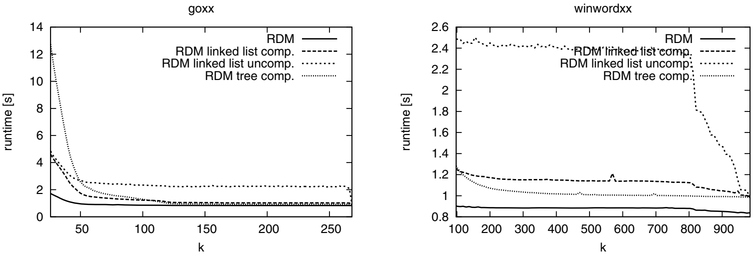


Fig. 4. The runtime for the various implementations of RDM on selected datasets

implementations in [6], each of them using the compressed and uncompressed layer partition. Surprisingly, both implementations using the binary trees were hopelessly slow, mainly because they require for each request, revealed or not, to update the path in the binary tree from the requested page to the root. Instead we use an approximation of RDM for the tree implementation using the compressed partition, where for revealed requests priorities do not change and this update is not necessary; the results shown are for this approximation. The runtime results for the two selected traces are given in Figure 4. As expected, our new implementation outperforms the previous ones, for small cache sizes by significant factors. Also, the implementations using the compressed layer partition significantly outperform their non-compressed counterparts.

OnOPT vs. LRU and FIFO. Having established that our new implementation is the fastest for OnOPT algorithms in general and RDM in particular, we compare it against FIFO and the two LRU implementations. The results in Figure 5 show that typically FIFO is the fastest algorithm while the LRUList is the slowest. While FIFO being the fastest is expected due to its processing pages in $O(1)$ time with very small constants, the fact that LRULinear outperforms LRUList despite its worst case of $O(k)$ time per page is explained by the overwhelming amount of cache hits (over the observed ratio of $1 - O(1/k)$ of revealed requests). For these requests, LRULinear only updates the last request for the requested page, whereas LRUList moves elements in the recency list which triggers higher constants in the runtime. Finally, we note that RDM typically is slower than LRULinear by small margins, which can be explained by the fact that both algorithms process revealed requests very fast and cache misses by scanning the memory; RDM has a slight overhead in runtime to update the layers and assign priorities. An interesting behavior is that for large cache sizes RDM is slightly faster than LRULinear, which we explain by a machine-specific optimization which does not write a value in a memory cell if the cell already stores the given value. Essentially, LRULinear always updates the last request for the current page, while RDM does not increase the timer for revealed requests meaning that no data associated with pages changes if many consecutive revealed requests occur. This typically happens for large cache sizes.

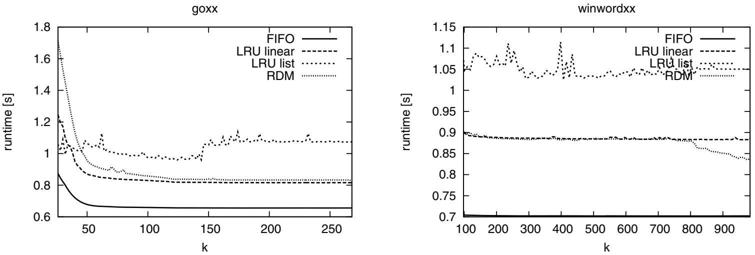


Fig. 5. The runtime for LRU, FIFO, and our RDM implementation

Misses with time penalty. We simulate a scenario where each cache miss inflicts a time penalty. We choose a typical cost of 9ms [15, Chapter 1.3.3] per cache miss. Again, we compare RDM against LRU and FIFO, however the runtime of the algorithm will be given by its actual runtime plus the penalty of 9ms for each miss, i.e. $total = runtime + \#misses \cdot 9ms$. Moreover, for both LRU and FIFO we set the runtime to zero, so they only pay the penalty for cache misses. In this scenario the total cost is dominated by this penalty. The results in Figure 6 show that despite the zero runtime for LRU and FIFO, RDM still outperforms them on many cache sizes; in general, these results hold for the other thirteen cache traces as well. This is because the runtime component for RDM is about one second, which corresponds to about 100 misses. Given that typically RDM outperforms LRU and FIFO by more than 100 misses, it becomes the best algorithm for most datasets if the cache size is not too large; for large cache sizes algorithms incur significantly fewer misses and the runtime component becomes more important.

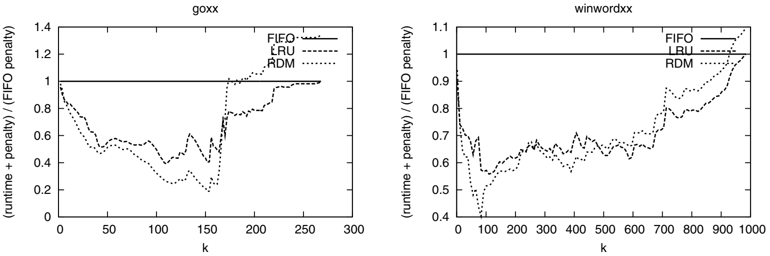


Fig. 6. RDM and LRU compared to FIFO when a cache miss costs 9ms

References

1. Achlioptas, D., Chrobak, M., Noga, J.: Competitive analysis of randomized paging algorithms. Theoretical Computer Science 234(1-2), 203–218 (2000)
2. Bein, W.W., Larmore, L.L., Noga, J., Reischuk, R.: Knowledge state algorithms. Algorithmica 60(3), 653–678 (2011)

3. Belady, L.A.: A study of replacement algorithms for virtual-storage computer. IBM Systems Journal 5(2), 78–101 (1966)
4. Borodin, A., Irani, S., Raghavan, P., Schieber, B.: Competitive paging with locality of reference. Journal of Computer and System Sciences 50(2), 244–258 (1995)
5. Boyar, J., Favrholdt, L.M., Larsen, K.S.: The relative worst-order ratio applied to paging. Journal of Computer and System Sciences 73(5), 818–843 (2007)
6. Brodal, G.S., Moruz, G., Negoescu, A.: OnlineMIN: A fast strongly competitive randomized paging algorithm. In: Proc. 9th Workshop on Approximation and Online Algorithms, pp. 164–175 (2011)
7. Fiat, A., Karp, R.M., Luby, M., McGeoch, L.A., Sleator, D.D., Young, N.E.: Competitive paging algorithms. Journal of Algorithms 12(4), 685–699 (1991)
8. Fiat, A., Rosen, Z.: Experimental studies of access graph based heuristics: Beating the LRU standard? In: Proc. 8th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 63–72 (1997)
9. Kaplan, S.F., Smaragdakis, Y., Wilson, P.R.: Flexible reference trace reduction for VM simulations. ACM Transactions on Modeling and Computer Simulation 13(1), 1–38 (2003)
10. Karlin, A.R., Manasse, M.S., Rudolph, L., Sleator, D.D.: Competitive snoopy caching. Algorithmica 3, 77–119 (1988)
11. Koutsoupias, E., Papadimitriou, C.H.: Beyond competitive analysis. SIAM Journal on Computing 30, 300–317 (2000)
12. McGeoch, L.A., Sleator, D.D.: A strongly competitive randomized paging algorithm. Algorithmica 6(6), 816–825 (1991)
13. Moruz, G., Negoescu, A.: Outperforming LRU via competitive analysis on paramtrized inputs for paging. In: Proc. 23rd Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 1669–1680 (2012)
14. Sleator, D.D., Tarjan, R.E.: Amortized efficiency of list update and paging rules. Communications of the ACM 28(2), 202–208 (1985)
15. Tanenbaum, A.S.: Modern operating systems, 3rd edn. Pearson Education (2008)
16. Young, N.E.: The k-server dual and loose competitiveness for paging. Algorithmica 11(6), 525–541 (1994)

Feasibility Pump Heuristics for Column Generation Approaches

Pierre Pesneau, Ruslan Sadykov, and François Vanderbeck

Univ. of Bordeaux, INRIA team Realopt

Abstract. Primal heuristics have become an essential component in mixed integer programming (MIP). Generic heuristic paradigms of the literature remain to be extended to the context of a column generation solution approach. As the Dantzig-Wolfe reformulation is typically tighter than the original compact formulation, techniques based on rounding its linear programming solution have better chance to yield good primal solutions. However, the dynamic generation of variables requires specific adaptation of heuristic paradigms. We focus here on "feasibility pump" approaches. We show how such methods can be implemented in a context of dynamically defined variables, and we report on numerically testing "feasibility pump" for cutting stock and generalized assignment problems.

Keywords: Primal Heuristic, Dantzig-Wolfe Decomposition, Branch-and-Price Algorithms.

1 Introduction

Heuristics are algorithms that attempt to derive "good" primal feasible solutions to a combinatorial optimization problem. They include constructive methods that build a solution and iterative improvement methods such as local search procedure that starts with an incumbent. The term "primal heuristic" generally refers to methods based on the tools of exact optimization, truncating an exact procedure or constructing solutions from the relaxation on which the exact approach relies: techniques range from greedy constructive procedures to rounding a solution of the linear programming (LP) relaxation, using the LP solution to define a target, or simply exploiting dual information to price choices. Today's MIP solvers rely heavily on generic primal heuristics: progress in primal heuristics is quoted as one of the main source of commercial solver performance enhancement in the last decade [6]. High quality primal values help pruning the enumeration by bound and by preprocessing (constraint propagation). They are also essential in tackling large scale real-life applications where the exact solver is given limited running time and a realistic ambition is to obtain good primal solutions.

Heuristics based on exact methods have found a new breath in the recent literature. The latest developments are reviewed in [5,12]. Here we focus on the so-called *feasibility pump algorithm* [1,11]. The method entails rounding the solution of the Linear Programming (LP) relaxation to the closest integer. It

R. Klasing (Ed.): SEA 2012, LNCS 7276, pp. 332–343, 2012.

might lead to an infeasible integer solution. But it defines a target for integer optimization, i.e., the LP is re-optimized with the objective of minimizing both the distance to that target and the original objective value. And the process iterates. Recently this paradigm has been extended from the context of binary integer programs to general integer programs [4,13]. Our purpose is to examine possible extensions of feasibility pump to the case where one works with a column generation approach for the Dantzig-Wolfe reformulation of the problem.

The column generation literature reports many application specific studies where primal heuristics are a key to success: some heuristics have been implemented either by taking decision in the space of the original compact formulation (using a so-called robust approach), others involve decision directly in the space of the Dantzig-Wolfe reformulation. Here, we focus on the latter direct approach that fully exploits the specificity of Dantzig-Wolfe decomposition: the price coordination mechanism that brings a global view and the more aggregate decisions that enable rapid progress in building a primal solution (fixing variables of the Dantzig-Wolfe reformulation has a much stronger impact than when fixing variables of the original formulation). Our aim is to extract generic methods for use in branch-and-price algorithms.

Making heuristic decision on the variables of the Dantzig-Wolfe (DW) reformulation requires particular attention to derive heuristics that are “compatible” with the pricing problem solver as highligthed herein. Otherwise, it may impair the effectiveness of the column generation approach that relied on the existence of an efficient pricing oracle. Hence, the generic primal heuristic paradigm must be adapted if one works with the column generation formulation. Alternatively, if one makes heuristic decision on the variables of the original compact formulation, then the generic primal heuristics for mixed integer programming apply directly. However, projecting the master solution in the original variable space is not always straightforward. First, the projection is not defined in the common case of identical subsystems. Second, because a modified original formulation (as induced by the primal heuristic) calls for a modified DW reformulation with possibly the same issues as when implementing the heuristic in the DW reformulation.

The rest of the paper is organized as follows. In Section 2, we review standard primal heuristics techniques based on mathematical programming and more specifically the feasibility pump heuristic. We then discuss in Section 3 the specific difficulties in implementing primal heuristics in a column generation context and we propose a simple strategy to get around these technical issues. In Section 4, we develop a general purpose feasibility pump heuristic for use in a column generation approach. This generic method is evaluated experimentally as reported in Section 5. In our conclusion, we analyze the results and discuss further work.

2 Primal Heuristics for MIP and Feasibility Pump

Assume a bounded integer program with n variables:

$$z := \min\{c\,x : A\,x \geq a,\ x \in I\!N^n\}\ .$$

Let x^{LP} be a solution to its Linear Programming (LP) relaxation, and let z^{LP} be its value. A "rounding heuristic" consists in iteratively selecting a fractional component of x^{LP} that must take integer value and fix it to an integer value or simply reset the lower (resp. upper) bound on this component. The method is defined by a rule for selecting the component and its fixed value or bound (such as *least fractional* or *guided search* among the rules reviewed in [5]). "Diving heuristics" differ from simple rounding heuristics by the fact that the LP is re-optimized after bounding or fixing one (or several) variables. Diving can be seen as a heuristic search in a LP-based branch-and-bound tree: the search plunges depth into the enumeration tree by selecting a branch heuristically at each node.

The *feasibility pump* heuristic can be seen as specific iterative rounding procedure [1,11]. It was originally developed for a binary integer program:

$$\min\{c\,x : A\,x \geq a,\ x \in \{0,1\}^n\}\ .$$

Feasibility pump entails exploring a sequence of trial solutions, $\tilde{x}$, obtained by rounding to the closed integer solution to the LP solution, x^{LP}, of a program with modified objective function. If the rounded trial integer solution is feasible the algorithm stops with this primal candidate. Otherwise, the rounded solution serves as a target to which one aims to minimize some distance measure. Assuming that the new objective function combines the original objective with the L_1 norm to the target solution, the modified problem takes the form:

$$\min\{c\,x + \epsilon(\sum_{j\in J^0} x_j + \sum_{j \in J^1}(1-x_j)) : A\,x \geq a,\ x\in[0,1]^n\}\ . \tag{1}$$

The index sets $J^0 \subset J$ and $J^1 \subset J$ form a partition of $J = \{1,\ldots,n\}$. They define respectively the components that take value 0 or 1 in the previous trial integer solution, $\tilde{x}$, obtained through rounding the LP solution of (1). Parameter ϵ controls the impact of the cost modifications relative to the original objective. Extending feasibility pump to a general integer program requires an adapted distance measure [4,13]. Assuming bounded integer variables $x_j \in I\!N$, with $l_j \leq x_j \leq u_j,\ j \in J$, the modified problem could take the form:

$$\min\{c\,x + \epsilon(\sum_{j:\tilde{x}_j = l_j}(x_j - l_j) + \sum_{j:\tilde{x}_j = u_j}(u_j - x_j) + \sum_{j:l_j<\tilde{x}_j<u_j} d_j) : A\,x \geq a, \tag{2}$$

$$d_j - \tilde{x}_j \leq x_j \leq d_j + \tilde{x}_j\ \forall j,\ x \in I\!R^n\ \ \} \tag{3}$$

where (3) are additional constraints needed to define auxiliary variables d_j.

3 Primal Heuristics Combined with Column Generation

In the context of a column generation approach, we assume a mixed integer program with decomposable structure (a subset of constraints is assumed to have a *block diagonal* structure):

$$[P]\quad \min\{cx : Ay \geq a, y = \sum_k x^k, \underbrace{B^k x^k \geq b^k\ \forall k,\ x^k \in I\!N^n}_{x^k\in X^k}\} \tag{4}$$

where $Ay \geq a$ represent "complicating constraints", while optimization over subsystems X^k defined by subsystem $B^k x^k \geq b^k$ is "relatively easy" (i.e., optimization over X^k is assumed to be tractable). In the sequel, to simplify the presentation, subsystems X^k are assumed to be bounded pure integer programs and G^k is an enumeration of the feasible solutions to X^k, i.e., $X^k = \{x^g\}_{g \in G^k}$. The structure of $[P]$ can be exploited to reformulate it as the master program:

$$\min\{ \sum_{k, g \in G^k} (cx^g)\lambda_g^k : \sum_{k, g \in G^k} (Ax^g)\lambda_g^k \geq a; \sum_{g \in G^k} \lambda_g^k = 1 \; \forall k; \; \lambda_g^k \in \{0,1\} \; \forall g, k\} \; . \tag{5}$$

When each block is identical, as is the case in many applications, let $Bx \geq b$ be the constraint set defining one block subsystem $X^k = X = \{Bx \geq b, x \in I\!N^n\} \; \forall k$, and the master takes the form:

$$[\mathrm{M}] \equiv \min\{\sum_{g \in G}(cx^g)\lambda_g : \sum_{g \in G}(Ax^g)\lambda_g \geq a; \; \sum_{g \in G} \lambda_g = K; \; \lambda_g \in I\!N \; \forall g\} \tag{6}$$

where $\lambda_g = \sum_k \lambda_g^k$, G is the set of *generators* of X, and K is the number of identical blocks. In the sequel, we assume identical subproblems and hence master formulation (6) unless specified otherwise. The master program $[M]$ is solved by Branch-and-Price: at each node of the Branch-and-Bound tree the linear relaxation of $[M]$ is solved by column generation. The reduced cost of a column $g \in G$ takes the form $(c - \pi A)\, x^g - \sigma$, where (π, σ) are the dual solution associated with constraints $Ax \geq a$ in $[M]$ and $\sum_{g \in G} \lambda_g = K$ respectively. Thus, the pricing problem is of the form: $\min\{(c - \pi A)\, x \; : \; Bx \geq b, x \in I\!N^n\}$.

The most commonly used generic primal heuristic in this column generation context is the so-called *restricted master heuristic*. The column generation formulation is restricted to a subset of generators $\overline{G}$ and associated variables, and it is solved as a static IP. The main drawback of this approach is that the resulting restricted master integer problem is often infeasible (the columns of $\overline{G}$ – typically defined by the LP solution – may not be combined to an integer solution). In an application specific context, an ad-hoc recourse can be designed to "repair" infeasibility. Such implementation has been developed for network design [8], vehicle routing [2,9,20] and delivery [19] problems. Alternatively, one can use the master LP solution as a base for column selection giving rise to a *rounding heuristics*. Such heuristics (sometimes coupled with local search) have been successfully applied [3,7,10,15] (on cutting stock, planning, and vehicle routing problems). However, reaching feasibility remains a difficult issue that is often handled in an application specific manner. Diving heuristics are generic ways of 'repairing" infeasibility as we highlight it in [14].

Deriving primal heuristics based on rounding the Dantzig-Wolfe reformulation LP solution raises some difficulties. Bounding a master variable or modifying its cost can be incompatible with the column generation approach in the sense that it can induce modifications to the subproblem that are not amenable to the pricing oracle. Depending on the application, the oracle may or may not still be applicable after some modifications to the problem structure. In the context of this paper, our aim is to derive a generic algorithm that applies without any required modifications to the pricing subproblem formulation. To achieve this goal, we restrict our problem modifications to operations that are easy-to-implement in the master and do not induce subproblem modifications: namely master variable lower bound setting and cost reduction.

Indeed, setting an upper bound on an existing column, i.e., enforcing $\lambda_g \leq u_g$, or a slightly more general constraint of the form (3) in the master, yields an associated dual variable that must be accounted for in pricing (by modeling an extra cost for a specific solution x^g). Alternatively, one must restrict the pricing problem to avoid regenerating x^g; indeed, if $\lambda_g \leq u_g$ is ignored, the column x^g might otherwise be wrongly regenerated as the best price solution. Both approaches induce significant modifications to the pricing procedure. However, setting a lower bound on an existing column of the form $\lambda_g \geq l_g$ is trivial: this constraint can safely be ignored when pricing. Indeed, ignoring the dual price "reward" for generating this column, means that the pricing oracle overestimates its reduced cost and might not generate it; but the column needs not be generated since it is already in the master. Similarly, we cannot increase the cost c_g of a variable λ_g beyond its true cost, because then λ_g will price out negatively according to the original pricing oracle, and hence it can be regenerated as the best subproblem solution, unless we modify the pricing problem to avoid this. On the other hand, decreasing the cost c_g of a variable λ_g is amenable to the unmodified column generation scheme, as the pricing oracle shall simply overestimate the reduced cost of such already included column.

An alternative approach is to perform the feasibility pump problem modifications in the space of the original formulation (4). Having solved the LP relaxation of (5), one can project its solution in the original space, defining

$$x^k = \sum_{g \in G^k} x^g \lambda_g^k \,, \tag{7}$$

and apply the primal heuristic procedures of Section 2 on this projected solution. There remains a key issue however. The projection is not uniquely defined in the case where there are K identical subproblems. Then, the LP solution of the aggregate master (6) can be disaggregated using for instance:

$$\lambda_g^k = \min\{1, \lambda_g - \sum_{\kappa=1}^{k-1} \lambda_g^k, (k - \sum_{\gamma \prec g} \lambda_\gamma)^+\} \quad \forall k = 1, \ldots, K, g \in G \,, \tag{8}$$

where $\prec$ defines a lexicographic ordering of columns $g \in G$ (see [23] for details). With such disaggregated λ_g^k values, one can use projection (7), but the reverse relation cannot be properly enforced, i.e., a restriction on x_j^k variables cannot be modeled in (6).

It is also important to observe that acting on the variables of the original formulation is a completely different decision space than acting on the DW reformulation variables. In particular, variable bounding or cost modification decisions have a more macroscopic effect when done in the DW reformulation. This can be both an advantage (faster progress to a integer solution) and a drawback (of quickly painting yourself in a corner). This study focus on primal heuristics for the DW reformulation because in many applications making more aggregate fixing is more an advantage than a drawback and secondly because that is where the need for innovation lies (primal heuristics in the original formulation can implemented as defined in the standard paradigm for MIPs).

From the above discussion, we conclude that to implement the feasibility pump paradigm, we shall restrict the method to using lower bound setting and cost reduction. Our algorithm shall combine rounding and diving paradigm: defining lower bound on master variables is implemented by defining a partial solution. Indeed, rounding down variable λ_g amounts to taking $\lfloor \lambda_g \rfloor$ copies of this column in the partial solution. The *residual master problem* that remains once the partial solution, denoted $\tilde{\lambda}$, is extracted takes the form:

$$[RM] \equiv \min\{\sum_{g\in G}(cx^g)\lambda_g : \sum_{g\in G}(Ax^g)\lambda_g \geq \tilde{a};\ \sum_{g\in G}\lambda_g = \tilde{K};\ \lambda_g \in I\!N \quad \forall g\} \qquad (9)$$

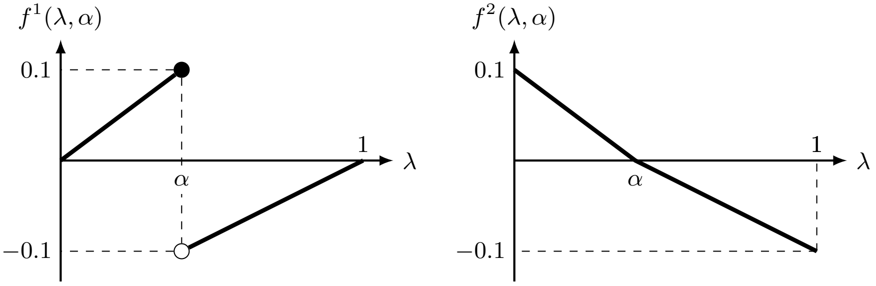
where $\tilde{a} = a - \sum_g (Ax^g)\tilde{\lambda}_g$ and $\tilde{K} = K - \sum_g \tilde{\lambda}_g$. The columns that are part of the partial solution remain in the residual problem for further selection if suitable. Thus, the master variable lower bounds are implemented implicitly by the definition of the partial solution.

A key feature in this primal heuristic is preprocessing: it is important to "cleanup" the residual problem after fixing a partial solution, deleting all columns that could not be part of an integer solution to the residual problem (and hence would lead to infeasibility if selected). Thus, preprocessing helps to avoid the primal heuristic dead-ending with a unfeasible solution. In this context, so-called "proper columns" are columns that could take integer value in an optimal solution to the residual master problem [22]. If the oracle assumes a bounded subproblem, one can tighten lower and upper bounds on subproblem variables to generate proper columns. In that case, we refine the bounds on subproblem variables by constraint propagation after fixing a partial master solution.

Observe that, as the *residual master problem* (9) that remains after a rounding operation might be modified by preprocessing, its re-optimization might not necessarily be trivial and it can lead to generating new columns. This mechanism yields the "missing" complementary columns to build feasible solutions. If the residual master is however infeasible for a given partial solution, re-optimization can be a way to prove it early through a Simplex phase 1 and/or preprocessing. Re-optimization of the master LP after fixing, however important feature for the success of the approach, can be time consuming. Tuning the level of approximation in this re-optimization allows one to control the computational effort.

4 A Generic Feasibility-Pump Algorithm

In Table 1, we propose a generic heuristic for use in a column generation context that exploits the main ideas of the feasibility pump paradigm. A target solution $\tilde{\lambda}$ is defined by rounding each component of the LP solution of the master (6) to the closest integer. If $\tilde{\lambda}$ defines a feasible solution to (6), we stop. Otherwise we use it as a target point. To induce a new master LP solution "closer" to target solution $\tilde{\lambda}$, we decrease the cost of columns that were rounded-up to define $\tilde{\lambda}$ and increase the cost of those that were rounded down. However, we do not increase column costs beyond their original value; otherwise, this would induce the regeneration of the same column at its initial cost. The two cost modification factor functions that we considered are presented in Figure 1. The first one is a direct adaptation of the cost modification arising in (1). The second is a complementary variant that aims at stabilizing the part of the solution that is currently integer, using the cost modification to make the current integer solution even more attractive.



$$f^1(\lambda,\alpha) = \begin{cases} 0.1\ \frac{\lambda}{\alpha} & \text{if } \lambda \le \alpha \\ -0.1\ \frac{(1-\lambda)}{(1-\alpha)} & \text{if } \lambda > \alpha \end{cases} \qquad f^2(\lambda,\alpha) = \begin{cases} 0.1\ (1-\frac{\lambda}{\alpha}) & \text{if } \lambda \le \alpha \\ -0.1\ \frac{(\lambda-\alpha)}{(1-\alpha)} & \text{if } \lambda > \alpha \end{cases}$$

Fig. 1. Two cost modification factor functions of the form $f(\lambda,\alpha)$ where $\lambda \in [0,1]$ is the factional part of a column value in the master LP solution and $\alpha \in (0,1)$ is a fractionality threshold parameter

At iteration t, the modified master program becomes:

$$[M^t] \equiv \min\{\sum_{g\in G^t} c_g^t \lambda_g : \sum_{g\in G^t} (Ax^g)\lambda_g \ge a^t;\ \sum_{g\in G^t} \lambda_g = K^t;\ \lambda_g \in I\!N \quad \forall g \in G^t\} \tag{10}$$

where G^t, a^t, K^t, and c_g^t are updated in the course of the algorithm. Initially, G^0 is the set of columns generated in the course of solving the linear relaxation of the master program, [M] given in (6), by column generation, $a^0 = a$, $K^0 = K$, and $c_g^0 = cx^g$ for all $g \in G^0$. Then, at iteration t, we compute the LP solution λ^t to (10), and its rounded value $\tilde{\lambda}$. If the later is not feasible for (6), we iterate the feasibility pump procedure. In our implementation, we define an initial partial solution by rounding down the LP solution to the master (6) before going into the feasibility pump heuristic (see *Step 2* in Table 1). In this way, our residual master program is close to a 0 - 1 problem and we omit the subtleties required to handle general integer problems. Finally, observe that the algorithm cycles with the same master LP solution if no columns are rounded up in the target solution and hence no column cost are decreased. To avoid this situation, we then

Table 1. Feasibility Pump heuristic. Let G^f denote the current set of fractional value columns, i.e., $G^f = \{g \in G^t : \lfloor \lambda_g^t \rfloor < \lambda_g^t < \lceil \lambda_g^t \rceil\})$, while $\tilde{\lambda}_g$ denotes the rounded value of λ_g^t, which can be obtained by rounded up or down the current LP value. $\hat{\lambda}$ denotes the current partial solution, while $f(.)$ is one of the function of Figure 1.

Step 1: Solve the LP relaxation of the master (6) by column generation. Set the iteration counter $t \leftarrow 0$. Initialize the column set G^0 to the columns generated in solving the master LP, the right-hand-sides to $a^0 \leftarrow a$ and $K^0 \leftarrow K$. Let λ^0 be the master LP solution. Initialize the master column current costs, $c_g^0 \leftarrow c_g$, and the current partial solution, $\hat{\lambda}_g \leftarrow 0$, for all $g \in G^0$. The fractionality threshold is initially set to $\alpha \leftarrow 0.5$.

Step 2: Fix a partial IP solution: $\hat{\lambda}_g \leftarrow \hat{\lambda}_g + \lfloor \lambda_g^t \rfloor$ for all $g \in G^t$. Reset $\tilde{\lambda} \leftarrow 0$; then set $\tilde{\lambda}_g \leftarrow \lfloor \lambda_g^t \rfloor$ and reset $\lambda_g^t \leftarrow \lambda_g^t - \lfloor \lambda_g^t \rfloor$.

Step 3: Update the master constraint right-hand-sides: $a^t \leftarrow a^t - \sum_{g \in G^t} Ax^g \tilde{\lambda}_g$, $K^t \leftarrow K^t - \sum_{g \in G^t} \tilde{\lambda}_g$. Apply preprocessing to the master, deleting non-proper columns from G^t and possibly update the pricing problems (possibly setting new bounds on subproblem variables in one aims to generate only proper columns). Using column generation, re-optimize the linear relaxation of the residual master program (10), possibly adding columns to G^t. If residual master problem is shown infeasible through preprocessing or Phase 1 of the Simplex algorithm, STOP. Else, reset λ^t to the current master LP solution and update G^f.

Step 4: Define the solution rounded to the closest integer: reset $\tilde{\lambda} \leftarrow 0$; then set $\tilde{\lambda}_g \leftarrow \lfloor \lambda_g^t \rfloor$ if $\lambda_g^t \leq \alpha$ and $\tilde{\lambda}_g = \lceil \lambda_g^t \rceil$ otherwise, for all $g \in G^t$.

Step 5: If no columns were rounded up, try to reset the fractionality threshold α. I.e., let $\alpha_{\min} = \min_{g \in G^f} \{\lambda_g^t - \lfloor \lambda_g^t \rfloor\}$ and $\alpha_{\max} = \max_{g \in G^f} \{\lambda_g^t - \lfloor \lambda_g^t \rfloor\}$. If $\alpha_{\min} < \alpha_{\max}$, decrease the fractionality threshold: $\alpha \leftarrow \frac{\alpha_{\min} + \alpha_{\max}}{2}$ and return to step 4. Otherwise, if $\alpha_{\min} = \alpha_{\max}$, apply a diversification step by fixing a column from $g \in G^f$ to value $\tilde{\lambda}_g = \lceil \lambda_g^t \rceil$ and go-to *Step 3*.

Step 6: If $(\hat{\lambda} + \tilde{\lambda})$ defines a complete primal solution, i.e., is a feasible integer solution to (6), record this solution and STOP.

Step 7: Define the updated column costs using rule: $c_g^{t+1} \leftarrow c_g^t$ for $g \in G^t \setminus G^f$ and

$$c_g^{t+1} \leftarrow \min\{c_g, f(\lambda_g^t - \lfloor \lambda_g^t \rfloor, \alpha)\, c_g^t\} \quad \forall g \in G^f \ .$$

Step 8: Let $t \leftarrow t + 1$ and re-optimize the linear relaxation of the residual master program (10) with modified cost using column generation; record its solution λ^t and update G^f.

Step 9: If the maximum number, T, of feasibility pump iterations has been reached, go-to *Step 2*.

Step 10: Go-to *Step 3*.

decrease the fractionality threshold parameter (see *Step 5* in Table 1), unless all fractional columns have the same fractionality. In the later case, we arbitrarily fix a column to its rounded up value to induce a diversification.

5 Computational Results

We built the above feasibility pump heuristic into *BaPCod* [21], a generic Branch-and-Price Code that we developed. We tested these procedures on standard models: the Cutting Stock Problem (CSP) and the Generalized Assignment Problem (GAP). For each model, we present average computational results on random instances similar to those of the literature. We tried both cost modification factor functions presented in Figure 1. In the numerical result tables, column "found" gives the fraction of instances for which the feasibility pump heuristic found a feasible primal solution. Tables report either the number of instances solved to optimality or the "gap" computed as the difference between the solution found and the column generation lower bound in per cent from the latter. This statistic is the average among instances for which a feasible solution has been found. Column "time" gives the average heuristic execution time in seconds over all instances, including those for which the heuristic fails to find a primal solution (those require typically larger computing time). For comparison, we present the performance of the diving heuristic presented in [14]: after fixing the lower integer part of the master solution, this heuristic rounds one column value selected according to the least fractional criteria, re-optimize the residual master, and reiterates until either finding a feasible solution or reaching infeasibility.

The results for the Cutting Stock Problem (CSP) are given in Table 2 for instances with 50 and 100 items respectively. For the CSP, the *master linear program* is:

$$\min\left\{\sum_{g\in G}\lambda_g : \sum_{g\in G} x_i^g\lambda_g \geq d_i \ \forall i; \quad \sum_{g\in G}\lambda_g \leq K; \quad \lambda_g \geq 0 \ \forall g\right\} \tag{11}$$

where G is the set of feasible solutions to the *pricing subproblem*:

$$\max\left\{\sum_i \pi_i x_i : \sum_i w_i x_i \leq W, \quad x_i \in I\!N, \ i = 1,\ldots,n\right\}$$

We solve the latter using the integer knapsack solver of Pisinger [17]. Note that for this application, the residual master problem is always feasible as one can use as many patterns as needed. Fifty random instances are generated using uniform distributions: $W = 10000$, $w_i \in U[500, 2500]$, $d_i \in U[1, 50]$. For these tests, the primal solution is always equal to the root dual bound or one unit above.

In the Generalized Assignment Problem (GAP), one searches for a minimum cost assignment of a set of jobs indexed by j to a set of machines indexed by m with limited capacity. The master linear program is:

Table 2. Results for Cutting Stock instances

n	max d_i	function	found	opt	gap	time
50	50	f^1	50/50	45/50	0.05	6.14
50	50	f^2	50/50	41/50	0.09	4.82
50	50	Pure Div	50/50	43/50	0.07	1.17
100	50	f^1	50/50	43/50	0.04	23.93
100	50	f^2	50/50	40/50	0.05	17.98
100	50	Pure Div	50/50	35/50	0.08	4.08

$$\min \left\{ \sum_{m,g \in G^m} c_g \lambda_g : \sum_{m,g \in G^m} x_j^g \lambda_g = 1 \, \forall j \in J; \; \sum_{g \in G^m} \lambda_g \leq 1 \, \forall m \in M; \; \lambda_g \geq 0 \, \forall g \right\},$$

where G^m is the set of feasible assignments to machine m solving the $0-1$ knapsack problem:

$$\max \left\{ \sum_{j \in J} (\pi_j - c_j^m) \, x_j^m : \sum_{j \in J} p_{jm} x_j^m \leq w_m; \quad x_j^m \in \{0,1\} \, \forall j \in J \right\} .$$

We solve the latter using the code of Pisinger [16]. Random instances were generated in the same way as the hard instances of type D in [18]. The results are shown in Table 3.

Table 3. Average results for 50 instances of type D for the Generalized Assignment Problem

m	n	function	found	gap	time
10	50	Pur Div.	34/50	1.00%	0.37
10	50	f^1	36/50	0.98%	1.81
10	50	f^2	48/50	1.14%	0.81
20	100	Pur Div.	35/50	0.65%	2.46
20	100	f^1	36/50	0.55%	14.56
20	100	f^2	42/50	0.75%	5.92

6 Conclusion

Our study demonstrates that the feasibility pump primal heuristics paradigm can be successfully extended to a column generation context. The key to such extension is to restrict problem modifications to those that are compatible with the column generation procedure: our implementation relies only on a cost reduction mechanism and implicitly on setting lower bound on master variables. As in the case for diving heuristics, the variable fixing done in our implementation of the feasibility pump heuristic can lead to a dead-end with an unfeasible residual master program. Compared to our previous experience using a pure diving heuristic, as reported in [14], our numerical preliminary experiment shows that feasibility pump leads to more feasible primal solutions and relatively good solutions. The diversification mechanisms that we experimented for diving heuristics in [14] (i.e., a limited backtracking when the heuristic dead-ends) could be adapted for feasibility pump. In future work, we also intend to test the approach on a larger scope of applications and to compare feasibility pump implementations in the master program, versus implementing cost modification in the space of the compact formulation, when this is feasible, i.e., in case such as the GAP where there are no multiple identical subproblems.

References

1. Achterberg, T., Berthold, T.: Improving the feasibility pump. Discrete Optim. 4(1), 77–86 (2007)
2. Agarwal, Y., Mathur, K., Salkin, H.M.: A set-partitioning-based exact algorithm for the vehicle routing problem. Networks 19(7), 731–749 (1989)
3. Belov, G., Scheithauer, G.: A cutting plane algorithm for the one-dimensional cutting stock problem with multiple stock lengths. European J. Oper. Res. 141(2), 274–294 (2002)
4. Bertacco, L., Fischetti, M., Lodi, A.: A feasibility pump heuristic for general mixed-integer problems. Discrete Optimization 4(1), 63–76 (2007)
5. Berthold, T.: Primal Heuristics for Mixed Integer Programs. Master's thesis, Technische Universität Berlin (2006)
6. Bixby, B.: Presentation of the gurobi optimizer. In: Integer Programming Down Under: Theory, Algorithms and Applications, Workshop at Newcastle NSW (2011)
7. Ceselli, A., Righini, G., Salani, M.: A column generation algorithm for a vehicle routing problem with economies of scale and additional constraints. In: Proceedings TRISTAN, Phuket, Thailand (June 2007)
8. Chabrier, A.: Heuristic branch-and-price-and-cut to solve a network design problem. In: Proceedings CPAIOR, Montreal, Canada (May 2003)
9. Chabrier, A., Danna, E., Le Pape, C.: Coopération entre génération de colonnes et recherche locale appliquées au problème de routage de véhicules. In: Huitièmes Journées Nationales sur la résolution de Problèmes NP-Complets (JNPC), Nice, France, pp. 83–97 (May 2002)
10. Dobson, G.: Worst-case analysis of greedy heuristics for integer programming with nonnegative data. Math. Oper. Res. 7(4), 515–531 (1982)
11. Fischetti, M., Glover, F., Lodi, A.: The feasibility pump. Math. Program. 104(1, ser. A), 91–104 (2005)
12. Fischetti, M., Lodi, A.: Heuristics in Mixed Integer Programming. Wiley Encyclopedia of Operations Research and Management Science, J.J. Cochran Edt., vol. 3, pp. 2199–2204. Wiley (2011)
13. Fischetti, M., Salvagnin, D.: Feasibility pump 2.0. Mathematical Programming Computation 1, 201–222 (2009)
14. Joncour, C., Michel, S., Sadykov, R., Sverdlov, D., Vanderbeck, F.: Column generation based primal heuristics. Electronic Notes in Discrete Mathematics 36, 695–702 (2010)
15. Perrot, N.: Integer Programming Column Generation Strategies for the Cutting Stock Problem and its Variants. PhD thesis, Université Bordeaux 1, France (2005)
16. Pisinger, D.: A minimal algorithm for the 0-1 knapsack problem. Operations Research 45(5), 758–767 (1997)
17. Pisinger, D.: A minimal algorithm for the bounded knapsack problem. INFORMS Journal on Computing 12(1), 75–82 (2000)
18. Savelsbergh, M.: A branch-and-price algorithm for the generalized assignment problem. Operations Research 45(6), 831–841 (1997)
19. Schmid, V., Doerner, K.F., Hartl, R.F., Savelsbergh, M.W.P., Stoecher, W.: An effective heuristic for ready mixed concrete delivery. In: Proceedings TRISTAN, Phuket, Thailand (June 2007)
20. Taillard, É.D.: A heuristic column generation method for the heterogeneous fleet VRP. RO Oper. Res. 33(1), 1–14 (1999)

21. Vanderbeck, F.: Bapcod - a generic branch-and-price code (2008), http://wiki.bordeaux.inria.fr/realopt/
22. Vanderbeck, F., Savelsbergh, M.W.P.: A generic view of dantzig-wolfe decomposition in mixed integer programming. Operations Research Letters 34(3), 296–306 (2006)
23. Vanderbeck, F., Wolsey, L.: Reformulation and decomposition of integer programs. In: 50 Years of Integer Programming 1958-2008, pp. 431–502. Springer, Berlin Heidelberg (2010)

Exact Graph Search Algorithms for Generalized Traveling Salesman Path Problems

Michael N. Rice and Vassilis J. Tsotras

University of California, Riverside (UCR), Riverside, CA, USA
{mrice,tsotras}@cs.ucr.edu

Abstract. The Generalized Traveling Salesman Path Problem (GTSPP) involves finding the shortest path from a location s to a location t that passes through at least one location from each of a set of generalized location categories (e.g., gas stations, grocery stores). This NP-hard problem type has many applications in transportation and location-based services. We present two exact algorithms for solving GTSPP instances, which rely on a unique product-graph search formulation. Our exact algorithms are exponential only in the number of categories (not in the total number of locations) and do not require the explicit construction of a cost matrix between locations, thus allowing us to efficiently solve many real-world problems to optimality. Experimental analysis on the road network of North America demonstrates that we can optimally solve large-scale, practical GTSPP instances typically in a matter of seconds, depending on the overall number and sizes of the categories.

1 Introduction

Within the last decade, the growing online presence of geospatial information systems has made possible many novel applications in the fields of transportation and location-based services. Many massive, online location databases are now being made publicly available for mining spatial locations based on categorical points of interest, thus paving the way for highly-advanced navigation solutions.

As an example, consider a traveler in a new city for the first time. On their way to do some sightseeing at a local attraction, they wish to visit a coffee house, a gas station, and an ATM (in no particular order). However, there may be many such locations to choose from for each of these location types. As the traveler likely does not care exactly *which* gas station, ATM, or coffee house they visit (since each provides the same general type of service[1]), a desirable solution is then any path which visits one of each of these location types with the least overall detour on the way to the destination. Such a scenario is a common occurrence for everyday personal navigation needs, and also has many additional applications in transportation, in general.

In this paper, we establish an algorithmic framework for efficiently solving such problem types on large-scale, real-world road networks. In Section 2, we

[1] The locations are user-defined and may be made more specific; e.g., only consider gas stations of a certain brand.

R. Klasing (Ed.): SEA 2012, LNCS 7276, pp. 344–355, 2012.

formalize this problem as the Generalized Traveling Salesman Path Problem (GTSPP), and we discuss related work and our contributions. Section 3 presents the foundation for our work by formulating GTSPP as a unique graph search problem, including a standard search algorithm for this approach. Section 4 extends these ideas into a more advanced search algorithm, based around the Contraction Hierarchies preprocessing technique. We present an experimental analysis of these algorithms on the road network of North America in Section 5. We conclude the paper with future research in Section 6.

2 Generalized Traveling Salesman Path Problems

Let $G = (V, E, w)$ be a directed graph, with $n = |V|$, $m = |E|$, and edge weight function $w : E \rightarrow \mathbb{R}_+$. Let $P_{s,t} = \langle v_1, v_2, \ldots, v_q \rangle$ be a path in G from $s = v_1 \in V$ to $t = v_q \in V$. Let $w(P_{s,t}) = \sum_{1 \leq i < q} w(v_i, v_{i+1})$ be the total weight, or cost, of $P_{s,t}$. The minimum-weight, or "shortest", path cost from s to t is $d(s, t)$.

A category set, $C = \{C_1, C_2, \ldots, C_k\}$, defines a set of node subsets where, for $1 \leq i \leq k$, $C_i = \{c_{i,1}, c_{i,2}, \ldots, c_{i,|C_i|}\} \subseteq V$ represents a distinct category of locations. A path, $P_{s,t}$, *satisfies* a category set C if, for $1 \leq i \leq k$, $P_{s,t} \cap C_i \neq \emptyset$ (i.e., $P_{s,t}$ contains at least one node from each category). This is formally written as $P_{s,t} \models C$. For any GTSPP instance $\langle s, t, C \rangle$, having category count $k = |C|$ and category density $g = \underset{1 \leq i \leq k}{max} \{|C_i|\}$, an optimal solution is a path $P'_{s,t}$ in G such that $P'_{s,t} \models C$ and, $\forall\, P_{s,t}$ in G where $P_{s,t} \models C$, $w(P'_{s,t}) \leq w(P_{s,t})$. This optimal solution path is formally referred to as $P^C_{s,t}$.

Related Work. The Generalized Traveling Salesman Problem (GTSP), also known as Errand Scheduling, Group TSP, Set TSP, One-of-a-Set TSP, Multiple-Choice TSP, and TSP with Neighborhoods, was originally introduced in the late 1960s [7,15] as a generalization of the well-known TSP formulation. Given a set of nodes partitioned into groups, or categories, the goal is to find a minimum-cost tour that visits exactly one node from each category. As TSP is a special case of GTSP in which each node defines its own category, then GTSP is also NP-hard. A review of many original applications of this problem type is presented in [8].

Initial solutions for this problem were based on exact dynamic programming formulations [7,14,15]. Other exact algorithms based on integer- and linear-programming techniques are presented in [5,9,10,13]. Much research has also been focused on transforming GTSP instances into standard TSP instances with roughly the same number of total nodes [1,4,11]. Under these transformations, an optimal solution to the transformed TSP instance is optimal for the original GTSP instance. However, most exact TSP algorithms remain exponential in the total number of nodes.

TSP problem variants having a fixed source node, s, and a fixed target node, t, are more commonly known as Traveling Salesman *Path* Problems. Therefore, in the remainder of our discussion we will be focused on the more specific Generalized Traveling Salesman Path Problem (GTSPP).

Our Contributions. Unfortunately, nearly all of the previous exact algorithms assume a pre-existing, complete graph on the set of category nodes (represented as a cost matrix). Such an assumption is invalid for most real-world navigation scenarios involving road networks, as these cost matrices must be computed explicitly from the underlying road network, and we do not know the category locations until query time (as they are likely to change from one use case to the next). Furthermore, computing such matrices requires $O(kg)$ graph searches, and can thus be quite time consuming, and potentially even prohibitive, in practice, especially for very large numbers of locations. For example, road networks can have categories with potentially millions of locations, and a complete matrix on such locations would require up to several terabytes of storage space.

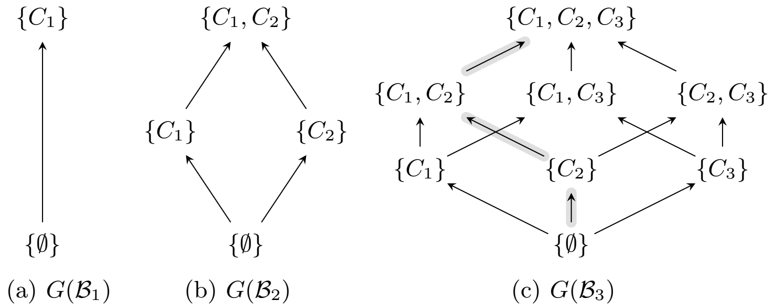
To avoid these difficulties, we reformulate GTSPP as a unique graph search problem which does not require the construction of a complete matrix between category locations. Using this as our foundation for a general algorithmic framework, we present two exact search algorithms for efficiently solving GTSPP instances on real-world road networks.

Additionally, while the number of locations to consider may be quite large in practice for many real-world GTSPP transportation and personal-navigation applications (e.g., $g = 100,000$), the number of categories is typically very small (e.g., $k = 5$). For such real-world problems in which $k \ll g$ (and often even $2^k \ll g$) is typically true, our proposed approach proves highly-advantageous because, unlike many of the algorithms discussed previously, our exact algorithms are exponential only in the number of categories, k.

3 GTSPP Product Graphs

Given any category set $C = \{C_1, C_2, \ldots, C_k\}$, let $\mathcal{B}_k = (\mathcal{P}(C), \subset)$ be the partially-ordered set (poset) defined by the power set of C when ordered by inclusion. Such a poset, $\mathcal{B}_k$, is called a Boolean lattice. The covering graph of a Boolean lattice poset $\mathcal{B}_k = (\mathcal{P}(C), \subset)$ on a category set C is a graph $G(\mathcal{B}_k) = (\mathcal{P}(C), E(\mathcal{B}_k))$, where $E(\mathcal{B}_k) = \{(c, c') \mid c, c' \in \mathcal{P}(C) \wedge c \subset c' \wedge \nexists c'' \in \mathcal{P}(C) : c \subset c'' \subset c'\}$. The covering graph defines a directed acyclic graph (DAG) on $\mathcal{B}_k$. We present examples of the covering graphs for several Boolean lattices in Fig. 1. Note that any path in the covering graph from the empty set to the full set represents a specific sequence of categories along the path, based on their order of accumulation (see Fig. 1). All $k!$ category sequences are thusly represented as paths in this graph. Also note that the Boolean lattice and its covering graph are exactly the same for *any* set of size k, as we can map the set members into the natural numbers $\{1, 2, \ldots, k\}$ (hence the subscript k, not C, in $\mathcal{B}_k$).

Given any graph $G = (V, E, w)$ and Boolean lattice $\mathcal{B}_k$ for a category set C of size k, we define the GTSPP product graph as $G_C = G \times G(\mathcal{B}_k) = (V \times \mathcal{P}(C), E_1 \cup E_2)$, where product nodes are represented as $\langle u, c \rangle$ such that $u \in V$ and $c \in \mathcal{P}(C)$, $E_1 = \{(\langle u, c \rangle, \langle v, c' \rangle) \mid c = c' \wedge (u, v) \in E\}$, and $E_2 = \{(\langle u, c \rangle, \langle v, c' \rangle) \mid u = v \wedge (c, c') \in E(\mathcal{B}_k) \wedge v \in c' \backslash c\}$. The E_1 edges represent a copy of each original edge from G for every subset of C. For all $(\langle u, c \rangle, \langle v, c' \rangle) \in E_1$,



(a) $G(\mathcal{B}_1)$ (b) $G(\mathcal{B}_2)$ (c) $G(\mathcal{B}_3)$

Fig. 1. Covering graphs for Boolean lattice posets $\mathcal{B}_1$, $\mathcal{B}_2$, and $\mathcal{B}_3$ (from left to right), respectively. The path highlighted in grey for $G(\mathcal{B}_3)$ represents the category traversal sequence $\langle C_2, C_1, C_3 \rangle$, based on the order of category accumulation along the path.

we define $w(\langle u, c \rangle, \langle v, c' \rangle) = w(u, v)$. The E_2 edges represent the accumulation of a new category (based on a corresponding covering graph edge) by inclusion of a specific node within that category. For all $(\langle u, c \rangle, \langle v, c' \rangle) \in E_2$, we define $w(\langle u, c \rangle, \langle v, c' \rangle) = 0$. Any path from $\langle s, \emptyset \rangle$ to $\langle t, C \rangle$ in G_C therefore represents a satisfying path in the original graph, based on a specific accumulation sequence of category nodes from each category.

We present a simple example of a GTSPP product graph in Fig. 2. For this problem instance, we have two unique categories (each with two unique nodes), and we must find the minimum-cost path from s to t which traverses one node from each category (as shown in green in the original graph). The resulting product graph is shown on the right of the figure. Edges from the set E_1 are shown as solid edges, whereas edges from the set E_2 are shown as dashed edges. The shortest path from $\langle s, \emptyset \rangle$ to $\langle t, C \rangle$ in the product graph is highlighted in grey, and its cost is equivalent to the optimal solution cost for this GTSPP instance.

Theorem 1. *Given any graph G and category set $C = \{C_1, C_2, \ldots, C_k\}$ (defined on G), the shortest path from $\langle s, \emptyset \rangle$ to $\langle t, C \rangle$ in the product graph G_C represents*

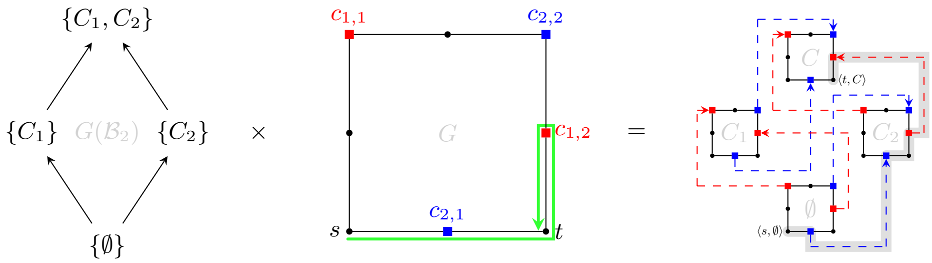


Fig. 2. Example product graph of graph G (with unit-cost edge weights) for category set $C = \{C_1, C_2\}$

an equivalent-cost, optimal solution for the GTSPP from s to t in the original graph G; i.e., $d(\langle s, \emptyset\rangle, \langle t, C\rangle) = w(P_{s,t}^C)$.

It follows from Theorem 1 that any shortest path algorithm will suffice to search the resulting product graph (e.g., Dijkstra's algorithm [3]).

Theorem 2. *Given any graph G and category set $C = \{C_1, C_2, \ldots, C_k\}$ (defined on G), a Dijkstra search from $\langle s, \emptyset\rangle$ to $\langle t, C\rangle$ in the product graph G_C runs in $O(2^k(m + nk + nlogn))$ time.*

Note that we do not need to explicitly construct the entire product graph to carry out the proposed search. We may instead perform an equivalent search in this product graph by materializing the nodes of the graph only as they are encountered implicitly during the search. This results in the potential for significant space savings for cases in which a solution path is found before most of the nodes are explored.

4 Product Graph Search Using Contraction Hierarchies

In this section, we further improve upon our proposed product-graph search approach by incorporating the graph preprocessing technique known as Contraction Hierarchies (CH) [6], originally designed for solving point-to-point (PTP) shortest path queries. We start with a brief overview of CH, followed by a discussion of how to integrate CH for efficiently solving GTSPP queries.

4.1 Contraction Hierarchies Overview

Preprocessing. CH preprocessing orders the nodes in the graph, $\phi : V \rightarrow \{1, \ldots, |V|\}$, and then *contracts* the nodes in this order. Contracting a node, v, removes it (temporarily) from the graph and adds so-called *shortcut* edges (if needed) to preserve shortest path costs in the remaining subgraph. For each pair of incoming and outgoing edges, (u, v) and (v, x), respectively, if the path $\langle u, v, x\rangle$ is a unique shortest path, then a new shortcut edge (u, x) is added with weight $w(u, v) + w(v, x)$ to bypass v in the remaining subgraph. The result is a new graph $G' = (V, E \cup E', w)$, where E' represents the added shortcut edges.

Query. The traditional CH shortest path query involves performing a forward Dijkstra search from s in the "upward" graph $G^{\uparrow} = (V, E^{\uparrow})$ where $E^{\uparrow} = \{(u, v) \in E \cup E' \mid \phi(u) < \phi(v)\}$ along with a simultaneous backward Dijkstra search from t in the "downward" graph $G^{\downarrow} = (V, E^{\downarrow})$ where $E^{\downarrow} = \{(u, v) \in E \cup E' \mid \phi(u) > \phi(v)\}$. Let $R_s^{\uparrow} = \{v \in V \mid \exists P_{s,v} \subseteq G^{\uparrow}\}$ be the set of all nodes reachable from s in the upward graph. Similarly, let $R_t^{\downarrow} = \{v \in V \mid \exists P_{v,t} \subseteq G^{\downarrow}\}$ be the set of all nodes from which t is reachable in the downward graph. Let $d_s^{\uparrow}(v)$ and $d_t^{\downarrow}(v)$ represent the shortest path cost from s to v in $G^{\uparrow}$ and from v to t in $G^{\downarrow}$, respectively. The shortest path cost is taken as $d(s, t) = \underset{\forall v \in R_s^{\uparrow} \cap R_t^{\downarrow}}{min} \{d_s^{\uparrow}(v) + d_t^{\downarrow}(v)\}$.

4.2 Contraction Hierarchies for GTSPP

Sweeping Search. We preface this discussion by first establishing an alternative PTP query approach, which does not require bidirectional Dijkstra search, but instead takes advantage of a nice structural property of the resulting CH search graphs.[2] Specifically, each search graph is acyclic by definition (i.e., each search can only increase in node rank order ϕ), thus allowing us to compute shortest path costs using only linear scans of the search graphs in topological order. Since the shortest path cost can be determined by considering only the reachable search spaces of s and t in $G^{\uparrow}$ and $G^{\downarrow}$ respectively, it suffices to consider only the union of these two search spaces $R = R_s^{\uparrow} \cup R_t^{\downarrow}$. Let $R_\phi = \langle v_1, v_2, \ldots, v_z \rangle$ be the nodes of R arranged in increasing rank order, ϕ, establishing a valid topological order. For any $v \in R$, let the value $d(v)$ represent the cost of the shortest path from s found so far during the search.

The search begins by initializing $d(s) = 0$ and $d(v) = \infty$ for all $v \neq s \in R$. The search progresses in two phases: an *upsweep* and a *downsweep* phase. The upsweep phase processes each node v_i in increasing order of ϕ. For each outgoing edge (v_i, x), such that $x \in R$ and $\phi(v_i) < \phi(x)$, we set $d(x) = min\{d(x), d(v_i) + w(v_i, x)\}$. The downsweep phase then processes each node v_i in the opposite (decreasing) order of ϕ. For each incoming edge (u, v_i), such that $u \in R$ and $\phi(u) > \phi(v_i)$, we set $d(v_i) = min\{d(v_i), d(u) + w(u, v_i)\}$.

Lemma 1. *Upon completion of the upsweep and downsweep phases, $d(v) = d(s, v)$ for all $v \in R$ such that $R_v^{\downarrow} \subseteq R$.*

This approach may be further extended to support many-to-many scenarios with multiple sources $S = \{s_1, s_2, \ldots, s_{|S|}\}$ and targets $T = \{t_1, t_2, \ldots, t_{|T|}\}$ by maintaining $|S|$ separate cost values for each node in the unioned search space $R = \{\bigcup_{i=1}^{|S|} R_{s_i}^{\uparrow}\} \cup \{\bigcup_{i=1}^{|T|} R_{t_i}^{\downarrow}\}$. Now, when relaxing the edges of a node during both the upsweep and downsweep phases on R_ϕ, we relax all $|S|$ costs before progressing to the next node.

LEvel Sweeping Search (LESS). We incorporate this alternative CH-based search algorithm into our GTSPP product-graph framework as follows. For the remainder of the paper, we shall assume that our product graph incorporates the preprocessed CH graph. By construction, each product graph is comprised of exactly $(k + 1)$ unique *levels* (as shown in Fig. 3). For $0 \leq i \leq k$, level $G_i = G_C[\{\langle u, c \rangle \in V(G_C) \mid i = |c|\}]$ is the subgraph induced by product nodes whose associated category subset has cardinality i. Each level is further comprised of smaller subgraphs $G^x = G_{|x|}[\{\langle u, c \rangle \in V(G_{|x|}) \mid c = x\}]$, which we shall call here *set-equivalent* (SE) subgraphs (also shown in Fig. 3). Within each level G_i, by definition of our product graph, there cannot exist any path between two distinct SE subgraphs (since the E_1 edges only connect nodes within the same SE subgraph and the E_2 edges only connect nodes between consecutive levels). For $i > 0$, this suggests that the costs within each SE subgraph of level G_i must therefore depend solely on the costs established in the previous level G_{i-1}.

[2] A similar approach has been discussed independently in [2].

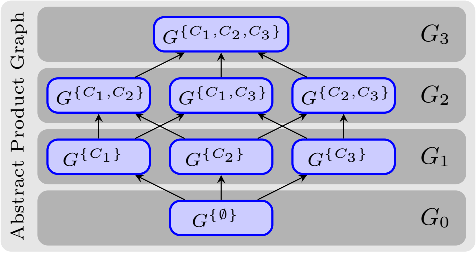


Fig. 3. An abstraction of the product graph G_C for $C = \{C_1, C_2, C_3\}$, illustrating levels (shown in dark grey) and SE subgraphs (shown in light blue)

We wish to take advantage of this useful property in our GTSPP search. Specifically, this allows us to process our product graph search one level at a time. Furthermore, for each level, we may process the costs for each SE subgraph independently, as these costs cannot influence one another. One solution would be to separately search each SE subgraph (using separate upsweep/downsweep phases), in any given order, for a given level. However, this would require 2^k separate upsweep and downsweep phases, one for each SE subgraph, and could be largely redundant as the separate sweeps may re-explore many of the same nodes and edges. An equivalent and more cache-efficient formulation is to instead perform only $(k+1)$ upsweep/downsweep phases (one for each level), maintaining a separate cost value per node for each SE subgraph within a given level. That is, when we process a node during a sweeping phase, we process all SE subgraph costs at the current level for that node before proceeding to the next node.

Since we only need to consider shortest paths that begin at s, travel through some nodes in C, and end at t, we may represent this search space as $R = \{R_s^{\uparrow}\} \cup \{\bigcup_{i=1}^{k} \bigcup_{j=1}^{|C_i|} \{R_{c_{i,j}}^{\downarrow} \cup R_{c_{i,j}}^{\uparrow}\}\} \cup \{R_t^{\downarrow}\}$. We begin by initializing $d(\langle s, \emptyset \rangle) = 0$, and for all other product nodes $\langle u, c \rangle \neq \langle s, \emptyset \rangle \in R \times \mathcal{P}(C)$, we initialize $d(\langle u, c \rangle) = \infty$. For $0 \leq i \leq k$, we process each level G_i by keeping track of exactly $\binom{k}{i}$ costs for each node in R: one for each SE subgraph in level G_i. Before beginning each sweeping phase on a given level G_i, if $i > 0$ we must transfer costs from the previous search level G_{i-1} via the established E_2 edges, according to our product graph definition. Specifically, for all $(\langle u, c \rangle, \langle v, c' \rangle) \in E_2 : |c'| = i$, we set $d(\langle v, c' \rangle) = min\{d(\langle v, c' \rangle), d(\langle u, c \rangle)\}$ to transfer the costs. We then proceed to perform an upsweep and then downsweep of R_ϕ (similar to before) taking care to relax all $\binom{k}{i}$ costs for each node in each sweeping phase.

Lemma 2. *Upon completion of the upsweep and downsweep phases for level G_i, $d(\langle v, c \rangle) = d(\langle s, \emptyset \rangle, \langle v, c \rangle)$ for all $\langle v, c \rangle \in R \times \mathcal{P}_i(C)$ such that $R_v^{\downarrow} \subseteq R$, where $\mathcal{P}_i(C)$ is the power set of C containing only subsets of at most cardinality i.*

Corollary 1. *Upon completion of the upsweep and downsweep phases for level G_k, $d(\langle t, C \rangle) = d(\langle s, \emptyset \rangle, \langle t, C \rangle)$ represents the optimal GTSPP solution cost.*

Theorem 3. *The LESS algorithm runs in $O(2^k(m' + nk))$ time, where $m' = |E \cup E'|$.*

Pruning. While our current solution is correct, its relative performance is expected to be highly sensitive to the density of a given GTSPP query. This is because our runtimes for this algorithm are directly proportional to the size of our search space, R, which typically grows in size proportional to the density, g, of the GTSPP instance under consideration (i.e., more distinct locations make for larger unioned search spaces). This expected behavior suggests that before the search it could be beneficial for us to try and aggressively prune any locations which we determine cannot possibly belong to any optimal GTSPP solution, in order to minimize the overall search space size and resulting runtime.

To achieve this, we require a method for estimating the cost of a solution which contains a given category node. A straightforward and fast approach is to utilize a constant-time heuristic function, $h : V \times V \to \mathbb{R}_{\geq 0}$, which returns a non-negative estimate on the shortest-path cost between any two nodes. We only require that the heuristic function be *admissible* such that $h(s,t) \leq d(s,t)$ for all $s, t \in V$ (i.e., it must always underestimate the shortest-path cost).

Given any admissible heuristic function, h, we must first establish an upper bound, μ, on the optimal GTSPP solution cost $w(P^C_{s,t})$. Starting at node $x_0 = s$ and ending at node $x_{k+1} = t$, for $1 \leq i \leq k$, we apply a greedy nearest-neighbor strategy to select a node $x_i = \underset{\forall c_{i,j} \in C_i}{argmin} \{h(x_{i-1}, c_{i,j})\}$, giving us the resulting node sequence $\langle x_0, x_1, \ldots, x_k, x_{k+1} \rangle$. We then compute the value $\mu = \sum_{0 \leq i \leq k} d(x_i, x_{i+1})$ by performing $(k+1)$ fast, point-to-point shortest path queries in the CH search graph (using the traditional bidirectional Dijkstra search). Since, by definition, the path established by the node sequence $\langle x_0, x_1, \ldots, x_k, x_{k+1} \rangle$ is a *satisfying* path for the instance $\langle s, t, C \rangle$, then μ is also therefore a valid upper bound on $w(P^C_{s,t})$.

We now prune each category, C_i, as follows. For each $c_{i,j} \in C_i$, if $h(s, c_{i,j}) + h(c_{i,j}, t) > \mu$, we remove node $c_{i,j}$ from C_i. This is because the value $h(s,v) + h(v,t)$ is a valid lower bound on any GTSPP solution which contains node v. After pruning, we may then carry out the proposed LESS algorithm, as before.

5 Experiments

In this section, we present experiments highlighting the performance characteristics of our proposed GTSPP algorithms. Specifically, we examine the performance impacts of our two primary measures of interest regarding GTSPP complexity: category density (g) and the number of categories (k). For each experiment, we consider four algorithms for comparison: unidirectional Dijkstra search (**U. Dijkstra**), bidirectional Dijkstra search (**B. Dijkstra**), CH-based Level-Sweeping Search (**LESS**), and LESS with Pruning (**P-LESS**).

For each experiment, we further consider two variants of GTSPP queries to model two possible extremes of locality: **non-local queries** in which $s \neq t$ are examined to model relatively long-distance routes and **local queries** in which $s = t$ are examined to model relatively short-distance routes.

Test Environment. All experiments were carried out on a 64-bit server machine running Linux CentOS 5.3 with 2 quad-core CPUs clocked at 2.53 GHz with 18 GB RAM (only one core was used per experiment). All programs were written in C++ and compiled using gcc version 4.1.2 with optimization level 3.

Test Dataset and Preprocessing. All experiments were performed on the road network of North America[3], with $21,133,774$ nodes and $52,523,592$ edges. The weight function, w, is based on travel time (in minutes). This dataset was derived from NAVTEQ data products, under their permission. We have chosen the Pre-Computed Cluster Distances (PCD) heuristic function from [12] as our pruning function, h. In brief, PCD partitions the graph into r partitions and computes an $r \times r$ cost matrix of the shortest path costs between the closest nodes from each pair of partitions. The heuristic function $h(u, v)$ returns the matrix value between the partitions of u and v, which is a lower bound on $d(u, v)$. For our experiments, we have chosen $r = 10,000$. PCD preprocessing required 7 minutes using the CH search graph, resulting in an overhead of 23 bytes per node. CH preprocessing required 18 minutes, resulting in an overhead of 35 bytes per node.

5.1 Category Density

We begin by examining the impact of category density, g, on the performance of our proposed algorithms. In Fig. 4(a) and Fig. 4(b) we present the results of our four algorithms for both non-local and local queries, respectively.

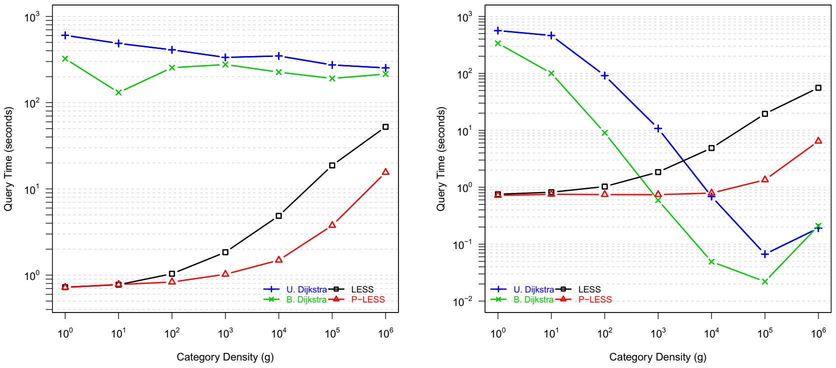
For all experiments in this section, we fixed the category count at $k = 5$. For every $0 \leq i \leq 6$, we constructed 100 random query instances in which each of the 5 categories were populated with $g = 10^i$ nodes selected uniformly at random. All source and target nodes, s and t, were additionally selected uniformly at random. The numbers presented in the figures represent average query times.

Starting with the non-local experiments of Fig. 4(a), we see that both Dijkstra variants have very high runtimes across all densities, but tend to improve as the density increases. B. Dijkstra is consistently faster than U. Dijkstra by a factor of 1.8, on average. However, it requires over 130 seconds, even in the best case.

In contrast, our advanced LESS algorithm can be seen to perform extremely well for low-density scenarios, but (as expected), begins to degrade as the density increases. Despite this degradation, it still outperforms the best Dijkstra algorithm by a factor of over two orders of magnitude, on average. The P-LESS approach reduces the runtimes even further, showing an additional 41% improvement, on average, over the unpruned LESS algorithm for these non-local queries, and requiring no more than 16 seconds in the worst case (for $g = 1,000,000$).

The story is slightly different, however, for the local query cases in Fig. 4(b). Here we see the Dijkstra algorithms show a much more significant overall improvement as the density increases (B. Dijkstra is less than 1 second for cases in which $g \geq 1,000$). This is due primarily to the fact that, as the density grows, so do the number of available zero-cost E_2 edges in the product graph. For such

[3] This includes only the US and Canada.



(a) $s \neq t, k = 5$ (b) $s = t, k = 5$

Fig. 4. Category Density Experiments for (a) non-local queries and (b) local queries

extremely-local cases (e.g., where $s = t$), this greatly benefits greedy search algorithms such as the Dijkstra variants which can then quickly transition along these increasingly-available E_2 edges to arrive at the nearby target in the final level. Alternatively, this scenario presents a slight disadvantage for both LESS-based strategies, which are not greedy by nature, but are instead forced to progress through the product graph search one level at a time, regardless of locality or density. Despite this, P-LESS is still the only algorithm which requires no more than 7 seconds across all of the densities tested here for such cases.

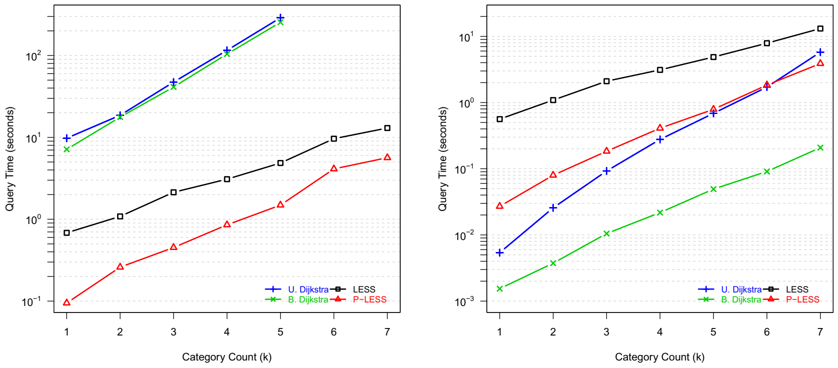
Furthermore, for such extremely local cases, we also see a marked improvement in our P-LESS approach over the unpruned LESS algorithm (e.g., pruning gives nearly an order of magnitude speed improvement for the highest-density scenario). This is anticipated from the fact that local cases are expected to have much smaller μ values, especially for high densities, leading to greater pruning.

5.2 Category Count

Next we examine the impact of the number of categories, k, on the performance of our algorithms. In Fig. 5(a) and Fig. 5(b) we present the results for both non-local and local queries, respectively (showing average query times, as before).

For all experiments in this section, we fixed the category density at $g = 10,000$. For every $1 \leq i \leq 7$, we constructed 100 random query instances, each with i categories populated with g nodes selected uniformly at random. All source and target nodes, s and t, were additionally selected uniformly at random.

Again, we start by reviewing the non-local query results in Fig. 5(a). As before, for non-local queries, the Dijkstra algorithms perform the worst overall, reaching nearly 300-second query times. Additionally, they are unable to even complete for cases with $k \geq 6$, due to memory exhaustion from such increasingly-large search spaces. Our LESS-based algorithms show similar improvements as before for such non-local queries, and appear to scale quite well across these



(a) $s \neq t, g = 10,000$

(b) $s = t, g = 10,000$

Fig. 5. Category Count Experiments for (a) non-local queries and (b) local queries

experiments, with the P-LESS algorithm requiring no more than 6 seconds for even the largest number of categories at $k = 7$.

For the local query results in Fig. 5(b), we see similar improvements for the Dijkstra search algorithms as before, with the B. Dijkstra algorithm requiring only up to 0.2 seconds in the worst case. This gives over an order of magnitude improvement, on average, compared to the U. Dijkstra search. Comparatively, our LESS-based algorithms perform the worst overall for these local scenarios, suggesting a similar pattern to that from the previous experiments, in which the greedy algorithms perform best for local queries, but worst for non-local queries.

Across both sets of experiments, the P-LESS algorithm again provides significant improvements over LESS, although its relative speedups appear to degrade with increasing category counts (going from a speedup of over 7 for $k = 1$ down to just over 2 for $k = 7$ for non-local queries, and over 20 down to just over 3 for local queries). As our current heuristic estimates for pruning are most accurate for fewer categories, this suggests that a more accurate heuristic may lead to further improvements over queries with larger numbers of categories.

6 Conclusion

We have demonstrated a new algorithmic framework based on a unique product-graph formulation, which allows us to solve real-world, large-scale GTSPP instances using various graph search algorithms. Our proposed algorithms are able to efficiently solve such problem instances to optimality, typically in a matter of seconds. These algorithms may also be used interchangeably, as needed, based on their respective performance advantages across various problem sizes and scales of locality. Specifically, our results suggest that, for highly-local, very-dense queries, a greedy Dijkstra search in our proposed product graph may be sufficient and effective in practice (and does not require preprocessing). However, for more consistently-efficient performance over longer distances and for various problem sizes, our proposed LESS algorithm (with pruning) is justifiably better.

Several promising areas of future research include pursuing more-aggressive pruning strategies and incorporating goal-directed search techniques (e.g., A^* search). Additionally, since each SE subgraph cost in a given level may be processed independently, this suggests that our level-sweeping search algorithm may further lend itself to strong parallelization, similar to that achieved in [2].

Acknowledgement. Work funded by NSF grants IIS-1144158 and IIS-0910859.

References

1. Behzad, A., Modarres, M.: A new efficient transformation of generalized traveling salesman problem into traveling salesman problem. In: Proceedings of the 15th International Conference of Systems Engineering, ICSE (2002)
2. Delling, D., Goldberg, A.V., Nowatzyk, A., Werneck, R.F.F.: Phast: Hardware-accelerated shortest path trees. In: IPDPS, pp. 921–931 (2011)
3. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik 1, 269–271 (1959)
4. Dimitrijevic, V., Saric, Z.: An efficient transformation of the generalized traveling salesman problem into the traveling salesman problem on digraphs. Inf. Sci. 102(1-4), 105–110 (1997)
5. Fischetti, M., Gonzalez, J.J.S., Toth, P.: A branch-and-cut algorithm for the symmetric generalized traveling salesman problem. Operations Research 45(3), 378–394 (1997)
6. Geisberger, R., Sanders, P., Schultes, D., Delling, D.: Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In: McGeoch, C.C. (ed.) WEA 2008. LNCS, vol. 5038, pp. 319–333. Springer, Heidelberg (2008)
7. Henry-Labordere, A.L.: The record balancing problem: A dynamic programming solution of a generalized traveling salesman problem. RIRO B-2, 43–49 (1969)
8. Laporte, G., Asef-Vaziri, A., Sriskandarajah, C.: Some applications of the generalized travelling salesman problem. The Journal of the Operational Research Society 47(12), 1461–1467 (1996)
9. Laporte, G., Mercure, H., Nobert, Y.: Generalized travelling salesman problem through n sets of nodes: The asymmetrical case. Discrete Applied Mathematics 18(2), 185–197 (1987)
10. Laporte, G., Nobert, Y.: Generalized travelling salesman problem through n sets of nodes: An integer programming approach. INFOR 21(1), 61–75 (1983)
11. Lien, Y.-N., Ma, Y.W.E., Wah, B.W.: Transformation of the generalized traveling-salesman problem into the standard traveling-salesman problem. Inf. Sci. 74(1-2), 177–189 (1993)
12. Maue, J., Sanders, P., Matijevic, D.: Goal Directed Shortest Path Queries Using _Precomputed _Cluster _Distances. In: Àlvarez, C., Serna, M. (eds.) WEA 2006. LNCS, vol. 4007, pp. 316–327. Springer, Heidelberg (2006)
13. Noon, C.E., Bean, J.C.: A lagrangian based approach for the asymmetric generalized traveling salesman problem. Operations Research 39(4), 623–632 (1991)
14. Saksena, J.P.: Mathematical model of scheduling clients through welfare agencies. CORS Journal 8, 185–200 (1970)
15. Srivastava, S.S., Kumar, S., Garg, R.C., Sen, P.: Generalized travelling salesman problem through n sets of nodes. CORS Journal 7, 97–101 (1969)

Control Complexity in Bucklin, Fallback, and Plurality Voting: An Experimental Approach[⋆]

Jörg Rothe and Lena Schend

Institut für Informatik, Universität Düsseldorf, 40225 Düsseldorf, Germany
{rothe,schend}@cs.uni-duesseldorf.de

Abstract. Walsh [23,22], Davies et al. [9], and Narodytska et al. [20] studied various voting systems empirically and showed that they can often be manipulated effectively, despite their manipulation problems being NP-hard. Such an experimental approach is sorely missing for NP-hard control problems, where control refers to attempts to tamper with the outcome of elections by adding/deleting/partitioning either voters or candidates. We experimentally tackle NP-hard control problems for Bucklin and fallback voting, which among natural voting systems with efficient winner determination are the systems currently known to display the broadest resistance to control in terms of NP-hardness [12,11]. We also investigate control resistance experimentally for plurality voting, one of the first voting systems analyzed with respect to electoral control [1,18].

Our findings indicate that NP-hard control problems can often be solved effectively in practice. Moreover, our experiments allow a more fine-grained analysis and comparison—across various control scenarios, vote distribution models, and voting systems—than merely stating NP-hardness for all these control problems.

1 Introduction and Motivation

Electoral control [1,18] refers to attempts to tamper with the outcome of elections by adding/deleting/partitioning either voters or candidates. To protect elections against such control attempts and other ways of manipulation (see, e.g., the surveys [14,2,13,16]), much work has been done recently to show that the attacker's task can be computationally hard: Certain voting systems are resistant to manipulation [13,16,8] or control [1,18,12,11] in certain scenarios. However, most of this work is concerned with NP-hardness results, which is a worst-case measure of complexity and leaves open the possibility that many elections can still be manipulated or controlled in a reasonable amount of time. Therefore, manipulation and control problems have also been tackled from different angles [21]. From a theoretical perspective, Zuckerman et al. [24] proposed approximation algorithms for NP-hard manipulation problems and Faliszewski et al. [15] showed that restricting to single-peaked electorates may strip manipulation and control problems off their NP-hardness shields. From an experimental perspective, Walsh [23,22], Davies et al. [9], and Narodytska et al. [20] (see also Coleman and

[⋆] This work was supported in part by DFG grants RO 1202/15-1 and RO 1202/12-1 (within the EUROCORES programme LogICCC of the ESF), SFF grant "Cooperative Normsetting" of HHU Düsseldorf, and a DAAD grant for a PPP project in the PROCOPE programme.

R. Klasing (Ed.): SEA 2012, LNCS 7276, pp. 356–368, 2012.

Teague [7]) studied various voting systems empirically, such as single transferable vote (STV), veto, Borda's, Nanson's, and Baldwin's rules, and showed that they can often be manipulated effectively, even though their manipulation problems are NP-hard. Such an experimental approach is sorely missing for NP-hard control problems to date.

This paper makes the first attempt to tackle NP-hard control problems via an experimental analysis. Among natural voting systems with efficient winner determination, the system currently known to display the broadest resistance (NP-hardness) to control is fallback voting, proposed by Brams and Sanver [5] via combining approval with Bucklin voting. Erdélyi et al. [12,11] showed that fallback voting is resistant to 20 out of the 22 standard types of control and that only Bucklin voting behaves almost as (and possibly equally) well. We empirically study the eight common voter control scenarios and four of the 14 common candidate control scenarios for Bucklin, fallback, and plurality voting. While the papers [23,22,9,20] focused on *constructive* manipulation problems only (where the aim is to make some favorite candidate win), we study both *constructive* and *destructive* control problems (the latter aiming at preventing some despised candidate's victory). When generating random elections in our experiments, we consider two probability distributions: the *Impartial Culture model* (where votes are distributed uniformly and are drawn independently) and the *Two Mainstreams model* (introduced here to model two mainstreams in society by adapting the *Pólya–Eggenberger urn model* [3]).

In general, our findings indicate that NP-hard control problems can often be solved effectively in practice. Our experiments also allow a more fine-grained analysis than merely stating NP-hardness for all the corresponding control problems: Specifically, we can compare constructive with destructive control, control across various voting systems in various control scenarios, and our two particular models of vote distribution.

After introducing some basic background on elections and electoral control in Section 2, we present the experimental setup and the algorithms implemented in Section 3. In Section 4 we outline and analyze our experimental control results. We discuss these results in Section 5, and give some conclusions and a brief outlook.

2 Preliminaries

Elections and Voting Systems: An *election* is a pair (C,V), where $C = \{c_1, c_2, \dots, c_m\}$ is a finite candidate set and $V = (v_1, v_2, \dots, v_n)$ is a finite list of voters expressing their preferences over the candidates in C. How the votes are represented depends on the voting system used. A *voting system* $\mathscr{E}$ determines how the voters' ballots are cast and who has won a given election (C,V), where the set $W \subseteq C$ of winners may be empty or have one or more elements. We call an election with votes cast according to $\mathscr{E}$ an $\mathscr{E}$ *election*. Here we focus on the systems Bucklin, fallback, and plurality voting.

Bucklin voting is a preference-based voting system named after James W. Bucklin [19]. "Preference-based" means that the voters' ballots are (strict) linear orders over all candidates in C. For example, if $C = \{c_1, c_2, c_3\}$ and a vote v is given by $c_2\, c_1\, c_3$, then this voter v strictly prefers c_2 to c_1 and c_1 to c_3. Let (C,V) be a given Bucklin election. The *level i score of a candidate* $c \in C$ ($score^i_{(C,V)}(c)$, for short) is the number of voters in V ranking c among their top i positions. Letting the *strict majority threshold* of a list V of votes be $maj(V) = \lfloor \|V\|/2 \rfloor + 1$, the *Bucklin score of* $c \in C$ is defined to be the smallest i such that $score^i_{(C,V)}(c) \geq maj(V)$. Every candidate with the smallest Bucklin score (say ℓ) and the highest level ℓ score is a *level* ℓ *Bucklin winner* (*BV winner*, for short). Note that there always exists a Bucklin winner and only level 1 Bucklin winners are always unique.

Fallback voting is a hybrid voting system introduced by Brams and Sanver [5]. It combines Bucklin voting with *approval voting* [4]. In a fallback election, each voter determines those candidates he or she approves of, and provides a linear order of the approved candidates. For example, if $C = \{c_1, c_2, c_3\}$ then a vote in a fallback election could be of the form $c_3\, c_1$, meaning that this voter approves of c_1 and c_3, strictly preferring c_3 to c_1, and disapproves of c_2. Winners are determined as follows in fallback voting: Given a fallback election (C,V), the notions of *level i score* of a candidate $c \in C$ and *level i fallback winner* are defined analogously as in Bucklin voting. If there is a level ℓ fallback winner with $\ell \leq \|C\|$, then he or she is a *fallback winner in* (C,V). Otherwise (i.e., if no fallback winner exists in (C,V)), every candidate with a highest *approval score* (which is the number of voters approving of this candidate) is a *fallback winner in* (C,V). The second case can occur in fallback elections, since the voters can prevent the candidates from gaining points by disapproving them, and so it is possible that no candidate reaches or exceeds the strict majority threshold on any level.

In *plurality voting*, the most preferred candidate in each vote gains one point, and the candidates with the most points are the *plurality winners*. Note that there always exists at least one plurality winner. This voting rule is preference-based as well, even though the ranking of the candidates after the top candidate is irrelevant.

Standard Control Scenarios and Their Problems: Unlike *manipulation* where voters cast insincere votes to strategically influence election outcomes [13,16], *electoral control* describes ways to tamper with the outcome of an election by changing the structure of the election itself [1,18]. These structural changes include adding, deleting, and partitioning either voters or candidates, and are exerted by an external actor, the "chair," having full knowledge of the voters' preferences (for a detailed discussion of why and where this assumption is appropriate for control complexity, see [18]). Bartholdi et al. [1] introduced the notion of *constructive control* where the chair's goal is to make a designated candidate end up winning alone the resulting election. The case where the chair's control action aims at preventing a given candidate from being a unique winner is called *destructive control* and has been introduced by Hemaspaandra et al. [18].

To study the complexity of control in different scenarios, a decision problem is defined for each standard type of electoral control. As these control scenarios and the related decision problems have been motivated and formally described in detail in many papers (see, e.g., [1,18,14,2,12,11]), we refrain from repeating these 22 definitions here, but rather refer to the literature. Here we only give two sample definitions of control problems, where for the first one we give the formal description:

$\mathscr{E}$-CONSTRUCTIVE CONTROL BY DELETING VOTERS ($\mathscr{E}$-CCDV)

Given: An $\mathscr{E}$ election (C,V), a designated candidate $c \in C$, and a positive integer k.

Question: Is there a subset $V' \subseteq V$ with $\|V'\| \leq k$ such that c is the unique $\mathscr{E}$ winner of election $(C, V - V')$?

Table 1. Known results on the control complexity of fallback, Bucklin, and plurality voting, for those control types considered here. Key: R = resistance, V = vulnerability, and S = susceptibility.

	Fallback Voting		Bucklin Voting		Plurality Voting	
Control by	Constr.	Destr.	Constr.	Destr.	Constr.	Destr.
Adding Voters	R	V	R	V	V	V
Deleting Voters	R	V	R	V	V	V
Partition of Voters	TE: R	TE: R	TE: R	TE: R	TE: V	TE: V
	TP: R	TP: R	TP: R	TP: S	TP: R	TP: R
Adding Candidates	R	R	R	R	R	R
Deleting Candidates	R	R	R	R	R	R

Control by partition of voters is modeled via a two-stage election. In the first stage, V is partitioned into V_1 and V_2, and the winners (subject to the tie-handling rule used, see below) of subelections (C,V_1) and (C,V_2) run against each other in the runoff, with respect to V. Hemaspaandra et al. [18] introduced the tie-handling rules "*ties promote*" (*TP*: all subelection winners participate in the runoff), and "*ties eliminate*" (*TE*: only a unique winner from either subelection can move on to the runoff; if there is more than one winner, none of them moves on).

Among the 14 standard candidate control problems [2] that are defined similarly, we confine ourselves here to constructive and destructive control by adding and by deleting candidates.

Let $\mathfrak{C}$ be a type of electoral control (e.g., constructive control by deleting voters). Using the notions defined by Bartholdi et al. [1] (see also [18]), we say a voting system $\mathscr{E}$ is *immune to* $\mathfrak{C}$ if the chair never succeeds in exerting control of type $\mathfrak{C}$. If $\mathscr{E}$ is not immune to $\mathfrak{C}$, it is *susceptible to* $\mathfrak{C}$. If $\mathscr{E}$ is susceptible to $\mathfrak{C}$, we say it is *vulnerable to* $\mathfrak{C}$ if the corresponding decision problem is decidable in deterministic polynomial time, and we say it is *resistant to* $\mathfrak{C}$ if the corresponding decision problem is NP-hard.

Control in Bucklin, Fallback, and Plurality Voting: Plurality voting is one of the first voting systems for which the complexity of constructive control [1] and destructive control [18] has been studied in the above scenarios. Control in fallback voting and Bucklin voting has been previously studied by Erdélyi et al. [12,11] with respect to classical complexity and also with respect to parameterized complexity [10]. In terms of NP-hardness, among natural systems with polynomial-time winner determination fallback voting has the most resistances to control (namely, 20 out of the 22 standard control types) and Bucklin voting behaves similarly well—just one case is open (destructive control by partition of voters in model TP). Table 1 gives an overview of known complexity results for electoral control in these three voting systems. The theoretical importance of NP-hardness notwithstanding, these results have only limited significance in practical applications, as an NP-hard control problem might still be easy to solve on the average, or on "typical" instances. That is why we challenge the NP-hardness results of Table 1 experimentally.

3 Experimental Setup

In this section we describe the experimental setting. As stated above, the instances of the problems corresponding to control by adding or by deleting either candidates or voters contain a parameter k bounding the number of candidates or voters that can be added or deleted, which is crucial for the running time of the implemented algorithms. To realize our experiments in an acceptable time frame, we confine ourselves to the case of $k = \lfloor m/3 \rfloor$ and $k = \lfloor n/3 \rfloor$, respectively, where m is the number of candidates and n is the number of voters.[1] Also, since we have to cope with NP-hard problems, a time limit has been implemented such that the algorithm stops when exceeding this limit, indicating by the output "timeout" that the search process is aborted unsuccessfully. Again, this allows us to handle the worst-case scenarios in a reasonable amount of time.

We randomly generated elections (C,V) with $m = \|C\|$ and $n = \|V\|$ for all combinations of $n,m \in \{4,8,16,32,64,128\}$. Each such combination is one data point for which we evaluated 500 of these elections, trying to determine for each given election whether or not control is possible, and if it is possible, we say that this election is *controllable*. This restriction to 500 elections per data point, again, results from practical issues balancing out manageability and informative value of the experiments conducted.

Election Generation and Distributions of Votes: Before we specify the different distribution models underlying our election generation, we explain how random votes can be cast in the considered voting systems and how many different votes can exist. Assuming that the generated election has m candidates, in Bucklin voting and in plurality voting a random vote can be obtained by generating a random permutation over the m different candidates, so the overall number of different votes in Bucklin and plurality elections is $m!$. In fallback voting, random votes can be generated as follows:

- randomly draw a preference p from all $m!$ possible preferences with m candidates;
- randomly draw the number, say $\ell \in \{0,1,\dots,m\}$, of approved candidates;
- the generated vote consists of the first ℓ candidates in p.

Thus, there can be $\sum_{\ell=0}^{m} \binom{m}{\ell} \ell!$ different votes in fallback elections with m candidates.

In the *Impartial Culture model* (*IC model*) we assume uniformly distributed votes and draw each vote independently out of all possible preferences. In addition, we introduce the *Two Mainstreams model* (*TM model*) by adapting the *Pólya–Eggenberger urn model* (*PE model*, see [3]; this has been used, e.g., by Walsh [23]) as follows:

- randomly draw two preferences out of an urn containing all possible, say t, preferences, where either $t = m!$ or $t = \sum_{\ell=0}^{m} \binom{m}{\ell} \ell!$, depending on the voting system;
- put each preference back into the urn with t additional copies;
- draw the votes out of this urn independently at random with replacement.

Each of the two preferences drawn in the first step can be interpreted as a representative of one "mainstream" in society (e.g., liberal and conservative). We consider this model to be better suited for investigating control problems than the PE model itself, since in PE with relatively high probability many (or even all) votes can be identical (including,

[1] Since every yes-instance for a given k is also a yes-instance for each $k' \geq k$, the number of yes-instances found in our experiments are a lower bound for the true number of yes-instances.

e.g., unregistered votes that may be added), which artificially would make control impossible or trivial to find and the problem easy. By contrast, in manipulation [23,22,9] this effect has less impact, as manipulators can deviate from this one mainstream if they so wish.

A High-Level Description of the Algorithms: Our algorithms are heuristics, designed so as to test the most "promising" cases (depending on the control type at hand) first, by using appropriate preorderings. Due to space limitations, we only roughly sketch them here. All algorithms for the different types of control share the same essential method of testing various subsets, and they differ only in the type of preordering and internal testing. Before actually searching for a successful sublist of voters or subset of candidates, the algorithms check conditions that, if true, indicate that the given instance is a no-instance. Let c be the distinguished candidate. Depending on the control type, some of the following conditions are tested:

Condition 1: The designated candidate is ranked last (for Bucklin and plurality), or is ranked last or disapproved (for fallback), in every vote.

Condition 2: For each $k' \leq k$, determine the smallest i and j such that

$$score^{i}_{(C,V)}(c') \geq \lfloor(\|V\|-k')/2\rfloor + 1 + k' \text{ and } score^{j}_{(C,V)}(c) \geq \lfloor(\|V\|-k')/2\rfloor + 1$$

hold for $c' \in C - \{c\}$. Note that $i \leq j-1$ for all $k' \leq k$.

Condition 3: For each $k' \leq k$ determine the smallest i and j such that

$$score^{i}_{(C,V)}(c') \geq \lfloor(\|V\| + k')/2\rfloor + 1 \text{ and } score^{j}_{(C,V)}(c) \geq \lfloor(\|V\| + k')/2\rfloor + 1 - k'$$

hold for $c' \in C - \{c\}$. Note that $i \leq j-1$ for all $k' \leq k$.

Condition 4: In the given election, the winner has a strict majority on the first level.

Condition 1 (respectively, Condition 4) is tested for every constructive (respectively, destructive) control type investigated here. Note that these conditions must hold in the election for both the registered and the unregistered voters for control by adding voters, and both for the original and the spoiler candidates for control by adding candidates. Condition 2 is tested in addition for the deleting-voters cases, whereas Condition 3 is tested in addition for the adding-voters cases. After having excluded these trivial cases, each of the algorithms searches for a successful sublist/subset of preordered versions of V or C. Let us describe this procedure only for constructive control by deleting voters in detail, where the voters are ordered ascending for c. That is, after the preordering v_1 is a voter ranking c worst and v_n is a voter ranking c best among all voters. (In fallback voting, the "worst" position for a candidate is to be not approved at all.) The algorithm now starts with deleting those votes c benefits least of. It follows the procedure of a depth-first search on a tree of height k that is structured as shown in Figure 1. In each node, it is tested whether deleting the votes on the path is a successful control action. For example, on path $s \rightarrow 1 \rightarrow 2 \rightarrow 3$ the algorithm tests the sublists $(v_1), (v_1, v_2), (v_1, v_2, v_3)$ and then tracks back testing the sublists $(v_1, v_2, v_4), (v_1, v_2, v_5)$, $(v_1, v_3), (v_1, v_3, v_4)$, and so on. The branches on the left side are visited first and, due to the preordering of the votes, these are the votes c benefits least of.

For the adding-voters cases, the unregistered voters are ordered in a descending order for the designated candidate, and the algorithm proceeds similarly as the algorithm for

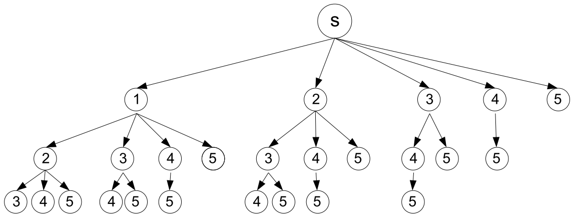


Fig. 1. Tree for $n = 5$ voters where up to $k = 3$ voters may be deleted. A node i corresponds to voter v_i after the preordering.

the deleting-voters cases. With this preordering, the algorithm first tests those voters the designated candidate can benefit most of when these are added to the voter list.

For the partition cases, the algorithm considers every possible sublist of the voter list up to size $k = \lfloor n/2 \rfloor$ as V_1, sets $V_2 = V - V_1$, and tests whether this is a successful control action or not. For the constructive cases, the voters are preordered descendingly with respect to the designated candidate, whereas for the destructive control cases no preordering is implemented.

In the candidate control scenarios, the candidates are also ordered with respect to the designated candidate, where a *descending order* here means that the first candidate has the most voters ranking him or her before the designated candidate and the last candidate has the fewest voters doing so. An *ascending order* is defined analogously. Again, in the adding-candidates case, the votes over all candidates (including the spoiler candidates) are considered. A descending ordering is used for finding control actions for constructive control by deleting candidates and for destructive control by adding candidates, whereas for the remaining candidate control cases an ascending order is used.

4 Summary of Experimental Results

We investigated the three voting systems only for those control types they are not known to be vulnerable to, which is indicated by an R- or an S-entry in Table 1. Due to space limitations we do not discuss the results for all types of control in detail.

Constructive Control by Deleting and by Adding Voters: We focus on presenting the results for control by deleting voters only, due to space limitations and since the results for control by adding voters are very similar for fallback and Bucklin voting.

In the Impartial Culture model, increasing the number of candidates decreases the number of yes-instances in the generated fallback elections. On the other hand, the number of yes-instances increases with the number of voters growing. In the Two Mainstreams model, the same correlations can be observed but here the overall numbers of yes-instances is smaller than in the IC model. Bucklin voting behaves very similarly, so for both distributions and both voting systems increasing the number of candidates makes successful actions of control by deleting voters less likely. In both voting systems and in both distribution models, timeouts occur whenever the number of voters exceeds 32. If the number of candidates is 128, we have timeouts already with 16 voters. The time limit is reached in about 11% of the elections analyzed in our experiments

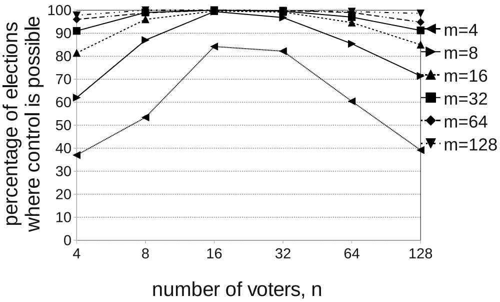


Fig. 2. Results for fallback voting in the IC model for destructive control by partition of voters in model TP, for a fixed number of candidates

(i.e., in about 8,225 out of 72,000 elections). Here again, the results for control by adding voters are very similar: Timeouts occur for the same election sizes and in about 10% of the elections the computation is aborted unsuccessfully. The algorithms for both control by adding and by deleting voters quickly find yes-instances, whereas the determination of no-instances becomes time-consuming for larger election sizes.

Control By Partition of Voters: As mentioned in Section 2, control by partition of voters comes in four problem variants, where each case must be investigated separately.

For constructive control by partition of voters in model TP we made the following observations: Similarly to control by deleting or by adding voters, the number of controllable elections increases with the number of voters increasing. This was observed for all three voting systems investigated. We have seen that in at most 13% of the tested plurality elections in the TM model a successful control action can be found. Note that no timeouts occur for up to 32 candidates, so more than 87% of the elections tested are demonstrably not controllable in these cases. For both distribution models, plurality elections produce fewer timeouts than the corresponding fallback or Bucklin elections. This suggests that the control problem for plurality voting is easier to solve on average than for fallback or Bucklin voting. Using the tie-handling model TE instead of TP, in both Bucklin and fallback voting an increase of yes-instances in the constructive cases is evident. By contrast, in the destructive counterparts no significant difference can be observed with respect to the tie-handling rule used.

The most striking results are those obtained for the destructive cases. Here we have that, for all three voting systems (and both tie-handling models for fallback and Bucklin voting) in the TM model, the average number of controllable elections is very high; and in the IC model, control is almost always possible, see Figure 2.

In light of the fact that for these cases the resistance proofs of Erdélyi et al. [12,11] for fallback and Bucklin voting tend to be the most involved ones (yielding the most complex instances for showing NP-hardness), these results might be surprising at first glance. However, one explanation for the observed results can be found in exactly this fact: The elections constructed in these reductions have a very complex structure which seems to be unlikely to occur in randomly generated elections (at least in elections generated under the distribution models discussed in this paper). Another explanation is that the problems used to reduce from in [12,11] tend to be easy to solve for small

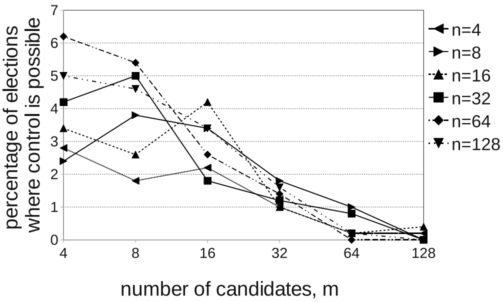


Fig. 3. Results for fallback voting in the TM model for constructive control by adding candidates, for a fixed number of voters

input sizes, but due to the complexity of the reduction, the resulting elections have many voters/candidates compared to the elections generated for the conducted experiments.

In the destructive cases, the number of timeouts is for all three voting systems with about 9% the lowest for all control types investigated. In Bucklin elections with uniformly distributed votes and for control by partition of voters in model TP, for only 3.32% of the elections no decision can be found within the time limit. As expected, in the corresponding constructive cases the number of timeouts is significantly higher.

Control By Adding Candidates: So far, the results for constructive control by adding candidates show the highest number of timeouts. For those election sizes where no timeouts occur (i.e., the determination of, respectively, yes- and no-instances is successful), we have that in none of the three voting systems many elections can be controlled successfully. In Figure 3, we see the results for fallback voting in the TM model, exemplifying the low numbers of yes-instances for this type of candidate control. In the IC model where, again, the overall number of controllable elections is higher, the highest percentage of controllable fallback elections is 11%. Bucklin elections behave similarly, but show more yes-instances: Up to 23% of the elections are controllable for less than 32 candidates in the IC model and up to 11% in the TM model. Plurality voting shows similar results as fallback voting with at most 20% yes-instances in the IC model and less than 4% in the TM model for those election sizes where no timeouts occur.

Turning now to the destructive variant of control by adding candidates, for Bucklin elections generated with the IC model, 71% is a lower bound for the number of controllable elections. For up to 16 candidates, a successful control action can be found in nearly all elections. The results for the TM model reconfirm the observation made before, namely that the tendencies in both models are similar, but with at most 77% and at least 42% of controllable elections the overall numbers are again lower than in the IC model. The latter results hold for fallback elections generated in the TM model as well, whereas in the IC model at least 53% and no more than 92% of the fallback elections are controllable. In the tested plurality elections generated with the IC model, similarly to Bucklin voting, more than 70% and nearly up to 100% are controllable. In the TM model, roughly between 50% and 60% of yes-instances are found for those election sizes where no timeouts occur, so between 40% and 50% of these plurality elections are not controllable. In this control scenario, for about 46% of the elections no definite output is given in the constructive case, whereas in only about 8% of the elections timeouts occur in the destructive case.

5 Discussion and Conclusions

We have empirically studied the complexity of NP-hard control problems for plurality, fallback, and Bucklin voting in the most important of the common control scenarios. This is the first such study for control problems in voting and complements the corresponding results [23,22,9,20,7] for manipulating elections. In general, our findings indicate that control can often be exerted effectively in practice, despite the NP-hardness of the corresponding problems. Our experiments also allow a more fine-grained analysis and comparison—across various control scenarios, vote distribution models, and voting systems—than merely stating NP-hardness for all these problems. Table 2 gives an overview of our experimental results. A detailed analysis and discussion follows.

IC versus TM: Comparing the results for the different distribution models, we see that in every voting system for all control types studied (except fallback voting in constructive control by deleting candidates) the overall number of yes-instances is higher in the IC than in the TM model. This may result from the fact that in elections with uniformly distributed votes, all candidates are likely to be approximately equally preferred by the voters. So both constructive and destructive control actions are easier to find. This also explains the observation that the IC model produces fewer timeouts.

Constructive versus Destructive Control: For all investigated types of control where both constructive and destructive control was investigated, we found that the destructive control types are experimentally much easier than their constructive counterparts, culminating in almost 100% of controllable elections for certain control types in the

Table 2. Overview of results: the "min" and "max" columns give the bottom and top percentage of yes-instances observed for the given control type; "to" gives the percentage of timeouts that occurred for the 18,000 tested elections; "n.i." stands for "not investigated" due to P-membership of the control problem. Results in boldface refer to elections generated in the TM model.

	FV			**BV**			**PV**		
	min	**max**	**to**	**min**	**max**	**to**	**min**	**max**	**to**
CCAC	1 / **0**	11 / **7**	51 / **50**	0 / **0**	23 / **11**	50 / **49**	0 / **0**	20 / **3**	50 / **34**
DCAC	53 / **39**	92 / **71**	11 / **14**	71 / **42**	99 / **77**	6 / **12**	70 / **47**	99 / **60**	7 / **25**
CCDC	13 / **15**	33 / **36**	37 / **37**	13 / **17**	58 / **45**	34 / **37**	5 / **22**	66 / **40**	37 / **35**
DCDC	8 / **12**	78/ **63**	15 / **22**	48 / **25**	99 / **77**	7 / **18**	7 / **4**	99 / **50**	10 / **35**
CCPV-TP	1 / **1**	53 / **20**	40 / **50**	1 / **0**	72 / **23**	31 / **48**	0 / **0**	54 / **13**	24 / **23**
DCPV-TP	37 / **27**	100 / **87**	6 / **17**	60 / **39**	100 / **88**	3 / **10**	55 / **15**	100 / **59**	4 / **35**
CCPV-TE	2 / **0**	97 / **34**	9 / **45**	2 / **0**	98 / **32**	8 / **44**	n.i.	n.i.	n.i.
DCPV-TE	50 / **34**	100 / **88**	4 / **16**	64 / **40**	100 / **89**	4 / **10**	n.i.	n.i.	n.i.
CCDV	2 / **1**	97 / **39**	16 / **12**	2 / **1**	100 / **42**	11 / **7**	n.i.	n.i.	n.i.
CCAV	4 / **1**	99 / **41**	13 / **13**	2 / **1**	99 / **41**	11 / **6**	n.i.	n.i.	n.i.

IC model. These findings confirm—and strengthen—the theoretical insight of Hemaspaandra et al. [18] that the destructive control problems disjunctively truth-table-reduce to their constructive counterparts and thus are never harder to solve, up to a polynomial factor (see also the corresponding observation of Conitzer et al. [8] regarding manipulation): In fact, destructive control tends to be even much easier than constructive control.

Comparison Across Voting Systems: For constructive control, we have seen that fallback and Bucklin voting show similar tendencies and numbers of yes-instances, especially regarding voter control. We also observed that their constructive voter control problems are in general harder to solve than those for plurality voting. In all three voting systems, constructive control by adding candidates seems to be the hardest control problem investigated so far, which leads to the interesting question of whether control by partition of candidates (left here for future research) is harder to solve on average.

Adding Candidates versus Deleting Candidates: Comparing control by adding candidates to control by deleting candidates in the constructive case we observed that the number of yes-instances for control by deleting candidates is significantly higher. These findings are perhaps not overly surprising, since in the voting systems considered here adding candidates to an election can only worsen the position of the designated candidate in the votes. That is, constructive control can be exerted successfully only if by adding candidates rivals of the designated candidate lose enough points so as to get defeated by him or her. This, in turn, can happen only if the designated candidate was already a highly preferred candidate in the original election.

Voter Control versus Candidate Control: For fallback and Bucklin voting, we can also compare constructive candidate and voter control directly. In both voting systems and both distribution models, the number of yes-instances for constructive control by adding voters is around four times higher than the number of yes-instances in the corresponding candidate control type, which confirms the argument above, saying that adding candidates cannot push the designated candidate directly. Constructive control by deleting voters can be successfully exerted more frequently when votes are less correlated, whereas the proportion of successful control actions for deleting candidates is about the same for both considered distribution models. The observed differences between these types of voter and candidate control may result from the fact that adding or deleting candidates only shifts the position of the designated candidate, which may not influence the outcome of the election as directly as increasing or decreasing the candidates' scores by adding or deleting voters does. This explains why voter control can be tackled more easily than candidate control by greedy approaches such as ours.

Concluding Remarks: Just as Walsh [22,23] observes for manipulation in the veto rule and in STV, for all types of control investigated in our experiments, the curves do not show the typical phase transition known for "really hard" computational problems such as the satisfiability problem (see [17,6] for a detailed discussion of this issue).

These observations raise the question of how other distribution models influence the outcome of such experiments. Furthermore, the algorithms implemented could be improved in terms of considering a higher number of elections per data point, increasing the election sizes, or allowing a higher number of voters or candidates to be deleted or added in the corresponding control scenarios. Besides this, other voting systems can be analyzed since only their winner determination has to be implemented in addition to a few minor adjustments such as trivial-case checks for the investigated control scenarios tailored to the voting system at hand. In future work, we will also investigate the remaining partition-of-candidates scenarios in Bucklin, fallback, and plurality voting.

Acknowledgments. We thank Toby Walsh for interesting discussions, Volker Aurich for giving us access to his computer lab, Guido Königstein for his help in setting up our experiments, and the anonymous reviewers for their helpful comments.

References

1. Bartholdi III, J., Tovey, C., Trick, M.: How hard is it to control an election? Mathematical and Computer Modelling 16(8/9), 27–40 (1992)
2. Baumeister, D., Erdélyi, G., Hemaspaandra, E., Hemaspaandra, L., Rothe, J.: Computational aspects of approval voting. In: Laslier, J., Sanver, R. (eds.) Handbook on Approval Voting, ch.10, pp. 199–251. Springer (2010)
3. Berg, S.: Paradox of voting under an urn model: The effect of homogeneity. Public Choice 47(2), 377–387 (1985)
4. Brams, S., Fishburn, P.: Approval Voting. Birkhäuser (1983)
5. Brams, S., Sanver, R.: Voting systems that combine approval and preference. In: Brams, S., Gehrlein, W., Roberts, F. (eds.) The Mathematics of Preference, Choice, and Order: Essays in Honor of Peter C. Fishburn, pp. 215–237. Springer (2009)
6. Cheeseman, P., Kanefsky, B., Taylor, W.: Where the really hard problems are. In: Proc. IJCAI 1991, pp. 331–337. Morgan Kaufmann (1991)
7. Coleman, T., Teague, V.: On the complexity of manipulating elections. In: Proc. CATS 2007., vol. 65, pp. 25–33 (2007)
8. Conitzer, V., Sandholm, T., Lang, J.: When are elections with few candidates hard to manipulate? Journal of the ACM 54(3), Article 14 (2007)
9. Davies, J., Katsirelos, G., Narodytska, N., Walsh, T.: Complexity of and algorithms for Borda manipulation. In: Proc. AAAI 2011, pp. 657–662. AAAI Press (August 2011)
10. Erdélyi, G., Fellows, M.: Parameterized control complexity in Bucklin voting and in fallback voting. In: Proc. COMSOC 2010, pp. 163–174. Universität Düsseldorf (2010)
11. Erdélyi, G., Piras, L., Rothe, J.: The complexity of voter partition in Bucklin and fallback voting: Solving three open problems. In: Proc. AAMAS 2011, pp. 837–844. IFAAMAS (2011)
12. Erdélyi, G., Rothe, J.: Control complexity in fallback voting. In: Proc. CATS 2010. CRPIT Series, vol. 32(8), pp. 39–48. Australian Computer Society (2010)
13. Faliszewski, P., Hemaspaandra, E., Hemaspaandra, L.: Using complexity to protect elections. Commun. ACM 53(11), 74–82 (2010)
14. Faliszewski, P., Hemaspaandra, E., Hemaspaandra, L., Rothe, J.: A richer understanding of the complexity of election systems. In: Ravi, S., Shukla, S. (eds.) Fundamental Problems in Computing: Essays in Honor of Professor Daniel J., ch.14, pp. 375–406. Springer (2009)
15. Faliszewski, P., Hemaspaandra, E., Hemaspaandra, L., Rothe, J.: The shield that never was: Societies with single-peaked preferences are more open to manipulation and control. Information and Computation 209(2), 89–107 (2011)
16. Faliszewski, P., Procaccia, A.: AI's war on manipulation: Are we winning? AI Magazine 31(4), 53–64 (2010)

17. Gent, I., Walsh, T.: Phase transitions from real computational problems. In: Proc. ISAI 1995, pp. 356–364 (1995)
18. Hemaspaandra, E., Hemaspaandra, L., Rothe, J.: Anyone but him: The complexity of precluding an alternative. Artificial Intelligence 171(5-6), 255–285 (2007)
19. Hoag, C., Hallett, G.: Proportional Representation. Macmillan (1926)
20. Narodytska, N., Walsh, T., Xia, L.: Manipulation of Nanson's and Baldwin's rules. In: Proc. AAAI 2011, pp. 713–718. AAAI Press (August 2011)
21. Rothe, J., Schend, L.: Typical-case challenges to complexity shields that are supposed to protect elections against manipulation and control: A survey. In: Website Proceedings of the Special Session on Computational Social Choice at the 12th International Symposium on Artificial Intelligence and Mathematics (2012)
22. Walsh, T.: Where are the really hard manipulation problems? The phase transition in manipulating the veto rule. In: Proc. IJCAI 2009, pp. 324–329. IJCAI (2009)
23. Walsh, T.: An empirical study of the manipulability of single transferable voting. In: Proc. ECAI 2010, pp. 257–262. IOS Press (2010)
24. Zuckerman, M., Procaccia, A., Rosenschein, J.: Algorithms for the coalitional manipulation problem. Artificial Intelligence 173(2), 392–412 (2009)

Advanced Coarsening Schemes for Graph Partitioning[⋆,⋆⋆]

Ilya Safro[1], Peter Sanders[2], and Christian Schulz[2]

[1] Mathematics and Computer Science Division, Argonne National Laboratory
safro@mcs.anl.gov
[2] Karlsruhe Institute of Technology, Institute for Theoretical Informatics,
Algorithmics II
{sanders,christian.schulz}@kit.edu

Abstract. The graph partitioning problem is widely used and studied in many practical and theoretical applications. Today multilevel strategies represent one of the most effective and efficient generic frameworks for solving this problem on large-scale graphs. Most of the attention in designing multilevel partitioning frameworks has been on the refinement phase. In this work we focus on the coarsening phase, which is responsible for creating structurally similar to the original but smaller graphs. We compare different matching- and AMG-based coarsening schemes, experiment with the algebraic distance between nodes, and demonstrate computational results on several classes of graphs that emphasize the running time and quality advantages of different coarsenings.

1 Introduction

Graph partitioning is a class of problems used in many fields of computer science and engineering. Applications include VLSI design, load balancing for parallel computations and network analysis. The goal is to partition the vertices of a graph into a certain number of disjoint sets of approximately the same size, so that a cut metric is minimized. This problem is NP-complete even for several restricted classes of graphs, and there is no constant factor approximation algorithm for general graphs [2]. In this paper we focus on a version of the problem that constrains the maximum block size to $(1+\epsilon)$ times the average block size and tries to minimize the total cut size, namely, the number of edges that run between blocks.

Because of the practical importance, many heuristics of different nature (spectral [3], combinatorial [4], evolutionist [5], etc.) have been developed to provide an approximate result in a reasonable computational time. However, only the introduction of the general-purpose multilevel methods during the 1990s has provided a breakthrough in efficiency and quality. Well-known software packages based on this approach include Metis [6] and Scotch [7] (to mention just few of them).

[⋆] Partially supported by DFG SA 933/10-1 and CSCAPES institute, a DOE project.

[⋆⋆] This paper is a short version of the technical report [1]

R. Klasing (Ed.): SEA 2012, LNCS 7276, pp. 369–380, 2012.

A multilevel algorithm consists of two main phases: *coarsening* – where the problem instance is gradually mapped to smaller ones to reduce the original complexity and *uncoarsening* – where the solution for the original instance is constructed by using the information inherited from the solutions created at the next coarser levels. In this work we focus on the coarsening phase, which is responsible for creating graphs that are smaller than but structurally similar to the given graph. We compare different coarsening schemes, introduce new elements to them, and demonstrate computational results. For this purpose algebraic multigrid (AMG) based coarsening components [8] have been integrated into the graph partitioning framework KaFFPa [9]. Similar integration has been done in [10]. We describe the difference between our approaches in [1].

2 Preliminaries

Consider an undirected graph $G = (V, E, c, \omega)$ with edge weights[1] $\omega : E \to \mathbb{R}_{>0}$, node weights $c : V \to \mathbb{R}_{\geq 0}$, $n = |V|$, and $m = |E|$. We extend c and ω to sets; in other words, $c(V') := \sum_{v \in V'} c(v)$ and $\omega(E') := \sum_{e \in E'} \omega(e)$. Here, $\Gamma(v) := \{u : \{v, u\} \in E\}$ denotes the neighbors of v. We are looking for *blocks* of nodes V_1,...,V_k that partition V, namely, $V_1 \cup \cdots \cup V_k = V$ and $V_i \cap V_j = \emptyset$ for $i \neq j$. The *balancing constraint* demands that $\forall i \in \{1, \cdots, k\} : c(V_i) \leq L_{\max} := (1 + \epsilon)c(V)/k + \max_{v \in V} c(v)$ for some parameter ϵ. The objective is to minimize the total *cut* $\sum_{i<j} \omega(E_{ij})$ where $E_{ij} := \{\{u, v\} \in E : u \in V_i, v \in V_j\}$. A vertex $v \in V_i$ that has a neighbor $w \in V_j, i \neq j$, is a boundary vertex. We denote by $\mathsf{nnzr}(A, i)$ and $\mathsf{nnzc}(A, i)$ the number of nonzero entries in the ith row or column of a matrix A, respectively.

A matching $M \subseteq E$ is a set of edges that do not share any common nodes; that is, the graph (V, M) has maximum degree one. *Contracting* an edge $\{u, v\}$ means replacing the nodes u and v by a new node x connected to the former neighbors of u and v, and setting $c(x) = c(u) + c(v)$. If replacing edges of the form $\{u, w\}$,$\{v, w\}$ would generate two parallel edges $\{x, w\}$, we insert a single edge with $\omega(\{x, w\}) = \omega(\{u, w\}) + \omega(\{v, w\})$. *Uncontracting* an edge e undoes its contraction.

Multilevel Graph Partitioning. In the multilevel framework we construct a hierarchy of decreasing-size graphs, $G_0,\ G_1, \ldots, G_k$, by *coarsening*, starting from the given graph G_0 such that each next-coarser graph G_i reflects structural properties of the previous graph G_{i-1}. At the coarsest level G_k is partitioned by a hybrid of external solvers, and starting from the $(k-1)$th level the solution is projected gradually (level by level) to the finest level. Each projection is followed by the *refinement*, which moves nodes between the blocks in order to reduce the size of the cut. This entire process is called a V-cycle. KaFFPa [9] extended the concept of *iterated multilevel algorithms* which was introduced for graph partitioning by Walshaw [11]. The main idea is to iterate coarsening

[1] Subscripts will be used for a short notation; i.e., ω_{ij} corresponds to the weight of $\{i, j\} \in E$.

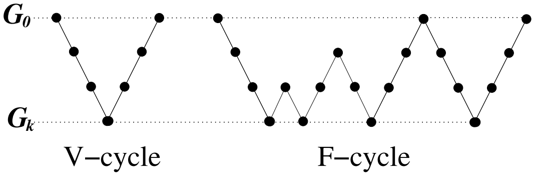


Fig. 1. V- and F-cycles schemes.

and uncoarsening taking into account the existing solution before next coarsening. Non-decreased partition quality is guaranteed at all stages of KaFFPa. In this paper, we consider also F-cycles [9] as a stronger and slower version of the multilevel framework for the graph partitioning problem.

3 Coarsening Schemes

One of the most important concerns of multilevel schemes is a measure of the connection strength between vertices. For matching-based coarsening schemes, experiments indicate that more sophisticated *edge rating* functions are superior to edge weight as a criterion for the matching algorithm [12]. To be more precise first the edges get rated using a rating function that indicates how much sense it makes to contract an edge. Then a matching algorithm tries to maximize the sum of the ratings of the edges to be contracted. The default configurations of KaFFPa employ the ratings

$$\begin{aligned}\text{expansion}^{*2}(\{u,v\}) &:= \omega(\{u,v\})^2/c(u)c(v), \text{ and}\\ \text{innerOuter}(\{u,v\}) &:= \omega(\{u,v\})/(\text{Out}(v)+\text{Out}(u)-2\omega(u,v)),\end{aligned}$$

where $\text{Out}(v) := \sum_{x\in\Gamma(v)}\omega(\{v,x\})$, since they yielded the best results in [12].

Algebraic Distance for Graph Partitioning. The notion of algebraic distance introduced in [8,13] is based on the principle of obtaining low-residual error components used in the Bootstrap AMG [14]. When a priori knowledge of the nature of this error is not available, slightly relaxed random vectors are used to approximate it. This principle was used for linear ordering problems to distinguish between local and global edges [8]. The main difference between the k-partitioning problem and other problems for which the algebraic distance has been tested so far is the balancing constraints. For many instances, it is important to keep the coarsening balanced; otherwise, even though the structural information will be captured by a sophisticated coarsening procedure, most of the actual computational work that constructs the approximate solution will be done by the refinement iterations. Bounding the number of refinement iterations may dramatically decrease its quality. Thus, a volume-normalized algebraic distance is introduced to take into account the balancing of vertices.

Given the Laplacian of a graph $L = D - W$, where W is a weighted adjacency matrix of a graph and D is the diagonal matrix with entries $D_{ii} = \sum_j \omega_{ij}$, we define its volume-normalized version denoted by $\tilde{L} = \tilde{D} - \tilde{W}$ based on volume-normalized edge weights $\tilde{\omega}_{ij} = \omega_{ij}/\sqrt{c(i)c(j)}$. We define an iteration matrix H for Jacobi over-relaxation (also known as a lazy random-walk matrix) as

$H = (1-\alpha)I + \alpha\tilde{D}^{-1}\tilde{W}$, where $0 \leq \alpha \leq 1$. The algebraic distance coupling ρ_{ij} is defined as $\rho_{ij} = \left(\sum_{r=1}^{R} |\chi_i^{(k,r)} - \chi_j^{(k,r)}|^2\right)^{\frac{1}{2}}$, where $\chi^{(k,r)} = H^k\chi^{(0,r)}$ is a relaxed randomly initialized test vector (i.e., $\chi^{(0,r)}$ is a random vector sampled over [-1/2, 1/2]), R is the number of test vectors, and k is the number of iterations. In our experimental settings we set $\alpha = 0.5$, $R = 5$, and $k = 20$.

3.1 Coarsening

To the best of our knowledge, the existing multilevel algorithms for combinatorial optimization problems (such as k-partitioning, linear ordering, clustering, and segmentation) can be divided into two classes: matching-based schemes [9] and algebraic multigrid (AMG)-inspired schemes [8].

AMG-inspired coarsening. One of the most traditional approaches for derivation of the coarse systems in AMG is the Galerkin operator, which projects the fine system of equations to the coarser scale. In the context of graphs this projection is defined as $L_c = PL_fP^T$, where L_f and L_c are the Laplacians of fine and coarse graphs $G_f = (V_f, E_f)$ and $G_c = (V_c, E_c)$, respectively. The (i, J)th entry of projection matrix P represents the strength of the connection between fine node i and coarse node J.

The coarsening begins by selecting a dominating set of (seed or coarse) nodes $C \subset V_f$ such that all other (fine) nodes in $F = V_f \setminus C$ are strongly coupled to C. This selection can be done by traversing all nodes and leaving node i in F (initially $F = V_f$, and $C = \emptyset$) that satisfy $\sum_{j\in C} 1/\rho_{ij} \geq \Theta \cdot \sum_{j\in V_f} 1/\rho_{ij}$, where Θ is a parameter of coupling strength.

The Galerkin operator construction differs from other AMG-based approaches for combinatorial optimization problems. Balancing constraints of the partitioning problem require a limited number of fine-to-coarse attractions between $i \in C$ (ith column in P) and its neighbors from F (nonzero entries in the ith column in P). In particular, this is important for graphs where the number of high-degree nodes in C is smaller than the number of parts in the desired partition. Another well-known problem of AMG that can affect the performance of the solver is the complexity of coarse levels. Consideration of the algebraic distance makes it possible to minimize the order of interpolation (the number of fractions a node from F can be divided to) to 1 or 2 only [8]. Algorithm 1 summarizes the construction of the ith row of P.

Algorithm 1 can be viewed as simplified version of bootstrap AMG [14] with the additional restriction on future volume of aggregates and adaptive interpolation order. $P_{iI(j)}$ thus represents the likelihood of i belonging to the $I(j)$th aggregate. The edge connecting two coarse aggregates p and q is assigned with the weight $w_{pq} = \sum_{k\neq l} P_{kp}w_{kl}P_{lq}$. The volume of the pth coarse aggregate is $\sum_j c(j)P_{jp}$. We emphasize the property of adaptivity of C (line 9 in Algorithm 1), which is updated if the balancing of aggregates is not satisfied.

Matching Based Coarsening. Another coarsening framework, which is more popular because of its simplicity and faster performance, is the *matching based scheme.* In this scheme a coarse graph is constructed by using contractions

Algorithm 1. Interpolation weights for P

```
input: $G$, $i \in V_f$, $P$
if $i \in C$ then $P_{iI(j)} \leftarrow 1$;
else
    $l \leftarrow$ list of at most $\kappa$ algebraically strongest connections of $i$ to $C$
    $\{e_1, e_2\} \leftarrow$ strongest pair of edges (w.r.t. $\rho_{e_1} + \rho_{e_2}$) in $l$ such that the
    $C$-neighbors are not over-loaded if $i$ is divided between them
    if $\{e_1, e_2\} \neq \emptyset$ then $l \leftarrow \{e_1, e_2\}$
    else $e_1 \leftarrow$ strongest connection of $i$ to $C$ such that the corresponding
    $C$-neighbor is not over-loaded if $i$ is aggregated with it; $l \leftarrow \{e_1\}$
    if $l$ is empty then move $i$ to $C$
    else $N_i^c \leftarrow C$-neighbors of $i$ that adjacent to edges in $l$
        $P_{iI(j)} \leftarrow 1/(\rho_{ij} \cdot \sum_{k \in N_i^c} 1/\rho_{ik})$ for $j \in N_i^c$
        update future volumes of $j \in N_i^c$
```

derived from a preprocessed edge matching. This scheme represents a special case of PL_fP^T in which $\mathsf{nnzr}(P, r) = 1$ for all rows r in P, and $1 \leq \mathsf{nnzc}(P, c) \leq 2$ for all columns c in P.

The *Global Paths Algorithm* (GPA), was proposed in [15] as a synthesis of Greedy and Path Growing algorithms. Experiments indicate that GPA yields better overall results in graph partitioning (w.r.t. final cut) than other matching algorithms [12]. Similar to the Greedy approach, GPA scans the edges in order of decreasing rating; but rather than immediately building a matching, it first constructs a collection of paths and even length cycles. It then computes optimal matchings on those using dynamic programming. For more details see [12].

The *RandomGPA Algorithm* is used by the KaFFPaEco configuration. It is a synthesis of the most simple random matching algorithm and the GPA algorithm. To be more precise this matching algorithm depends on the number of blocks the graph has to be partitioned in. It matches the first $\max\{2, 7 - \log k\}$ levels, where k is the number of blocks, using the random matching algorithm and switches to the GPA algorithm afterwards. The random matching algorithm traverses the nodes in a random order and if the current node is not already matched it chooses a random unmatched neighbor for the matching.

The Coarsest Level. Coarsening is terminated when the graph is small enough to be directly partitioned. We use the same initial partitioning scheme as in KaFFPa [9], namely, the libraries Scotch and Metis for initial partitioning. For AMG, some modifications have been made since Scotch and Metis cannot deal with fractional numbers and Metis expects $\omega_{ij} \geq 1$. To overcome this problem, we apply a scheme based on randomized rounding (for details see [1]).

3.2 Uncoarsening

Recall that uncoarsening undoes contraction. For AMG-based coarsening this means that fine nodes have to be assigned to blocks of the partition of the finer graph in the hierarchy. We assign a fine node v to the block that minimizes $\text{cut}_B \cdot p_B(v)$, where cut_B is the cut after v would be assigned to block B and $p_B(v)$ is a penalty function to avoid blocks that are heavily overloaded. To be more precise, after some experiments we fixed the penalty function to $p_B(v) = 2^{\max(0,100\frac{c(B)+c(v)}{L_{\max}})}$, where $L_{\max}$ is the upper bound for the block weight. Note that slight imbalances (e.g. overloaded blocks), can usually be fixed by the refinement algorithms implemented within KaFFPa. For matching-based coarsening the uncoarsening is straightforward: a vertex is assigned to the block of the corresponding coarse vertex.

Karlsruhe Fast Flow Partitioner (KaFFPa). Since we integrated different coarsening schemes into the multilevel graph partitioner KaFFPa [9], we give a very rough overview over the techniques KaFFPa uses during uncoarsening. After a matching is uncontracted, local search-based refinement algorithms move nodes between block boundaries in order to reduce the cut while maintaining the balancing constraint. These local improvement algorithms are usually variants of the FM algorithm [4]. KaFFPa additionally uses more advanced refinement algorithms. The first method is based on max-flow min-cut computations between pairs of blocks, in other words, a method to improve a given bipartition. Roughly speaking a max-flow min-cut algorithm is applied to an area around the initial cut of a bipartition. The second method for improving a given partition is called multi-try FM which is a FM variant achieving a very localized search (for details see [9]).

4 Experimental Evaluation

The AMG coarsening was implemented separately based on the coarsening for linear ordering solvers from [8] and was integrated into KaFFPa [9]. The computational experiments have been performed with six configurations of KaFFPa (see Table 1). All configurations use the FM algorithm and flows for the refinement. The strong configurations further employ flows using larger areas, multi-try FM and F-cycles. Throughout this section, because of the different running times, we concentrate on two groups of comparison: for fast versions (AMG-ECO, ECO, ECO-ALG) and for strong versions (AMG, STRONG, F-CYCLE). To be precise, usually the running time of F-CYCLE is bigger than that of STRONG and AMG. However, the running time gap between fast and strong versions is even more significant on the average. Since the main goal of this paper is to introduce the AMG coarsening with different uncoarsenings, most of the comparisons will be of type AMG vs respective non-AMG ratios (between corresponding averages over 10 trials for each configuration). A comprehensive comparison of the F-CYCLE and the STRONG configuration can be found in [9]. All experiments are performed with fixed imbalance factor 3%. Detailed results of all benchmarks presented here can be found in [16].

Benchmark I: Walshaw's Partitioning Archive. This collection of real-world instances [17] is the most popular graph partitioning benchmark in the literature. Over the years many different heuristics have been tested and adjusted on this

Table 1. Description of the six configurations used for the computational experiments

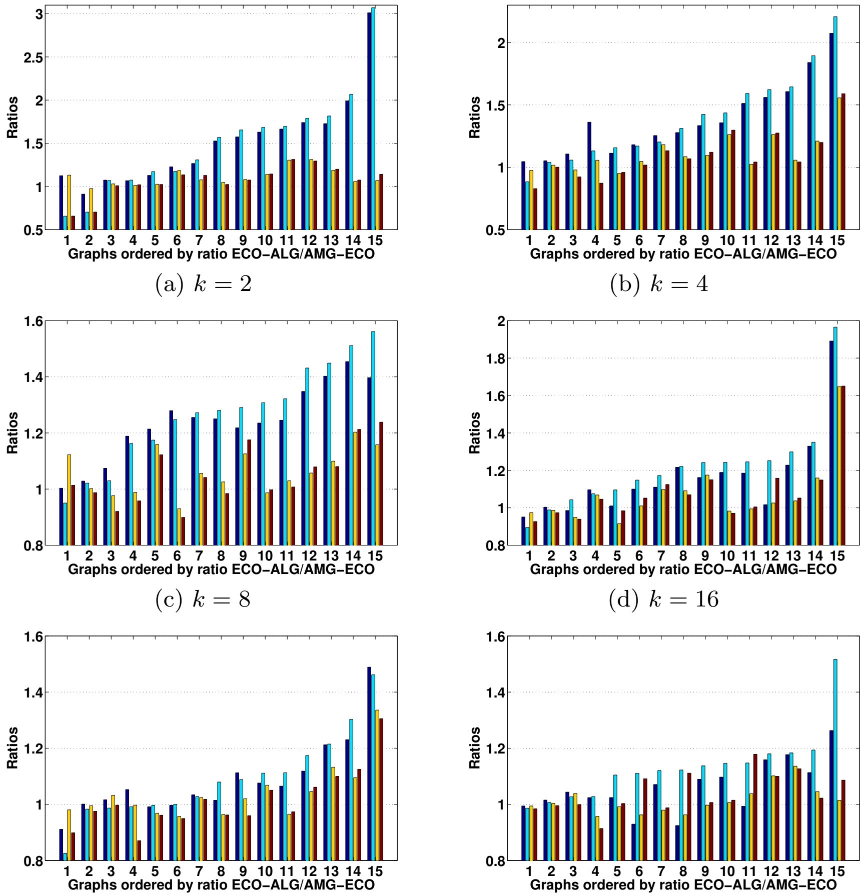
ECO	KaFFPaEco configuration for good trade-off of quality and runtime.
ECO-ALG	Same refinement as in ECO, coarsening uses the GPA algorithm at each level and the edge rating function employs algebraic distances; i.e., it uses the rating function ex_alg(e) := expansion$^{*2}(e)/\rho_e$.
F-CYCLE	KaFFPaStrong configuration with strong refinement schemes and the F-cycle scheme for high partition quality; this configuration achieved the best known partitions for many instances from Benchmark I in 2010 [9].
STRONG	Uses the same refinement and matching schemes as in the F-CYCLE configuration; however, here only one single V-cycle is performed.
AMG-ECO	AMG coarsening based on algebraic distances with interpolation order at most 2, refinement as in ECO.
AMG	Same coarsening as in AMG-ECO, same refinement as in STRONG.

benchmark, so that many heuristics are able to obtain good results on these graphs. In contrast to Benchmarks II and III, the comparison of our methods has not demonstrated surprisingly new results. Overall, we observed that uncoarsening performance of fast versions (ECO, ECO-ALG, AMG-ECO) are more or less similar to each other and algebraic distance can improve the quality. The uncoarsening of a STRONG V-cycle is somewhat slower than AMG because of the density of coarse levels. Full results are summarized in [1].

Benchmark II: Scale-free networks. In scale-free networks the distribution of vertex degrees asymptotically follows the power-law distribution. These types of networks often contain irregular parts and long-range links that can confuse both contraction and AMG coarsening schemes. Since Walshaw's benchmark doesn't contain graphs derived from such networks, we evaluate our algorithms on 15 graphs collected from [17,18].

Because of the large running time of the strong configurations on these graphs, we compare only the fast versions of AMG and matching-based coarsenings. The results of the comparison on scale-free graphs are presented in Figures 2 and Table 2. Each figure corresponds to a different number of blocks k. The horizontal axes represent graphs from the benchmark. The vertical axes are for ratios that represent comparison of averages of final results for a pair of methods. Each graph corresponds to one quadruple of bars. First, second, third and fourth bars represent averages of ratios ECO/AMG-ECO, ECO-ALG/AMG-ECO after finest refinement, ECO/AMG-ECO, ECO-ALG/AMG-ECO before finest refinement, respectively.

Benchmark III: Potentially Hard Graphs for Fast k-partitioning Algorithms. We present a simple strategy for checking the quality of multilevel schemes. To construct a potentially hard instance for gradual multilevel projections, we consider a mixture of graphs that are weakly connected with each other. These graphs have to possess different structural properties (such as finite-element faces, power-law degree distribution, and density) to ensure nonuniform coarsening and mutual aggregation of well-separated graph regions. Such mixtures of



(a) $k = 2$

(b) $k = 4$

(c) $k = 8$

(d) $k = 16$

(e) $k = 32$

(f) $k = 64$

Fig. 2. Comparison of coarsening schemes on scale-free graphs. Figures (a)-(f) contain results of comparison for k = 2, 4, 8, 16, 32, and 64, respectively. Each quadruple of bars correspond to one graph. First, second, third and fourth bars represent averages of ratios ECO/AMG-ECO, ECO-ALG/AMG-ECO after refinement, ECO/AMG-ECO, and ECO-ALG/AMG-ECO before refinement, respectively.

structures (e.g., multi-mode networks) may have a twofold effect. First, they can force the algorithm to contract incorrect edges; and second, they can attract a "too strong" refinement to reach a local optimum, which can contradict better local optimums at finer levels. The last situation has been observed in different variations also in multilevel linear ordering algorithms [19].

We created a benchmark (available through [17]) with potentially hard mixtures. Each graph in this benchmark represents a star-like structure of different graphs $S_0, \ldots, S_t$. Graphs $S_1, \ldots, S_t$ are weakly connected to the center S_0 by random edges. Since all the constituent graphs are sparse, a faster aggregation of them has been achieved by adding more than one random edge to each boundary

Table 2. Computational comparison for scale-free graphs

k	$\frac{\text{ECO}}{\text{ECO-ALG}}$ quality	$\frac{\text{ECO}}{\text{ECO-ALG}}$ full time	$\frac{\text{ECO}}{\text{ECO-ALG}}$ uncoarsening time	$\frac{\text{ECO-ALG}}{\text{AMG-ECO}}$ quality	$\frac{\text{ECO-ALG}}{\text{AMG-ECO}}$ uncoarsening time
2	1.38	0.77	1.62	1.16	3.62
4	1.24	1.32	1.85	1.11	2.14
8	1.15	1.29	1.45	1.07	1.94
16	1.09	1.27	1.33	1.06	1.69
32	1.06	1.18	1.23	1.00	1.60
64	1.06	1.13	1.13	1.01	2.99

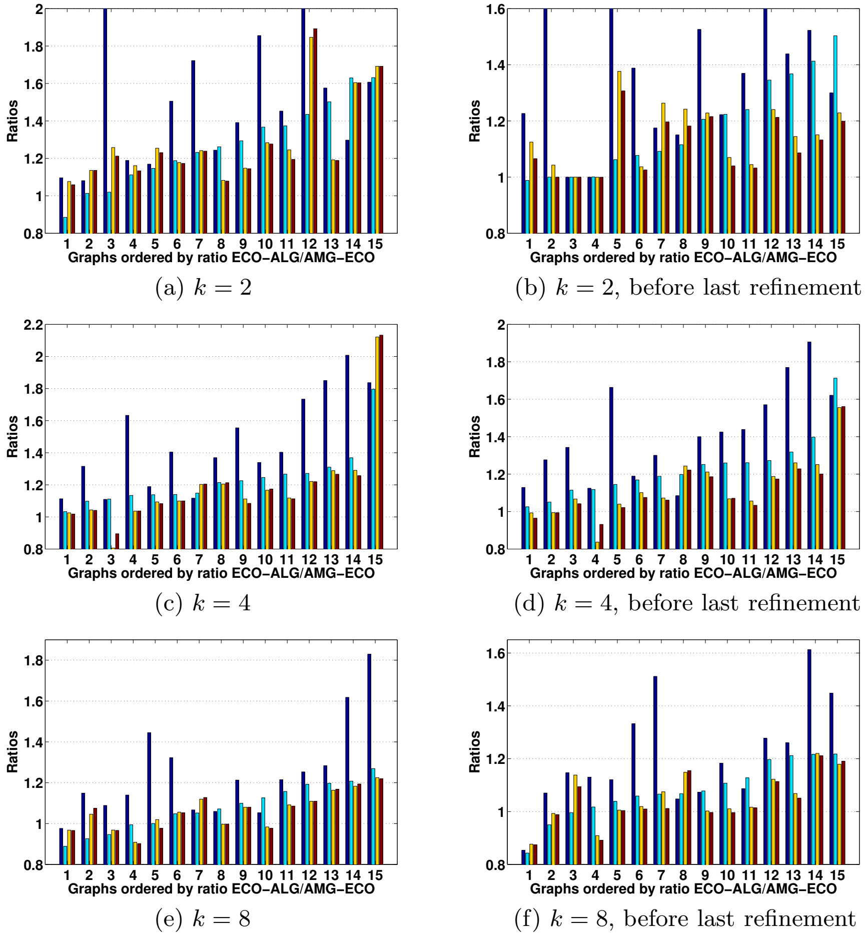
Table 3. Computational comparison for potentially hard graphs

k	$\frac{\text{ECO}}{\text{ECO-ALG}}$ quality	$\frac{\text{ECO}}{\text{ECO-ALG}}$ full time	$\frac{\text{ECO-ALG}}{\text{AMG-ECO}}$ quality	$\frac{\text{ECO-ALG}}{\text{AMG-ECO}}$ uncoarsening time	$\frac{\text{STRONG}}{\text{AMG}}$ quality	$\frac{\text{STRONG}}{\text{AMG}}$ uncoarsening time	$\frac{\text{F-CYCLE}}{\text{AMG}}$ quality
2	1.42	0.51	1.18	0.55	1.15	2.11	1.11
4	1.15	0.88	1.23	0.64	1.13	1.69	1.12
8	1.12	1.08	1.08	0.98	1.05	1.37	1.04

node. The total number of edges between each S_i and S_0 was less than 3% out of the total number of edges in S_i. We considered the mixtures of the following structures: social networks, finite-element graphs, VLSI chips, peer-to-peer networks, and matrices from optimization solvers.

The comparison on this benchmark is demonstrated in Figure 3. Each graph corresponds to one quadruple of bars. The first, second, third and the fourth bar represent averages over 10 ratios of ECO/AMG-ECO, ECO-ALG/AMG-ECO, STRONG/AMG, and F-cycle/AMG, respectively. In almost all experiments we observed that introduction of algebraic distance as a measure of connectivity plays a crucial role in both fast versions AMG-ECO and ECO-ALG since it helps to separate the subgraphs and postpone their aggregation into one mixture. We also observe that both fast and slow AMG coarsenings almost always lead to better results. Note that in contrast to Benchmarks I and II, the uncoarsening of ECO-ALG is significantly faster than that of AMG-ECO.

Role of the algebraic distance. In this work the importance of the algebraic distance as a measure of connectivity strength for graph partitioning algorithms has been justified in all experimental settings. In particular, the most significant gap was observed between ECO and ECO-ALG (see all benchmarks), versions which confirms preliminary experiments in [13], where the algebraic distance has been used at the finest level only. The price for improvement in the quality is the additional running time for Jacobi over-relaxation, which can be implemented by using the most suitable (parallel) matrix-vector multiplication method. However, in cases of strong configurations and/or large irregular instances, the difference in the running time becomes less influential as it is not comparable to the amount of work in the refinement phase. For example, for the largest graph in Benchmark I (auto, $|V| = 448695$, $|E| = 3314611$) the ECO coarsening is approximately



(a) $k = 2$

(b) $k = 2$, before last refinement

(c) $k = 4$

(d) $k = 4$, before last refinement

(e) $k = 8$

(f) $k = 8$, before last refinement

Fig. 3. Comparison of coarsening schemes on hard examples. Figures (a,c,e) contain results of comparison before applying finest level refinement. Figure (b,d,f) contain results of comparison of final results. Each quadruple of bars correspond to one graph. First, second, third and fourth bars represent averages of ratios ECO/AMG-ECO, ECO-ALG/AMG-ECO, STRONG/AMG, and F-cycle/AMG, respectively. Four exceptionally high ratios on both Figures are between 3.5 and 5.7.

10 times faster than that in the ECO-ALG; but for both configurations when $k = 64$, it takes less than 3% of the total time. Note that for irregular instances from Benchmark II, already starting $k = 4$ the total running time for ECO becomes bigger than in ECO-ALG (see Table 2).

Does AMG coarsening help? The positive answer to this question is given mostly by Benchmarks II and III, that contain relatively complex and irregular instances (Tables 2 and 3). On Benchmark III we have demonstrated that the AMG configuration is superior to F-CYCLE, which runs significantly longer. This result is in contrast to Benchmark I, in which we did not observe any particular class of graphs that corresponded to stable significant difference in favor of one of the methods in pairs ECO-ALG vs AMG-ECO and STRONG vs AMG. However, we note that in both Benchmarks I and II several graphs exhibited that AMG versions yield to the respective matching for large k. The problem is eliminated when we stabilize ρ by using more relaxations according to Theorem 4.2 in [8]. We cannot present here the exact comparison of coarsening running times because their underlying implementations are very different. Theoretically, however, if in both matching and AMG configurations the algebraic distance is used and when the order of interpolation in AMG is limited by 2 (and usually it is 1, meaning that the coarse graphs are not dense like in [10]), the exact complexity of AMG coarsening is not supposed to be bigger than that of matching.

5 Conclusions

We introduced a new coarsening scheme for multilevel graph partitioning based on the AMG coarsening and the algebraic distance connectivity measure. Both matching and AMG coarsening schemes have been compared under fast and strong configurations of refinement. In addition to known benchmarks, we introduced a new benchmark with potentially hard graphs for large-scale graph partitioning solvers (available through [17]). As the main conclusion of this work, we emphasize the success of the proposed AMG coarsening and the algebraic distance connectivity measure between nodes demonstrated on highly irregular instances. One has to take into account the trade-off between increased running time when using algebraic distance and improved quality of the partitions. The increasing running time becomes less tangible with growth of graph size compared with the complexity of the refinement phase.

References

1. Safro, I., Sanders, P., Schulz, C.: Advanced coarsening schemes for graph partitioning. Technical Report ANL/MCS-P2016-0112, Argonne National Laboratory (2012)
2. Bui, T.N., Jones, C.: Finding good approximate vertex and edge partitions is NP-hard. Inf. Process. Lett. 42(3), 153–159 (1992)
3. Pothen, A., Simon, H.D., Liou, K.P.: Partitioning sparse matrices with eigenvectors of graphs. SIAM J. Matrix Anal. Appl. 11(3), 430–452 (1990)
4. Fiduccia, C.M., Mattheyses, R.M.: A Linear-Time Heuristic for Improving Network Partitions. In: 19th Conference on Design Automation, pp. 175–181 (1982)
5. Sanders, P., Schulz, C.: Distributed Evolutionary Graph Partitioning. In: 12th Workshop on Algorithm Engineering and Experimentation (2011)
6. Schloegel, K., Karypis, G., Kumar, V.: Graph partitioning for high performance scientific simulations. In: Dongarra, J., et al. (eds.) CRPC Par. Comp. Handbook. Morgan Kaufmann (2000)
7. Pellegrini, F.: Scotch home page, `http://www.labri.fr/pelegrin/scotch`
8. Ron, D., Safro, I., Brandt, A.: Relaxation-based coarsening and multiscale graph organization. Multiscale Modeling & Simulation 9(1), 407–423 (2011)

9. Sanders, P., Schulz, C.: Engineering Multilevel Graph Partitioning Algorithms. In: Demetrescu, C., Halldórsson, M.M. (eds.) ESA 2011. LNCS, vol. 6942, pp. 469–480. Springer, Heidelberg (2011)
10. Chevalier, C., Safro, I.: Comparison of Coarsening Schemes for Multilevel Graph Partitioning. In: Stützle, T. (ed.) LION 3. LNCS, vol. 5851, pp. 191–205. Springer, Heidelberg (2009)
11. Walshaw, C.: Multilevel refinement for combinatorial optimisation problems. Annals of Operations Research 131(1), 325–372 (2004)
12. Holtgrewe, M., Sanders, P., Schulz, C.: Engineering a Scalable High Quality Graph Partitioner. In: 24th IEEE International Parallal and Distributed Processing Symposium (2010)
13. Chen, J., Safro, I.: Algebraic distance on graphs. SIAM Journal on Scientific Computing 33(6), 3468–3490 (2011)
14. Brandt, A.: Multiscale scientific computation: Review 2001. In: Barth, T., Haimes, R., Chan, T. (eds.) Proceeding of the Yosemite Educational Symposium on Multiscale and Multiresolution Methods. Springer (October 2000)
15. Maue, J., Sanders, P.: Engineering Algorithms for Approximate Weighted Matching. In: Demetrescu, C. (ed.) WEA 2007. LNCS, vol. 4525, pp. 242–255. Springer, Heidelberg (2007)
16. Safro, I., Sanders, P., Schulz, C.: Benchmark with Potentially Hard Graphs for Partitioning Problem, http://www.mcs.anl.gov/~safro/hardpart.html
17. Bader, D., Meyerhenke, H., Sanders, P., Wagner, D.: 10th DIMACS Implementation Challenge - Graph Partitioning and Graph Clustering, http://www.cc.gatech.edu/dimacs10/
18. Lescovec, J.: Stanford Network Analysis Package (SNAP), http://snap.stanford.edu/index.html
19. Safro, I., Ron, D., Brandt, A.: Multilevel algorithms for linear ordering problems. Journal of Experimental Algorithmics 13, 1.4–1.20 (2008)

A Heuristic for Non-convex Variance-Based Clustering Criteria

Rodrigo F. Toso[1], Casimir A. Kulikowski[1], and Ilya B. Muchnik[2]

[1] Department of Computer Science, Rutgers University, Piscataway, NJ 08854, USA
{rtoso,kulikows}@cs.rutgers.edu
[2] DIMACS, Rutgers University, Piscataway, NJ 08854, USA
muchnik@dimacs.rutgers.edu

Abstract. We address the clustering problem in the context of exploratory data analysis, where data sets are investigated under different and desirably contrasting perspectives. In this scenario where, for flexibility, solutions are evaluated by criterion functions, we introduce and evaluate a generalized and efficient version of the incremental one-by-one clustering algorithm of MacQueen (1967). Unlike the widely adopted two-phase algorithm developed by Lloyd (1957), our approach does not rely on the gradient of the criterion function being optimized, offering the key advantage of being able to deal with non-convex criteria. After an extensive experimental analysis using real-world data sets with a more flexible, non-convex criterion function, we obtained results that are considerably better than those produced with the k-means criterion, making our algorithm an invaluable tool for exploratory clustering applications.

1 Introduction

In clustering, one seeks to partition a given set $\mathcal{D} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ containing d-dimensional unlabeled samples into k nonempty subsets or clusters so as to aggregate samples that are most similar into a common cluster. In the variational approach to clustering [2], the quality of a solution (k-partition) is evaluated by a criterion function (or functional), and the optimization process consists in finding a k-partition that minimizes such functional. Under this framework, the most successful criteria are based on the sufficient statistics of each cluster $\mathcal{D}_i$, that is, their sample prior probabilities $\hat{p}_{\mathcal{D}_i}$, means $\hat{\boldsymbol{\mu}}_{\mathcal{D}_i}$, and variances $\hat{\sigma}^2_{\mathcal{D}_i}$, which yield not only mathematically motivated but also perceptually confirmable descriptions of the data. Examples of functionals include not only the widely studied minimum sum-of-squares clustering criterion $J_1 = \sum_{i=1}^{k} \hat{p}_{\mathcal{D}_i} \hat{\sigma}^2_{\mathcal{D}_i}$, but also variants rooted in the theory of sampling such as $J_2 = \sum_{i=1}^{k} \hat{p}_{\mathcal{D}_i} \hat{\sigma}_{\mathcal{D}_i}$, introduced by Neyman [14], and $J_3 = \sum_{i=1}^{k} \hat{p}^2_{\mathcal{D}_i} \hat{\sigma}^2_{\mathcal{D}_i}$, proposed by Kiseleva et al. [9]. The key difference between J_1 when compared to J_2 and J_3 is that the former only employs the first two sufficient statistics of the clusters to discriminate samples (namely, their priors and means), while the latter makes use of the estimated variances as well. As a result, the decision boundaries resulting from J_2 and J_3 are quadratic and thus more flexible than the linear boundaries produced by J_1.

R. Klasing (Ed.): SEA 2012, LNCS 7276, pp. 381–392, 2012.

There are plenty of algorithms for optimizing J_1, mostly local search methods. Along this line, perhaps the most successful one is the two-phase algorithm by Lloyd [11], commonly referred to as the k-means clustering algorithm. This gradient-based procedure was recently extended to work successfully with J_2 as well [12]. However, gradient-based optimization algorithms can only work with convex functionals, a fact that holds true for both J_1 and J_2, but fails for J_3, recently shown to be non-convex [12]. Even though there exist more complex algorithms that provide performance guarantees for J_3 (see e.g., [17,6]), to our knowledge, no implementations validating such approaches have been reported in the literature, perhaps due to their inherent complexities.

This paper is motivated by the lack of a simple but yet fairly efficient local search procedure with the added support for non-convex criteria such as J_3. Hence, we propose an efficient iterative implementation of the one-by-one clustering procedure of MacQueen [13] where each sample is greedily assigned to the cluster that most improves the current functional value, with the cluster statistics being updated before the next sample is considered. This process is iterated until a predefined convergence condition is met. The main advantage of this approach is the ability to deal with functionals for which convexity does not hold. Throughout the minimization of non-convex criteria with gradient-based methods including the two-phase procedure, the actual criterion may increase when a sample is moved to the cluster whose membership function value (i.e., the gradient of the functional) is minimized. In fact, this is likely to happen in early iterations [12], resulting in poor solutions. Working directly on the functional avoids this problem since the procedure is monotonically decreasing and does not rely on convexity.

We address the concerns about efficiency by showing that our implementation of the one-by-one approach offers the same theoretical running time found in a common, straightforward implementation of the two-phase method. To validate our algorithm in practice, we first briefly discuss an experimental comparison with the two-phase procedure, where our contribution is found slower though slightly more accurate in the majority of the data sets adopted. Next, our approach is set to optimize the non-convex criterion J_3, which we show that can offer interesting alternative interpretations of the data when compared to J_1. This way, our algorithm has the potential to be an invaluable tool in exploratory data analysis.

Our paper is organized as follows. In the next section, we establish the notation and discuss the background of our work. Then, we introduce our iterative one-by-one heuristic in Sec. 3. The experimental evaluation is conducted in Sec. 4, and in Sec. 5 we offer a summary and future research possibilities.

2 Notation and Background

Henceforth, we represent a k-partition $\{\mathcal{D}_1, \ldots, \mathcal{D}_k\}$ by the set of functions

$$h_{\mathcal{D}_i}(\mathbf{x}) = \begin{cases} 1, \text{if } \mathbf{x} \in \mathcal{D}_i \\ 0, \text{otherwise}, \end{cases} \tag{1}$$

where $i = 1, \ldots, k$. Additionally, $h_{\mathcal{D}_i}(\mathbf{x}) = 1 \iff h_{\mathcal{D}_j}(\mathbf{x}) = 0$ for all $j \neq i$. With the number of samples in a given cluster $\mathcal{D}_i$ denoted by $n_{\mathcal{D}_i} = \sum_{\mathbf{x} \in \mathcal{D}_i} h_{\mathcal{D}_i}(\mathbf{x})$, the prior probability of $\mathcal{D}_i$ is estimated as $\hat{p}_{\mathcal{D}_i} = \frac{n_{\mathcal{D}_i}}{n}$. By employing the first and second sample central moments

$$\mathcal{M}^{(1)}_{\mathcal{D}_i} = \hat{\boldsymbol{\mu}}_{\mathcal{D}_i} = \frac{1}{n_{\mathcal{D}_i}} \sum_{\mathbf{x} \in \mathcal{D}_i} \mathbf{x}, \text{ and}$$

$$\mathcal{M}^{(2)}_{\mathcal{D}_i} = \frac{1}{n_{\mathcal{D}_i}} \sum_{\mathbf{x} \in \mathcal{D}_i} ||\mathbf{x}||^2,$$

respectively, the sample cluster variance is computed as $\hat{\sigma}^2_{\mathcal{D}_i} = \mathcal{M}^{(2)}_{\mathcal{D}_i} - ||\mathcal{M}^{(1)}_{\mathcal{D}_i}||^2$. Note that $\hat{p}_{\mathcal{D}_i} \in \mathbb{R}$, $\hat{\boldsymbol{\mu}}_{\mathcal{D}_i} \in \mathbb{R}^d$, and $\hat{\sigma}^2_{\mathcal{D}_i} \in \mathbb{R}$, for all $i = 1, \ldots, k$.

In the variational approach to clustering [2], functionals can be written as

$$J(\hat{p}_{\mathcal{D}_1}, \hat{\boldsymbol{\mu}}_{\mathcal{D}_1}, \hat{\sigma}^2_{\mathcal{D}_1}, \ldots, \hat{p}_{\mathcal{D}_k}, \hat{\boldsymbol{\mu}}_{\mathcal{D}_k}, \hat{\sigma}^2_{\mathcal{D}_k}) = \sum_{i=1}^{k} J_{\mathcal{D}_i}(\hat{p}_{\mathcal{D}_i}, \hat{\boldsymbol{\mu}}_{\mathcal{D}_i}, \hat{\sigma}^2_{\mathcal{D}_i}). \tag{2}$$

Among the functionals derived from Eq. 2 is the minimum sum-of-squares clustering criterion, defined as

$$\begin{aligned} J_1 &= \sum_{i=1}^{k} \hat{p}_{\mathcal{D}_i} \hat{\sigma}^2_{\mathcal{D}_i} & (3) \\ &= \sum_{i=1}^{k} \sum_{\mathbf{x} \in \mathcal{D}_i} ||\mathbf{x} - \hat{\boldsymbol{\mu}}_{\mathcal{D}_i}||^2, & (4) \end{aligned}$$

where Eq. 4 follows from [12]. While Eq. 4 explicitly denotes the similarity of a sample with respect to a cluster through a membership function, Eq. 3 quantifies the similarity of each cluster directly. The former is in fact the gradient of the latter, which, in this case, is convex [2]. In J_1, the separating hyperplane (also known as decision boundary) between two clusters $\mathcal{D}_i$ and $\mathcal{D}_j$ with respect to a sample $\mathbf{x}$ is given by the linear equation

$$||\mathbf{x} - \hat{\boldsymbol{\mu}}_{\mathcal{D}_i}||^2 - ||\mathbf{x} - \hat{\boldsymbol{\mu}}_{\mathcal{D}_j}||^2 = 0. \tag{5}$$

The so called k-means clustering algorithm is perhaps the most studied local search heuristic for the minimization of J_1. In fact, k-means usually refers to (variants of) one of the following two algorithms. The first is an iterative two-phase procedure due to Lloyd [11] that is initialized with a k-partition and operates in two alternating phases: (1) given the set of samples $\mathcal{D}$ and the k means (centers) representing the current clusters, it reassigns each sample to the closest center; and (2) with the resulting updated k-partition of $\mathcal{D}$, it updates the centers. This process is iteratively executed until a stopping condition is met. An efficient implementation of this algorithm is studied in [8]. The second variant is an incremental one-by-one procedure which utilizes the first k samples of $\mathcal{D}$ as the cluster centers. Each subsequent sample is assigned to the closest center, which is then updated to reflect the change. This procedure was introduced by MacQueen [13] and is a single-pass, online procedure, i.e., samples are considered only once. In [18], an efficient iterative variant of this approach was given. A comprehensive survey on the origins and variants of k-means clustering algorithms can be found in [3].

The main difference between the two approaches above is when the cluster centers are updated: in a separate phase, after all the samples have been considered (two-phase), or sample after sample (one-by-one). It is here that the main drawbacks of the one-by-one method appear. The computational time required to update the cluster centers after each sample is analyzed may be prohibitively large, especially in medium to large-sized data sets. Another concern is clustering quality, since its ability to escape from local minima has also been questioned [5]. In contrast, significant improvements have been made in the two-phase algorithm when tied with J_1, such as tuning it to run faster [8,15], to be less susceptible to local minima [4], or to be more general [10,16].

We now discuss two alternative criteria whose key characteristic lies on their separating hyperplanes. The first, initially proposed by Neyman [14] for one-dimensional sampling, was recently generalized to multidimensional data [12] and is defined as

$$J_2 = \sum_{i=1}^{k} \hat{p}_{\mathcal{D}_i} \sqrt{\hat{\sigma}^2_{\mathcal{D}_i}}. \tag{6}$$

The decision boundaries produced by criterion J_2 are given by

$$\left[\frac{\hat{\sigma}_{\mathcal{D}_i}}{2} + \frac{1}{2\hat{\sigma}_{\mathcal{D}_i}}(||\mathbf{x} - \hat{\boldsymbol{\mu}}_{\mathcal{D}_i}||^2)\right] - \left[\frac{\hat{\sigma}_{\mathcal{D}_j}}{2} + \frac{1}{2\hat{\sigma}_{\mathcal{D}_j}}(||\mathbf{x} - \hat{\boldsymbol{\mu}}_{\mathcal{D}_j}||^2)\right] = 0. \tag{7}$$

Since convexity holds for J_2, successful experimental progress has been made in the application of the two-phase procedure with such criterion [12].

The second criterion, introduced by Kiseleva et al. [9], is written as

$$J_3 = \sum_{i=1}^{k} \hat{p}^2_{\mathcal{D}_i} \hat{\sigma}^2_{\mathcal{D}_i}, \tag{8}$$

and discriminates according with the following decision boundaries:

$$\left[\hat{p}^2_{\mathcal{D}_i}\hat{\sigma}^2_{\mathcal{D}_i} + \hat{p}^2_{\mathcal{D}_i}(||\mathbf{x} - \hat{\boldsymbol{\mu}}_{\mathcal{D}_i}||^2)\right] - \left[\hat{p}^2_{\mathcal{D}_j}\hat{\sigma}^2_{\mathcal{D}_j} + \hat{p}^2_{\mathcal{D}_j}(||\mathbf{x} - \hat{\boldsymbol{\mu}}_{\mathcal{D}_j}||^2)\right] = 0. \tag{9}$$

This criterion was extended to multidimensional clustering [17], and a recent manuscript has shown that it is non-convex, thus rendering the two-phase gradient approach useless [12]. Next, we introduce a local search heuristic that can tackle J_3 as well as any variance-based clustering criteria.

3 Heuristic for Variance-Based Criteria

In this section, we present our iterative one-by-one algorithm together with data structures that allow for a fast evaluation of the change in the criterion function when a sample is considered as a member of another cluster. The algorithm minimizes any additive criterion function specializing Eq. 2, and is a generalized variant of the k-means algorithm for J_1 studied in [18].

Before introducing the main algorithm, we first present an efficient procedure to update a given functional value to reflect the case where an arbitrary sample $\mathbf{x} \in \mathcal{D}_j$ is reassigned to cluster $\mathcal{D}_i$ ($i \neq j$). In Algorithm 1, we use the notation $J^{(\mathbf{x}\to\mathcal{D}_i)}$ to indicate that $\mathbf{x}$, currently in cluster $\mathcal{D}_j$, will be considered in cluster $\mathcal{D}_i$. For efficiency, similarly to [18,5], we maintain the unnormalized statistics of each cluster, namely $n_{\mathcal{D}_i} = \sum_{\mathbf{x}\in\mathcal{D}} h_{\mathcal{D}_i}(\mathbf{x})$, $\mathbf{m}_{\mathcal{D}_i} = \sum_{\mathbf{x}\in\mathcal{D}_i} \mathbf{x}$, and $s^2_{\mathcal{D}_i} = \sum_{\mathbf{x}\in\mathcal{D}_i} ||\mathbf{x}||^2$. Such equations can be efficiently updated when a sample is moved from one cluster to another. Further, it is straightforward to compute the estimated priors, means, and variances of any cluster given these auxiliary statistics (see Sec. 2). We note that in [18,5], only $n_{\mathcal{D}_i}$ needs to be maintained given that their algorithms are tied with J_1.

Algorithm 1. Computes $J^{(\mathbf{x}\to\mathcal{D}_i)}$

```
Input: sample $\mathbf{x} \in \mathcal{D}_j$, target cluster $\mathcal{D}_i$, current criterion value $J^*$, and cluster statis-
    tics: $n_{\mathcal{D}_j}$, $\mathbf{m}_{\mathcal{D}_j}$, $s^2_{\mathcal{D}_j}$, $\hat{p}_{\mathcal{D}_j}$, $\hat{\boldsymbol{\mu}}_{\mathcal{D}_j}$, $\hat{\sigma}^2_{\mathcal{D}_j}$, $n_{\mathcal{D}_i}$, $\mathbf{m}_{\mathcal{D}_i}$, $s^2_{\mathcal{D}_i}$, $\hat{p}_{\mathcal{D}_i}$, $\hat{\boldsymbol{\mu}}_{\mathcal{D}_i}$, and $\hat{\sigma}^2_{\mathcal{D}_i}$.
1: Let $n'_{\mathcal{D}_j} := n_{\mathcal{D}_j} - 1$ and $n'_{\mathcal{D}_i} := n_{\mathcal{D}_i} + 1$.
2: Let $\mathbf{m}'_{\mathcal{D}_j} := \mathbf{m}_{\mathcal{D}_j} - \mathbf{x}$ and $\mathbf{m}'_{\mathcal{D}_i} := \mathbf{m}_{\mathcal{D}_i} + \mathbf{x}$.
3: Let $(s^2_{\mathcal{D}_j})' := s^2_{\mathcal{D}_j} - ||\mathbf{x}||^2$ and $(s^2_{\mathcal{D}_i})' := s^2_{\mathcal{D}_i} + ||\mathbf{x}||^2$.
4: Let $\hat{p}'_{\mathcal{D}_j} := \frac{n'_{\mathcal{D}_j}}{n}$ and $\hat{p}'_{\mathcal{D}_i} := \frac{n'_{\mathcal{D}_i}}{n}$.
5: Let $\hat{\boldsymbol{\mu}}'_{\mathcal{D}_j} := \frac{1}{n'_{\mathcal{D}_j}}\mathbf{m}_{\mathcal{D}'_j}$ and $\hat{\boldsymbol{\mu}}'_{\mathcal{D}_i} := \frac{1}{n'_{\mathcal{D}_i}}\mathbf{m}'_{\mathcal{D}_i}$.
6: Let $(\hat{\sigma}^2_{\mathcal{D}_j})' := \frac{1}{n'_{\mathcal{D}_j}}(s^2_{\mathcal{D}_j})' - ||\hat{\boldsymbol{\mu}}'_{\mathcal{D}_j}||^2$ and $(\hat{\sigma}^2_{\mathcal{D}_i})' := \frac{1}{n'_{\mathcal{D}_i}}(s^2_{\mathcal{D}_i})' - ||\hat{\boldsymbol{\mu}}'_{\mathcal{D}_i}||^2$.
7: Compute $J^{(\mathbf{x}\to\mathcal{D}_i)}$ with the updated statistics for clusters $\mathcal{D}_i$ and $\mathcal{D}_j$.
```

In this procedure, the auxiliary statistics are updated in lines 1–3 in $\Theta(d)$ time. Then, the sufficient statistics are computed in lines 4–6, again in $\Theta(d)$ time. Finally, in line 7, assuming that $J_{\mathcal{D}_i}(\hat{p}_{\mathcal{D}_i}, \hat{\boldsymbol{\mu}}_{\mathcal{D}_i}, \hat{\sigma}^2_{\mathcal{D}_i})$ can be evaluated in $\Theta(1)$ for all $i = 1, \ldots, k$, the functional J^* is updated in $\Theta(1)$. As an example, for J_1, we have that $J_1^{(\mathbf{x}\to\mathcal{D}_i)} = J_1^* - \hat{p}_{\mathcal{D}_i}\hat{\sigma}^2_{\mathcal{D}_i} - \hat{p}_{\mathcal{D}_j}\hat{\sigma}^2_{\mathcal{D}_j} + \hat{p}'_{\mathcal{D}_i}(\hat{\sigma}^2_{\mathcal{D}_i})' + \hat{p}'_{\mathcal{D}_j}(\hat{\sigma}^2_{\mathcal{D}_j})'$. This shows that Algorithm 1 runs in $\Theta(d)$. Correctness follows from Sec. 2.

Let us now present our clustering heuristic in Algorithm 2. The procedure is initialized with a k-partition that is used to compute the auxiliary and the sample statistics in lines 1 and 2, respectively. In the main loop (lines 5–17), every sample $\mathbf{x} \in \mathcal{D}$ is considered as follows: Algorithm 1 is used to assess the functional value when the current sample is tentatively moved to each cluster $\mathcal{D}_1, \ldots, \mathcal{D}_k$ (lines 6–8). If there exists a cluster $\mathcal{D}_{min}$ for which the objective

Algorithm 2. Minimizes a clustering criterion function.

```
Input: an initial $k$-partition (Equation 1).
Compute $n_{\mathcal{D}_i}$, $\mathbf{m}_{\mathcal{D}_i}$, and $s^2_{\mathcal{D}_i}$ $\forall\ i = 1, \dots, k$.
Compute $\hat{p}_{\mathcal{D}_i}$, $\hat{\boldsymbol{\mu}}_{\mathcal{D}_i}$, and $\hat{\sigma}^2_{\mathcal{D}_i}$ $\forall\ i = 1, \dots, k$.
Set $J^* := J(\hat{p}_{\mathcal{D}_1}, \hat{\boldsymbol{\mu}}_{\mathcal{D}_1}, \hat{\sigma}^2_{\mathcal{D}_1}, \dots, \hat{p}_{\mathcal{D}_k}, \hat{\boldsymbol{\mu}}_{\mathcal{D}_k}, \hat{\sigma}^2_{\mathcal{D}_k})$.
while convergence criterion not reached do
  for all $\mathbf{x} \in \mathcal{D}$ do
    for all $i \mid h_{\mathcal{D}_i}(\mathbf{x}) = 0$ do
      Compute $J^{(\mathbf{x} \to \mathcal{D}_i)}$ via Algorithm 1.
    end for
    if $\exists\ i \mid J^{(\mathbf{x} \to \mathcal{D}_i)} < J^*$ then
      Let $min = i \mid \min_i J^{(\mathbf{x} \to \mathcal{D}_i)}$.     (i.e., $\mathbf{x} \to \mathcal{D}_{min}$ mostly improves $J^*$.)
      Let $j = i \mid h_{\mathcal{D}_i}(\mathbf{x}) = 1$.     (i.e., $\mathcal{D}_j$ is the current cluster of $\mathbf{x}$.)
      Set $h_{\mathcal{D}_{min}}(\mathbf{x}) := 1$ and $h_{\mathcal{D}_j}(\mathbf{x}) := 0$.     (i.e., assign $\mathbf{x}$ to cluster $\mathcal{D}_{min}$.)
      Update: $n_{\mathcal{D}_{min}}$, $n_{\mathcal{D}_j}$, $\mathbf{m}_{\mathcal{D}_{min}}$, $\mathbf{m}_{\mathcal{D}_j}$ $s^2_{\mathcal{D}_{min}}$, $s^2_{\mathcal{D}_j}$.
      Update: $\hat{p}_{\mathcal{D}_{min}}$, $\hat{p}_{\mathcal{D}_j}$, $\hat{\boldsymbol{\mu}}_{\mathcal{D}_{min}}$, $\hat{\boldsymbol{\mu}}_{\mathcal{D}_j}$, $\hat{\sigma}^2_{\mathcal{D}_{min}}$, $\hat{\sigma}^2_{\mathcal{D}_j}$.
      Set $J^* := J^{(\mathbf{x} \to \mathcal{D}_{min})}$.
    end if
  end for
end while
```

function can be improved, the sample is reassigned to such cluster and all the statistics are updated. The algorithm stops when a convergence goal is reached.

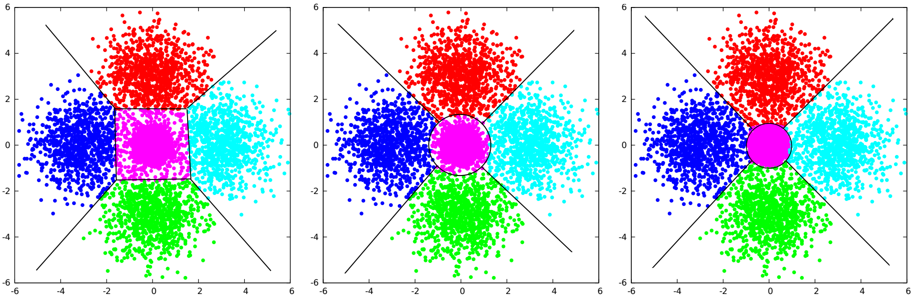
Because the procedure is monotonically decreasing, convergence is guaranteed. The running time to execute one iteration of Algorithm 2 is $\Theta(nkd)$, the same than an iteration of a simple two-phase procedure [5]. Concerns with running times of our heuristic are justifiable because: (1) the procedures based on the two-phase approach are tuned to a particular functional (J_1) that does not rely on the variances of each cluster; and (2) they do not update the cluster statistics on the fly (i.e., after considering each sample). In a final note, procedures adopting the one-by-one approach are reportedly more susceptible to local minima [5]. We address these concerns in the following empirical study.

4 Experimental Results

In this section, we validate Algorithm 2, henceforth denoted `one-by-one`, and its effectiveness in optimizing the non-convex criterion J_3, through a series of computational experiments focused on the quality of the solutions and the computational time needed to obtain them. All algorithms were coded in C++, compiled with `g++` version 4.1.2, and run on a single 2.3 GHz CPU with 128 GBytes of RAM. The algorithms were stopped whenever no sample changed cluster in a given iteration.

In a preamble, we compared our algorithm with a simple implementation of the two-phase approach (the so called k-means procedure, but offering the flexibility of working with any convex membership function such as those from J_1 or J_2). This first step indicated a tradeoff between quality and running time, with `one-by-one` outperforming the simple two-phase algorithm in the first, though by a small margin, but being outperformed in the latter. The results for clustering quality were unexpected, since in the vast majority of data sets the procedures found surprisingly similar results, contradicting previous reports that the one-by-one approach was more prone to local minima. As for the computational times, the two-phase implementation ran about 2.5 times faster than `one-by-one` in real-world data sets. These results are more carefully presented in Appendix 5.

After having shown that the optimization performance of `one-by-one` is comparable with (and even slightly better than) that of the two-phase approach, we now move on to the main reason behind our work on this implementation: criterion J_3, a promising non-convex functional that, to our knowledge, was never empirically studied before due to the lack of an appropriate clustering algorithm.



(a) Criterion J_1. **(b)** Criterion J_2. **(c)** Criterion J_3.

Fig. 1. Decision boundaries for a mixture of five equiprobable Gaussian distributions. The central cluster has a quarter of the variance of the external clusters.

To begin, we used `one-by-one` to provide a visual comparison between the three criteria mentioned in this paper, illustrating in Fig. 1 how their decision boundaries discriminate samples. Here, J_2 and J_3 built quadratic boundaries around the central cluster (of smaller variance) and linear hyperplanes between the external clusters (of the same variance), since Eqs. 7 and 9 become linear when $\hat{\sigma}^2_{\mathcal{D}_i} = \hat{\sigma}^2_{\mathcal{D}_j}$. For J_1, all boundaries are linear and thus unable to provide a proper discrimination for the central cluster.

4.1 Clustering Quality Analysis

In the subsequent experiments, we selected twelve real-world *classification* data sets (those with available class labels) from the UCI Machine Learning Repository [1] having fairly heterogeneous parameters as shown in Table 1.

To collect evidence backing `one-by-one` as a good local search procedure for variance-based criteria in general, we ran the algorithm on 1,000 randomly generated initial k-partitions and stored both the initial (random) and the final (optimized) criterion value for each run. For J_1, the ratio between the initial and local optimal solution was 0.4131; for J_2, 0.5748, and for J_3, 0.3640. Though no conclusion can be made about the effectiveness of each criterion in particular, it is clear that `one-by-one` effectively reduces the criterion value over its course.

Next, the available class labels from the selected data sets were used in order to compare the functionals under the following measures of clustering quality: accuracy, widely adopted by the classification community, and the Adjusted Rand Index or ARI [7], a pair-counting measure adjusted for chance that is extensively adopted by the clustering community. (See Vinh et al. [19].)

Table 1. Description and solution quality for real-world data sets obtained from the UCI Repository [1]. Quality measures are averaged over 1,000 runs with random initial k-partitions.

Dataset	Parameters			Accuracy			Adjusted Rand Index		
	k	d	n	J_1	J_2	J_3	J_1	J_2	J_3
arcene	2	10000	200	0.6191	0.6173	**0.6750**	0.0559	0.0536	**0.1180**
breast-cancer	2	30	569	0.8541	0.8735	**0.8770**	0.4914	0.5502	**0.5613**
credit	2	42	653	0.5513	**0.5865**	0.5819	0.0019	**0.0226**	0.0193
inflammations	4	6	20	0.6773	0.6606	**0.7776**	0.4204	0.4008	**0.6414**
internet-ads	2	1558	2359	**0.8953**	0.8279	0.7961	**0.4975**	0.3434	0.2771
iris	3	4	150	**0.8933**	**0.8933**	**0.8933**	**0.7302**	**0.7302**	0.7282
lenses	2	6	24	**0.6036**	0.6011	0.6012	0.0346	0.0326	**0.0382**
optdigits	10	64	5619	0.7792	0.7702	**0.7959**	0.6619	0.6498	**0.6810**
pendigits	10	16	10992	0.6857	0.6960	**0.7704**	0.5487	0.5746	**0.6155**
segmentation	7	19	2310	0.5612	0.5516	**0.5685**	0.3771	0.3758	**0.4028**
spambase	2	57	4601	0.6359	**0.6590**	0.6564	0.0394	**0.0773**	0.0726
voting	2	16	232	**0.8966**	0.8875	0.8865	**0.6274**	0.5988	0.5959
Average				0.7211	0.7187	**0.7400**	0.3739	0.3675	**0.3959**
Wins				4	3	7	3	3	7

In Table 1, we note that J_3 significantly outperforms both J_1 and J_2 on average, being about 2% better than its counterparts in both quality measures. Although we chose not to display the individual standard deviations for each data set, the average standard deviation in accuracy across all datasets was 0.0294, 0.0291, and 0.0276 for J_1, J_2, and J_3 respectively; for ARI, 0.0284, 0.0282, and 0.0208 respectively. In this regard, J_3 also offered a more stable operation across the different initial solutions.

4.2 Runtime Analysis

The purpose of this section is twofold: to assess the running time of `one-by-one` on real-world problems, and to check whether different clustering criteria have any impact in the running time of `one-by-one`. In Table 2, we report both the average running time, in milliseconds, to optimize each criterion function, and the average number of iterations completed upon termination.

Table 2. Average running times (ms) until convergence, for real-world data, reported as the average over 1,000 runs with random initial k-partitions.

Dataset	Parameters			Running times			Iterations		
	k	d	n	J_1	J_2	J_3	J_1	J_2	J_3
arcene	2	10000	200	139.66	153.90	208.03	2.427	2.715	3.724
breast-cancer	2	30	569	2.48	3.16	2.07	4.000	4.376	3.022
credit	2	42	653	5.55	4.24	2.86	6.000	4.066	3.006
inflammations	4	6	20	0.59	0.58	0.94	4.352	4.513	8.652
internet-ads	2	1558	2359	572.91	855.59	458.24	5.075	7.936	4.000
iris	3	4	150	0.30	0.47	0.34	3.670	3.761	4.123
lenses	2	6	24	0.02	0.04	0.04	2.488	2.472	2.649
optdigits	10	64	5619	1562.97	1475.40	2020.39	15.812	14.575	20.040
pendigits	10	16	10992	811.74	959.45	611.11	15.671	17.647	11.915
segmentation	7	19	2310	126.19	112.43	102.56	15.585	13.244	12.596
spambase	2	57	4601	84.57	52.29	36.65	9.000	5.349	3.519
voting	2	16	232	0.95	0.81	0.62	4.039	4.456	3.501

Even for larger datasets such as optdigits, pendigits, internet-ads, and segmentation, the running times have barely reached the two-second mark; in fact, only for optdigits the algorithm needed more than a second to converge. Thus, we are confident that `one-by-one` can be used in real-world clustering applications. It is important to note that we saw no significant change in the running time when a different clustering criteria was selected. The average number of iterations spent before convergence also corroborates to our conclusions.

5 Summary and Future Research

We have proposed a local search heuristic for the clustering problem of minimizing additive criterion functions taking as parameters the sample priors, means, and variances of each cluster. The main contribution of this paper is an algorithm that can effectively handle both convex and non-convex clustering criteria.

The theoretical running time of our accelerated technique is the same achieved by the widely studied two-phase procedure, though preliminary results on both synthetic and real-world data sets have indicated a tradeoff between running time and clustering quality: the simple two-phase implementation was considerably faster, but found marginally worse solutions in terms of quality.

However, because gradient-based algorithms like those based on the two-phase approach are not suitable for non-convex criteria, to our knowledge, we have introduced the first local search procedure designed to optimize functionals like J_3. Moreover, with such algorithm we were able to show that J_3 is indeed very promising, providing outstanding results for data sets of heterogeneous real-world applications including digit recognition, image segmentation, and discovery of medical conditions.

Future research paths include a more extensive experimentation with functionals J_2 and J_3 to better understand their strengths and weaknesses. The proposed method is also a natural choice for applications that might benefit from new additive variations of J_1, J_2, and J_3, and also for criteria with additional penalty factors or regularization terms.

References

1. Asuncion, A., Newman, D.J.: UCI Machine Learning Repository (2009)
2. Bauman, E.V., Dorofeyuk, A.A.: Variational approach to the problem of automatic classification for a class of additive functionals. Automation and Remote Control 8, 133–141 (1978)
3. Bock, H.-H.: Origins and extensions of the k-means algorithm in cluster analysis. Electronic Journal for History of Probability and Statistics 4(2) (2008)
4. Bradley, P.S., Fayyad, U.M.: Refining initial points for k-means clustering. In: Proceedings of the 15th International Conference on Machine Learning, pp. 91–99. Morgan Kaufmann Publishers Inc. (1998)
5. Duda, R.O., Hart, P.E., Storck, D.G.: Pattern Classification, 2nd edn. Wiley Interscience (2000)
6. Efros, M., Schulman, L.J.: Deterministic clustering with data nets. Technical Report 04-050, Electronic Colloquium on Computational Complexity (2004)
7. Hubert, L., Arabie, P.: Comparing partitions. Journal of Classification 2, 193–218 (1985)
8. Kanungo, T., Mount, D.M., Netanyahu, N.S., Piatko, C.D., Silverman, R., Wu, A.Y.: An efficient k-means clustering algorithm: analysis and implementation. IEEE Transactions on Pattern Analysis and Machine Intelligence 24(7), 881–892 (2002)
9. Kiseleva, N.E., Muchnik, I.B., Novikov, S.G.: Stratified samples in the problem of representative types. Automation and Remote Control 47, 684–693 (1986)
10. Likas, A., Vlassis, N., Verbeek, J.J.: The global k-means algorithm. Pattern Recognition 36, 451–461 (2003)
11. Lloyd, S.P.: Least squares quantization in PCM. Technical report, Bell Telephone Labs Memorandum (1957)
12. Lytkin, N.I., Kulikowski, C.A., Muchnik, I.B.: Variance-based criteria for clustering and their application to the analysis of management styles of mutual funds based on time series of daily returns. Technical Report 2008-01, DIMACS (2008)
13. MacQueen, J.: Some methods for classification and analysis of multivariate observations. In: Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability, vol. 1, pp. 281–297. University of California Press (1967)
14. Neyman, J.: On the two different aspects of the representative method: the method of stratified sampling and the method of purposive selection. Journal of the Royal Statistical Society 97, 558–625 (1934)
15. Pelleg, D., Moore, A.: Accelerating exact k-means algorithms with geometric reasoning. In: Proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 277–281. ACM (1999)
16. Pelleg, D., Moore, A.: x-means: Extending k-means with efficient estimation of the number of clusters. In: Proceedings of the 17th International Conference on Machine Learning, pp. 727–734. Morgan Kaufmann Publishers Inc. (2000)

17. Schulman, L.J.: Clustering for edge-cost minimization. In: Proceedings of the 32nd Annual ACM Symposium on Theory of Computing, pp. 547–555. ACM (2000)
18. Späth, H.: Cluster analysis algorithms for data reduction and classification of objects. E. Horwood (1980)
19. Vinh, N.X., Epps, J., Bailey, J.: Information theoretic measures for clusterings comparison: is a correction for chance necessary? In: Proceedings of the 26th Annual International Conference on Machine Learning, pp. 1073–1080. ACM (2009)

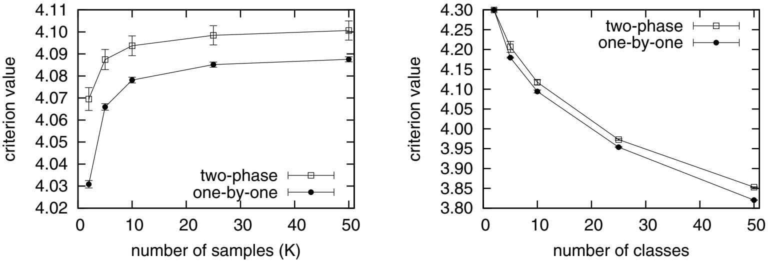
Appendix: One-by-One versus Two-Phase Approaches

To provide a fair and balanced comparison between our procedure and the two-phase approach, denoted by `two-phase`, we set `one-by-one` to optimize J_1. No accelerated two-phase implementations (e.g., [8,15]) were considered because they are tied to functional J_1.

5.1 Synthetic Data

Synthetic data sets were generated following [8,15] in order to determine how each algorithm scales with respect to the number of clusters (k), attributes (d), and samples (n). Samples were drawn from an even mixture of Gaussian distributions $\mathcal{N}_i(\boldsymbol{\mu}_i, \sigma^2 I), i = 1, \ldots, k$, and assembled follows. Each cluster center $\boldsymbol{\mu}_i$ was sampled from a uniform distribution inside the d-hypercube $[0, 1]^d$. Covariance matrices were fixed as $\sigma^2 I$, with $\sigma^2 \in \{0.05, 0.10, 0.30\}$, which, for $k \leq d$, translates into clusters with no, little, and some overlap, respectively. The remaining parameters were varied as follows: $n \in \{2000, 5000, 10000, 25000, 50000\}$, $k, d \in \{2, 5, 10, 25, 50\}$. For each 4-tuple (n, d, k, σ^2), we generated 10 random data sets, and for each data set, 30 random initial k-partitions.

Clustering Quality Analysis. The average performance of each algorithm is shown in Fig. 2, where we plot the final criterion obtained by each algorithm when varying the number of samples (Fig. 2a) and number of clusters (Fig. 2b).

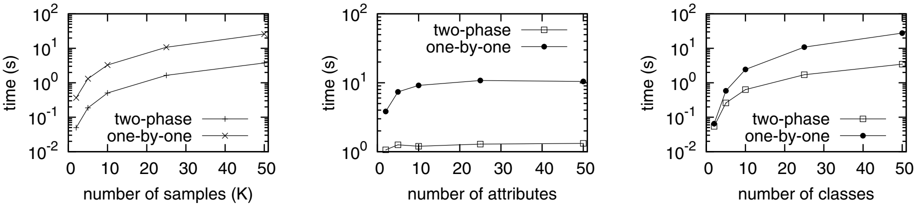


(a) Varying n. **(b)** Varying k.

Fig. 2. Average criterion function value for different instance parameters

Varying the number of attributes resulted in nearly identical curves. On average, `one-by-one` marginally outperformed `two-phase` regarding solution quality while also obtaining much smaller standard deviations.

Runtime Analysis. In Fig. 3, we plot the average running times to generate the solutions reported in the previous experiment. Here, as predicted, `two-phase` was indeed faster than `one-by-one`.



(a) Varying n. **(b)** Varying d. **(c)** Varying k.

Fig. 3. Average running times, displayed in logarithmic scale

5.2 Real-World Data

We set both implementations to optimize J_1 and J_2 on the same real-world instances from Sec. 4. Based on the four average performance indicators in Table 3, `one-by-one` is again slightly more accurate but about 2.5 times slower.

Table 3. Performance indicators on real-world instances

Performance criterion	J_1		J_2	
	`two-phase`	`one-by-one`	`two-phase`	`one-by-one`
Ratio final/initial criterion value	0.4160	**0.4131**	0.5828	**0.5748**
Accuracy	0.7171	**0.7211**	0.7099	**0.7187**
Adjusted Rand Index	**0.3745**	0.3739	0.3665	**0.3675**
Normalized runtime	**1.0000**	2.5863	**1.0000**	2.0911

A Decomposition Approach for Solving Critical Clique Detection Problems*

Jose L. Walteros and Panos M. Pardalos

Center for Applied Optimization,
Department of Industrial and Systems Engineering, University of Florida,
303 Weil Hall, Gainesville, FL, USA
jwalteros@ufl.edu, pardalos@ise.ufl.edu
http://www.ise.ufl.edu/cao

Abstract. The problem of detecting critical elements in a network involves the identification of a subset of elements (nodes, arcs, paths, cliques, etc.) whose deletion minimizes a connectivity measure over the induced network. This problem has attracted significant attention in recent years because of its applications in several fields such as telecommunications, social network analysis, and epidemic control. In this paper we examine the problem of detecting critical cliques (CCP). We first introduce a mathematical formulation for the CCP as an integer linear program. Additionally, we propose a two-stage decomposition strategy that first identifies a candidate clique partition and then uses this partition to reformulate and solve the problem as a generalized critical node problem (GCNP). To generate candidate clique partitions we test two heuristic approaches and solve the resulting (GCNP) using a commercial optimizer. We test our approach in a testbed of 13 instances ranging from 25 to 100 nodes.

Keywords: Critical element detection, critical clique detection, clique partitioning.

1 Introduction

The problem of detecting *critical elements* (nodes, arcs, paths, clusters, cliques, etc.) in a network has recently become a major endeavor. Identifying these elements can be crucial for studying many structural characteristics of a network such as connectivity, centrality, robustness, and vulnerability, as well as for identifying dominant clusters and/or partitions.

There is a wide variety of applications for which the detection of critical elements may be of great value. For example, analyzing beforehand how well a network would perform under certain disruptive events plays a vital role in the design and the operation such network. In order to detect vulnerability issues, it is particularly important to analyze how well connected a network remains after a disruptive event takes place destroying or impairing a set of elements in the

* This research is partially supported by NSF, DTRA and DURIP grants.

R. Klasing (Ed.): SEA 2012, LNCS 7276, pp. 393–404, 2012.

network. The main strategy is to identify which is the set of critical elements that must be protected or reinforced in order to mitigate the negative impact that the absence of such elements may produce in the network. Applications of this kind arise in many different contexts and fields such as in social networks analysis [4], homeland security [12], evacuation planning [16], immunization strategies [21], transportation [15], and power grid construction [19], among others.

In general, most of the critical element detection problems fall into the following definition. Given a connected undirected network $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ and $\mathcal{E}$ are the set of nodes and edges, respectively, the critical element detection problem involves finding a subset of elements $\mathcal{A} = \{1, \dots, k\}$ $(k < |\mathcal{V}|)$ such that its deletion minimizes a given connectivity measure over $\mathcal{G}$.

Several measures have been used to assess the level of disconnection of the residual network. There are mainly two classes in which these measures can be categorized. The measures from the first class can be associated mostly with network flow problems (e.g., shortest path problems and maximum flow problems) [5,10,16,22]. For these cases, the critical elements are the ones whose deletion results in the maximum increase of the shortest path, or consequently, the maximum decrease of the flow capacity between two predefined nodes s and t. This kind of measures are commonly used in the context of network interdiction, and are generally designed to tackle arc interdiction problems (detecting critical arcs).

On the other hand, the measures of the second class are associated with topological characteristics of the network. For example, one can account for the total number of pairwise connections (i.e., the total number of node pairs that are connected in the network by at least one path) [2,7], the total cost of pairwise connectivity (i.e., a weighted sum of the pairwise connections) [2,7], the size of the largest connected component (i.e., the number of nodes that belong to the largest maximal connected subgraph of $\mathcal{G}$) [17,20], and the total number of connected components [1,20]. The measures of this class are the ones that we will consider in this work.

Among all the critical element detection problems, the one of detecting critical nodes (CNP) is the one that has attracted more attention. From the complexity point of view, the CNP is proven to be $\mathcal{NP}$-hard on general networks for most of the connectivity measures described above [2,7,8]. There are few cases, though, for which the CNP is solvable in polynomial time (see, [7,20]). Existing methodologies for solving the CNP include heuristics (and metaheuristics) [1,2,4], mathematical programing [2,5,10,16,17,22], dynamic programing [7,20], approximated algorithms [8], and simulation approaches [14].

A simple heuristic approach regarding the CNP was explored by Albert et al. [1]. This work aims at analyzing the tolerance of complex networks with respect to strategic node deletions. Instead of finding the collection of nodes that must be removed to impair the connectivity of the network, the authors analyze the resulting consequences over the network when (i) a set of randomly chosen nodes is removed and (ii) when the nodes with large degree are removed.

Recent endeavors using mathematical programing techniques can be found in [2] and [17]. In their work, Aurslevan et al. [2] provide a prove of the $\mathcal{NP}$-hardness of the CNP for the pairwise connectivity measure. They also introduce a linear integer formulation and a fast constructive heuristic. An alternative formulation was presented in [17]. In this paper, the authors provide a detailed polyhedral analysis for different valid inequalities as well as a branch-and-cut framework.

The use of dynamic programming has been studied by Shen and Smith [20] and Di Summa et al. [7]. In both studies, the authors provide a detailed complexity analysis of the CNP over trees and other structures. They also prove that the cardinality version of some CNP variations over trees are polynomially solvable via dynamic programing.

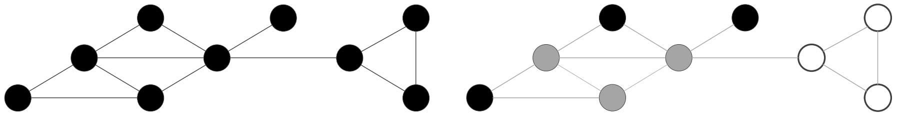
From the approximation algorithms perspective, a variation of the CNP problem is presented in [8]. In this work, the authors propose a reformulation for the CNP where the objective function is set to minimize the number of nodes (or edges) that must be removed in order to achieve a certain degradation (disruption) in the connectivity of the network. In addition to these reformulations, a thorough complexity and inapproximability analysis is presented as well as a pseudo-approximation scheme.

The main purpose of this paper is to extend the scope of previous works related with the CNP, and analyze the problem of detecting critical cliques on networks. We organize this paper as follows. In Section 2, we introduce the critical clique detection problem (CCP) including a complexity analysis regarding the $\mathcal{NP}$-completeness of the CCP. We also introduce a integer linear formulation and its respective variations for two of the connectivity measures described above. In Section 3, we present a decomposition approach for solving the CCP. The proposed approach is based on a reduction from the CCP to a generalized critical node problem (GCNP) by means of a clique partitioning problem. We also present two algorithms that can be used to obtain candidate clique partitions, as well as a formulation for the GCNP that is used to solve the resulting problem. In Section 4, we present our computational results, and finally, in Section 5 we provide conclusions an further directions for subsequent projects.

2 The Critical Clique Detection Problem (CCP)

Given a connected undirected network $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ and $\mathcal{E}$ are the set of nodes and edges, respectively, and an integer k, the CCP involves finding a set of k disjoint cliques such that its deletion results in the maximum network disconnection. Additional constraints regarding the structure of the cliques can also be imposed, for instance, an upper bound on the size of the critical cliques. Notice that the CCP can be seen as a generalization of the CNP, where the objective is to find cliques instead of nodes. The CNP is then the case where the size of the cliques is limited to be one. Figure 1 presents an example of the CCP over a 9-node graph, where $k = 2$. Figure 1(a) displays the original network, and Figure 1(b) the optimal solution where the cliques selected are colored in gray and white.

Among the different connectivity measures described above which can be used as objective functions we discuss two: the total number of pairwise connections and the size of the largest component. A description of these objectives follows:



(a) Original graph (b) Optimal solution

Fig. 1. Example for a 9-node graph

2.1 Objective Functions

Before presenting the objectives, we need to introduce the following definitions. For any subset $\mathcal{V}' \subseteq \mathcal{V}$ we define $\mathcal{E}(\mathcal{V}') \subseteq \mathcal{E}$ as the set of edges such that, for each edge $e \in \mathcal{E}(\mathcal{V}')$, both endpoints of e belong to $\mathcal{V}'$. We also define the induced graph $\mathcal{G}(\mathcal{V}')$ as the graph comprised by the set of nodes $\mathcal{V}'$, and the set of edges $\mathcal{E}(\mathcal{V}')$. We assume that two nodes $i, j \in \mathcal{V}$ are connected over $\mathcal{G}$ if there exist at least one path that connects i with j in $\mathcal{G}$. Let $\mathcal{Q}$ be the set of maximal connected components of $\mathcal{G}$. We define a maximal connected component $\mathcal{C}_q \in \mathcal{Q}$ as a subset $\mathcal{C}_q \subseteq \mathcal{V}$ of nodes such that every pair of nodes $i, j \in \mathcal{C}_q$ is connected over $\mathcal{G}(\mathcal{C}_q)$, and such that, for every node $l \in \mathcal{V} \setminus \mathcal{C}_q$, there is no edge connecting l with any node $i \in \mathcal{C}_q$. From now on we will refer to the maximal connected components only as components unless otherwise stated. Let $\sigma_q = |\mathcal{C}_q|$ be the number of nodes of component $\mathcal{C}_q \in \mathcal{Q}$. We define the number of pairwise connections of component $\mathcal{C}_q \in \mathcal{Q}$ as $\binom{\sigma_q}{2} = \sigma_q(\sigma_q - 1)/2$. Let $\mathcal{T} = \{\mathcal{K}_1, \ldots, \mathcal{K}_k\}$ be the set of k critical cliques of $\mathcal{G}$, $\mathcal{V}(\mathcal{K}_t) \subseteq \mathcal{V}$ be the subset of nodes that comprise clique $\mathcal{K}_t \in \mathcal{T}$, and $\mathcal{V}(\mathcal{T}) \subseteq \mathcal{V}$ be the set of all the nodes that belong to the critical cliques. Finally, let $\mathcal{G}^{\mathcal{T}} = (\mathcal{V} \setminus \mathcal{V}(\mathcal{T}), \mathcal{E}(\mathcal{V} \setminus \mathcal{V}(\mathcal{T})))$ be the resulting network after the deletion of the critical cliques, and $\mathcal{Q}^{\mathcal{T}}$ the corresponding set of remaining components. The definitions of the two objectives used follow:

Minimize the Total Pairwise Connectivity (TPW): Given a network $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and an integer k, we try to find a collection of cliques $\mathcal{T}$, of size $|\mathcal{T}| \leq k$, such that the sum of the pairwise connections of all the components left is minimized:

$$\min \sum_{q \in \mathcal{Q}^{\mathcal{T}}} \sigma_q(\sigma_q - 1)/2 \tag{1}$$

Minimize the Size of the Largest Component (MinMS): Given a network $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and an integer k, we try to find a collection of cliques $\mathcal{T}$, of size $|\mathcal{T}| \leq k$, such that the size of the largest component is minimized:

$$\min \max_{q \in \mathcal{Q}^{\mathcal{T}}} \{\sigma_q\} \tag{2}$$

2.2 $\mathcal{NP}$-Completeness of the CCP

We now prove that the decision version of the CCP problem is $\mathcal{NP}$-complete. The decision version of the CCP, defined as the α–CCP can be stated as follows.

Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and a positive integer k, is there a set of disjoint cliques $\mathcal{T} = \{\mathcal{K}_1, \mathcal{K}_2, \ldots, \mathcal{K}_t\}, t \leq k$ such that the size of the largest component of $\mathcal{G}^{\mathcal{T}}$ is at most α? Note that in this case we use expression (2) as the connectivity measure, although, we could easily adapt this result for (1) as well.

Clearly, the α–CCP belongs to the class $\mathcal{NP}$ for any given $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. Notice that given a collection of cliques $\mathcal{T}$ in $\mathcal{G}$, identifying the size of each component of $\mathcal{G}^{\mathcal{T}}$ can be done in polynomial time by means of a breadth first search algorithm [13].

To prove that the α–CCP belongs to the $\mathcal{NP}$–complete class, we propose the following reduction from the *clique partitioning problem* known to be $\mathcal{NP}$–complete [9]. The clique partitioning problem is defined as follows: Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and a positive integer k, is it possible to partition set $\mathcal{V}$ into $t \leq k$ disjoint cliques $\mathcal{K}_1, \mathcal{K}_2, \ldots, \mathcal{K}_t$?

It can be easily argued that the α–CCP generalizes the partition into cliques problem. Note that for $\alpha = 0$, there is a collection of at most k cliques $\mathcal{K}_1, \ldots, \mathcal{K}_k$ such that every component left in the network has size zero (i.e., $|\mathcal{Q}^{\mathcal{T}}| = 0$) if and only if, every vertex in $\mathcal{V}$ belongs to one of the following subsets induced by the critical cliques $\mathcal{V}(\mathcal{K}_1), \mathcal{V}(\mathcal{K}_2), \ldots, \mathcal{V}(\mathcal{K}_k)$. Thereby, the α–CCP is $\mathcal{NP}$-complete.

2.3 CCP formulations

When studying combinatorial problems, using a linear integer formulation is in general a natural starting point. Despite the inherent difficulty of these problems, techniques such as branch and bound, branch and cut, and others are proven to be very efficient approaches to obtain solutions for instances of manageable size. We now present an integer linear formulation for the CCP as well as the respective modifications to tackle the two objectives proposed above.

Let $\mathcal{V}(e)$ be the set of endpoints of edge $e \in \mathcal{E}$ and $\mathcal{T}$ be the set of critical cliques such that $|\mathcal{T}| = k$. Let x_i^t be a binary variable that takes the value of one if node i is assigned to clique $\mathcal{K}_t \in \mathcal{T}$, and zero otherwise. Let y_{ij} be a binary variable that takes the value of one if nodes i and j, belong to the same component in the residual graph, and zero otherwise. Let z_i be an auxiliary binary variable that takes the value of one if node i does not belong to a critical clique, and zero otherwise. The formulation for the CCP for the TPW objective is as follows:

$$\min \sum_{i,j \in \mathcal{V}} y_{ij} \tag{3}$$

$$\text{s.t. } x_i^t + x_j^t \leq 1 \qquad e \in \mathcal{V} \times \mathcal{V} \setminus \mathcal{E}, i, j \in \mathcal{V}(e), t \in \mathcal{T} \tag{4}$$

$$z_i + \sum_{t \in \mathcal{T}} x_i^t = 1 \qquad i \in \mathcal{V} \tag{5}$$

$$y_{ij} \geq z_i + z_j - 1 \qquad e \in \mathcal{E}, i, j \in \mathcal{V}(e) \tag{6}$$

$$y_{ij} + y_{jl} - y_{il} \leq 1 \qquad i, j, l \in \mathcal{V} \tag{7}$$

$$y_{ij} - y_{jl} + y_{il} \leq 1 \qquad i, j, l \in \mathcal{V} \tag{8}$$

$$-y_{ij} + y_{jl} + y_{il} \leq 1 \qquad i, j, l \in \mathcal{V} \tag{9}$$

$$x_i^t \in \{0,1\} \qquad i \in \mathcal{V}, t \in \mathcal{T} \tag{10}$$

$$z_i \in \{0,1\} \qquad i \in \mathcal{V} \tag{11}$$

$$y_{ij} \in \{0,1\} \qquad i,j \in \mathcal{V} \tag{12}$$

where the objective function (3) minimizes the sum of pairwise connections. Note that since y_{ij} is equal to 1 if nodes i and j belong to the same component, $\sum_{i,j\in V} y_{ij}$ is equivalent to $\sum_{q\in\mathcal{Q}^{\mathcal{T}}} \sigma_q(\sigma_q - 1)$. Constraint (4) ensures that if there is no edge $e \in \mathcal{E}$ between nodes i and j (i.e., $e \in \mathcal{V}\times\mathcal{V}\setminus\mathcal{E}$), both nodes cannot be assigned to the same clique. Constraint (5) ensures that if node i is not assigned to a clique, its corresponding variable z_i must be equal to one. Constraints (6) define the relationship between y variables and z variables. Constraints (7–9) define the triangular relationship of y variables (i.e., if in the residual network node i is connected to node j and node j is connected to node k, then node i must also be connected to node k). And finally constraints (10–12) define the domain of the variables used. We will refer to this problem as CCP–TPW.

To use the MinMS as the objective, we can adapt the proposed model by introducing a new variable β defined as the size of the largest component. Then the model can be formulated as follows:

$$\min \quad \beta \tag{13}$$

$$\text{s.t.} \quad (4-12)$$

$$\sum_{i\in V} y_{ij} \leq \beta \qquad i \in \mathcal{V} \tag{14}$$

where objective function (13) combined with constraints (14) enforces the minimization of the size of the largest component. We will refer to this problem as CCP–MinMS.

Formulations CCP–TPW and CCP–MinMS are relatively large in size with respect to the size of the network (they require $O(|\mathcal{V}|^2)$ variables and $O(|\mathcal{V}|^3)$ constraints). To solve these formulations, it is common to use a cutting plane generation scheme that sequentially includes constraints (7–9) as needed. Moreover, it is easy to see that we can strengthen these formulations using some valid inequalities originally designed for similar problems [11,17], as well as symmetry breaking constraints.

As an alternative solution approach, we also provide a decomposition strategy for the CCP.

3 Decomposition Approach for Solving the CCP

The decomposition strategy proposed in this paper is based on the following theorem:

Theorem 1. *The set of critical cliques of any feasible solution* $(\mathbf{x}, \mathbf{y}, \mathbf{z})$ *belongs to at least one clique partition of the original network* $\mathcal{G}$.

Proof. Let $\mathcal{T}$ be the set of critical cliques of solution $(\mathbf{x}, \mathbf{y}, \mathbf{z})$, and $\mathcal{V}(\mathcal{T})$ be the set of nodes comprising the cliques. Let $\bar{\mathcal{R}}$ be any clique partition of the residual network $\mathcal{G}(\mathcal{V} \setminus \mathcal{T})$. Note that $\mathcal{R} = \mathcal{T} \cup \bar{\mathcal{R}}$ is a clique partition of $\mathcal{G}$ as $\mathcal{T}$ and $\bar{\mathcal{R}}$ are two disjoint sets of cliques that cover all the nodes in $\mathcal{G}$. □

Since every set of critical cliques can be associated with a clique partition, we propose to solve the CCP by: (i) generating a clique partition, (ii) collapsing each clique of the given partition into a single node forming a network $\mathcal{H}$, and (iii) using an exact or heuristic method, for solving a generalized CNP over $\mathcal{H}$ (see Algorithm 1). We now analyze each of the steps of this approach.

Algorithm 1. `CCPCollapseAlgorithm`($\mathcal{G}$)

```
$\mathcal{R} \leftarrow$ generate a clique partition
$\mathcal{H} \leftarrow$ collapse($\mathcal{R}$)
$\mathcal{T}^* \leftarrow$ SolveGeneralizedCNP($\mathcal{H}$)
return $\mathcal{T}^*$
```

3.1 Constructing Clique Partitions

The main component of this approach is the way in which the clique partition is generated. This is because, in order to obtain a good solution, we would like to generate a clique partition containing the optimal set of critical cliques (or at least a good proxy). We propose to heuristically generate candidate clique partitions. The idea behind our approach is that if we want to greedily reduce the number of pairwise connections, we can either aim at eliminating a large clique, or a clique with a large degree (i.e., a clique with many edges emanating from it), we propose two different algorithms for partitioning the network following this analysis.

The first approach approach is to use as a clique partition the solution of a maximum edge clique partition problem (Max-ECP). The Max-ECP problem looks for a clique partition that maximizes the number of edges within the cliques. Even though the Max-ECP is proven to be $\mathcal{NP}$–hard, there are several approximation algorithms to solve this problem. We decided to use the 2-approximation algorithm proposed by [6] that we called `MaxECP` (see Algorithm 2). Since Algorithm 2 requires solving sequentially a maximum clique problem, we used the approximation algorithm proposed in [3]. Note that it is also possible to get both, the clique partitioning or/and the maximum clique, by solving the corresponding mathematical problems, or by means of any other technique (exact or heuristic).

For the second approach, we propose to use a clique partition based on the degree of the cliques. We use a heuristic that greedily finds a clique with a large degree in $\mathcal{G}$ (see Algorithm 3). Once we find this clique, we remove it from the network and continue following the same process until all the nodes are eliminated (see Algorithm 4).

Algorithm 2. `MaxECP`$(\mathcal{G})$ [6]

```
$\mathcal{T} \leftarrow \emptyset$
repeat
    Select the maximum clique $\bar{\mathcal{K}}$ in $\mathcal{G}(\mathcal{V} \setminus \mathcal{V}(\mathcal{T}))$.
    $\mathcal{T} \leftarrow \bar{\mathcal{K}} \cup \mathcal{T}$
    $\mathcal{G} \leftarrow \mathcal{G}(\mathcal{V} \setminus \mathcal{V}(\mathcal{T}))$
until $\mathcal{G} = \emptyset$
return $\mathcal{T}$
```

Algorithm 3. `GreedyGetClique`$(\mathcal{G})$

```
$\mathcal{K} \leftarrow \emptyset$
repeat
    Select vertex $i$ with maximum degree in the subgraph induced by $\mathcal{G}$.
    $\mathcal{K} \leftarrow \mathcal{K} \cup \{i\}$
    $\mathcal{N}(i) \leftarrow$ neighbors of $i$
    $\mathcal{G} \leftarrow \mathcal{G} \cap \mathcal{N}(i)$
until $\mathcal{G} = \emptyset$
return $\mathcal{K}$
```

Algorithm 4. `MaxDegree`$(\mathcal{G} = (\mathcal{V}, \mathcal{E}))$

```
$i \leftarrow 1$
while $V \neq \emptyset$ do
    $\mathcal{K}_t \leftarrow$ GreedyGetClique$(\mathcal{V})$
    $\mathcal{V} \leftarrow \mathcal{V} \setminus \mathcal{K}_t$
    $t \leftarrow t + 1$
end while
```

3.2 Clique Collapsing

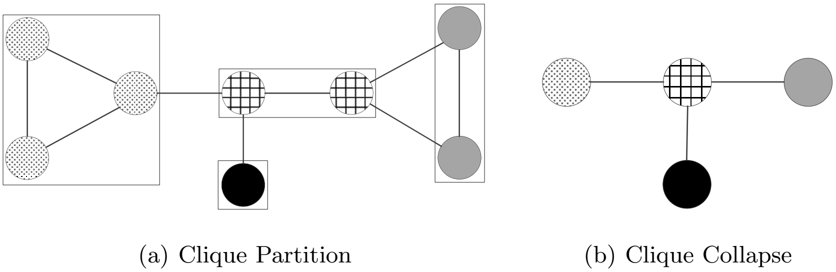
First, assume that we have a clique partition $\mathcal{R} = \{\mathcal{K}_1, \mathcal{K}_2, \ldots, \mathcal{K}_l\}$. We can collapse each of the cliques in $\mathcal{R}$ into a single node. Let $\mathcal{H}$ be a network comprised by these nodes. Let $\mathcal{V}^{\mathcal{R}}$ be the set of nodes representing the cliques and $\mathcal{E}^{\mathcal{R}}$ be the edges connecting the nodes in $\mathcal{V}^{\mathcal{R}}$. There exists an edge (i, j) in $\mathcal{E}^{R}$ if there exists at least one edge in $\mathcal{E}$ connecting a node in K_i with a node in $\mathcal{K}_j$. Let $\mathcal{H} = (\mathcal{V}^{\mathcal{R}}, \mathcal{E}^{\mathcal{R}})$ be the network induced by the collapsed nodes. Finally, let $s(\mathcal{K}_i)$ be the size of clique $\mathcal{K}_i$. Figure 2 provides an example of the clique collapsing, given a clique partition.

3.3 CNP Generalization for Solving the CCP

Assuming that we have a partition $\mathcal{R}$, once we have the collapsed network $\mathcal{H}$ we can obtain the solution of the CCP by solving a generalized CNP problem. We will discuss only the reformulation for the CCP-TPW case, although, this result can be trivially extended for the CCP-MinMS.

Notice that if we want to count the total number of pairwise connections in $\mathcal{H}$, we need to take into account the connections at the interior of each node in

$\mathcal{V}^{\mathcal{R}}$ (recall that at the point, each clique is now represented by a node), as well as the connection associated with each edge in $\mathcal{E}^{\mathcal{R}}$. For the sake of clarity, we abuse the notation in this formulation using i and j when referring to the collapsed nodes in $\mathcal{V}^{\mathcal{R}}$ and by defining x_i as a binary variable that takes the value of one if collapsed node i is removed and zero otherwise. Within each clique $\mathcal{K}_i \in \mathcal{R}$, the total number of connections is given by $p_i = \binom{s(\mathcal{K}_i)}{2} = (s(\mathcal{K}_i)(s(\mathcal{K}_i) - 1)/2)$. Moreover, note that if nodes i and j are connected in $\mathcal{H}$, the number of pairwise connections represented by edge $(i, j) \in \mathcal{E}^{\mathcal{R}}$ is given now by $t_{ij} = s(K_i)s(K_j)$. Thus, the generalized formulation for the CNP follows.


(a) Clique Partition

(b) Clique Collapse

Fig. 2. Clique collapse

$$\min \sum_{i \in \mathcal{V}^{\mathcal{R}}} p_i(1 - x_i) + \sum_{i,j \in \mathcal{V}^{\mathcal{R}}} t_{ij} y_{ij} \tag{15}$$

$$\text{s.t. } y_{ij} + x_i + x_j \geq 1 \qquad \forall (i, j) \in \mathcal{E}^{\mathcal{R}} \tag{16}$$

$$y_{ij} + y_{jk} - y_{ki} \leq 1 \qquad \forall (i, j, k) \in \mathcal{V}^{\mathcal{R}} \tag{17}$$

$$y_{ij} - y_{jk} + y_{ki} \leq 1 \qquad \forall (i, j, k) \in \mathcal{V}^{\mathcal{R}} \tag{18}$$

$$-y_{ij} + y_{jk} + y_{ki} \leq 1 \qquad \forall (i, j, k) \in \mathcal{V}^{\mathcal{R}} \tag{19}$$

$$\sum_{i \in \mathcal{V}^{\mathcal{R}}} x_i \leq k \tag{20}$$

$$x_i \in \{0, 1\} \qquad \forall i \in \mathcal{V}^{\mathcal{R}} \tag{21}$$

$$y_{ij} \in \{0, 1\} \qquad \forall (i, j) \in \mathcal{V}^{\mathcal{R}} \tag{22}$$

where objective (15) accounts for the minimization of the total pairwise connections taking into account the connections at the interior of the cliques. Constraints (16–22) are defined exactly as in [2].

4 Computational Experiments

We tested efficacy of our approach on 13 randomly generated networks ranging in size from 25 to 100 nodes. All the networks were generated using the algorithm proposed by Palmer and Steffan [18] such that the degree of the nodes follows a power-law distribution. We solved the IP formulations and applied the decomposition strategy for the CCP-TWP and the CCP-MinMS for different values of k. We solved first the IP formulations using the commercial optimizer CPLEX 12.0, fixing a time limit of four hours (14,400 seconds). We also implemented both the `MaxDegree` and the `MaxECP` algorithms to generate the clique partitions and used CPLEX to solve the resulting GCNP formulations. The computational results are listed in Table 1. For the cases in which the optimizer fails to obtain an optimal solution within the time frame, we provide the best integer solution found.

Table 1. Computational results. The optimal solutions are listed in bold and the time is described in seconds. (*) indicates that the optimizer was not able to find an integer solution within the time limit

			CCP-TPW						CCP-MinMS					
			IP Form.		`MaxDeg`		`MaxECP`		IP Form.		`MaxDeg`		`MaxECP`	
$\|\mathcal{V}\|$	$\|\mathcal{E}\|$	k	Best	Time	Best	Time	Best	Time	Best	Time	Best	Time	Best	Time
25	75	3	**80**	381.03	105	0.59	120	0.16	**13**	658.33	15	0.55	16	0.20
25	75	5	**25**	96.52	36	0.17	39	0.08	**6**	184.51	9	0.25	9	0.14
25	75	7	**2**	14,400	4	0.05	5	0.05	**1**	8.23	3	0.14	3	0.08
50	100	5	**398**	9,730.05	497	126.48	535	89.94	**28**	7,125.01	32	1,482.40	33	450.59
50	100	10	**18**	3,470.09	26	1.16	76	9.94	**2**	6,256.77	4	7.01	9	34.58
50	150	5	502	14,400	561	47.69	561	17.77	49	14,400	34	247.77	34	62.86
50	150	10	**54**	12,701.60	76	3.73	114	6.32	10	14,400	10	16.22	13	20.67
75	150	10	1,545	14,400	836	465.68	947	270.75	52	14,400	41	14,153.12	43	12,981.10
75	150	15	423	14,400	50	13.37	132	640.41	4	14,400	6	461.36	11	13,984.49
75	200	20	50	14,400	7	3.35	39	31.74	**1**	13,7028.50	3	63.72	5	306.00
100	200	15	1,152	14,400	295	6,385.16	412	9,685.28	*	14,400	28	4,295.45	39	6,458.30
100	200	30	*	14,400	6	9.79	17	19.90	*	14,400	2	59.49	3	3,647.48
100	300	30	*	14,400	17	18.23	18	13.49	*	14,400	3	718.34	3	443.96

We were able to obtain optimal solutions for 6 instances out of 13 for both, the CCP-TPW and the CCP-MinMS. Furthermore, note that with the proposed approach, we obtained good solutions for most of the instances. Notice that for the CCP-TPW case, since the total number of pairwise connections grows quadratically with respect to the size of the remaining components, a near optimal solution having just a few additional nodes may have a significantly larger number of pairwise connections compared to the optimal solution.

In terms of the running times, the decomposition approach ran significantly faster than the IP formulation. Moreover, we found that the execution time for finding clique partitions is negligible (less that a second) compared with the time used by CPLEX to solve the GCNP. Note that the time required to solve the GCNP can be significantly reduced by using a simple variation of the heuristic proposed in [2]. Finally, we observe that the clique partitions obtained with MaxDegree yield better results for both objectives.

5 Concluding Remarks

This study was motivated by the increasing interest of solving critical element detection problems. We introduced the problem of identifying critical cliques

(CCP) over networks considering two connectivity measures: the total pairwise connectivity and the size of the largest component. To address this problem, we formulated it as an integer program. In addition, we proposed a decomposition strategy for solving large-scale instances that first generates a clique partition and then reformulates and solves the problem as a generalized critical node problem (GCNP). We introduced two heuristics for obtaining clique partition candidates. The resulting GCNP is then solved using a commercial optimizer. We evaluated the performance of our approach by solving 13 randomly generated instances ranging in size from 25 to 100 nodes.

Future research may involve testing additional methodologies for obtaining clique partitions, as well as testing the performance of the proposed approach when additional constraints over the cliques are included.

References

1. Albert, R., Jeong, H., Barabasi, A.-L.: Error and attack tolerance of complex networks. Nature 406(6794), 378–382 (2000)
2. Arulselvan, A., Commander, C.W., Elefteriadou, L., Pardalos, P.M.: Detecting critical nodes in sparse graphs. Computers and Operations Research 36(7), 2193–2200 (2009)
3. Boppana, R., Halldorsson, M.: Approximating maximum independent sets by excluding subgraphs. BIT 32, 180–196 (1992)
4. Borgatti, S.P.: Identifying sets of key players in a social network. Comput. Math. Organ. Theory 12, 21–34 (2006)
5. Corley, H., Sha, D.Y.: Most vital links and nodes in weighted networks. Operations Research Letters 1(4), 157–160 (1982)
6. Dessmark, A., Jansson, J., Lingas, A., Lundell, E.-M., Persson, M.: On the approximability of maximum and minimum edge clique partition problems. International Journal of Foundations of Computer Science 18 (2006, 2007)
7. Di Summa, M., Grosso, A., Locatelli, M.: Complexity of the critical node problem over trees. Computers and Operations Research 38(12), 1766–1774 (2011)
8. Dinh, T.N., Xuan, Y., Thai, M.T., Pardalos, P.M., Znati, T.: On new approaches of assessing network vulnerability: Hardness and approximation. IEEE/ACM Transactions on Networking PP(99) (2011)
9. Garey, M.R., Johnson, D.S.: Computers and Intractability; A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York (1990)
10. Grötschel, M., Monma, C., Stoer, M.: Design of survivable networks. In: Ball, C.M.M.O., Magnanti, T.L., Nemhauser, G. (eds.) Network Models. Handbooks in Operations Research and Management Science, vol. 7, pp. 617–672. Elsevier (1995)
11. Grötschel, M., Wakabayashi, Y.: Facets of the clique partitioning polytope. Mathematical Programming 47, 367–387 (1990)
12. Grubesic, T.H., Matisziw, T.C., Murray, A.T., Snediker, D.: Comparative approaches for assessing network vulnerability. International Regional Science Review 31(1), 88–112 (2008)
13. Hopcroft, J., Tarjan, R.: Algorithm 447: efficient algorithms for graph manipulation. Commun. ACM 16, 372–378 (1973)
14. Houck, D.J., Kim, E., O'Reilly, G.P., Picklesimer, D.D., Uzunalioglu, H.: A network survivability model for critical national infrastructures. Bell Labs Technical Journal 8(4), 153–172 (2004)

15. Jenelius, E., Petersen, T., Mattsson, L.-G.: Importance and exposure in road network vulnerability analysis. Transportation Research Part A: Policy and Practice 40(7), 537–560 (2006)
16. Matisziw, T.C., Murray, A.T.: Modeling s-t path availability to support disaster vulnerability assessment of network infrastructure. Comput. Oper. Res. 36, 16–26 (2009)
17. Oosten, M., Rutten, J.H.G.C., Spieksma, F.C.R.: Disconnecting graphs by removing vertices: a polyhedral approach. Statistica Neerlandica 61(1), 35–60 (2007)
18. Palmer, C., Steffan, J.: Generating network topologies that obey power laws. In: Global Telecommunications Conference, GLOBECOM 2000, vol. 1, pp. 434–438. IEEE (2000)
19. Salmeron, J., Wood, K., Baldick, R.: Analysis of electric grid security under terrorist threat. IEEE Transactions on Power Systems 19(2), 905–912 (2004)
20. Shen, S., Smith, J.C.: Polynomial-time algorithms for solving a class of critical node problems on trees and series-parallel graphs. Networks (2011)
21. Tao, Z., Zhongqian, F., Binghong, W.: Epidemic dynamics on complex networks. Progress in Natural Science 16(5) (2005)
22. Wollmer, R.: Removing arcs from a network. Operations Research 12(6), 934–940 (1964)

Author Index