

CONFERENCES IN RESEARCH AND PRACTICE IN
INFORMATION TECHNOLOGY

VOLUME 51

THEORY OF COMPUTING 2006

AUSTRALIAN COMPUTER SCIENCE COMMUNICATIONS, VOLUME 28, NUMBER 4.



AUSTRALIAN
COMPUTER
SOCIETY



CO_{mputing}
R_{esearch}
& E_{ducation}

THEORY OF COMPUTING 2006

Proceedings of the 12th Computing:
The Australasian Theory Symposium (CATS2006),
Hobart, Tasmania, Australia, 16-19 January 2006

Joachim Gudmundsson and Barry Jay, Eds.

Volume 51 in the Conferences in Research and Practice in Information Technology Series.
Published by the Australian Computer Society Inc.



Published in association with the ACM Digital Library.

**Proceedings of the Twelfth Computing: The Australasian Theory Symposium (CATS2006),
Hobart, Tasmania, Australia, 16-19 January 2006**

Conferences in Research and Practice in Information Technology, Volume 51.

Copyright ©2006, Australian Computer Society. Reproduction for academic, not-for profit purposes permitted provided the copyright text at the foot of the first page of each paper is included.

Editors:
Joachim Gudmundsson
IMAGEN program
National ICT Australia Ltd
Locked Bay 9013
Alexandria NSW 1435
Australia
Email: Joachim.Gudmundsson@nicta.com.au

Barry Jay
Faculty of Information Technology
University of Technology, Sydney
PO Box 123
Broadway NSW 2007
Australia
Email: cbj@it.uts.edu.au

Series Editor: John F. Roddick,
Conferences in Research and Practice in Information Technology
Flinders University,
PO Box 2100, Adelaide 5001
South Australia.
crpit@infoeng.flinders.edu.au

Publisher: Australian Computer Society Inc.
PO Box Q534, QVB Post Office
Sydney 1230
New South Wales
Australia.

Conferences in Research and Practice in Information Technology, Volume 51.
ISSN 1445-1336.
ISBN 1-920-68233-3.

Printed, November 2005 by Flinders Press, PO Box 2100, Bedford Park, SA 5042, South Australia.
Cover Design by Modern Planet Design, (08) 8340 1361.

The *Conferences in Research and Practice in Information Technology* series aims to disseminate the results of peer-reviewed research in all areas of Information Technology. Further details can be found at <http://crpit.com/>.

Table of Contents

Proceedings of the Twelfth Computing: The Australasian Theory Symposium (CATS2006), Hobart, Tasmania, Australia, 16-19 January 2006

Preface	vii
Programme Committee	viii
Organising Committee	ix
CORE - Computing Research and Education	xi
ACSW Conferences and the Australian Computer Science Communications	xii
ACSW and CATS 2006 Sponsors	xv

Keynote Paper

Computational Geometric and Combinatorial Approaches to Digital Halftoning	3
<i>Tetsuo Asano</i>	

Accepted Papers

Geometric spanners with few edges and degree five	7
<i>Michiel Smid</i>	
Graph Orientation Algorithms to Minimize the Maximum Outdegree	11
<i>Yuichi Asahiro, Eiji Miyano, Hirotaka Ono and Kouhei Zenmyo</i>	
Multilayer Grid Embeddings of Iterated Line Digraphs	21
<i>Toro Hasunuma</i>	
Compositional Type Systems for Stack-Based Low-Level Languages	27
<i>Ando Saabas and Tarmo Uustalu</i>	
Mechanically Verifying Correctness of CPS Compilation	41
<i>Ye Henry Tian</i>	
Formalising the L4 microkernel API	53
<i>Rafal Kolanski and Gerwin Klein</i>	
Combinatorial Generation by Fusing Loopless Algorithms	69
<i>Tadao Takaoka and Stephen Violich</i>	
The Busy Beaver, the Placid Platypus and other Crazy Creatures	79
<i>James Harland</i>	
A Polynomial Algorithm for Codes Based on Directed Graphs	87
<i>A.V. Kelarev</i>	
On the complexity of the DNA Simplified Partial Digest Problem	93
<i>Jacek Blazewicz and Marta Kasprzak</i>	

On the Approximability of Maximum and Minimum Edge Clique Partition Problems	101
<i>Anders Dessmark, Jesper Jansson, Andrzej Lingas, Eva-Marta Lundell and Mia Persson</i>	
Faster Algorithms for Finding Missing Patterns	107
<i>Shuai Cheng Li</i>	
On the Logical Implication of Multivalued Dependencies with Null Values	113
<i>Sebastian Link</i>	
Boolean equation solving as graph traversal	123
<i>Brian Herlihy, Peter Schachte and Harald Søndergaard</i>	
Learnability of Term Rewrite Systems from Positive Examples	133
<i>M.R.K. Krishna Rao</i>	
On-demand Bounded Broadcast Scheduling with Tight Deadlines	139
<i>Chung Keung Poon, Feifeng Zheng and Yinfeng Xu</i>	
Greedy algorithms for on-line set-covering and related problems.....	145
<i>Giorgio Ausiello, Aristotelis Giannakos and Vangelis Th. Paschos</i>	
Author Index	153

Preface

This volume contains the contributed papers presented at the 2006 Computing: The Australasian Theory Symposium (CATS 2006), held in Hobart, Australia, 16-19 January, 2006. In addition, the volume also includes an abstract of the keynote by Tetsuo Asano on *Computational Geometric and Combinatorial Approaches to Digital Halftoning*.

CATS is the premier theoretical computer science conference in Australasia and it is an established part of the Australasian Computer Science Week. This is the 12th CATS meeting and we thank the University of Tasmania for hosting the meeting. Previous CATSs were held in Sydney (1994), Melbourne (1996), Sydney (1997), Perth (1998), Auckland (1999), Canberra (2000), Gold Coast (2001), Monash (2002), Adelaide (2003), Otago (2004) and Newcastle (2005).

The scientific program contains 17 papers chosen from 32 submissions. Each paper was reviewed by at least three referees, and evaluated on the quality, originality and relevance to the symposium. The challenging task of selecting the papers for presentation was performed by the members of our program committee and external reviewers.

We would like to thank everyone involved in putting CATS 2006 together; especially the program committee, those who submitted papers and the external reviewers.

We wish all the participants an interesting conference.

Joachim Gudmundsson
National ICT Australia Ltd
Barry Jay
University of Technology, Sydney
CATS 2006 Program Chairs
January, 2006

Programme Committee

Chair

Joachim Gudmundsson, National ICT Australia Ltd, Australia
Barry Jay, University of Technology, Sydney, Australia.

Members

Tetsuo Asano, Japan Advanced Institute of Science and Technology, Japan
Mike Atkinson, University of Otago, New Zealand
Ljiljana Brankovic, University of Newcastle, Australia
Prosenjit Bose, Carleton University, Canada
Rod Downey, Victoria University of Wellington, New Zealand
James Harland, Royal Melbourne Institute of Technology, Australia
Mike Johnson, Macquarie University, Australia
Paul Kelly, Imperial College London, United Kingdom
Delia Kesner, Université Paris 7, France
Ling Li, Curtin University, Australia
Eugenio Moggi, University of Genoa, Italy
Jens Palsberg, University of California, U.S.A.
Andrew Solomon, University of Technology, Sydney, Australia
Gerhard Woeginger, Technical University of Eindhoven, The Netherlands.

Additional Reviewers

Mads Dam
Andrzej Filinski
Giovanna Guerrini
Kohei Honda
Andrei Kerlarev
Paul Kennedy
Bruce Litow
Ion Petre
Andy Song
Richard Webber.

Organising Committee

Welcome

On behalf of the Tasmanian Organising Committee of ACSW2006 I would like to welcome all the delegates to the conferences of this busy and interesting week, in particular those coming from overseas.

The location of the various conferences and other events at the Wrest Point Hotel allows delegates to move quickly from event to event, and to easily and comfortably gather in groups for those conversations and interactions that are so important for the exchange of ideas and the promotion of cooperation, not to mention social pleasure.

We trust you will have a thoroughly enjoyable time.

Professor Young Ju Choi
Chair, Organising Committee
January, 2006

General Chair

Professor Young Ju Choi, School of Computing, University of Tasmania, Australia

Organising Committee Members

Ms Nicole Clark
Dr Julian Dermoudy
Mr Tony Gray
Mr Neville Holmes
Mr Ian McMahon
Ms Julia Mollison
Professor Arthur Sale
Ms Soon-ja Yeom

CORE - Computing Research and Education

CORE welcomes all delegates to ACSW2006 in Hobart.

ACSW, the Australasian Computer Science Week continues to grow with new conferences becoming entrenched in the week. As the premier annual Computer Science event in Australia and New Zealand, it provides an unparalleled opportunity for the wide community of Computer Science academics and researchers to meet, network, promote IT research and be exposed to the latest research in other areas of IT. The research presented at each conference is of the highest standard and essential for the growth and future of our region, in an ever more competitive world.

CORE is expanding its awards. The Distinguished Service Award first offered in late 2004 will be offered every second year and next at the 2007 Conference. Along with the Chris Wallace Research Award, we are offering an annual teaching award for the first time.

CORE has continued to play a part in the Federation of Australian Scientific and Technological Societies and by participating in events such as Science Meets Parliament, CORE is becoming recognised by the wider community and will continue to do so. A major contribution from many members in 2005 was a submission to the RQF Forum with some of our ideas appearing in the draft. CORE and members of the Executive have also been interviewed as representatives of the Computer Science community for several other Government and industry inquiries and initiatives.

Thank you all for your contributions in 2005 and we look forward to an exciting 2006.

Jenny Edwards

President, Computing Research and Education

January, 2006

ACSW Conferences and the Australian Computer Science Communications

The Australasian Computer Science Week of conferences has been running in some form continuously since 1978. This makes it one of the longest running conferences in computer science. The proceedings of the week have been published as the *Australian Computer Science Communications* since 1979 (with the 1978 proceedings often referred to as *Volume 0*). Thus the sequence number of the Australasian Computer Science Conference is always one greater than the volume of the Communications. Below is a list of the conferences, their locations and hosts.

- 2008. Communications Volume Number 30. Proposed Host and Venue - University of Wollongong, NSW.
- 2007. Volume 29. Host and Venue - University of Ballarat, VIC.
- 2006. **Volume 28. Host and Venue - University of Tasmania, TAS.**
- 2005. Volume 27. Host - University of Newcastle, NSW. APBC held separately from 2005.
- 2004. Volume 26. Host and Venue - University of Otago, Dunedin, New Zealand. First running of APCCM.
- 2003. Volume 25. Hosts - Flinders University, University of Adelaide and University of South Australia. Venue - Adelaide Convention Centre, Adelaide, SA. First running of APBC. Incorporation of ACE. ACSAC held separately from 2003.
- 2002. Volume 24. Host and Venue - Monash University, Melbourne, VIC.
- 2001. Volume 23. Hosts - Bond University and Griffith University (Gold Coast). Venue - Gold Coast, QLD.
- 2000. Volume 22. Hosts - Australian National University and University of Canberra. Venue - ANU, Canberra, ACT. First running of AUIC.
- 1999. Volume 21. Host and Venue - University of Auckland, New Zealand.
- 1998. Volume 20. Hosts - University of Western Australia, Murdoch University, Edith Cowan University and Curtin University. Venue - Perth, WA.
- 1997. Volume 19. Hosts - Macquarie University and University of Technology, Sydney. Venue - Sydney, NSW. ADC held with DASFAA (rather than ACSW) in 1997.
- 1996. Volume 18. Host - University of Melbourne and RMIT University. Venue - Melbourne, Australia. CATS joins ACSW.
- 1995. Volume 17. Hosts - Flinders University, University of Adelaide and University of South Australia. Venue - Glenelg, SA.
- 1994. Volume 16. Host and Venue - University of Canterbury, Christchurch, New Zealand. CATS run for the first time separately in Sydney.
- 1993. Volume 15. Hosts - Griffith University and Queensland University of Technology. Venue - Nathan, QLD.
- 1992. Volume 14. Host and Venue - University of Tasmania, TAS. (ADC held separately at La Trobe University).
- 1991. Volume 13. Host and Venue - University of New South Wales, NSW.
- 1990. Volume 12. Host and Venue - Monash University, Melbourne, VIC. Joined by Database and Information Systems Conference which in 1992 became ADC (which stayed with ACSW) and ACIS (which now operates independently).
- 1989. Volume 11. Host and Venue - University of Wollongong, NSW.
- 1988. Volume 10. Host and Venue - University of Queensland, QLD.
- 1987. Volume 9. Host and Venue - Deakin University, VIC.
- 1986. Volume 8. Host and Venue - Australian National University, Canberra, ACT.
- 1985. Volume 7. Hosts - University of Melbourne and Monash University. Venue - Melbourne, VIC.
- 1984. Volume 6. Host and Venue - University of Adelaide, SA.
- 1983. Volume 5. Host and Venue - University of Sydney, NSW.
- 1982. Volume 4. Host and Venue - University of Western Australia, WA.
- 1981. Volume 3. Host and Venue - University of Queensland, QLD.
- 1980. Volume 2. Host and Venue - Australian National University, Canberra, ACT.
- 1979. Volume 1. Host and Venue - University of Tasmania, TAS.
- 1978. Volume 0. Host and Venue - University of New South Wales, NSW.

Conference Acronyms

ACE. Australian/Australasian Conference on Computing Education.

ACSAC. Asia-Pacific Computer Systems Architecture Conference (previously Australian Computer Architecture Conference (ACAC)).

ACSC. Australian/Australasian Computer Science Conference.

ACSW. Australian/Australasian Computer Science Week.

ADC. Australian/Australasian Database Conference.

APBC. Asia-Pacific Bioinformatics Conference.

APCCM. Asia-Pacific Conference on Conceptual Modelling.

AUIC. Australian/Australasian User Interface Conference.

CATS. Computing - The Australian/Australasian Theory Symposium.

Note that various name changes have occurred, most notably the change of the names of conferences to reflect a wider geographical area.

ACSW and CATS 2006 Sponsors

We wish to thank the following sponsors for their contribution towards this conference. For an up-to-date overview of sponsors of ACSW 2006 and CATS 2006, please see <http://www.comp.utas.edu.au/acsw06/>.



University of Tasmania, Australia



AUSTRALIAN
COMPUTER
SOCIETY

Australian Computer Society



CORE - Computing Research and Education



Faculty of Information Technology



National ICT Australia Ltd

KEYNOTE PAPER

Computational Geometric and Combinatorial Approaches to Digital Halftoning

Tetsuo Asano

School of Information Science,
JAIST (Japan Advanced Institute of Science and Technology)
1-1 Asahidai, Nomi, Ishikawa, 923-1292 Japan

Digital halftoning is a technique to convert a continuous-tone image into a binary image consisting of black and white dots. It is an important technique for printing machines and printers to output an image with few intensity levels or colors which looks similar to an input image. In this talk I will explain how computational geometry and combinatorial optimization can contribute to digital halftoning or what geometric and combinatorial problems are related to digital halftoning (Aronov et al. 2004, Asano et al. 2004).

Conventional halftoning algorithms are classified into two categories depending on resolution of printing devices. In a low-resolution printer such as an ink-jet printer individual dots are rather clearly separated. On the other hand dots are too small in a high-resolution printer such as off-set printer to make fine control over their positions. Therefore, dots should form clusters whose sizes are determined by their corresponding intensity levels. Such a halftoning algorithm is called a cluster-dot halftoning.

This algorithm consists in partitioning the output image plane into repetitive polygons called screen elements, usually of the same shape such as rectangles or parallelograms. Each screen element is then filled in by dots according to the corresponding intensity levels. Dots in a screen element is clustered around some center point to form a rounded figure. Denoting by k the area or the number of pixels of a screen element, only $k + 1$ different intensity levels instead of 2^k levels are reproduced since the gray level in a screen element is determined only by the number of dots in the region. So, large screen element is required to have effective tone scale. On the contrary the size of a screen element should be small for effective resolution. This suggests a serious tradeoff between effective resolution and effective tone scale. So, it is required to resolve it by introducing adaptive mechanism to determine cluster sizes.

In most of the conventional cluster-dot halftoning algorithms the output image plane is partitioned into screen elements in a fixed manner independent of given input images. A key idea of our algorithm is to partition the output plane into screen elements of various sizes to reflect spatial frequency distribution of an input image. This adaptive method is a solution to balance effective resolution and effective tone scale in the following sense. The two indices are both important, but one is more important than the other depending on spatial frequency distribution of an input image. That is, resolution is more important in a high-frequency part to have a sharp contour,

so that the sizes of screen elements should be kept small. On the other hand, tone scale is more meaningful in a low-frequency part with intensity levels changing smoothly, and so larger sizes of screen elements are preferred. All these requirements suggest the following geometric optimization problem. Given a continuous-tone image A and a scaling factor to define the size of an output image, we first compute spatial frequency distribution by applying Laplacian or Sobel differential operator. Then, each grid in the output image plane is associated with a disc of radius reflecting the Laplacian value at the corresponding point. Now, we have a number of discs of various radii. Then, the problem is to choose a set of discs to cover the output plane in an optimal way. The optimality criterion should reflect how large area is covered by exactly one disc from the set, which implies minimization of the area of unoccupied region and intersection among chosen discs to make the resulting screen elements rounded figures.

Optimization of a dither mask used in a so-called Ordered Dither algorithm for halftoning is also an interesting topic. The problem is how to arrange n^2 integers from 0 to $n^2 - 1$ as uniformly as possible over an $n \times n$ matrix. Again we introduce a discrepancy-based measure to evaluate the uniformity. The measure is based on the observation that if those integers are uniformly distributed over a matrix then the average of elements in a rigid submatrix (or region) of a fixed size must be the same wherever we take such a submatrix. So, we define the discrepancy of a matrix to be the largest difference of the average in such a region with the average in the whole matrix.

Different schema to achieve low discrepancy for a family of square regions are described. More concretely, we prove that the discrepancy for a family of 2×2 regions can be 0 if and only if the matrix size is even. For families of larger regions there is a scheme to achieve 0-discrepancy for regions of size $k \times k$ in a matrix of size $n \times n$ if the integer k divides n . On the other hand, the discrepancy cannot be 0 if the matrix size n and region size k are relatively prime.

References

- Aronov, B., Asano, T., Kikuchi, Y., Nandy, S. C., Sasahara, S. & Uno, T. (2004) 'A Generalization of Magic Squares with Applications to Digital Halftoning' in Proc. of 15th International Symposium on Algorithms and Computation, ISAAC 2004, pp. 89-100. To appear in Theory of Computing System.
- Asano, T., Katoh, N., Tamaki, H. & Tokuyama, T. (2004) 'The structure and number of global roundings of a graph', Theoretical Computer Science, 325(3):425-437.

ACCEPTED PAPERS

Geometric spanners with few edges and degree five

Michiel Smid

School of Computer Science
Carleton University
Ottawa, Ontario, Canada K1S 5B6
E-mail: michiel@scs.carleton.ca

Abstract

An $O(n \log n)$ -time algorithm is presented that, when given a set S of n points in \mathbb{R}^d and an integer k with $0 \leq k \leq n$, computes a graph with vertex set S , that contains at most $n - 1 + k$ edges, has stretch factor $O(n/(k+1))$, and whose degree is at most five. This generalizes a recent result of Aronov *et al.*, who obtained this result for two-dimensional point sets.

Keywords: Computational geometry, spanners, minimum spanning trees.

1 Introduction

Given a set S of n points in \mathbb{R}^d and a real number $t \geq 1$, a graph G with vertex set S is called a t -spanner for S , if for any two points p and q in S , there exists a path in G between p and q whose length is at most t times the Euclidean distance $|pq|$ between p and q . Here, the length of a path is defined to be the sum of the Euclidean lengths of all edges on the path. A path in G between p and q whose length is at most $t|pq|$ is called a t -spanner path. The *stretch factor* (or *dilation*) of G is defined to be the smallest value of t for which G is a t -spanner.

The problem of constructing a t -spanner for any given point set has been studied intensively. Salowe (1991) and Vaidya (1991) were the first to show that, for any constant $t > 1$ and for any constant dimension $d \geq 2$, a t -spanner with $O(n)$ edges can be computed in $O(n \log n)$ time, where the constant factors in the Big-Oh bounds depend on the stretch factor t and the dimension d . Since then, many more algorithms have been discovered that compute spanners with $O(n)$ edges and that have other properties; see the survey papers by Eppstein (2000), Gudmundsson and Knauer (2006), and Smid (2000).

Das and Heffernan (1996) considered a dual version of the spanner problem: Given a bound on the number of edges, what is the smallest stretch factor that can be obtained? They present an $O(n \log n)$ -time algorithm that constructs, when given any set S of n points in \mathbb{R}^d and any real constant $\epsilon > 0$, a graph that contains at most $(1 + \epsilon)n$ edges, whose degree is

at most three, and whose stretch factor is bounded by a constant that only depends on ϵ and d . Das and Heffernan left open the problem of determining the smallest possible stretch factor when $n + o(n)$ edges are allowed.

Since any graph with a finite stretch factor is connected, it must have at least $n - 1$ edges. Let S be a set of n points in the plane that are regularly spaced around a circle. Eppstein (2000) shows that every connected graph with vertex set S and consisting of $n - 1$ edges (i.e., every spanning tree of S) has stretch factor $\Omega(n)$. He also shows that, for any set S of n points in \mathbb{R}^d , the minimum spanning tree has stretch factor $O(n)$; in fact, the stretch factor can easily be shown to be at most $n - 1$; see Lemma 2 below.

Aronov *et al.* (2005) generalize these results for the case when the points are in \mathbb{R}^2 . They show that, for any set S of n points in the plane and for any integer k with $0 \leq k \leq n$, in $O(n \log n)$ time, a graph with vertex set S and consisting of $n - 1 + k$ edges can be computed, whose stretch factor is $O(n/(k+1))$. They also show that there exists a set S of n points, such that every connected graph with vertex set S and consisting of $n - 1 + k$ edges has stretch factor $\Omega(n/(k+1))$.

The algorithm of Aronov *et al.* is based on properties of the minimum spanning tree and the Delaunay triangulation. In particular, it uses the facts that, for two-dimensional point sets, (i) these structures can be computed in $O(n \log n)$ time, (ii) the stretch factor of the Delaunay triangulation is bounded by a constant, and (iii) the Delaunay triangulation is a planar graph. As a result, their analysis is only valid for two-dimensional point sets: First, in dimension $d \geq 3$, it is unlikely that the minimum spanning tree can be computed in $O(n \log n)$ time; see Erickson (1995). Second, for $d \geq 3$, no non-trivial upper bound on the stretch factor of the Delaunay triangulation is known. Finally, again for $d \geq 3$, the Delaunay triangulation is not a planar graph; in particular, it may have $\Theta(n^2)$ edges.

In this paper, we show that, by using a minimum spanning tree of a bounded degree spanner for S (as opposed to a minimum spanning tree of the point set itself), the result of Aronov *et al.* is in fact valid for any constant dimension $d \geq 2$. Moreover, we show that this result can be obtained by a graph having degree five.

2 Properties of the minimum spanning tree of a spanner

Let S be a set of n points in \mathbb{R}^d , let $t \geq 1$ be a real number, and let G be an arbitrary t -spanner for S . Let T be a minimum spanning tree of G . In this section, we prove some properties of T that will lead to our generalization of the result by Aronov *et al.*

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

Copyright 2006, Australian Computer Society, Inc. This paper appeared at Computing: The Australasian Theory Symposium (CATS2006), Hobart, Australia. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 51. Barry Jay and Joachim Gudmundsson, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.

These properties basically state that T has “approximately” the same properties as an exact minimum spanning tree of the point set S .

Lemma 1 *Let p and q be two distinct points of S . Then every edge on the path in T between p and q has length at most $t|pq|$.*

Proof. Let P be the path in T between p and q , and let (x, y) be an arbitrary edge on P . We will prove by contradiction that $|xy| \leq t|pq|$. Hence, we assume that $|xy| > t|pq|$.

Let Q be a t -spanner path in G between p and q . Since the length of Q is at most $t|pq|$, every edge of Q has length at most $t|pq|$. In particular, (x, y) is not an edge of Q . We may assume without loss of generality that x is between p and y on the path P . Starting at x , follow the path P towards p , and let x' be the first vertex that is on Q . Similarly, starting at y , follow the path P towards q , and let y' be the first vertex that is on Q . Let P' be the subpath of P between the vertices x' and y' , and let Q' be the subpath of Q between the vertices x' and y' . Then, P' and Q' do not have any edge in common, and these two subpaths form a simple cycle in G that contains the edge (x, y) .

Let G' be the graph obtained from T , by adding all edges of Q' (that are not in T yet), and deleting the edge (x, y) . Then G' is a connected subgraph of G on the point set S , and, since the weight of Q' is less than the weight of (x, y) , the weight of G' is less than the weight of T . This is a contradiction and, thus, we have shown that $|xy| \leq t|pq|$. ■

Lemma 2 *The minimum spanning tree T of the t -spanner G is a $(t(n-1))$ -spanner for S .*

Proof. Let p and q be two distinct points of S , and let P be the path in T between p and q . By Lemma 1, each edge of P has length at most $t|pq|$. Since P contains at most $n-1$ edges, it follows that the length of P is at most $t(n-1)|pq|$. ■

Lemma 3 *Let m be an integer with $1 \leq m \leq n-1$, and let T' and T'' be two vertex-disjoint subtrees of T , each consisting of at most m vertices. Let p be a vertex of T' , let q be a vertex of T'' , and let P be the path in T between p and q . If x is a vertex of T' that is on the subpath of P within T' , and y is a vertex of T'' that is on the subpath of P within T'' , then*

$$|xy| \leq (2t(m-1) + 1)|pq|.$$

Proof. Let P' be the subpath of P between p and x . By Lemma 1, each edge of P' has length at most $t|pq|$. Since P' contains at most $m-1$ edges, it follows that this path has length at most $t(m-1)|pq|$. On the other hand, since P' is a path between p and x , its length is at least $|px|$. Thus, we have $|px| \leq t(m-1)|pq|$. A symmetric argument can be used to show that $|qy| \leq t(m-1)|pq|$. Therefore, we have

$$\begin{aligned} |xy| &\leq |xp| + |pq| + |qy| \\ &\leq t(m-1)|pq| + |pq| + t(m-1)|pq|, \end{aligned}$$

completing the proof of the lemma. ■

3 A graph with $n + O(k)$ edges and stretch factor $O(n/k)$

Let S be a set of n points in \mathbb{R}^d , and let k be an integer with $1 \leq k \leq n$. Fix a constant $t > 1$, and let G be a

t -spanner for S whose degree is bounded by a constant that only depends on the dimension d . Clearly, the minimum spanning tree T of G has bounded degree as well. Thus, T contains a *centroid edge*, i.e., an edge whose removal from T yields two subtrees, each consisting of at most αn vertices, for some constant $\alpha < 1$ that depends on the degree of T . In fact, a centroid edge can be computed in $O(n)$ time. By repeatedly choosing a centroid edge in the currently largest subtree, we can remove $\ell = O(k)$ edges from T , and obtain vertex-disjoint subtrees T_0, T_1, \dots, T_ℓ , each containing $O(n/k)$ vertices. Observe that the vertex sets of these subtrees form a partition of S . Let X be the set of endpoints of the ℓ edges that are removed from T . Then, the size of X is at most 2ℓ , which is $O(k)$.

We define G' to be the graph with vertex set S that is the union of

1. the trees T_0, T_1, \dots, T_ℓ , and
2. a t -spanner G'' for the set X , consisting of $O(k)$ edges.

We first observe that the number of edges of G' is bounded from above by $n - 1 + O(k)$.

Lemma 4 *The graph G' has stretch factor $O(n/k)$.*

Proof. Let p and q be two distinct points of S . Let i and j be the indices such that p is a vertex of the subtree T_i and q is a vertex of the subtree T_j .

First assume that $i = j$. Let P be the path in T_i between p and q . Then, P is a path in G' . By Lemma 1, each edge on P has length at most $t|pq|$. Since T_i contains $O(n/k)$ vertices, the number of edges on P is $O(n/k)$. Therefore, since t is a constant, the length of P is $O(n/k) \cdot |pq|$.

Now assume that $i \neq j$. Let P be the path in T between p and q . Let (x, x') be the edge of P for which x is a vertex of T_i , but x' is not a vertex of T_i . Similarly, let (y, y') be the edge of P for which y is a vertex of T_j , but y' is not a vertex of T_j . Then, both (x, x') and (y, y') are edges of T that have been removed when the subtrees were constructed. Hence, x and y are both contained in X and, therefore, are vertices of G'' . Let P_i be the path in T_i between p and x , let P_{xy} be a t -spanner path in G'' between x and y , and let P_j be the path in T_j between y and q . The concatenation Q of P_i , P_{xy} , and P_j is a path in G' between p and q .

Since both P_i and P_j are subpaths of P , it follows from Lemma 1 that each edge on P_i and P_j has length at most $t|pq|$. Since T_i and T_j contain $O(n/k)$ vertices, it follows that the sum of the lengths of P_i and P_j is $O(n/k) \cdot |pq|$. The length of P_{xy} is at most $t|xy|$ which, by Lemma 3, is also $O(n/k) \cdot |pq|$. Thus, the length of Q is $O(n/k) \cdot |pq|$. ■

We take for G the t -spanner of Das and Heffernan (1996). This spanner can be computed in $O(n \log n)$ time, and each vertex has degree at most three. Given G , its minimum spanning tree T can be computed in $O(n \log n)$ time. Since a centroid edge can be computed in $O(n)$ time, the subtrees T_0, T_1, \dots, T_ℓ can be computed in $O(n \log n)$ time. Finally, we take for G'' the t -spanner of Das and Heffernan. This spanner G'' can be computed in $O(k \log k) = O(n \log n)$ time, and each vertex has degree at most three.

For these choices of G and G'' , the graph G' has stretch factor $O(n/k)$, it contains $n - 1 + O(k)$ edges, and it can be computed in $O(n \log n)$ time. We analyze the degree of G' : Consider any vertex p of G' . If $p \notin X$, then the degree of p in G' is equal to the degree of p in T , which is at most three. Assume that

$p \in X$. The graph G'' contains at most three edges that are incident to p . Similarly, the tree T contains at most three edges that are incident to p , but, since $p \in X$, at least one of these three edges is not an edge of G' . Therefore, the degree of p in G' is at most five. Thus, each vertex of G' has degree at most five.

Let c be a constant such that the graph G' contains at most $n - 1 + ck$ edges.

4 The main result

We are now ready to prove the main result of this paper. Let S be a set of n points in \mathbb{R}^d , and let k be an integer with $0 \leq k \leq n$. Consider the constant c that was introduced above.

First assume that $k < c$. Let G be a t -spanner for S , for some constant t , in which each vertex has degree at most three, and let G' be a minimum spanning tree of G . Then, G' has $n - 1 \leq n - 1 + k$ edges, degree at most three and, by Lemma 2, the stretch factor of G' is at most $t(n - 1)$, which is $O(n/(k + 1))$.

If $c \leq k \leq n$, then we apply the results of Section 3 with k replaced by k/c . This gives a graph G' with at most $n - 1 + k$ edges, degree at most five, and stretch factor $O(n/(k + 1))$. Thus, we have proved the following result:

Theorem 1 *Let S be a set of n points in \mathbb{R}^d , and let k be an integer with $0 \leq k \leq n$. In $O(n \log n)$ time, a graph with vertex set S can be computed that has the following properties:*

1. *The graph contains at most $n - 1 + k$ edges.*
2. *The graph has stretch factor $O(n/(k + 1))$.*
3. *Each vertex of the graph has degree at most five.*

Aronov *et al.* (2005) gave an example of a set S of n points in the plane, such that every connected graph with vertex set S and consisting of $n - 1 + k$ edges has stretch factor $\Omega(n/(k + 1))$. Therefore, the result in Theorem 1 is optimal. An interesting open problem is whether the degree can be reduced.

Acknowledgements

The author would like to thank the members of the Algorithms Seminar Group at Carleton University for pointing out a serious error in a previous version of this paper.

References

- Aronov, B., de Berg, M., Cheong, O., Gudmundsson, J., Haverkort, H. & Vigneron, A. (2005), Sparse geometric graphs with small dilation, in 'Proceedings of the 16th International Symposium on Algorithms and Computation', Lecture Notes in Computer Science, Springer-Verlag, Berlin.
- Das, G. & Heffernan, P. J. (1996), 'Constructing degree-3 spanners with other sparseness properties', *International Journal of Foundations of Computer Science* **7**, 121–135.
- Eppstein, D. (2000), Spanning trees and spanners, in J.-R. Sack & J. Urrutia, eds, 'Handbook of Computational Geometry', Elsevier Science, Amsterdam, pp. 425–461.
- Erickson, J. (1995), On the relative complexities of some geometric problems, in 'Proceedings of the 7th Canadian Conference on Computational Geometry', pp. 85–90.

Gudmundsson, J. & Knauer, C. (2006), Dilation and detours in geometric networks, in T. F. Gonzalez, ed., 'Handbook on Approximation Algorithms and Metaheuristics', Chapman & Hall/CRC, Boca Raton.

Salowe, J. S. (1991), 'Constructing multidimensional spanner graphs', *International Journal of Computational Geometry & Applications* **1**, 99–107.

Smid, M. (2000), Closest-point problems in computational geometry, in J.-R. Sack & J. Urrutia, eds, 'Handbook of Computational Geometry', Elsevier Science, Amsterdam, pp. 877–935.

Vaidya, P. M. (1991), 'A sparse graph almost as good as the complete graph on points in K dimensions', *Discrete & Computational Geometry* **6**, 369–381.

Graph Orientation Algorithms to Minimize the Maximum Outdegree*

Yuichi Asahiro¹Eiji Miyano²Hirotaka Ono³Kouhei Zenmyo⁴

¹ Department of Social Information Systems, Kyushu Sangyo University,
Fukuoka 813-8503, Japan. asahiro@is.kyusan-u.ac.jp

² Department of Systems Innovation and Informatics, Kyushu Institute of Technology,
Fukuoka 820-8502, Japan. miyano@ces.kyutech.ac.jp

³ Department of Computer Science and Communication Engineering, Kyushu University,
Fukuoka 812-8581, Japan. ono@csce.kyushu-u.ac.jp

⁴ Department of Systems Innovation and Informatics, Kyushu Institute of Technology,
Fukuoka 820-8502, Japan. kouhei@theory.ces.kyutech.ac.jp

Abstract

We study the problem of orienting the edges of a weighted graph such that the maximum weighted outdegree of vertices is minimized. This problem, which has applications in the guard arrangement for example, can be shown to be \mathcal{NP} -hard generally. In this paper we first give optimal orientation algorithms which run in polynomial time for the following special cases: (i) the input is an unweighted graph, or more generally, a graph with identically weighted edges, and (ii) the input graph is a tree. Then, by using those algorithms as sub-procedures, we provide a simple, combinatorial, $\min\{\frac{w_{max}}{w_{min}}, (2-\varepsilon)\}$ -approximation algorithm for the general case, where w_{max} and w_{min} are the maximum and the minimum weights of edges, respectively, and ε is some small positive real number that depends on the input.

Keywords: graph orientation, min-max optimization, \mathcal{NP} -hardness, approximation algorithms.

1 Introduction

1.1 Brief History of Graph Orientation

Let $G = (V, E, w)$ be a simple, undirected, weighted graph with a vertex set V , an edge set E , and a positive integral weight function $w : E \rightarrow \mathbb{Z}^+$, where each edge is a pair $\{u, v\}$ of vertices $u, v \in V$. An *orientation* Λ of the graph G is an assignment of direction to each edge $\{u, v\} \in E$. The graph orientation is a well-studied area in the fields of graph theory and combinatorial optimization, and has a long history. In 1939, Robbins stated a seminal result on the relation between the orientation and the *connectivity*: A graph has a strongly connected orientation if and only if it is 2-edge-connected. Thereafter, a variety of classes of questions have been introduced and investigated in the literature, including the characterization of oriented graphs satisfying the specified connectivity, and the problem of finding orientations with topological properties such as the *tightness* (an orientation of G whose diameter is the same as the

diameter of G is called tight), the *degree constraint* and the *acyclicity*. For example, as a classical result, Nash-Williams (1960) characterized graphs having k -edge-connected orientations. Chung, Garey and Tarjan (1985) provided a linear-time algorithm for checking whether a graph has a strongly connected orientation and finding one if it does. In 1978 Chvátal and Thomassen introduced the following problem called *Oriented Diameter*: Given a graph G , find a strongly connected orientation of G with the minimum diameter. They proved that the problem is \mathcal{NP} -hard for general graphs. Then, Fomin, Matala and Rapaport (2004) showed that the problem remains \mathcal{NP} -hard even if the graph is restricted to a subset of chordal graphs and gave approximability and non-approximability results.

The orientation with the degree constraint is also popular. Chrobak and Eppstein (1991) studied the problem of orienting the edges of a planar graph such that the outdegree of each vertex is bounded, and proved that a 3-bounded outdegree orientation and a 5-bounded outdegree *acyclic* orientation can be surely constructed in linear time for every planar graph. Recently, Biedl, Chan, Ganjali, Hajiaghayi, and Wood (2005) studied the problem of determining a *balanced acyclic* orientation of unweighted graphs, where balanced means that the difference between the (unweighted) indegree and outdegree of each vertex is minimized, and proved that it is \mathcal{NP} -hard and there is a $\frac{13}{8}$ -approximation algorithm. The \mathcal{NP} -hardness of Biedl et al.'s result is for graphs with maximum degree six. Kára, Kratochvíl, and Wood (2005) closed the gap, by proving the \mathcal{NP} -hardness for graphs with maximum degree four, and also showed that it remains \mathcal{NP} -hard for planar graphs with maximum degree six, and so on. The orientation with the degree constraint has several applications in the fields of data structures and graph drawing as mentioned in (Chrobak & Eppstein 1991, Biedl, Chan, Ganjali, Hajiaghayi, & Wood 2005).

1.2 Our Problems and Results

In this paper we propose a new variant of the graph orientation by considering a natural objective function, the *Minimum Outdegree Orientation* problem (MOO):

MOO

Instance: A simple, weighted, undirected graph $G = (V, E, w)$.

Question: Find an orientation Λ of G which minimizes the maximum weighted outdegree of vertices.

Copyright ©2006, Australian Computer Society, Inc. This paper appeared at Computing: The Australasian Theory Symposium (CATS2006), Hobart, Australia. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 51. Barry Jay and Joachim Gudmundsson, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.

*Supported in part by Grants-in-Aid for Scientific Research on Priority Areas 16092223, Young Scientists (B)'s 15700019, 15700021, and 17700022, and Scientific Research (B) 14380145 from the Japanese Ministry of Education, Science, Sports and Culture.

The MOO is originally motivated by the *Capacitated Guard Arrangement* problem, which is one of the *Art Gallery* problems (Chvátal 1975, O'Rourke 1987): The original Art Gallery problem for a polygon P is to find a minimum set Q of points in P such that every point of P is visible from some point in Q and to place one guard on each point in Q , $|Q|$ guards in total. If P can be viewed as a graph (a set of line segments such as a mesh) and the guards have to be placed only on its vertices (or intersections of line segments), then the Art Gallery problem can be straightforwardly formulated by the *Vertex Cover* problem, i.e., a guard placed on a vertex must watch (cover) all the edges incident with the vertex and the goal is to minimize the number of guards arranged. In the *Capacitated Guard Arrangement*, guards are positioned on all vertices but they can cover only the specified number of edges, and its goal is to minimize the capacity of each guard, which is represented by the MOO.

In this paper we show the following results:

- We prove that, unfortunately, the MOO is generally \mathcal{NP} -hard.
- But, fortunately, we can obtain optimal orientation algorithms which run in polynomial time for the following special cases: (i) the input is an unweighted graph, or more generally, a graph with identically weighted edges, and (ii) the input graph is a tree.
- Furthermore, by using those algorithms as sub-procedures, we provide a simple, combinatorial, $\min\{\frac{w_{max}}{w_{min}}, (2 - \varepsilon)\}$ -approximation algorithm for the general case, where w_{max} and w_{min} are the maximum and the minimum weights of edges, respectively, and ε is some small positive real number that depends on the input.

Note that Venkateswaran (2004) previously investigated the unweighted version of the MOO, for which he also provided an $O(|E|^2)$ orientation algorithm. In this paper, we show that the $O(|E|^2)$ bound can be reduced.

1.3 Related Work

The difficulty of solving the MOO exactly and/or approximately can be closely related to the intractability of the *minimum makespan scheduling*, which is a central problem in the scheduling area, and well studied from the viewpoint of the approximability. In the *scheduling on unrelated parallel machines* ($R||C_{max}$ in the now-standard notation), given a set J of jobs, a set M of machines, and the time $p_{ij} \in \mathbb{Z}^+$ taken to process job $j \in J$ on machine $i \in M$, its goal is to find a job scheduling so as to minimize the makespan, i.e., the maximum processing time of any machine. Lenstra, Shmoys, and Tardos (1990) gave a polynomial time 2-approximation algorithm that is based on the LP-formulation for the general version of $R||C_{max}$ and its $\frac{3}{2}$ inapproximability result. Schuurman and Woeginger (1999) stated that it is even interesting to improve on the results of (Lenstra, Shmoys, & Tardos 1990) in the so-called *restricted assignment* variant of $R||C_{max}$, in which the processing time p_{ij} of job j on machine i is identically fixed p_j , but the job can only be processed on a subset of the machines. In the MOO, the processing time p_j of job j corresponds to the weight $w(\{u, v\})$ of edge $\{u, v\}$ and its assignable machines correspond to two terminals u and v . Hence, an orientation of $\{u, v\}$ is regarded as a job assignment. Only for the simpler problems of the restricted $R||C_{max}$, an FPTAS (Horowitz & Sahni 1976) or a polynomial time

algorithm (Pinedo 2002) were provided, but there are a lot of unknown questions for the general cases.

1.4 Organization

The rest of this paper is organized as follows. Section 2 introduces some notations. Then we prove the \mathcal{NP} -hardness of the general MOO in Section 3. In Section 4 we consider easy subclasses of the MOO and provide two polynomial time algorithms for them. In Section 5 we give a new combinatorial $\min\{\frac{w_{max}}{w_{min}}, (2 - \varepsilon)\}$ -approximation algorithm for the general MOO based on the polynomial time algorithms of Section 4. Finally, we conclude in Section 6.

2 Preliminaries

Let $G = (V, E, w)$ be a simple, undirected, weighted graph, where V , E , and w denote a set of vertices, a set of edges, and an integral weight function, $w : E \rightarrow \mathbb{Z}^+$, respectively. Let w_{max} and w_{min} be the maximum and the minimum weights of edges, respectively. Throughout the paper, let $|V| = n$ and $|E| = m$ for the input graph. By $\{u, v\}$ for $u, v \in V$ we denote the undirected edge with ends in u and v , and by (u, v) the directed arc, directed from u toward v . Let $d(v)$ represent a degree of a vertex v and $D(G)$ the maximum degree of a graph G . An *orientation* Λ of the undirected graph G is an assignment of direction to each edge $\{u, v\} \in E$, i.e., (u, v) or (v, u) . Equivalently, we can regard the orientation Λ as a set of directed arcs such that Λ includes exactly either one of (u, v) or (v, u) for each $\{u, v\} \in E$. A *directed path* P of length k from a vertex v_0 to a vertex v_k in a directed graph $G = (V, A, w)$ is a set $\{(v_{i-1}, v_i) \mid (v_{i-1}, v_i) \in A, i = 1, 2, \dots, k\}$ of arcs, which is also denoted by a sequence $\langle v_0, v_1, \dots, v_k \rangle$ for simplicity. For the path P , the path of its reverse order is denoted by \bar{P} , i.e., $\bar{P} = \langle v_k, v_{k-1}, \dots, v_0 \rangle$.

We say a vertex i *dominates* j if an arc (i, j) is in orientation Λ , that we represent by $i \rightarrow j$. $\delta_\Lambda^+(v)$ and $\delta_\Lambda^-(v)$ under an orientation Λ denote the total weights of outgoing arcs and that of incoming arcs of a vertex v in the weighted directed graph $G(V, A, w)$, which we call the *weighted outdegree* and the *weighted indegree* of v , respectively. Let $\delta(v) = \delta_\Lambda^+(v) + \delta_\Lambda^-(v)$. (Note that $\delta(v)$ does not change depending on Λ .) Then the *cost* of an orientation Λ for a graph G is defined to be $\Delta_\Lambda(G) = \max_{v \in V} \{\delta_\Lambda^+(v)\}$.

By $G[V']$ we denote the subgraph induced by $V' \subseteq V$ for G , simply represented by $G[V'] \subseteq G$. Let $W(G) = \sum_{\{u, v\} \in E} w(\{u, v\})$ and $\ell(G) = W(G)/|V|$ be the total weight of edges and the average weighted outdegrees of vertices in G , respectively. Also, as for all the induced subgraphs H 's of G , let $L(G) = \max_{H \subseteq G} \{\ell(H)\}^*$.

Every orientation has the following trivial lower bounds caused by the maximum weight of edges, and by the average weighted outdegrees of vertices:

Proposition 1 *For an undirected weighted graph G and any orientation Λ , $\Delta_\Lambda(G) \geq w_{max}$.*

Proposition 2 *For an undirected weighted graph G and any orientation Λ , $\Delta_\Lambda(G) \geq \lceil \ell(G) \rceil$.*

Let OPT denote an optimal orientation. We say a graph orientation algorithm is a σ -approximation algorithm if $ALG(G)/OPT(G) \leq \sigma$ holds for any undirected graph G , where $ALG(G)$ is the objective value

*Note that $L(G)$ can be obtained by a polynomial time algorithm (Gallo, Grigoriadis, & Tarjan 1989), though $L(G)$ is introduced only for the analysis here.

of a solution obtained by the algorithm for G , and $OPT(G)$ is that of an optimal solution.

3 \mathcal{NP} -Hardness

In this section we show the \mathcal{NP} -hardness of the MOO for the most general case. Let us consider the following decision version of the MOO:

MOO(k)

Instance: A simple undirected graph $G = (V, E, w)$, and an integer k .

Question: Is there an orientation Λ such that $\Delta_\Lambda(G) \leq k$?

The proof of its \mathcal{NP} -hardness is by a polynomial time reduction from the *Partition* problem.

Partition

Instance: A set $S = \{s_1, s_2, \dots, s_n\}$ of n positive integers.

Question: Is there a subset $S' \subseteq S$ such that $\sum_{s_i \in S'} s_i = \sum_{s_i \in S-S'} s_i$?

Theorem 3 *The MOO(k) is \mathcal{NP} -complete.*

Proof. Let us consider a restricted set of instances of the Partition that satisfy the following two conditions (1) $\sum_{s_i \in S} s_i$ is even, and (2) for all i , $s_i < \sum_{s_i \in S} s_i/2$. Even with these restrictions, the Partition is still \mathcal{NP} -hard because an instance which does not satisfy either of these conditions can be trivially solved in polynomial time.

From an instance $S = \{s_1, s_2, \dots, s_n\}$ of the Partition, we construct a weighted undirected graph $G = (V, E, w)$. For example, if $S = \{1, 2, 4, 5, 6\}$, the constructed graph G is as shown in Figure 1. The detailed construction is as follows: The vertex set V of G is divided into three types of vertices: (i) Item vertices v_1, v_2, \dots, v_n associated with n items in S , (ii) Subset vertices a and b , and (iii) Auxiliary vertices u_1, u_2 , and u_3 . The total number of vertices is $n + 5$. Let us define $K = \sum_{s_i \in S} s_i/2$. The edge set E contains the following four types of edges: (i) n edges $\{a, v_i\}$'s with weight s_i , i.e., $w(\{a, v_i\}) = s_i$ for all i 's, (ii) n edges $\{b, v_i\}$'s with weight s_i for all i 's, (iii) three edges $\{u_1, u_2\}$, $\{u_2, u_3\}$, and $\{u_3, u_1\}$ with weight K , and (iv) n edges $\{u_1, v_i\}$'s with weight $K - s_i$ for all i 's. The total number of edges is $3n + 3$. Finally, we set $k = K$. This construction of G can be obviously executed in polynomial time.

Since clearly the MOO(k) is in \mathcal{NP} , we only show in the next its \mathcal{NP} -hardness: We prove that there is $S' \subseteq S$ such that $\sum_{s_i \in S'} s_i = K$ if and only if there is an orientation Λ of G such that $\Delta_\Lambda(G) = K$.

Lemma 4 *If there exists a subset $S' \subseteq S$ such that $\sum_{s_i \in S'} s_i = K$, then there exists an orientation Λ of G such that $\Delta_\Lambda(G) = K$.*

Proof. Suppose that there exists a subset $S' \subseteq S$ such that $\sum_{s_i \in S'} s_i = K$. Consider the following orientation Λ based on S' : (i) $u_1 \rightarrow u_2, u_2 \rightarrow u_3, u_3 \rightarrow u_1$, (ii) $v_i \rightarrow u_1$ for all i 's. (iii) If $s_i \in S'$, $v_i \rightarrow a$ and $b \rightarrow v_i$; otherwise $a \rightarrow v_i$ and $v_i \rightarrow b$, for all i 's.

(i) For the auxiliary vertices, one can verify that $\delta_\Lambda^+(u_1) = \delta_\Lambda^+(u_2) = \delta_\Lambda^+(u_3) = K$ holds. (ii) As for each item vertex v_i , since $v_i \rightarrow u_1$ and either of $v_i \rightarrow a$ or $v_i \rightarrow b$ holds, $\delta_\Lambda^+(v_i) = (K - s_i) + s_i = K$. (iii) For each $s_i \in S'$, $b \rightarrow v_i$ holds, and for each $s_i \in S - S'$, $v_i \rightarrow b$ also holds. Therefore $\delta_\Lambda^+(b) =$

$\sum_{s_i \in S'} w(v_i, b) = \sum_{s_i \in S'} s_i = K$. As for the vertex a , we can show $\delta_\Lambda^+(a) = K$ by a similar discussion. \square

Lemma 5 *If there does not exist a subset $S' \subseteq S$ such that $\sum_{s_i \in S'} s_i = K$, then there does not exist an orientation Λ of G such that $\Delta_\Lambda(G) \leq K$.*

Proof. Suppose that there does not exist a subset $S' \subseteq S$ such that $\sum_{s_i \in S'} s_i = K$. We show that $\Delta_\Lambda(G) > K$ for any orientation Λ in the following.

First of all, let us consider a special condition (C) that either of $b \rightarrow v_i$ and $v_i \rightarrow a$, or, $a \rightarrow v_i$ and $v_i \rightarrow b$ holds for all the item vertices v_i 's in the orientation Λ . Note that the former condition corresponds to the case that the item $s_i \in S'$ and the latter $s_i \in S - S'$. For any subset S' , $\sum_{s_i \in S'} s_i \neq K$ by the above assumption, which means $\sum_{s_i \in S'} s_i > K$ or $\sum_{s_i \in S-S'} s_i > K$. Therefore under any orientation Λ that satisfies the condition (C), $\delta_\Lambda^+(a) > K$ or $\delta_\Lambda^+(b) > K$, hence $\Delta_\Lambda(G) > K$.

Next, as the remaining cases, we consider orientations that do not satisfy the special condition (C). Those orientations are divided into the following two cases, (i) there is an item vertex v_j such that $v_j \rightarrow a$ and $v_j \rightarrow b$, and (ii) there is an item vertex v_j such that $a \rightarrow v_j$ and $b \rightarrow v_j$.

Take a look at the case (i) that $v_j \rightarrow a$ and $v_j \rightarrow b$ for some j in the orientation Λ . Since $v_j \rightarrow a, v_j \rightarrow b$, $w((v_j, a)) = w((v_j, b)) = s_j$ and $w((v_j, u_1)) = K - s_j$, one can see that $\delta_\Lambda^+(v_j) \leq K$ holds if and only if $u_1 \rightarrow v_j$. Assuming $u_1 \rightarrow v_j$, however, either of $\delta_\Lambda^+(u_1) \geq 2K - s_j$ or $\max\{\delta_\Lambda^+(u_2), \delta_\Lambda^+(u_3)\} = 2K$ holds whatever orientation of the subgraph $G[\{u_1, u_2, u_3\}]$ is selected. Since $s_j < K$, we conclude that $\Delta_\Lambda(G) > K$ in this case.

Let us proceed to the case (ii). Consider an orientation Λ under which $a \rightarrow v_j$ and $b \rightarrow v_j$ for exactly one vertex v_j (and there exist no vertices satisfying the condition of the case (i)). Let Λ' be an orientation that satisfies the condition (C) in which orientation for edges except $\{b, v_j\}$ is the same as Λ , and $v_j \rightarrow b$ instead of $b \rightarrow v_j$ in Λ . $K < \max\{\delta_{\Lambda'}^+(a), \delta_{\Lambda'}^+(b)\} \leq \max\{\delta_\Lambda^+(a), \delta_\Lambda^+(b)\}$ holds since the replacement of $v_j \rightarrow b$ by $b \rightarrow v_j$ only increases the weighted outdegree of the vertex b . Therefore, also in this case $\Delta_\Lambda(G) > K$. In the case that more than one such vertex v_j 's exist, we can show this lemma by a similar discussion. \square

From the above two lemmas, the \mathcal{NP} -hardness of the MOO(k) is shown, that concludes Theorem 3. \square

4 Optimal Algorithms for Special Cases

In this section we present two polynomial time algorithms when an instance is (1) a weighted tree, and (2) an unweighted graph, or more generally, a graph such that all the weights of their edges are identical. The basic ideas of those algorithms are simple but they will play important roles in our approximation algorithms for the most general case.

4.1 Trees

Recall that the maximum weighted outdegree of a graph G under every orientation is at least the maximum weight w_{max} of their edges as mentioned in Proposition 1. We can efficiently find an orientation Λ such that $\Delta_\Lambda(G) = w_{max}$ if G is a tree:

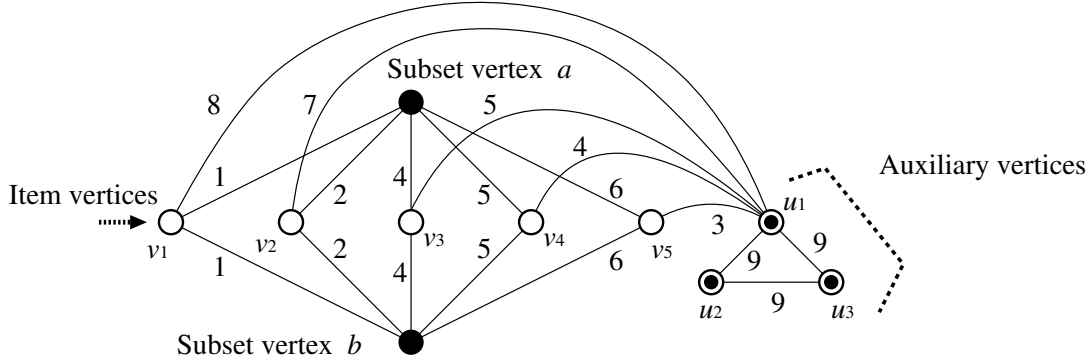


Figure 1: Reduction from an instance $S = \{1, 2, 4, 5, 6\}$ of the Partition problem.

Theorem 6 *For trees, an optimal solution can be obtained in $O(n)$ time.*

Proof. All we have to do is to orient all edges toward a root chosen arbitrary, which can be obviously achieved in linear time. (This linear time algorithm will be refereed to as **Convergence** later.) \square

4.2 Identical Weights

Here, in order to make our basic idea clear, we give our elementary algorithm that is optimal if all the weights of edges are identical. A similar optimal algorithm has been independently shown in (Venkateswaran 2004), but our proof of the optimality is much simpler. Note that now we consider connected graphs as input, $n - 1 \leq m$.

Theorem 7 (Venkateswaran 2004) *If all the weights of edges are identical, an optimal solution can be obtained in $O(m^2)$ time.*

Proof. We can consider the weight of the edges is one without loss of generality, and hence the weighted outdegree in this case is the same as the outdegree of the unweighted graph. The following **Reverse** is an algorithm to solve this case optimally, whose basic strategy is quite straightforward: Observe an unweighted graph and an orientation illustrated in Figure 2-(a). One can see that the outdegrees of the four vertices u_0, u_1, u_2 and u_3 are $(5, 3, 3, 1)$, respectively, and the maximum outdegree is five. However, if we reverse the orientation of the directed path $\langle u_0, u_1, u_2, u_3 \rangle$ as shown in Figure 2-(b), the outdegrees become $(4, 3, 3, 2)$ and the maximum outdegree decreases to four without increasing the outdegrees of intermediate vertices u_1 and u_2 . **Reverse** repeatedly finds such a directed path and reduces its maximum outdegree by reversing its direction:

Algorithm Reverse:

Input: An unweighted graph $G = (V, E)$.

Output: An arc set Λ which determines directions of edges in E .

Step 0: Set $\Lambda = \emptyset$.

Step 1: Find arbitrary orientation of the graph G and update Λ .

Step 2: Compute the (weighted) outdegree $\delta_\Lambda^+(v)$ for each vertex v . Let u be a vertex having maximum outdegree among all vertices (in case of ties, select one vertex arbitrary).

Step 3: Find a directed path $P = \langle u, v_1, \dots, v_k \rangle$ of length $k(k \geq 1)$ that satisfies

- $\delta_\Lambda^+(v_i) \leq \delta_\Lambda^+(u)$ for $1 \leq i \leq k - 1$, and
- $\delta_\Lambda^+(v_k) \leq \delta_\Lambda^+(u) - 2$.

If such a path P exists, then set $\Lambda = \Lambda \setminus P \cup \bar{P}$ (i.e., orient the path P in reverse order) and goto Step 2. Otherwise output Λ and halt.

At first, we estimate the running time. Step 0 is done in $O(1)$ time. Both of Steps 1 and 2 require $O(m)$ time. Step 3 can be done with a breadth first search and it also takes $O(m)$ time. Therefore, the number of iterations of Steps 2 and 3 determines the total amount of time. Consider a subset of vertices $M = \{v \mid v \in V, \delta_\Lambda^+(v) \text{ is the maximum, or it is (the maximum } - 1)\}$. (This set does not appear in the description of the algorithm and does only for this analysis.)

In Step 3, a vertex u in M is selected as the starting vertex of a directed path. The modification of the orientation Λ in Step 3, (i) decreases the outdegree of the vertex u by one and then u still belongs to M at the next iteration, and (ii) increases the outdegree of some vertex $v_k \in V - M$ and then v_k may become to be in M at the next iteration. Therefore, the outdegree of a vertex in M monotonically decreases after it belongs to M . As mentioned before, the outdegrees of intermediate vertices v_1, \dots, v_{k-1} in the path remain unchanged in Step 3.

From Proposition 2 and the above observation, each vertex v can be the starting vertex of such paths by the limited number of times $d(v) - m/n$, since now we consider the weight of each edge is one. Summing up $d(v) - m/n$ over all the vertices gives an upper bound on the number of iterations of Steps 2 and 3, that is $\sum_{v \in V} d(v) - m = 2m - m = m$. Therefore, since Steps 2 and 3 takes $O(m)$ time, the total running time of the algorithm is $O(m^2)$ time.

The rest of the proof is to show optimality of the algorithm. We begin with the following proposition.

Proposition 8 *For a graph $G(V, E, w)$ and its induced subgraph $G'(V', E', w)$, $\Delta_{OPT}(G) \geq \Delta_{OPT'}(G')$, where OPT and OPT' are optimal solutions for G and G' , respectively.*

Proof. Assume $\Delta_{OPT}(G) < \Delta_{OPT'}(G')$. We can obtain an orientation Λ for G' from OPT by extracting arcs connecting between two vertices in V' . Since G' is a subgraph of G and $\delta_{OPT}^+(v) \leq \Delta_{OPT}(G)$ for every vertex $v \in V'$, $\delta_\Lambda^+(v) \leq \Delta_{OPT}(G) < \Delta_{OPT'}(G')$ holds for every vertex $v \in V'$. This contradicts that OPT' is an optimal solution for G' . \square

Let v_p be the vertex having the maximum outdegree $p = \delta_\Lambda^+(v_p)$ under an orientation Λ obtained

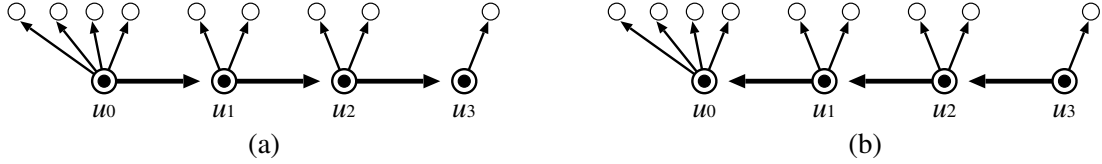


Figure 2: (a) $(\delta^+(u_0), \delta^+(u_1), \delta^+(u_2), \delta^+(u_3)) = (5, 3, 3, 1)$ and the maximum outdegree is five, but (b) $(\delta^+(u_0), \delta^+(u_1), \delta^+(u_2), \delta^+(u_3)) = (4, 3, 3, 2)$ and the maximum outdegree decreases to four.

by the algorithm. All the vertices reachable from v_p by following a directed path have the outdegree of at least $p - 1$ from the halting criteria in Step 3. Let the set of those vertices and v_p be V' and consider an induced subgraph $G[V']$.

Lemma 9 $\Delta_{OPT}(G[V']) = \Delta_\Lambda(G[V']) = p$, that is, $\Delta_\Lambda(G[V'])$ is optimal.

Proof. The number of edges in $G[V']$ is at least $p + (p - 1)(|V'| - 1) = (p - 1)|V'| + 1$. From Proposition 2, $\Delta_{OPT}(G[V']) \geq \lceil ((p - 1)|V'| + 1)/|V'| \rceil = \lceil p - 1 + 1/|V'| \rceil = p = \Delta_\Lambda(G[V'])$. \square

We can define such a subgraph $G[V']$ for each vertex having the maximum outdegree of p under the orientation Λ , and Lemma 9 holds for every such subgraph. Since p is the maximum outdegree under Λ and $\Delta_{OPT}(G) \geq \Delta_{OPT}(G[V']) = p$ for any such $G[V']$'s from Proposition 8, the output Λ of the algorithm is an optimal orientation for G . This ends the proof of Theorem 7. \square

A Faster Algorithm. Algorithm **Reverse** can find an optimal orientation for unweighted graphs in polynomial time, however, there might be a possibility of improvements: In Step 3 of **Reverse**, the algorithm finds just one simple directed path in $O(m)$ time and iterates that m times, but if we can find several edge-disjoint directed paths at one blow, it might reduce the number of iterations. To this end, we consider the following network for a given orientation Λ and a parameter k : Let A denote the arc set of weighted directed graph $G' = (V, A)$ obtained by applying the orientation Λ to the input graph $G = (V, E)$. For G' , we consider two subsets $V_k^+ = \{v \mid \delta_\Lambda^+(v) > k\}$ and $V_k^- = \{v \mid \delta_\Lambda^+(v) < k\}$ of V . Note that $V = V_k^+ \cup V_k^- \cup \{v \mid \delta_\Lambda^+(v) = k\}$. Then the network $\mathcal{N}_k(\Lambda)$ we construct is defined as $\mathcal{N}_k(\Lambda) = (\tilde{V}, \tilde{E}) = (\{s\} \cup \{t\} \cup V, A \cup A_k^+ \cup A_k^-)$, where

$$A_k^+ = \bigcup_{v \in V_k^+} \{e_{(v,i)}^+ = (s, v) \mid i = k + 1, \dots, \delta_\Lambda^+(v)\},$$

$$A_k^- = \bigcup_{v \in V_k^-} \{e_{(v,i)}^- = (v, t) \mid i = \delta_\Lambda^+(v) + 1, \dots, k\},$$

and the capacities $\text{cap}(e) = 1$ for $\forall e \in \tilde{E}$. See Figure 3. The above two sets of arcs may contain parallel arcs for each vertex in V_k^+ and V_k^- . The number of vertices is $|V| + 2$ and that of arcs is at most $3|E|$ in $\mathcal{N}_k(\Lambda)$ (although the exact number of arcs depends on k and Λ).

Lemma 10 *The size of the maximum flow for a network $\mathcal{N}_k(\Lambda)$ is $f_k = \sum_{u \in V_k^+} (\delta_\Lambda^+(u) - k)$ if and only if the answer of $MOO(k)$ is “yes”. (The proof will be given later.)*

Since the network \mathcal{N}_k is a unit capacity flow network, this lemma says that the followings are equivalent for an undirected graph G : (1) G under an orientation includes f_k edge-disjoint directed paths between V_k^+ and V_k^- . (2) G has an orientation with the maximum outdegree bounded by k . The maximum flow problem for a unit capacity flow network can be solved in $O(|E_{\mathcal{N}}|^{3/2})$ time, where $|E_{\mathcal{N}}|$ is the number of edges of the flow network (Even & Tarjan 1975). Note that this algorithm can work for networks with parallel edges in the same upper bound. Since \mathcal{N}_k has at most $3m$ edges, we immediately obtain the following theorem.

Theorem 11 *The $MOO(k)$ can be solved in $O(m^{3/2})$ time, if all the edge weights are identical.* \square

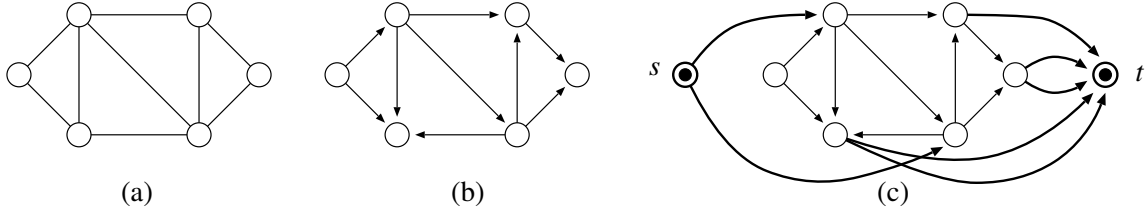
We can find the optimal k and an orientation by doing the binary search of Theorem 11 as its engine.

Corollary 12 *The MOO can be solved in $O(m^{3/2} \cdot \log \Delta_{OPT}(G))$ time, if all the edge weights are identical.* \square

Proof of Lemma 10. (Only-if part) Suppose that the size of the maximum flow is f_k . Since the network $\mathcal{N}_k(\Lambda)$ has only edges with capacity one, there exist f_k edge-disjoint paths from s to t . Therefore, by the construction of the network, each $u \in V_k^+$ has $(\delta_\Lambda^+(u) - k)$ directed paths to some vertices in V_k^- , that are edge-disjoint to each other. By applying the procedure of reversing in Step 3 of **Reverse** to those paths in order, we can reduce the outdegree of the vertex u to k . Since the outdegree of any $v \in V_k^-$ does not exceed k by this operation, the maximum outdegree of the resulting orientation is k . (If part) In this case, if we apply the algorithm **Reverse** to the input graph G with Λ as the initial orientation instead of a random one, we can find an orientation OPT such that $\Delta_{OPT}(G) \leq k$.

Consider the collection of directed paths processed in Step 3 of **Reverse** to obtain OPT , that are represented by P_1, P_2, \dots, P_h and supposed to be obtained in this order. Here it is important to note that some edge $\{u, v\}$ may appear several times in those paths but its directions differ; (u, v) may be included in some paths though (v, u) is also in others. Also we assume that the sequence P_1, P_2, \dots, P_h is minimal in a sense that for $1 \leq i \leq h - 1$, even after processing P_i the maximum outdegree is still greater than k but just after processing P_h the maximum outdegree decreases to k . Although **Reverse** may process a sequence of paths that is not minimal, in such a case it is sufficient to consider only its minimal subset. Because of the optimality of **Reverse**, $h = f_k$. In the following, we show that the sequence of the paths P_1, P_2, \dots, P_h can be transformed into a sequence of paths in which every path is edge-disjoint to others. Moreover the algorithm **Reverse** runs correctly for the modified sequence.

After reversing P_i to \bar{P}_i in Step 3, we suppose to obtain an orientation Λ_i , i.e., at the beginning of the

Figure 3: (a) Input graph G (b) An orientation Λ (c) The network $\mathcal{N}_2(\Lambda)$

i -th execution of Step 3, we have Λ_{i-1} (Λ in case $i = 0$) and update it to Λ_i in Step 3, and eventually we obtain $OPT = \Lambda_h$.

We divide the sequence of paths P_1, P_2, \dots, P_h into $\Delta_\Lambda(G) - k$ ($\stackrel{\text{def}}{=} z$) groups based on the decrease of the maximum outdegree of vertices, $P_{h_0} (= P_1), \dots, P_{h_1-1}, P_{h_1}, \dots, P_{h_2-1}, P_{h_2}, \dots, P_{h_z-1} (= P_h)$ such that $\Delta_{\Lambda_i}(G) = \Delta_{\Lambda_{i+1}}(G)$ for $h_j \leq i \leq h_{j+1} - 2$ and $\Delta_{\Lambda_{i-1}}(G) - 1 = \Delta_{\Lambda_i}(G)$ for $i = h_j$, where $0 \leq j \leq z - 1$.

Let us consider the first group of the paths, $P_{h_0} (= P_1), \dots, P_{h_1-1}$. We show that even if they are not edge-disjoint, we can transform them to edge-disjoint ones. Suppose that they are not edge-disjoint. Let $2 \leq q \leq h_1 - 1$ be the (smallest) index of a path such that P_1, \dots, P_{q-1} are edge-disjoint but P_r and P_q are not for some $1 \leq r \leq q - 1$. We focus on the subsequence P_1, \dots, P_q of paths.

Based on the rule of the grouping of the paths, each path P_i in the first group starts from a vertex u with $\delta_{\Lambda_{i-1}}^+(u) = \Delta_\Lambda(G)$. Therefore, each vertex can be the starting vertex only once in the group, because, otherwise its outdegree decreases by at least two that contradicts the path starting from it is processed in the first group.

Step 3 of **Reverse** only changes the outdegrees of the first and the last vertices of a path. Especially, as for the last vertices, their outdegrees increase but never reaches to $\Delta_\Lambda(G)$, namely, such vertices can not be a start vertex of such a path in the first group of paths. Therefore, the another sequence $P_1, \dots, P_{r-1}, P_{r+1}, \dots, P_{q-1}, P_r, P_q$ is also valid in terms of that **Reverse** possibly runs following this sequence and the result at the end is same as that of the original sequence P_1, \dots, P_q . Hence we can assume that $r = q - 1$, namely, P_{q-1} and P_q are not edge-disjoint and it is the first occurrence in the group, without loss of generality.

We assume that P_{q-1} and P_q share only one edge $\{x, y\}$. The case more than one edge are shared among these can be discussed similarly and is omitted. Let the two paths P_{q-1} and P_q be $\langle u_{q-1}, \dots, x, y, \dots, v_{q-1} \rangle$ and $\langle u_q, \dots, y, x, \dots, v_q \rangle$, respectively. Note that the direction of the edge $\{x, y\}$ differs in those paths because of the reversing procedure. See Figure 4. The vertices $u_{q-1}, v_{q-1}, u_q, v_q, x$, and y are distinct from the observations that if two of them are identical, either one of P_{q-1} and P_q is not such a path processed in the first group; for example, if v_{q-1} and u_q are identical, then $\delta_{\Lambda_{q-1}}^+(u_q) = \delta_{\Lambda_{q-2}}^+(u_q) + 1 < \delta_{\Lambda_{q-2}}^+(u_{q-1})$ that implies the path P_q is not processed in the first group.

From the rule for grouping the paths, it holds that

$$\delta_{\Lambda_{q-2}}^+(u_{q-1}) = \delta_{\Lambda_{q-2}}^+(u_q) = \delta_{\Lambda_{q-1}}^+(u_q). \quad (1)$$

Then, since the path P_{q-1} is reversed at the $(q - 1)$ -th execution of Step 3 of **Reverse**, the following also holds

$$\delta_{\Lambda_{q-2}}^+(u_{q-1}) \geq \delta_{\Lambda_{q-2}}^+(v_{q-1}) + 2. \quad (2)$$

By reversing P_{q-1} to $\overline{P_{q-1}}$, we obtain (by an orientation Λ_{q-1})

$$\delta_{\Lambda_{q-1}}^+(u_{q-1}) = \delta_{\Lambda_{q-2}}^+(u_{q-1}) - 1, \quad (3)$$

$$\delta_{\Lambda_{q-1}}^+(v_{q-1}) = \delta_{\Lambda_{q-2}}^+(v_{q-1}) + 1, \quad \text{and} \quad (4)$$

$$\delta_{\Lambda_{q-1}}^+(v_q) = \delta_{\Lambda_{q-2}}^+(v_q). \quad (5)$$

Also, the path P_q is reversed at the q -th iteration of Step 3 of **Reverse**, that derives

$$\delta_{\Lambda_{q-1}}^+(u_q) \geq \delta_{\Lambda_{q-1}}^+(v_q) + 2. \quad (6)$$

Let us decompose the two paths P_{q-1} and P_q to the following three portions, respectively: $P_{q-1} = (P_{q-1}^{(1)}, P_{q-1}^{(2)}, P_{q-1}^{(3)})$, where $P_{q-1}^{(1)} = \langle u_{q-1}, \dots, x \rangle$, $P_{q-1}^{(2)} = \langle x, y \rangle$, and $P_{q-1}^{(3)} = \langle y, \dots, v_{q-1} \rangle$, and $P_q = (P_q^{(1)}, P_q^{(2)}, P_q^{(3)})$, where $P_q^{(1)} = \langle u_q, \dots, y \rangle$, $P_q^{(2)} = \langle y, x \rangle$, and $P_q^{(3)} = \langle x, \dots, v_q \rangle$.

Consider alternating paths $P' = (P_{q-1}^{(1)}, P_q^{(3)}) = \langle u_{q-1}, \dots, x, \dots, v_q \rangle$ and $P'' = (P_q^{(1)}, P_{q-1}^{(3)}) = \langle u_q, \dots, y, \dots, v_{q-1} \rangle$. At the $(q - 1)$ -th iteration of Step 3 of **Reverse**, P' is also a candidate of a path processed, because $\delta_{\Lambda_{q-2}}^+(u_{q-1}) = \delta_{\Lambda_{q-1}}^+(u_q) \geq \delta_{\Lambda_{q-1}}^+(v_q) + 2 = \delta_{\Lambda_{q-2}}^+(v_q) + 2$ based on the above conditions (1), (6), and (5).

Consider processing $P_1, \dots, P_{q-2}, P', P''$ in this order instead of $P_1, \dots, P_{q-2}, P_{q-1}$ in Step 3 of **Reverse**. Let the resulting orientation be Λ' . We want to show that the path P'' is also a candidate of path processed at the q -th execution of Step 3 of **Reverse**, i.e., it holds that $\delta_{\Lambda'}^+(u_q) \geq \delta_{\Lambda'}^+(v_{q-1}) + 2$. By reversing P' to $\overline{P'}$ at the $(q - 1)$ -th execution of the step, the weighted outdegrees of the vertices are

$$\delta_{\Lambda'}^+(u_{q-1}) = \delta_{\Lambda_{q-2}}^+(u_{q-1}) - 1,$$

$$\delta_{\Lambda'}^+(v_q) = \delta_{\Lambda_{q-2}}^+(v_q) + 1,$$

$$\delta_{\Lambda'}^+(v_{q-1}) = \delta_{\Lambda_{q-2}}^+(v_{q-1}), \quad \text{and}$$

$$\delta_{\Lambda'}^+(u_q) = \delta_{\Lambda_{q-2}}^+(u_q).$$

From these and the above conditions (1) and (2), it holds that $\delta_{\Lambda'}^+(u_q) = \delta_{\Lambda_{q-2}}^+(u_q) = \delta_{\Lambda_{q-2}}^+(u_{q-1}) \geq \delta_{\Lambda_{q-2}}^+(v_{q-1}) + 2 = \delta_{\Lambda'}^+(v_{q-1}) + 2$. Therefore P'' is an alternate candidate for the reverse operation at the q -th iteration of Step 3 of **Reverse**, that is, $P_1, \dots, P_{q-2}, P', P''$ is also a valid sequence of paths processed by **Reverse**. In addition to that the resulting orientation Λ'' is the same as the original orientation Λ_q at the end of the sequence.

By the above procedure, although the set of paths $P_1, \dots, P_{q-2}, P', P''$ is not yet edge-disjoint, the number of shared edges decreased. Therefore, by repeatedly applying the above procedure, we can transform the first (original) group of paths to disjoint one

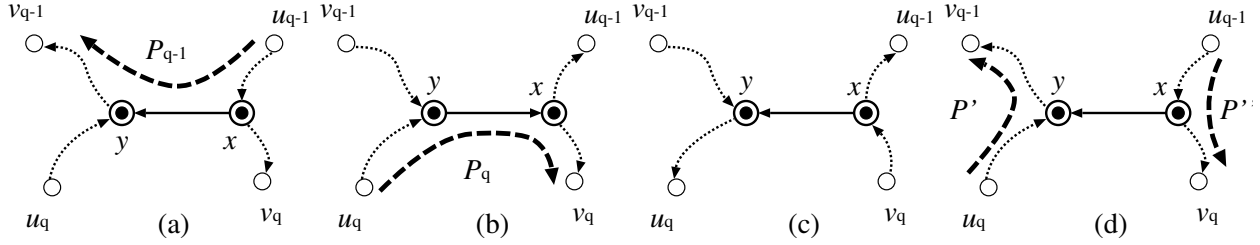


Figure 4: (a) A path P_{q-1} (b) Reversing P_{q-1} and the next candidate path P_q (c) The obtained orientation Λ_q (d) Alternating paths P' and P'' instead of P_{q-1} and P_q

without changing the temporal orientation(solution) Λ_{h-1} at the end of the first group.

The above discussion can be applied to the case across two groups, say, considering P_{h-1} and P_h with distinct four vertices u_{h-1} , v_{h-1} , u_h , and v_h at the ends of the paths. In such a case, similar to the above condition (1), it holds that $\delta_{\Lambda_{h-2}}^+(u_{h-1}) > \delta_{\Lambda_{h-2}}^+(u_h)$ because of the grouping scheme for paths. The other conditions in this case are similar to the above (2) to (6). Then a similar discussion derives $\delta_{\Lambda_{h-2}}^+(u_{h-1}) \geq \delta_{\Lambda_{h-2}}^+(v_h) + 2$ and $\delta_{\Lambda_{h-1}}^+(u_h) \geq \delta_{\Lambda_{h-1}}^+(v_{h-1}) + 2$, that makes us possibly construct a pair of alternating paths similar to P' and P'' in the above; one is from u_{h-1} to v_h and the other is from u_h to v_{h-1} . In this case, two of the vertices u_{h-1} , v_{h-1} , u_h , and v_h may be identical, but a similar discussion can be done.

As a result, if we apply the above procedure repeatedly, we can obtain an edge-disjoint sequence of paths from the original sequence P_1, \dots, P_h . Then, the edge-disjoint sequence is possible to be produced by an execution of **Reverse** for the input graph and the initial orientation Λ . **Reverse** output an optimal solution and now we consider the minimality of the sequence and the number of such paths is equal to $f_k (= h)$. Therefore, by construction of the network $\mathcal{N}_k(\Lambda)$, there exist f_k edge-disjoint paths from vertices in V_k^+ to vertices in V_k^- . That is, the size of the maximum flow is equal to f_k . \square

5 Approximation Algorithms

In this section we present two approximation algorithms for the general case of the MOO using the algorithms presented in the previous section as sub-procedures. One can notice that algorithm **Reverse** can be applied to a general weighted graph if we ignore its weights of edges. This simple idea achieves the following approximation guarantee:

Theorem 13 *Algorithm Reverse is a w_{max}/w_{min} -approximation algorithm for general input graphs.*

Proof. Let an input graph be G and an optimal orientation for G be OPT . Consider two weighted graphs G_{min} and G_{max} that are obtained by replacing all the edge weights to w_{min} and w_{max} , respectively. It is important to note that OPT is not always optimal for G_{min} or G_{max} .

Suppose that algorithm **Reverse** outputs an orientation Λ for the input graph G . Then, from the optimality of Λ for both G_{min} and G_{max} ,

$$\begin{aligned} \Delta_\Lambda(G_{min}) &\leq \Delta_{OPT}(G_{min}) \leq \\ \Delta_{OPT}(G) &\leq \Delta_\Lambda(G) \leq \Delta_\Lambda(G_{max}) \end{aligned}$$

holds, and hence

$$\frac{\Delta_\Lambda(G)}{\Delta_{OPT}(G)} \leq \frac{\Delta_\Lambda(G_{max})}{\Delta_\Lambda(G_{min})}.$$

The oriented graphs of G_{max} and G_{min} have the same structure except for their edge weights, and therefore

$$\frac{\Delta_\Lambda(G)}{\Delta_{OPT}(G)} \leq \frac{\Delta_\Lambda(G_{max})}{\Delta_\Lambda(G_{min})} = \frac{w_{max}}{w_{min}}.$$

\square

Since **Reverse** does not work well when $w_{max} \gg w_{min}$ and its performance is heavily dependent to the edge weights of the input graph, we would like to design another approximation algorithm with a ‘stable’ worst case ratio. Indeed a quite simple strategy can achieve an approximation ratio of 2: For ease of exposition, observe a weighted graph G_a illustrated in Figure 5-(a), which consists of four vertices, v_1 , v_2 , v_3 , and v_4 , and six edges whose weights are $w(\{v_1, v_2\}) = 1$, $w(\{v_1, v_3\}) = 1$, $w(\{v_1, v_4\}) = 1$, $w(\{v_2, v_3\}) = 1$, $w(\{v_2, v_4\}) = 2$, and $w(\{v_3, v_4\}) = 3$. The average weight of the edges is $\ell(G_a) = 9/4$, which is a trivial lower bound of the MOO, as shown in Proposition 2. The outline of the 2-approximation algorithm is as follows: (i) First we choose a vertex v whose weighted degree $\delta(v)$ is at most $2\ell(G_a) = 9/2$ since such a vertex surely exists in G_a . In this case we choose vertex v_1 . (ii) All the edges incident with v_1 are oriented outwards from v_1 to its neighbors, and v_1 is removed from G_a . (iii) We recalculate the average weight of the remaining graph $G_a - \{v_1\}$, and iterate those stages while edges not oriented are remaining. As a result, we select v_1 , v_2 , and v_3 in this order, and the oriented graph is shown in Figure 5-(b).

The above simple procedure guarantees that the maximum weighted outdegree is at most $2\ell(H)$ for each induced subgraph $H \subseteq G_a$, which implies that the approximation ratio is 2. Furthermore, from this observation, we might reduce the approximation ratio to $2 - \varepsilon$ for some positive ε if it is possible to select at least one vertex whose degree is less than twice the average weights of induced subgraphs in each iteration; however, it is impossible. There is an apparent counterexample as illustrated in Figure 5-(c). Its average weight is 2 and the weighted degree of each vertex is 4, double the former. In order to improve the approximation ratio we require further ideas.

Throughout the following, by $\delta_G(u)$ we denote the total weight of edges that connect to a vertex u in a graph G . Here we provide our approximation algorithm, **ALGMOO**, that can overcome the approximation ratio of 2:

Algorithm ALGMOO:

Input: A weighted graph $G = (V, E, w)$.

Output: An arc set Λ which determines directions of edges in E .

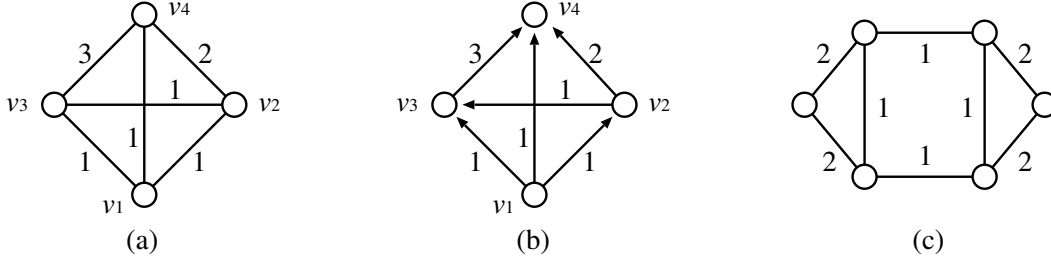


Figure 5: (a) Input graph G_a (b) Output graph oriented by a simple 2-approximation algorithm (c) Worst case example

Step 0: Set $G' = G$, $\Lambda = \emptyset$, and $\ell = \ell(G)$.

Step 1: Repeat the following while there exists a vertex u in G' such that $\delta_{G'}(u) \leq \lceil 2\ell \rceil - 1$,

- For each edge $\{u, v\}$ for some v in G' , add (u, v) to Λ . Then, remove the vertex u and all edges incident to u from G' .

Step 2: There are three cases:

(Case 2-1): No vertex is in G' , i.e., all the vertices are removed in Step 1. In this case, output Λ and halt.

(Case 2-2): For every vertex v in G' , $\delta_{G'}(v) = \lceil 2\ell \rceil$. In this case, proceed to Step 3.

(Case 2-3): There is at least one vertex v such that $\delta_{G'}(v) \geq \lceil 2\ell \rceil + 1$. In this case, update $\ell = \ell(G')$ and goto Step 1.

Step 3: Find a cycle $\langle v_0, v_1, \dots, v_k, v_0 \rangle$ in G' . If such a cycle does not exist, goto Step 4. Add $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k), (v_k, v_0)$ to Λ , and remove edges $\{v_0, v_1\}, \{v_1, v_2\}, \dots, \{v_{k-1}, v_k\}, \{v_k, v_0\}$ from G' and repeat this step.

Step 4: (At the beginning of this step G' is a forest.) Apply algorithm **Convergence** to each connected component of G' . Let an orientation obtained for G' by **Convergence** be Λ' . Output $\Lambda \cup \Lambda'$ and halt.

Theorem 14 Algorithm **ALGMOO** runs in $O(m^2)$ time and is a $(2 - 1/\lceil L(G) \rceil)$ -approximation algorithm.

Proof. Running Time: Steps 0, 1, and 2 require $O(m)$ time, respectively. The number of iterations of Steps 1 and 2 is at most $O(n)$ because one vertex is removed from the graph in single iteration or the algorithm proceeds to Step 3. In Step 3, finding a cycle is done by a breadth-first search which takes $O(m)$ time, and this step is repeated at most $O(m)$ times. In Step 4, we need $O(n)$ time because algorithm **Convergence** runs in $O(c)$ time for a connected component having c vertices, and the total number of vertices in the forest is at most n . In summary, $O(nm)$ time for Steps 0 through 2, $O(m^2)$ time for Step 3, and $O(n)$ time for Step 4 are required respectively, and hence the running time in total is $O(m^2)$.

Approximation Ratio: Let G_i be a graph at the beginning of the i -th iteration of Steps 1 and 2, which is represented in **ALGMOO** by G' , e.g., $G_1 = G$. Suppose that the number of iterations of Steps 1 and 2 is j (≥ 1). At the i -th iteration ($i \geq 2$) of Steps 1 and 2, for every vertex u , $\delta_{G_i}(u) \geq \lceil 2\ell(G_{i-1}) \rceil$, and some vertex v has $\delta_{G_i}(v) \geq \lceil 2\ell(G_{i-1}) \rceil + 1$. This means that, for

all $i \leq j - 1$, $\lceil \ell(G_i) \rceil \leq \lceil \ell(G_{i+1}) \rceil$. Then, since G_j is a subgraph of the input graph $G (= G_1)$, from Propositions 2 and 8, $\Delta_{OPT}(G) \geq \Delta_{OPT_j}(G_j) \geq \lceil \ell(G_j) \rceil$, where OPT_j is an optimal solution for G_j .

The proof is based on the following lemma by which we can conclude

$$\begin{aligned} \Delta_\Lambda(G) &= \max_{u \in V} \{\delta_\Lambda^+(u)\} \\ &\leq \lceil 2\ell(G_j) \rceil - 1 \\ &\leq 2\lceil \ell(G_j) \rceil - 1 \\ &\leq (2 - \frac{1}{\lceil \ell(G_j) \rceil}) \lceil \ell(G_j) \rceil \\ &\leq (2 - \frac{1}{\lceil L(G) \rceil}) \Delta_{OPT}(G) \end{aligned}$$

that we would like to show.

Lemma 15 For every vertex u , $\delta_\Lambda^+(u) \leq \lceil 2\ell(G_j) \rceil - 1$, or Λ is optimal.

Proof. (Step 1) For $1 \leq i \leq j$, a vertex u removed from the graph at the i -th iteration of Steps 1 and 2 has $\delta_\Lambda^+(u) \leq \lceil 2\ell(G_i) \rceil - 1 \leq \lceil 2\ell(G_j) \rceil - 1$. (Note that $\delta_\Lambda^+(u) \leq \lceil 2\ell(G_j) \rceil - 1$ does not imply that $\delta_G(u) \leq \lceil 2\ell(G) \rceil - 1$.)

(Step 2) If the algorithm terminates (Case 2-1), every vertex u satisfies the condition $\delta_\Lambda^+(u) \leq \lceil 2\ell(G_j) \rceil - 1$ based on the analysis of Step 1 above.

(Step 3) At the beginning of this step, for any edge $\{u, v\}$ in the original graph G that connect to a vertex u in G' , neither of (u, v) nor (v, u) is in Λ , or (v, u) is in Λ , i.e., “current” weighted outdegree of u is zero. Also $\delta_{G'}(u) = \lceil 2\ell(G_j) \rceil$ because of the condition in (Case 2-2) of Step 2.

Consider a vertex v_i that is contained in some cycle $\langle v_0, \dots, v_k, v_0 \rangle$ whose orientation is determined in Step 3. Since the orientation Λ contains (v_{i-1}, v_i) (or (v_k, v_0) when $i = 0$), $w(\{v_{i-1}, v_i\}) \geq w_{min}$, and $\delta_{G'}(v_i) = \lceil 2\ell(G_j) \rceil$, it holds that $\delta_\Lambda^+(v_i) \leq \lceil 2\ell(G_j) \rceil - w_{min} \leq \lceil 2\ell(G_j) \rceil - 1$.

(Step 4 and the overall performance) There are two cases on the vertex set V' of G' at the beginning of the Step 4: (i) All the vertices in V' are included at least one cycle whose orientation is determined in Step 3, and (ii) otherwise, i.e., some vertex x is not included in such cycles.

In the case (i), every vertex u in V' has $\delta_\Lambda^+(u) \leq \lceil 2\ell(G_j) \rceil - 1$ based on the analysis of Step 3 without regard to the orientation determined in Step 4. Then, also from the analysis for Steps 1 through 3 in the above, we can see that the weighted outdegree of all the vertices in G_j under Λ is at most $\lceil 2\ell(G_j) \rceil - 1$.

In the case (ii), If $\delta_\Lambda^+(x) \leq \lceil 2\ell(G_j) \rceil - 1$ for every such vertex x , similar discussion as for the case (i)

can be done and we can conclude that the weighted outdegree of all the vertices in G_j under Λ is at most $\lceil 2\ell(G_j) \rceil - 1$.

Let us assume that $\delta_\Lambda^+(x) = \lceil 2\ell(G_j) \rceil$. We observe that x is a vertex in a tree and $\delta_{G'}(x) = \lceil 2\ell(G_j) \rceil$ at the beginning of Step 4. Based on algorithm **Convergence**, x must be a leaf vertex in that tree to have weighted outdegree $\lceil 2\ell(G_j) \rceil$, because if x is an internal vertex (including root), then at least one edge $\{x, y\}$ for some y is directed as (y, x) in Λ so that $\delta_\Lambda^+(x) \leq \lceil 2\ell(G_j) \rceil - w(y, x) \leq \lceil 2\ell(G_j) \rceil - w_{\min}$. This implies that there is an edge $\{x, z\}$ for some z such that $w(\{x, z\}) = \lceil 2\ell(G_j) \rceil$. Therefore since such an edge exists in the input graph, from Proposition 1, $\Delta_{OPT}(G) \geq w_{\max} \geq \lceil 2\ell(G_j) \rceil = \Delta_\Lambda(G)$, that is, Λ is optimal. \square

This ends the proof of the theorem. \square

The proof of Lemma 15 is for general input graphs. If all the $\delta_G(v)$'s are equal, (i.e., regular in a sense of weights), a better ratio can be obtained.

Corollary 16 *If all the $\delta_G(v)$'s are equal for the input graph, algorithm ALGMOO is a $(2 - w_{\min}/\lceil L(G) \rceil)$ -approximation algorithm.*

Proof. The proof is very similar to that of Lemma 15. If all the $\delta_G(v)$'s are equal, the algorithm skips Step 1 and the output (an orientation) of the algorithm is determined in only Steps 3 and 4.

In the above proof of Theorem 14, we showed that either of (1) for every vertex v processed in Steps 3 and 4, $\delta_\Lambda^+(v) \leq \lceil 2\ell(G_j) \rceil - w_{\min}$, or (2) Λ is optimal. Therefore,

$$\begin{aligned} \Delta_\Lambda(G) &= \max_{u \in V} \{\delta_\Lambda^+(u)\} \\ &\leq \lceil 2\ell(G_j) \rceil - w_{\min} \\ &\leq (2 - \frac{w_{\min}}{\lceil \ell(G_j) \rceil}) \lceil \ell(G_j) \rceil \\ &\leq (2 - \frac{w_{\min}}{\lceil L(G) \rceil}) \Delta_{OPT}(G). \end{aligned}$$

\square

Remark. There is a tight example for algorithm ALGMOO. That is, when run on the instance, ALGMOO outputs an orientation whose maximum weighted outdegree is at least $(2 - 1/L(G))\Delta_{OPT}(G)$.

Consider a weighted graph G with n vertices v_0, v_1, \dots, v_{n-1} illustrated in Figure 6. Edges of G are $\{v_i, v_{i+1}\}$ with weight f for $1 \leq i \leq n-2$, $\{v_{n-1}, v_1\}$ with weight f , and $\{v_0, v_i\}$ with weight 1 for $1 \leq i \leq n-1$. Here f has to meet a condition $2f+1 < n-1$. For this graph, $\ell(G) = L(G) = (f+1)(n-1)/n$ that derives $L(G) < f+1$.

Since $\lceil 2\ell(G) \rceil - 1 \geq 2f+1$, the algorithm first determines directions of three edges, say, $\{v_1, v_2\}$, $\{v_2, v_3\}$, and $\{v_0, v_2\}$ for a vertex v_2 as (v_2, v_1) , (v_2, v_3) , and (v_2, v_0) . Therefore, in the final orientation Λ obtained, $\delta_\Lambda^+(v_2) = 2f+1$ and hence $\Delta_\Lambda(G) \geq 2f+1$.

Consider an orientation $\Gamma = \{(v_i, v_{i+1}) \mid 1 \leq i \leq n-2\} \cup \{(v_{n-1}, v_1)\} \cup \{(v_i, v_0) \mid 1 \leq i \leq n-1\}$. We can easily observe that $\Delta_\Gamma(G) = f+1$. Therefore,

$$\frac{\Delta_\Lambda(G)}{\Delta_{OPT}(G)} \geq \frac{\Delta_\Lambda(G)}{\Delta_\Gamma(G)} \geq 2 - \frac{1}{f+1} > 2 - \frac{1}{L(G)}.$$

\square

6 Conclusion

In this paper we have studied a variant of the graph orientation whose objective is to minimize the maximum weighted outdegree of vertices. We then proved its \mathcal{NP} -hardness, and presented an approximation algorithm with an approximation guarantee of $2 - \varepsilon$ but ε depends on the average weights of an input graph. One of the interesting, but challenging open problems is to improve the approximation factor to $2 - \phi$ for ϕ that does not depend on the input.

Acknowledgments

We would like to thank Masafumi Yamashita and Jesper Jansson for their helpful discussions and comments.

References

- Biedl, T., Chan, T., Ganjali, Y., Hajiaghayi, M.T., & Wood, D.R. (2005), 'Balanced vertex-orderings of graphs,' *Discrete Applied Mathematics* **48** (1), 27–48.
- Chrobak, M. & Eppstein, D. (1991), 'Planar orientations with low out-degree and compaction of adjacency matrices,' *Theoretical Computer Science* **86** (2), 243–266.
- Chung, F.R.K., Garey, M.R., & Tarjan, R.E. (1985), 'Strongly connected orientations of mixed multi-graphs,' *Networks* **15**, 477–484.
- Chvátal, V. (1975), 'A combinatorial theorem in plane geometry,' *J. Combinatorial Theory, series B* **18**, 39–41.
- Chvátal, V. & Thomassen, C. (1978), 'Distances in orientation of graphs,' *J. Combinatorial Theory, series B* **24**, 61–75.
- Even, S., & Tarjan, R.E. (1975), 'Network flow and testing graph connectivity,' *SIAM J. Computing* **4**, 507–518.
- Fomin, F.V., Matamala, M., & Rapaport, I. (2004), 'Complexity of approximating the oriented diameter of chordal graphs,' *J. Graph Theory* **45** (4), 255–269.
- Gallo, G., Grigoriadis, M.D., & Tarjan, R.E. (1989), 'A fast parametric maximum flow algorithm and applications,' *SIAM J. Computing* **18**, 30–55.
- Horowitz, E., & Sahni, S. (1976), 'Exact and approximate algorithms for scheduling nonidentical processors,' *J. ACM* **23** (2), 317–327.
- Kára, J., Kratochvíl, J., & Wood, D.R. (2005), 'On the complexity of the balanced vertex ordering problem,' in *Proc. COCOON2005, LNCS 3595*, 849–858.
- König, J.-C., Krumme, D.W., & Lazard, E. (1998), 'Diameter-preserving orientations of the torus,' *Networks* **32** (1), 1–11.
- Lenstra, J.K., Shmoys, D.B., & Tardos, É. (1990), 'Approximation algorithms for scheduling unrelated parallel machines,' *Mathematical Programming* **46** (3), 259–271.
- Nash-Williams, C.St.J.A. (1960), 'On orientations, connectivity and odd vertex pairings in finite graphs,' *Canadian J. Math.* **12**, 555–567.

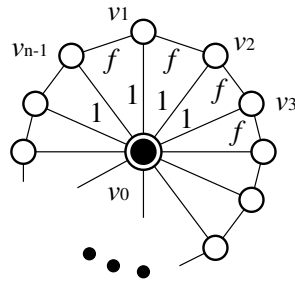


Figure 6: Worst case example for ALGMOO

- O'Rourke, J. (1987), *Art Gallery Theorems and Algorithms*, Oxford University Press.
- Pinedo, M. (2002), *Scheduling: Theory, Algorithms, and Systems*, Prentice Hall, 2nd Ed.
- Robbins, H.E. (1939), 'A theorem on graphs with an application to a problem of traffic control,' *American Math. Monthly* **46**, 281–283.
- Schuurman, P., & Woeginger, G.J. (1999), 'Polynomial time approximation algorithms for machine scheduling: Ten open problems,' *J. Scheduling* **2**, 203–213.
- Venkateswaran, V. (2004), 'Minimizing maximum in-degree,' *Discrete Applied Mathematics* **143**, 374–378.

Multilayer Grid Embeddings of Iterated Line Digraphs

Toru Hasunuma

Department of Mathematical and Natural Sciences
The University of Tokushima,
1-1 Minamijosanjima, Tokushima 770-8502 Japan,
Email: hasunuma@ias.tokushima-u.ac.jp

Abstract

In this paper, we show that for any fixed d -regular digraph G , every iterated line digraph $L^k(G)$ ($k \geq 1$) can be embedded in d layers using $O(n^2)$ area, where n is the number of vertices in $L^k(G)$. Also, we present $\Omega(\frac{n^2}{\log^2 n})$ lower bound on the area of $L^k(G)$ for any fixed number of layers. Besides, we apply the results to specific families of iterated line digraphs such as de Bruijn digraphs, Kautz digraphs, and wrapped butterfly digraphs.

Keywords: Multilayer grid embedding, Iterated line digraphs, VLSI-layout, Interconnection networks, Bisection width.

1 Introduction

1.1 Multilayer Grid Embeddings

As a VLSI-layout model, Aggarwal *et al.* (1991) defined the k -PCB model (PCB is an abbreviation for a printed circuit board). Consider k stacked grid layers. In the k -PCB model, an embedding of a graph consists of a placement of the vertices as horizontal line segments in grids so that each horizontal line segment is placed in the same position on each grid layer, and drawing of each edge as a path in grids (along grid edges) so that there is no crossing. A path corresponding to an edge is not necessary in a single layer. It can change the layer to the upper or lower layer at contact cuts. An embedding of a graph in the k -PCB model is called a k -layer grid embedding of the graph. The area of a multilayer grid embedding of a graph is the minimum size of a grid which covers the embedding. Although the k -PCB model originally permits the existence of contact cuts, Aggarwal *et al.* (1991) mentioned that the existence of contact cuts is undesirable from a practical point of view. Thus, in this paper, we only treat multilayer grid embeddings *without contact cuts*. That is, any edge is drawn as a grid-path in a single grid layer. Throughout the paper, multilayer grid embeddings mean multilayer grid embeddings without contact cuts.

Figure 1 illustrates an example of a 2-layer grid embedding of a graph, where each line style indicates a layer in which an arc is drawn. Multilayer grid embedding is related to the three-dimensional orthogonal graph drawing (Biedl, Thiele & Wood 2005). In fact, we can define a multilayer grid embedding as

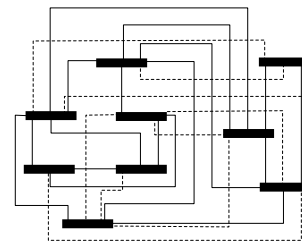


Figure 1: A 2-layer grid embedding of a graph.

a three-dimensional orthogonal graph drawing with several restrictions.

The *thickness* of a graph is the minimum number k such that the graph is a union of k planar graphs. Aggarwal *et al.* (1991) showed the following results. (In what follows, we use n to denote the number of vertices in a graph.)

- Every graph with thickness at most 2 can be embedded in two layers in $O(n^2)$ area.
- There is a graph with thickness at most 2 which needs $\Omega(n^2)$ area for k -layer embedding, where k is any fixed integer.
- Every graph with thickness at most t can be embedded in t layers in $O(n^3)$ area with respect to prespecified placements of the vertices.

Also, Aggarwal *et al.* (1991) proved $\Omega(n^3)$ lower bound on the area of permutation layouts with respect to prespecified placements of the vertices. From this result, the above $O(n^3)$ bound is optimal for the area with respect to prespecified placements of the vertices. In general setting, it still remains open whether any graph with thickness at most t can be embedded in t layers in $O(n^2)$ area.

1.2 Iterated Line Digraphs

Let G be a digraph. A digraph can have loops and symmetric arcs but not multiple arcs. The vertex set and the arc set of G are denoted by $V(G)$ and $A(G)$, respectively. For $v \in V(G)$, the number of arcs with tail (respectively, head) v in G is the outdegree (respectively, indegree) of v . A d -regular digraph is a digraph in which for every vertex v , both the outdegree and the indegree of v are d . The *line digraph* $L(G)$ of G is defined as follows. The vertex set of $L(G)$ is the arc set of G , i.e., $V(L(G)) = A(G)$. The vertex (u, v) is a predecessor of every vertex of the form (v, w) , i.e., $A(L(G)) = \{((u, v), (v, w)) \mid (u, v), (v, w) \in A(G)\}$. When we regard “ L ” as an operation on digraphs, it is called the *line digraph operation*. The k -iterated

line digraph $L^k(G)$ of G is the digraph obtained from G by iteratively applying the line digraph operation k times. Figure 2 shows an example of iterated line digraphs.

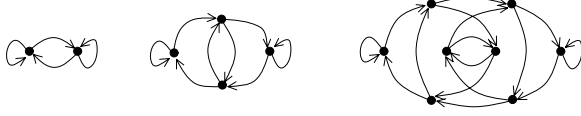


Figure 2: A digraph G , the line digraph $L(G)$, and the 2-iterated line digraph $L^2(G)$.

The line digraph operation is a useful tool for constructing large digraphs with bounded degree, small diameter (Fiol, Yebra & Alegre 1984), and high connectivity (Fábrega & Fiol 1989). In fact, the class of iterated line digraphs of a regular digraph contains well-known interconnection networks for massively parallel computers such as de Bruijn digraphs, Kautz digraphs (Bermond & Peyrat 1989) and wrapped butterfly digraphs (Leighton 1992, Xu 2001).

1.3 Our Results

In this paper, we treat a class of iterated line digraphs of a regular digraph and study on multilayer grid embeddings of such a class. A multilayer grid embedding of a digraph can be similarly defined as that of the underlying graph.

Let G be a d -regular digraph. Then any iterated line digraph $L^k(G)$ is also d -regular. A d -regular digraph can be decomposed into d 1-regular digraphs, and a 1-regular digraph is clearly planar. Thus, the thickness of $L^k(G)$ is at most d . By the result of Agarwal *et al.* (1991), we can see that $L^k(G)$ can be embedded in d layers in $O(n^3)$ area. In this paper, we show that such an upper bound can be improved, i.e., any iterated line digraph $L^k(G)$ can be embedded in d layers in $O(n^2)$ area. Also, we present $\Omega(\frac{n^2}{\log^2 n})$ lower bound on the area of $L^k(G)$ for any fixed number of layers. In order to derive the lower bound, we show $\Omega(\frac{n}{\log n})$ lower bound on the bisection width of $L^k(G)$, which deserves an independent interest, since the bisection width of a network is an important indicator of its power as a computer network (Bornstein, Litman, Maggs & Sitaraman 2001, Leighton 1992).

This paper is organized as follows. The basic definitions and terminology are given in Section 2. In Section 3, we present an algorithm to embed $L^k(G)$ in $O(n^2)$ area. In Section 4, we show a $\Omega(\frac{n^2}{\log^2 n})$ lower bound on the area for an embedding of $L^k(G)$. In Section 5, we apply the algorithm in Section 3 to specific families of iterated line digraphs. Finally, Section 6 concludes the paper with some remarks.

2 Preliminaries

Let $G = (V, A)$ be a digraph. If $(u, v) \in A(G)$, then u is the *tail* of (u, v) ; v is the *head* of (u, v) . If $u, v \in V(G)$, then u is a *predecessor* of v if $(u, v) \in A(G)$, and u is a *successor* of v if $(v, u) \in A(G)$. The set of arcs with tail u is $A_G^+(u) = \{(u, v) \in A(G)\}$, while the set of arcs with head u is $A_G^-(u) = \{(v, u) \in A(G)\}$.

A *walk* of length k in G is a sequence of vertices (v_0, v_1, \dots, v_k) , where $(v_{i-1}, v_i) \in A(G)$ for each i satisfying $1 \leq i \leq k$. A *path* is a walk in which no

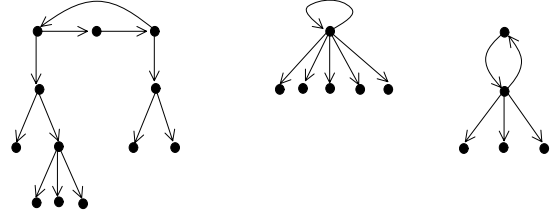


Figure 3: A cycle-rooted tree, a loop-star, and a 2-cycle-star.

vertex is repeated. Let $u, v \in V(G)$. The *distance* $d_G(u, v)$ from u to v in G is the minimum length of a path from u to v in G . The *diameter* of G , denoted by $\text{diam}(G)$, is the maximum distance for any two vertices in G , i.e., $\text{diam}(G) = \max_{u, v \in V(G)} d_G(u, v)$.

The *underlying graph* $U(G)$ of G is a graph obtained from G by replacing each arc with a corresponding edge. Thus, $U(G)$ has loops (respectively, multiple edges with multiplicity two) if G has loops (respectively, symmetric arcs). A digraph is called *weakly connected* if the underlying graph is connected. A (directed) *tree* is a weakly connected digraph in which there is a unique vertex with indegree 0, called the *root*, such that all other vertices have indegree 1. The *depth* of a tree is the maximum length of a path from the root to a vertex with outdegree 0. A (directed) *star* is a tree with depth 1. A *cycle-rooted tree* is a weakly connected digraph in which every vertex has indegree 1. In particular, a digraph obtained from a star and a cycle of length k by identifying the root of the star and a vertex in the cycle is called a *k-cycle-star*. Besides, 1-cycle-star is simply called a *loop-star* (see Figure 3).

Except for marginal grid-points, the degree of every grid-point is four. The upper (respectively, lower) vertical edge incident on a grid-point p is *N-grid-edge* (respectively, *S-grid-edge*) of p . The left (respectively, right) horizontal edge incident on p is *W-grid-edge* (respectively, *E-grid-edge*) of p .

In a multilayer grid embedding of G , each vertex is placed as a horizontal line segment in the same position on each layer. We denote by $\text{lseg}(v)$ the horizontal line segment corresponding to a vertex v of G . The *length* of a horizontal line segment is defined as the number of grid points that are contained in the line segment in a grid. In a multilayer grid embedding of G , each arc is drawn as a path in a grid. We denote by $\text{gpath}(u, v)$ the drawn path corresponding to an arc (u, v) of G . The *starting grid-point* of $\text{gpath}(u, v)$ is the grid-point in $\text{lseg}(u)$ which is incident on a grid-edge used in $\text{gpath}(u, v)$.

3 An Upper Bound on the Area

Theorem 3.1 *For any fixed d -regular digraph G , every $L^k(G)$ ($k \geq 1$) can be embedded in d layers in $O(n^2)$ area, where n is the number of vertices in $L^k(G)$.*

Proof. We present an algorithm for a d -layer grid embedding of $L^k(G)$ for all $k \geq 1$. Let G be a d -regular digraph with p vertices. For each vertex v in G , color the d arcs in $A_G^-(v)$ using d colors. Let F_1, F_2, \dots, F_d be the d subdigraph induced by arcs of the same color. Each F_i has the property that every vertex in F_i has indegree one. This means that each F_i is a disjoint union of cycle rooted trees. Clearly, a cycle-rooted tree is planar. Thus, each F_i is also planar.

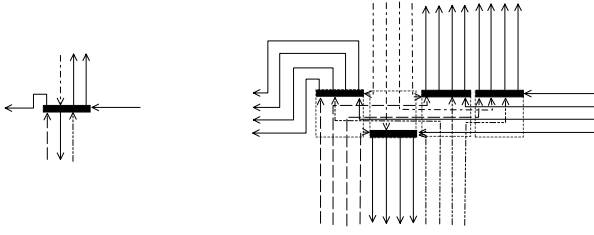


Figure 4: Parts of 4-layer grid embeddings of G and $L(G)$.

Now consider a d -layer grid embedding of G under the following conditions;

- every arc in F_i is drawn as a grid-path in the i -th grid,
- for every vertex v of G , the length of $\text{lseg}(v)$ is d , and for any two arcs (v, u_1) and (v, u_2) in $A_G^+(v)$, $\text{gpath}(v, u_1)$ and $\text{gpath}(v, u_2)$ have distinct starting grid-points in $\text{lseg}(v)$.
- for any arc (v, u) in $A_G^+(v)$, $\text{gpath}(v, u)$ does not contain the W -grid-edge of the leftside grid-point in $\text{lseg}(v)$ and does not contain the E -grid-edge of the rightside grid-point in $\text{lseg}(v)$.

The area of such an embedding of G is not essential for the proof of the theorem, since we consider the class $\{L^k(G) \mid k \geq 1\}$, where G is any fixed regular digraph. Thus, for example, such an embedding can be obtained heuristically. The left figure in Figure 4 is an example of a part of a 4-layer grid embedding of G , where each line-style indicates a layer in which an arc is drawn.

Based on the embedding of G , we construct an embedding of $L(G)$ as follows.

1. Magnify the scale of the embedding of G ; the height d times and the width d times. Each grid-point in a grid is magnified to a rectangular of size $d \times d$. Thus, a horizontal line segment is magnified to a rectangular of size $d \times d^2$.
2. For each (v, u) in $A(G)$ (which is a vertex in $L(G)$), place the corresponding horizontal line segment of length d in the area of the rectangular magnified from the starting grid-point of $\text{gpath}(v, u)$. If $\text{gpath}(v, u)$ use the N -grid-edge (respectively, S -grid-edge) of the starting grid-point in the embedding of G , then the upper-horizontal (respectively, lower-horizontal) line segment in the area of the rectangular is used for $\text{lseg}(v, u)$.

By the above manipulation, we define the placement of the vertices in $L(G)$. Next, we consider the drawing of the arcs in $L(G)$.

3. For drawing of the arcs in $L(G)$, replace each magnified grid-path with a set of d parallel grid-paths. For the magnified grid-path of $\text{gpath}(v, u)$, we will make the set of d parallel grid-paths correspond to the set of d arcs in $\{(v, u), (u, w) \mid (u, w) \in A(G)\}$. Since every grid-path in the set is correctly incident on $\text{lseg}((v, u))$, it remains to join to their head line segments, i.e., $\text{lseg}((u, w))$ for $(u, w) \in A(G)$. Each line segment $\text{lseg}((u, w))$ is placed in the area of the rectangular magnified from $\text{lseg}(u)$.

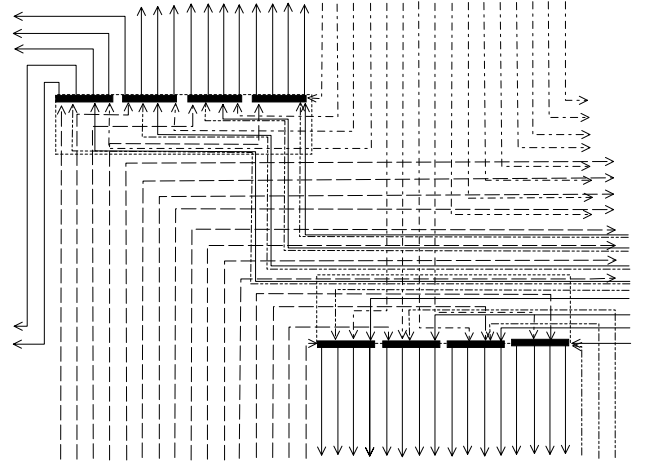


Figure 5: A part of the 4-layer grid embedding of $L^2(G)$ (corresponding to a left-half of the part of embedding of $L(G)$ in Fig.4).

The rectangular magnified from $\text{lseg}(u)$ has the size $d \times d^2$. Thus, it can be checked that we can join d parallel arcs to the d line segments $\text{lseg}((u, w))$, $(u, w) \in A(G)$ in one-to-one manner without crossing (see Figure 4).

By the above manipulation, no crossing in each grid is produced, since the drawing of arcs is based on the drawing of arcs in G . Thus, we obtain a d -layer grid embedding of $L(G)$. By iteratively applying the similar magnifying technique, a d -multilayer grid embedding of $L^k(G)$ for all $k \geq 1$ is obtained (see Figure 5).

Let A be the area of a d -layer grid embedding of G . In the i -th iterative step, we obtain $d^{2i}A$ -area embedding of $L^i(G)$ using d layers. Since $n = |V(L^i(G))| = pd^i$, where $p = |V(G)|$, it holds that $d^{2i}A = \frac{A}{p}n^2 = O(n^2)$. (Note that A and p can be treated as constants, since we consider the class $\{L^k(G) \mid k \geq 1\}$ for any fixed digraph G .) \square

4 A Lower Bound on the Area

In order to derive a lower bound on the area, we use the following result by Aggarwal *et al.* (1991).

Theorem 4.1 (Aggarwal, Klawe & Shor 1991)

Let $c(G)$ be the crossing number of G . Then, for any fixed t , every t -layer grid embedding of G requires $\Omega(c(G))$ area.

A cut (V_1, V_2) of a graph $H = (V, E)$ is the set of edges between V_1 and V_2 , i.e., $\{uv \in E(H) \mid u \in V_1, v \in V_2\}$, where $V_1 \cup V_2 = V(H)$ and $V_1 \cap V_2 = \emptyset$. The bisection width of H is $\min_{|V_1| \geq n/3, |V_2| \geq n/3} |(V_1, V_2)|$. (There is another definition slightly different from this definition of the bisection width (Bornstein *et al.* 2001). However, our result correctly holds for such a definition.)

Pach *et al.* (1996) presented the following relation between the crossing number and the bisection width.

Theorem 4.2 (Pach, Shahrokhi & Szegedy 1996)

Let $c(G)$ and $b(G)$ be the crossing number and the bisection width of G , respectively. Also, let d_1, d_2, \dots, d_n be the degree sequences of G . Then, $b(G)^2 \leq (1.58)^2 (16c(G) + \sum_{i=1}^n d_i^2)$.

The bisection width of a digraph is similarly defined as that of the underlying graph of the digraph. In what follows, we derive a lower bound on the bisection width of $L^k(G)$. (Precisely, simple graphs are assumed in Theorem 4.2. On the other hand, the underlying graph $U(L^k(G))$ may not be simple. However, multiplicity of a multiple edge in $U(L^k(G))$ is at most two. Also, the numbers of loops and multiple edges in $U(L^k(G))$ depend only on G . Thus, our discussion on digraphs below can be combined with the above theorem.)

Theorem 4.3 *Let G be a d -regular digraph. The bisection width of $L^k(G)$ is $\Omega(\frac{n}{\log n})$.*

Proof. Let G_1 and G_2 be digraphs with the same number of vertices. An embedding of G_1 to G_2 consists of a one-to-one mapping f from $V(G_1)$ to $V(G_2)$ and a mapping of each arc (u, v) in G_1 to a path from $f(u)$ to $f(v)$ in G_2 . The complete symmetric digraph K_n^* is an n -vertex digraph in which for any ordered pair (u, v) of vertices, there is an arc from u to v .

Let G be a d -regular digraph with p vertices. In what follows, we will show that there is an embedding of K_n^* , where $n = pd^k = |V(L^k(G))|$, into $L^k(G)$ so that for any arc e of $L^k(G)$, there are at most $O(n \log n)$ paths corresponding to arcs of K_n^* that contain e . This means that the bisection width of $L^k(G)$ is $\Omega(n^2/n \log n) = \Omega(n/\log n)$, since the bisection width of K_n^* is $\Omega(n^2)$.

As a one-to-one mapping from $V(K_n^*)$ to $V(L^k(G))$, we employ any one-to-one mapping. Let ρ be the employed mapping. For an arc $(v_i, v_j) \in A(K_n^*)$, we map it to a shortest path from $\rho(v_i)$ to $\rho(v_j)$ in $L^k(G)$. This mapping from $A(K_n^*)$ to the set of paths in $L^k(G)$ is denoted by ρ^* . By applying the line digraph operation to a digraph one time, the diameter of the digraph increase by at most one (Fiol et al. 1984). Thus, $\text{diam}(L^k(G))$ is at most $\text{diam}(G) + k$. Thus, the number of paths of length at most $\text{diam}(G) + k$ which contain e is an upper bound on the cardinality of the set $\{(u_i, u_j) \in A(K_n^*) \mid e \in \rho^*((u_i, u_j))\}$.

Let $e = (u, v)$. Consider a path of length i which contains e as the k -th arc. Such a path is uniquely decomposed into a path of length $k - 1$ which ends at u , the arc e , and a path of length $(i - k)$ which starts at v . Since G is d -regular, the numbers of paths of length $k - 1$ which ends at u and the paths of length $(i - k)$ which starts at v are at most d^{k-1} and d^{i-k} , respectively. Thus, the number of paths of length i which contains e as the k -th arc is at most $d^{k-1}d^{i-k} = d^{i-1}$. Therefore, the number of paths of length i which contains e is at most $d^{i-1}i$. Hence, the number of paths of length at most $\text{diam}(G) + k$ which contains e is at most $\sum_{i=1}^{\text{diam}(G)+k} d^{i-1}i$. Here,

$$\begin{aligned} \sum_{i=1}^t d^{i-1}i &= 1 + 2d + 3d^2 + \dots + td^{t-1}, \\ &= (1 + d + d^2 + \dots + d^{t-1}) \\ &\quad + (d + d^2 + \dots + d^{t-1}) + \dots + d^{t-1} \\ &= \frac{d^t - 1}{d - 1} + d \left(\frac{d^{t-1} - 1}{d - 1} \right) + \dots + d^{t-1} \\ &= \frac{1}{d-1} \sum_{i=0}^{t-1} d^i (d^{t-i} - 1) \\ &= \frac{1}{d-1} \left(td^t - \frac{d^t - 1}{d-1} \right). \end{aligned}$$

Thus, $\sum_{i=1}^t d^{i-1}i \leq td^t$. Therefore, $\sum_{i=1}^{\text{diam}(G)+k} d^{i-1}i \leq (\text{diam}(G) + k)d^{\text{diam}(G)+k}$. Since $n = pd^k$, $\text{diam}(G) + k = O(\log n)$ and $d^{\text{diam}(G)+k} = d^{\text{diam}(G)}d^k = O(n)$. Hence, $(\text{diam}(G) + k)d^{\text{diam}(G)+k} = O(n \log n)$. \square

From Theorems 4.2, and 4.3, a lower bound on the crossing number of $L^k(G)$ is obtained. (Note that $\sum_{i=1}^n d_i^2 = 4d^2n = O(n)$ for $U(L^k(G))$, since $L^k(G)$ is d -regular.)

Corollary 4.4 *Let G be a d -regular digraph. The crossing number of $L^k(G)$ is $\Omega(\frac{n^2}{\log^2 n})$.*

Therefore, by Theorem 4.1 and Corollary 4.4, we obtain a lower bound on the area for multilayer grid embeddings of iterated line digraphs.

Theorem 4.5 *For any fixed d -regular digraph G and for any fixed t , every t -layer grid embedding of $L^k(G)$ requires $\Omega(\frac{n^2}{\log^2 n})$ area.*

5 Specific Families of Iterated Line Digraphs

5.1 De Bruijn and Kautz Digraphs

The de Bruijn and Kautz digraphs have been noticed as interconnection networks because of their various nice properties (Bermond & Peyrat 1989, Xu 2001). The *de Bruijn digraph* $B(d, D)$ is a digraph whose vertices are the words of length D on an alphabet of d letters (for example, $\{0, 1, \dots, d-1\}$), and in which there is an arc from each vertex $(v_0, v_1, \dots, v_{D-1})$ to the d vertices $(v_1, \dots, v_{D-1}, \alpha)$, where $\alpha \in \{0, 1, \dots, d-1\}$. Thus, $B(d, D)$ is d -regular, and $|V(B(d, D))| = d^D$.

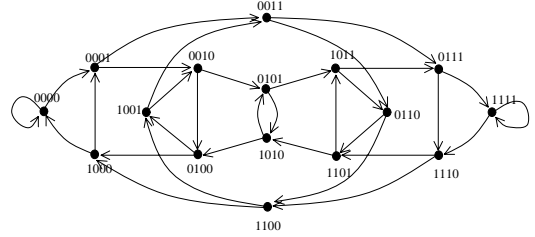


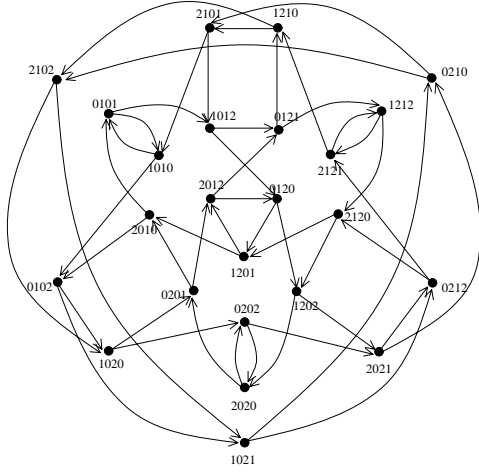
Figure 6: $B(2, 4)$.

The *Kautz digraph* $K(d, D)$ is a digraph whose vertices are the words of length D without two consecutive identical letters on an alphabet of $d + 1$ letters, and in which there is an arc from each vertex $(v_0, v_1, \dots, v_{D-1})$ to the d vertices $(v_1, \dots, v_{D-1}, \alpha)$, where $\alpha \in \{0, 1, \dots, d\}$ and $\alpha \neq v_{D-1}$. Thus, $K(d, D)$ is d -regular and $|V(K(d, D))| = d^{D-1}(d + 1)$. Figures 6 and 7 illustrate examples of de Bruijn and Kautz digraphs, respectively.

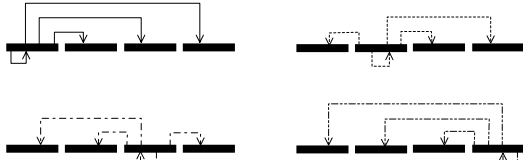
Fiol, Yebra and Alegre (1984) showed that $B(d, D)$ and $K(d, D)$ can be also defined as $B(d, D) = L^{D-1}(K_d^\circ)$ and $K(d, D) = L^{D-1}(K_{d+1}^*)$, where K_{d+1}^* is the complete symmetric digraph with $d + 1$ vertices, and K_d° is the complete digraph with d vertices, i.e., the digraph obtained from K_d^* by adding a loop to each vertex. Figure 2 also illustrates $B(2, 1)$, $B(2, 2)$ and $B(2, 3)$.

Theorem 5.1

- $B(d, D)$ can be embedded in d layers in $d^{2D}(d+1)$ area.
- $K(d, D)$ can be embedded in d layers in $d^{2D-1}(d+1)(d+2)$ area.

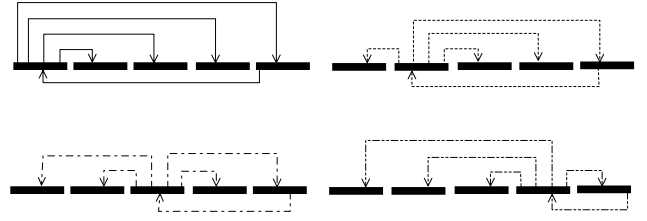
Figure 7: $K(2, 4)$.

Proof. The complete digraph K_d° is decomposed into d loop-stars. We draw each loop-star on one layer. For a d -multilayer grid embedding of K_d° , place d horizontal line segments of length d horizontally. Then, except for loops, draw each arc above the horizontal line segments. Finally, draw each loop below the horizontal line segments. (Figure 8 illustrates the 4-layer grid embedding of K_4° .) The width of this embedding is d^2 and the height of the embedding is $(d+1)$. Therefore, the area for this embedding of K_d° is $d^2(d+1)$. Since $B(d, D) = L^{D-1}(K_d^\circ)$, by applying the algorithm in Section 3, a d -multilayer grid embedding of $B(d, D)$ using $d^2(d+1) \times d^{2(D-1)} = d^{2D}(d+1)$ area is obtained.

Figure 8: The 4-layer grid embedding of K_4° .

The complete symmetric digraph K_{d+1}^* is decomposed into d 2-cycle-stars such that a vertex is commonly contained in 2-cycles of all the 2-cycle-stars. Let x be such a vertex in K_{d+1}^* . Each 2-cycle-star is drawn on one layer. For d -multilayer grid embedding of K_{d+1}^* , place d horizontal line segments of length d horizontally such that $\text{lseg}(x)$ is placed in the most right position. Then, for each 2-cycle-star, draw the arcs above the horizontal line segments except for one arc in the 2-cycle. The remaining arcs are drawn below the horizontal line segments. (Figure 9 illustrates the 4-layer grid embedding of K_5^* .) The width of the embedding is $d(d+1)$ and the height of the embedding is $(d+2)$. Thus, the area for the embedding is $d(d+1)(d+2)$. Since $K(d, D) = L^{D-1}(K_{d+1}^*)$, by applying the algorithm presented in Section 3, $K(d, D)$ can be embedded in d layers in $d(d+1)(d+2) \times d^{2(D-1)} = d^{2D-1}(d+1)(d+2)$ area. \square

Using n to denote the number of vertices, Theorem

Figure 9: The 4-layer grid embedding of K_5^* .

5.1 can be restated as follows.

- $B(d, D)$ can be embedded in d layers in $(d+1)n^2$ area.
- $K(d, D)$ can be embedded in d layers in $(d+2)n^2$ area.

Remark: For a loop (v, v) in G , we can draw the paths $((v, v), (v, w))$ in $L(G)$ inside the magnified rectangular corresponding to $\text{lseg}(v)$. Thus, the area required for a d -layer grid embedding of $B(d, D)$ can be improved to dn^2 by the special treatment for loops.

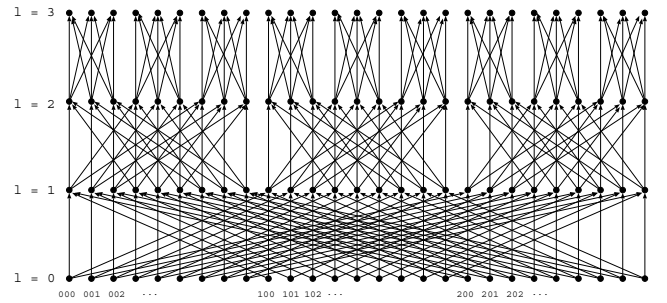
5.2 Wrapped Butterfly Digraphs

The d -ary butterfly graph $b(d, r)$ is one of the most popular interconnection networks and is defined as follows:

$$V(b(d, r)) = \{(v_0, \dots, v_{r-1}, l) \mid v_i \in \{0, 1, \dots, d-1\}, 0 \leq i < r, 0 \leq l \leq r\},$$

$$E(b(d, r)) = \{(u_0, \dots, u_{r-1}, l), (v_0, \dots, v_{r-1}, l+1) \mid u_i = v_i \text{ for } i \neq l\}.$$

The d -ary butterfly digraph $\vec{b}(d, r)$ is a digraph obtained from $b(d, r)$ by orienting each edge according to increasing values of l (see Figure 10).

Figure 10: $\vec{b}(3, 3)$.

The d -ary wrapped butterfly graph $wb(d, r)$ is the graph obtained from $b(d, r)$ by identifying each vertex of the lowest level ($l = 0$) with the corresponding vertex of the highest level ($l = r$) (Leighton 1992, Xu 2001). Similarly, the d -ary wrapped butterfly digraph $\vec{wb}(d, r)$ is obtained from $\vec{b}(d, r)$. Thus, $\vec{wb}(d, r)$ is d -regular and $|V(\vec{wb}(d, r))| = d^r r$.

Bermond *et al.* (1998) showed that $\vec{wb}(d, r)$ can be also defined as $L^{r-1}(K_d^\circ \otimes C_r)$, where \otimes denotes the Kronecker product and C_r is a cycle of length r . The Kronecker product of two digraphs G_1 and G_2 , denoted by $G_1 \otimes G_2$, is defined as follows:

$$V(G_1 \otimes G_2) = V(G_1) \times V(G_2),$$

$$A(G_1 \otimes G_2) = \{((u_1, u_2), (v_1, v_2)) \mid (u_1, v_1) \in A(G_1) \text{ and } (u_2, v_2) \in A(G_2)\}.$$

Theorem 5.2 $\vec{wb}(d, r)$ can be embedded in d layers in $d^{2r-1}(d+1)(dr+2)$ area.

Proof. Let $V(K_d^o) = \{v_1, v_2, \dots, v_d\}$. Also, let $V(C_r) = \{w_1, w_2, \dots, w_r\}$. Then $V(\vec{wb}(d, r)) = \{(v_i, w_j) \mid 1 \leq i \leq d, 1 \leq j \leq r\}$. For each j , place horizontal line segments $\text{lseg}((v_i, w_j))$ of length d , $1 \leq i \leq d$, horizontally. Also, $\text{lseg}((v_i, w_{j+1}))$, $1 \leq i \leq d$, are placed in parallel with $\text{lseg}((v_i, w_j))$, $1 \leq i \leq d$ such that the distance between them is d .

Since $K_d^* \otimes C_r$ is decomposed into d r -cycle-rooted trees, draw each r -cycle-rooted tree in one layer. For each r -cycle-rooted tree, the arcs connecting a vertex in $\{(v_i, w_j) \mid 1 \leq i \leq d\}$ and a vertex in $\{(v_i, w_{j+1}) \mid 1 \leq i \leq d\}$ can be correctly drawn in the grid space between $\{\text{lseg}((v_i, w_j)) \mid 1 \leq i \leq d\}$ and $\{\text{lseg}((v_i, w_{j+1})) \mid 1 \leq i \leq d\}$ for $j = 1, 2, \dots, r-1$. For the arcs connecting a vertex in $\{(v_i, w_r) \mid 1 \leq i \leq d\}$ and a vertex in $\{(v_i, w_1) \mid 1 \leq i \leq d\}$, $\lfloor d/2 \rfloor$ arcs are drawn in anti-clockwise manner, and $\lceil d/2 \rceil$ arcs are drawn in clockwise manner. Figure 11 illustrates such an embedding of $K_3^* \otimes C_3$.

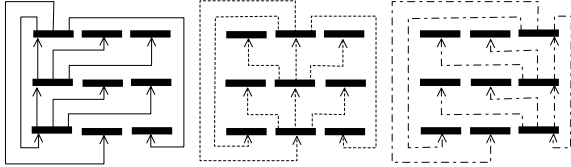


Figure 11: The 3-layer grid embedding of $K_3^* \otimes C_3$.

In this embedding of $K_d^* \otimes C_r$, the width is $d \times d + \lfloor d/2 \rfloor + \lceil d/2 \rceil = d(d+1)$, and the height is $d \times (r-1) + 1 + 2\lfloor d/2 \rfloor \leq dr+2$. Therefore, by the algorithm in Section 3, $\vec{wb}(d, r)$ can be embedded in d layers using $d(d+1)(dr+2) \times d^{2(r-1)} = d^{2r-1}(d+1)(dr+2)$ area. \square

6 Concluding remarks

We have shown that for any fixed d -regular digraph G , every iterated line digraph $L^k(G)$ ($k \geq 1$) can be embedded in d layers using $O(n^2)$ area, where n is the number of vertices in $L^k(G)$. Also, we have presented $\Omega(\frac{n^2}{\log^2 n})$ lower bound on the area of $L^k(G)$ for any fixed number of layers. Besides, we have applied the results to specific families of iterated line digraphs. It remains unknown whether our bounds can be improved.

For iterated line digraphs, several kinds of layouts have been studied (Hasunuma 2002, Hasunuma 2003). Queue layouts can be applied to three-dimensional straight-line grid drawings. Book-embeddings (also called stack-layouts) are related to multilayer grid embeddings. A book consists of a line called the *spine* and half-planes sharing the spine as a common boundary, called *pages*. A book-embedding of a graph is defined by a linear ordering of the vertices, and an assignment of the edges to pages so that there is no crossing of edges in each page. It is not difficult to

see that a $2d$ -page book-embedding of a graph can be applied to obtain a d -layer grid embedding of a graph by considering two pages as one layer.

References

- Aggarwal, A., Klawe, M. & Shor, P. (1991), Multi-layer grid embeddings for VLSI, *Algorithmica* 6, 129–151.
- Aggarwal, A., Klawe, M., Lichtenstein, D., Linial, N. & Wigderson, A. (1991), A lower bound on the area of permutation layouts, *Algorithmica* 6, 241–255.
- Bermond, J.-C., Darrot, E., Delmas, O. & Perennes, P. (1998), Hamiltonian circuits in the directed wrapped butterfly networks, *Discrete Applied Mathematics* 84, 21–42.
- Bermond, J.-C. & Peyrat, C. (1989), De Bruijn and Kautz networks: A competitor for the hypercube?, in F. André, J.P. Verjus, eds, 'Hypercube and Distributed Computers', North Holland, pp. 279–293.
- Biedl, T., Thiele, T. & Wood, D.R. (2005), Three-dimensional orthogonal graph drawing with optimal volume, *Algorithmica*, published online.
- Bornstein, C.F., Litman, A., Maggs, B.M. & Sitaraman, R.K. (2001), On the bisection width and expansion of butterfly networks, *Theory of Computing Systems* 34, 491–518.
- Fábrega, J. & Fiol, M.A. (1989), Maximally connected digraphs, *J. Graph Theory* 13, 657–668.
- Fiol, M.A., Yebra, J.L.A. & Alegre, I. (1984), Line digraph iterations and the (d, k) digraph problem, *IEEE Trans. Comput.* 33, 400–403.
- Hasunuma, T. (2002), Embedding iterated line digraphs in books, *Networks* 40, 51–62.
- Hasunuma, T. (2003), Laying out iterated line digraphs using queues, in '11th International Symposium on Graph Drawing (GD2003)', G. Liotta, ed., Lecture Notes in Computer Science, Vol. 2912, Springer pp. 202–213.
- Leighton, F.T. (1992), Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes, Morgan Kaufmann Publishers.
- Pach, J., Shahrokhi, F. & Szegedy, M. (1996), Applications of the crossing number, *Algorithmica* 16, 111–117.
- Xu, J. (2001), Topological structure and analysis of interconnection networks, Kluwer Academic Publishers.

Compositional Type Systems for Stack-Based Low-Level Languages

Ando Saabas

Tarmo Uustalu

Institute of Cybernetics at Tallinn University of Technology
 Akadeemia tee 21, 12618 Tallinn, Estonia
 Email: {ando|tarmo}@cs.ioc.ee

Abstract

It is widely believed that low-level languages with jumps must be difficult to reason about by being inherently non-modular. We have recently argued that this is untrue and proposed a novel method for developing compositional natural semantics and Hoare logics for low-level languages and demonstrated its viability on the example of a simple low-level language with expressions (Saabas & Uustalu 2005). The central idea is to use the implicit structure of finite disjoint unions present in low-level code as an (ambiguous) phrase structure.

Here we apply our method to a stack-based language and develop it further. We define a compositional natural semantics and Hoare logic for this language and go then on to show that, in addition to Hoare logics, one can also derive compositional type systems as weaker specification languages with the same method. We describe type systems for stack-error freedom and secure information flow.

Keywords: low-level languages, compositionality, Hoare logics, type systems, dataflow analyses, certified code, compilation of proofs, typings from compilation

1 Introduction

The advent of the paradigm of proof-carrying (or, more generally, certified) code has generated significant interest in reasoning about low-level code. This is because software is usually distributed in compiled form for the sake of self-containedness, but also because certification of compiled code instead of source programs eliminates the need for the software consumer to trust a compiler. Low-level languages are widely believed to be difficult to reason about as inherently non-modular. The lack of modularity is attributed to low-level code being flat (a set of labelled instructions with no explicit structure) and to the presence of general jumps. If a language is non-modular, it cannot have a compositional semantics or logic or type system.

We have recently argued that the non-modularity premise is untrue and proposed to exploit a very basic implicit structure present in low-level code as the “phrase structure” for semantic descriptions and logics of low-level languages (Saabas & Uustalu 2005). The structure in question is given by finite unions of pieces of code with non-overlapping support: a piece

of code is either a single instruction or a finite union of non-overlapping pieces of code. Despite its banality and ambiguity (any piece of code can be parsed in many ways), this structure is perfectly viable from the point of metatheory and attractive from the point-of-view of practical reasoning about programs: it supports the idea that properties of a large piece of code should be provable from properties of its constituent small pieces (which can be established by different parties). An additional bonus of the method is that it supports compiling high-level programs together with proofs; in the compilation, the structure of a high-level source program hints the optimal way to structure its low-level equivalent.

In (Saabas & Uustalu 2005), we demonstrated this method on the example of a simple low-level language GOTO with expressions. In this paper, we develop it further and consider an operand-stack based language PUSH. This language, although fairly similar on the surface, is more demanding because of the possibility of abnormal terminations due to stack errors (wrong operand types, stack underflow), but it is also richer in that, for PUSH, it makes sense to study not only logics as calculi for correctness, but also type systems as calculi for attesting weaker properties such as basic safety (stack-error freedom) or properties usually established by dataflow analyses.

The technical contribution of the paper is as follows. We define a structured version SPUSH of PUSH and equip it with a compositional natural semantics discriminating between normal and abnormal terminations and agreeing with the non-compositional small-step semantics of PUSH. We also define an error-free partial-correctness Hoare logic for SPUSH and prove it to be sound and (relatively) complete wrt. the natural semantics. For a compilation from WHILE to SPUSH, we show that it preserves WHILE proofs in a constructive sense (so that proof compilation is possible) and reflects SPUSH proofs. Beyond the logic, we also describe two type systems for SPUSH. The first system is a weakening of the Hoare logic and attests stack-error freedom, which we show sound and also complete wrt. an appropriate abstracted natural semantics. We also show that our compilation from WHILE can be augmented to accompany the SPUSH code delivered with a typing derivation attesting that it is stack-error free. The second type system is equivalent to a secure information flow analysis.

The cornerstone technical ideas of the paper are: (i) low-level languages can be handled in a compositional way by exploiting an implicit phrase structure that they do have anyway, (ii) natural semantics can be made sensitive to abnormal terminations by introducing a special abnormal evaluation relation, (iii) Hoare logics and type systems should be derived systematically from natural semantics descriptions, (iv) the abstract interpretations that underlie dataflow

analyses can be described as abstract natural semantics and the analyses themselves as type systems. Not all of these ideas are new, but we believe that the paper combines them in a useful fashion.

The organization of the papers is the following. In Section 2, we introduce the syntax of the language PUSH and its non-compositional small-step semantics. In Section 3, we describe the syntax and the compositional natural (big-step) semantics of the structured version SPUSH. In Section 4, we describe the corresponding Hoare logic. In Sections 5 and 6, we discuss the abstract natural semantics and the type system for safe stack usage. In Section 7, we discuss a compilation of WHILE programs to SPUSH pieces of code and the corresponding compilation of proofs and type derivation generation. Section 8 discusses the abstract natural semantics and type system for secure information flow. Section 9 is a brief overview of the related work while 10 concludes.

2 The language and its small-step semantics

As advertised, our object of study is a simple operand-stack based low-level language, which we call PUSH.

The building blocks of the syntax of PUSH are labels $\ell \in \mathbf{Label}$, which are natural numbers, and instructions $instr \in \mathbf{Instr}$. We also assume having a countable set of program variables (registers) $x \in \mathbf{Var}$. The instructions of the language are defined by the grammar

$$instr ::= load\ x \mid store\ x \mid push\ n \\ \mid add \mid eq \mid \dots \mid goto\ \ell \mid gotoF\ \ell$$

A piece of code $c \in \mathbf{Code}$ is a finite set of labelled instructions, i.e., a set of pairs of a label and an instruction: $\mathbf{Code} =_{df} \mathcal{P}_{fin}(\mathbf{Label} \times \mathbf{Instr})$. A piece of code c is wellformed, if no label in it labels two different instructions, i.e., if $(\ell, instr), (\ell, instr') \in c$ imply $instr = instr'$. The domain of a piece of code is the set of labels in it: $dom(c) =_{df} \{\ell \mid (\ell, instr) \in c\}$.

Semantic descriptions of imperative languages are defined in terms of states. A state for PUSH consists of a label ℓ , stack ζ and store σ , which record the pc value and the content of the operand stack and the store at a moment: $\mathbf{State} =_{df} \mathbf{Label} \times \mathbf{Stack} \times \mathbf{Store}$. A stack is a list whose elements can be both boolean or integer values: $\mathbf{Stack} =_{df} (\mathbb{Z} \cup \mathbb{B})^*$. (We use the notation X^* for lists over X , $[]$ for the empty list, $x :: xs$ for the list with head x and tail xs and $xs ++ ys$ for the concatenation of xs and ys .) Variables can only be of integer type and must always be defined: $\mathbf{Store} =_{df} \mathbf{Var} \rightarrow \mathbb{Z}$.

If a language is low-level, its semantics is usually described in an operational form that is small-step (there is no non-trivial notion of big steps one could talk of). The small-step semantics of PUSH is formulated via a single-step reduction relation $- \mapsto \subseteq \mathbf{State} \times \mathbf{Code} \times \mathbf{State}$ defined in Figure 1. The associated multi-step reduction relation \rightarrow^* is its reflexive-transitive closure. It is immediate that \rightarrow is deterministic, there is always at most one step possible. A state can be terminal ($c \vdash (\ell, \sigma) \not\mapsto$) for two reasons: (i) we have $\ell \notin dom(c)$, which signifies normal termination, or (ii) we have $\ell \in dom(c)$ but the rule for the instruction at ℓ does not apply because of wrong types or shortage of potential operands on the stack, which signifies abnormal termination. (The possibility of abnormal terminations was not present in the language GOTO of (Saabas & Uustalu 2005).) The obvious shortcoming of this semantics is that it is entirely non-compositional (there is no phrase structure to follow) and that all of the code must be known at all times because of the jump instructions.

$$\begin{array}{c} \frac{(\ell, store\ x) \in c \quad n \in \mathbb{Z}}{c \vdash (\ell, n :: \zeta, \sigma) \mapsto (\ell + 1, \zeta, \sigma[x \mapsto n])} \text{ store} \\ \frac{(\ell, load\ x) \in c}{c \vdash (\ell, \zeta, \sigma) \mapsto (\ell + 1, \sigma(x) :: \zeta, \sigma)} \text{ load} \\ \frac{(\ell, push\ n) \in c}{c \vdash (\ell, \zeta, \sigma) \mapsto (\ell + 1, n :: \zeta, \sigma)} \text{ push} \\ \frac{(\ell, add) \in c \quad n_0, n_1 \in \mathbb{Z}}{c \vdash (\ell, n_0 :: n_1 :: \zeta, \sigma) \mapsto (\ell + 1, n_0 + n_1 :: \zeta, \sigma)} \text{ add} \\ \frac{(\ell, eq) \in c \quad n_0, n_1 \in \mathbb{Z}}{c \vdash (\ell, n_0 :: n_1 :: \zeta, \sigma) \mapsto (\ell + 1, n_0 = n_1 :: \zeta, \sigma)} \text{ eq} \\ \dots \\ \frac{(\ell, goto\ m) \in c}{c \vdash (\ell, \zeta, \sigma) \mapsto (m, \zeta, \sigma)} \text{ goto} \\ \frac{(\ell, gotoF\ m) \in c}{c \vdash (\ell, tt :: \zeta, \sigma) \mapsto (\ell + 1, \zeta, \sigma)} \text{ gotoF}^{tt} \\ \frac{(\ell, gotoF\ m) \in c}{c \vdash (\ell, ff :: \zeta, \sigma) \mapsto (m, \zeta, \sigma)} \text{ gotoF}^{ff} \end{array}$$

Figure 1: Single-step reduction rules of PUSH

3 Structured version and natural semantics

To overcome the non-compositionality problem of the semantics described above, some structure needs to be introduced into PUSH code. As was shown in (Saabas & Uustalu 2005), a useful structure to use for defining the semantics of a low-level language compositionally is that of finite unions of non-overlapping pieces of code. This is present in the code anyway, but it is ambiguous (any set is a finite union of sets in many ways) and implicit, so one has to choose and make the choices explicit. Hence we define a corresponding structured version of PUSH, which we call SPUSH. Structured pieces of code $sc \in \mathbf{SCode}$ are defined by the following grammar

$$sc ::= (\ell, instr) \mid \mathbf{0} \mid sc_0 \oplus sc_1$$

which stipulates that a piece of code is either a single labelled instruction or a finite union of pieces of code. We define the domain $dom(sc)$ of a piece of code sc to be the set of all labels in the code: $dom(\mathbf{0}) = \emptyset$, $dom((\ell, instr)) = \{\ell\}$, $dom(sc_0 \oplus sc_1) = dom(sc_0) \cup dom(sc_1)$.

A piece of code is wellformed iff the labels of all of its instructions are different: a single instruction is always wellformed, $\mathbf{0}$ is wellformed and $sc_0 \oplus sc_1$ is wellformed iff both sc_0 and sc_1 are wellformed and $dom(sc_0) \cap dom(sc_1) = \emptyset$. Note that contiguity is not required for wellformedness, the domain of a piece of code does not have to be an interval.

The compositional semantic description we give for SPUSH is a (big-step) natural semantics. Since there is the possibility of abnormal terminations and we want to distinguish between non-terminations and abnormal terminations, we define two evaluation relations, $\succ \rightarrow, \succ \dashv \rightarrow \subseteq \mathbf{State} \times \mathbf{SCode} \times \mathbf{State}$, one for normal, the other for abnormal terminating evaluations. Both relate possible initial states for evaluating a piece of code to the corresponding terminal states. The two relations are defined (mutually inductively) by the rules in Figure 2. Of course, alternatively one could say that we have just one evaluation relation but indexed by a doubleton for distinguishing between the two flavors of termination.

The $load_{ns}$ and $push_{ns}$ rules should be self-explanatory. Both $store\ x$ and add can potentially cause an error, therefore there are two rules for them, for normal and abnormal evaluation.

$$\begin{array}{c}
\frac{}{(\ell, \zeta, \sigma) \succ (\ell, \text{load } x) \rightarrow (\ell + 1, \sigma(x) :: \zeta, \sigma)} \text{load}_{\text{ns}} \\
\frac{n \in \mathbb{Z}}{(\ell, n :: \zeta, \sigma) \succ (\ell, \text{store } x) \rightarrow (\ell + 1, \zeta, \sigma[x \mapsto n])} \text{store}_{\text{ns}} \quad \frac{\forall n \in \mathbb{Z}, \zeta' \in (\mathbb{Z} \cup \mathbb{B})^*. \zeta \neq n :: \zeta'}{(\ell, \zeta, \sigma) \succ (\ell, \text{store } x) \dashv (\ell, \zeta, \sigma)} \text{store}_{\text{ns}}^{ab} \\
\frac{}{(\ell, \zeta, \sigma) \succ (\ell, \text{push } n) \rightarrow (\ell + 1, n :: \zeta, \sigma)} \text{push}_{\text{ns}} \\
\frac{n_0, n_1 \in \mathbb{Z}}{(\ell, n_0 :: n_1 :: \zeta, \sigma) \succ (\ell, \text{add}) \rightarrow (\ell + 1, n_0 + n_1 :: \zeta, \sigma)} \text{add}_{\text{ns}} \quad \frac{\forall n_0, n_1 \in \mathbb{Z}, \zeta' \in (\mathbb{Z} \cup \mathbb{B})^*. \zeta \neq n_0 :: n_1 :: \zeta'}{(\ell, \zeta, \sigma) \succ (\ell, \text{add}) \dashv (\ell, \zeta, \sigma)} \text{add}_{\text{ns}}^{ab} \\
\dots \\
\left[\begin{array}{c} \frac{}{(m, \zeta, \sigma) \succ (\ell, \text{goto } m) \rightarrow (\ell', \zeta', \sigma')} \\ \frac{}{(\ell, \zeta, \sigma) \succ (\ell, \text{goto } m) \rightarrow (\ell', \zeta', \sigma')} \\ \frac{}{(m, \zeta, \sigma) \succ (\ell, \text{goto } m) \dashv (\ell', \zeta', \sigma')} \\ \frac{}{(\ell, \zeta, \sigma) \succ (\ell, \text{goto } m) \dashv (\ell', \zeta', \sigma')} \end{array} \right] \quad \frac{m \neq \ell}{(\ell, \zeta, \sigma) \succ (\ell, \text{goto } m) \rightarrow (m, \zeta, \sigma)} \text{goto}_{\text{ns}}^{\neq} \\
\left[\begin{array}{c} \frac{}{(\ell, \text{tt} :: \zeta, \sigma) \succ (\ell, \text{gotoF } m) \rightarrow (\ell + 1, \zeta, \sigma)} \\ \frac{}{(m, \zeta, \sigma) \succ (\ell, \text{gotoF } m) \rightarrow (\ell', \zeta', \sigma')} \\ \frac{}{(\ell, \text{ff} :: \zeta, \sigma) \succ (\ell, \text{gotoF } m) \rightarrow (\ell', \zeta', \sigma')} \\ \frac{}{(m, \zeta, \sigma) \succ (\ell, \text{gotoF } m) \dashv (\ell', \zeta', \sigma')} \\ \frac{}{(\ell, \text{ff} :: \zeta, \sigma) \succ (\ell, \text{gotoF } m) \dashv (\ell', \zeta', \sigma')} \\ \frac{\forall b \in \mathbb{B}, \zeta' \in (\mathbb{Z} \cup \mathbb{B})^*. \zeta \neq b :: \zeta'}{(\ell, \zeta, \sigma) \succ (\ell, \text{gotoF } m) \dashv (\ell, \zeta, \sigma)} \end{array} \right] \quad \frac{m \neq \ell}{(\ell, \text{tt} :: \zeta, \sigma) \succ (\ell, \text{gotoF } m) \rightarrow (\ell + 1, \zeta, \sigma)} \text{gotoF}_{\text{ns}}^{\neq \text{tt}} \\
\frac{m \neq \ell}{(\ell, \text{ff} :: \zeta, \sigma) \succ (\ell, \text{gotoF } m) \rightarrow (m, \zeta, \sigma)} \text{gotoF}_{\text{ns}}^{\neq \text{ff}} \\
\frac{m \neq \ell \quad \forall b \in \mathbb{B}, \zeta' \in (\mathbb{Z} \cup \mathbb{B})^*. \zeta \neq b :: \zeta'}{(\ell, \zeta, \sigma) \succ (\ell, \text{gotoF } m) \dashv (\ell, \zeta, \sigma)} \text{gotoF}_{\text{ns}}^{\neq ab} \\
\frac{\text{ffs} \in \{\text{ff}\}^*}{(\ell, \text{ffs} ++ \text{tt} :: \zeta, \sigma) \succ (\ell, \text{gotoF } \ell) \rightarrow (\ell + 1, \zeta, \sigma)} \text{gotoF}_{\text{ns}}^{\text{ff}} \\
\frac{\text{ffs} \in \{\text{ff}\}^* \quad \forall b \in \mathbb{B}, \zeta' \in (\mathbb{Z} \cup \mathbb{B})^*. \zeta \neq b :: \zeta'}{(\ell, \text{ffs} ++ \zeta, \sigma) \succ (\ell, \text{gotoF } \ell) \dashv (\ell, \zeta, \sigma)} \text{gotoF}_{\text{ns}}^{\text{ff}, ab} \\
\frac{\ell \in \text{dom}(sc_i) \quad (\ell, \zeta, \sigma) \succ sc_i \rightarrow (\ell'', \zeta'', \sigma'') \quad (\ell'', \zeta'', \sigma'') \succ sc_0 \oplus sc_1 \rightarrow (\ell', \zeta', \sigma')}{(\ell, \zeta, \sigma) \succ sc_0 \oplus sc_1 \rightarrow (\ell', \zeta', \sigma')} \oplus_{\text{ns}} \\
\frac{\ell \in \text{dom}(sc_i) \quad (\ell, \zeta, \sigma) \succ sc_i \dashv (\ell', \zeta', \sigma')}{(\ell, \zeta, \sigma) \succ sc_0 \oplus sc_1 \dashv (\ell', \zeta', \sigma')} \oplus_{\text{ns}}^{abn} \\
\frac{\ell \in \text{dom}(sc_i) \quad (\ell, \zeta, \sigma) \succ sc_i \rightarrow (\ell'', \zeta'', \sigma'') \quad (\ell'', \zeta'', \sigma'') \succ sc_0 \oplus sc_1 \dashv (\ell', \zeta', \sigma')}{(\ell, \zeta, \sigma) \succ sc_0 \oplus sc_1 \dashv (\ell', \zeta', \sigma')} \oplus_{\text{ns}}^{abl} \\
\frac{\ell \notin \text{dom}(sc)}{(\ell, \zeta, \sigma) \succ sc \rightarrow (\ell, \zeta, \sigma)} \text{ood}_{\text{ns}}
\end{array}$$

Figure 2: Natural semantics rules of SPUSH

We have spelled out the rules for `goto m` and `gotoF m` instructions in two different ways: a recursive style (in square brackets) and a direct style. The two styles are equivalent, but we comment only the direct style. The recursive style could be seen as a formal explanation of the direct style. The issue is that, differently from other single-instruction pieces of code, a `goto` or `gotoF` instruction can loop back on itself. This happens when the labelling label and the target label coincide.

The side condition in the $\text{goto}_{\text{ns}}^{\neq}$ rule states that a `goto m` instruction only terminates, if it does not loop back on itself. The $\text{gotoF}_{\text{ns}}^{\neq \text{tt}}$ rule should be self-explanatory, however the `gotoF m` rules for the case there is a `ff` on the top of the stack should be explained. The complication here is that just like `goto m`, `gotoF m` can loop back on itself. Unlike `goto m` however, it cannot loop infinitely, since every successful jump removes an element from the stack. Instead it can either exit the loop at some point (when it encounters a `tt` on top of the stack), or cause an error if it either encounters an integer on the stack or the stack runs empty. Therefore, two rules ($\text{gotoF}_{\text{ns}}^{\neq}$ and $\text{gotoF}_{\text{ns}}^{\neq \text{ab}}$) are needed for normal and abnormal behavior of `gotoF m` for the case when it loops back on itself. The rule $\text{gotoF}_{\text{ns}}^{\neq \text{ab}}$ covers the case when there is no boolean value at the top of the stack.

The rule \oplus_{ns} says that, to evaluate the union $sc_0 \oplus sc_1$ starting from some state such that the `pc` is in the domain of sc_i , one first needs to evaluate sc_i , and then evaluate the whole union again, but starting from the new intermediate state reached. Finally, the ood_{ns} rule is needed to reflect the case where the reduction sequence is normally terminated because the `pc` has landed outside the domain of the code.

It is fairly straightforward that the `pc` in the final state of a normally terminating evaluation of a code is outside its domain while the `pc` in the final state of an abnormally terminating evaluation is inside. Evaluation is deterministic in the sense that any piece of code terminates either normally or abnormally in a definite state, if it terminates at all.

Every SPUSH piece of code can be mapped into a PUSH piece of code using a forgetful function $U \in \mathbf{SCode} \rightarrow \mathbf{Code}$, defined by $U((\ell, \text{instr})) =_{\text{df}} \{(\ell, \text{instr})\}$, $U(\mathbf{0}) =_{\text{df}} \emptyset$, $U(sc_0 \oplus sc_1) =_{\text{df}} U(sc_0) \oplus U(sc_1)$. The compositional natural semantics of SPUSH agrees with the non-compositional semantics of PUSH in the following technical sense.

Theorem 1 (Preservation of evaluations by U) (i) If $(\ell, \zeta, \sigma) \succ_{sc} (\ell', \zeta', \sigma')$, then $U(sc) \vdash (\ell, \zeta, \sigma) \rightarrow^* (\ell', \zeta', \sigma') \not\vdash$ and $\ell' \notin \text{dom}(sc)$. (ii) If $(\ell, \zeta, \sigma) \succ_{sc} \dashv (\ell', \zeta', \sigma')$, then $U(sc) \vdash (\ell, \zeta, \sigma) \rightarrow^* (\ell', \zeta', \sigma') \not\vdash$ and $\ell' \in \text{dom}(sc)$.

Proof. By induction on the derivation of $(\ell, \zeta, \sigma) \succ_{sc} (\ell', \zeta', \sigma')$ or $(\ell, \zeta, \sigma) \succ_{sc} \dashv (\ell', \zeta', \sigma')$. \square

Theorem 2 (Reflection of stuck reduction sequences by U) (i) If $U(sc) \vdash (\ell, \zeta, \sigma) \rightarrow^* (\ell', \zeta', \sigma') \not\vdash$ and $\ell' \notin \text{dom}(sc)$, then $(\ell, \zeta, \sigma) \succ_{sc} (\ell', \zeta', \sigma')$. (ii) If $U(sc) \vdash (\ell, \zeta, \sigma) \rightarrow^* (\ell', \zeta', \sigma') \not\vdash$ and $\ell' \in \text{dom}(sc)$, then $(\ell, \zeta, \sigma) \succ_{sc} \dashv (\ell', \zeta', \sigma')$.

Proof. By induction on the structure of sc and subordinate induction on the length of the reduction sequence. \square

From these theorems it is immediate that the SPUSH semantics of a structured version of a piece of PUSH code cannot depend on the way it is structured: if $U(sc) = U(sc')$, then sc and sc' have exactly the same evaluations (although with different derivations).

4 Hoare logic

The compositional natural semantics of SPUSH is a good basis for developing a compositional Hoare logic of it. Just as evaluations relate an initial and a terminal state, Hoare triples relate pre- and postconditions about states. Since a state contains a `pc` value and stack content, it must be possible to refer to these in assertions. In our logic, we have special individual constants pc and st to refer to them. Using the constant pc , we can make assertions about particular program points by constraining the state to correspond to a certain `pc` value. This allows us to make assertions only about program points through which the particular piece of code is entered or exited, thus eliminating the need for global contexts of invariants and making reasoning modular.

The logic we define is an error-free partial correctness logic: for a Hoare triple to be derivable, the postcondition must be satisfied by the terminal state of any normal evaluation and abnormal evaluations from the allowed initial states must be impossible. (We would get a more expressive partial correctness logic with triples with two postconditions, one for normal terminations, the other for abnormal terminations; in the case of a programming language with error-handling constructs, that approach is the only reasonable one, see, e.g., (Schröder & Mossakowski 2004). Our logic corresponds to the case where the abnormal postcondition is always \perp , so there is no need to ever spell it out. A different version where it is always \top would correspond to error-ignoring partial correctness.)

The signature of the Hoare logic contains, as extra-arithmetical and extra-list constants, special individual constants pc , st and the program variables **Var**, to refer to the values of the program counter, stack and program variables in a state. The assertions $P, Q \in \mathbf{Assn}$ are formulae over that signature in an ambient logical language containing the signature of arithmetic and lists of integers and booleans. We use the notation $Q[x_0, \dots, x_n \mapsto t_0, \dots, t_n]$ to denote that every occurrence of x_i in Q has been replaced with t_i . The derivable Hoare triples $\{ \} - \{ \} \subseteq \mathbf{Assn} \times \mathbf{SCode} \times \mathbf{Assn}$ are defined inductively by the rules in Figure 3.

The extra disjunct $pc \neq \ell \wedge Q$ in the rules for primitive instructions is required because of the semantic rule ood_{ns} : if we evaluate the instruction starting from outside the domain of the instruction (i.e. $pc \neq \ell$), we have immediately terminated and have hence remained in the same state, therefore any assertion holding before evaluating the instruction will also hold after. The disjunct $m = \ell$ in the rule for `goto m` accounts for the case when `goto m` loops back on itself. We have a similar case with the `gotoF m` rule, but here the situation is more subtle. As explained in Section 3, when `gotoF m` loops back on itself, it can either exit normally to the next instruction (in case there is some number of `ff`s on the stack, followed by a `tt`), or raise an error. The disjunct $m = \ell \wedge \dots$ accounts for that case.

The rule for unions can be seen as mix of the while and sequence rules of the Hoare logic of WHILE: if, evaluating either sc_0 or sc_1 starting from a state that satisfies P and has the `pc` value in the domain of sc_0 resp. sc_1 , we end in a state satisfying P , then, after evaluating their union $sc_0 \oplus sc_1$ starting from a state satisfying P , we are guaranteed to be in a state satisfying P . Furthermore, we know that we are then outside the domains of both sc_0 and sc_1 . The rule of consequence is the same as in the standard Hoare logic. Note that we have circumvented the inevitable *incompleteness* of any axiomatization of logics containing arithmetic by invoking semantic entailment

instead of deducibility in the premises of the consequence rule.

The Hoare logic is sound and complete.

Theorem 3 (Soundness of Hoare logic) *If $\{P\} sc \{Q\}$ and $(\ell, \zeta, \sigma) \models_\alpha P$, then (i) for any (ℓ', ζ', σ') such that $(\ell, \zeta, \sigma) \succ_{sc} (\ell', \zeta', \sigma')$, we have $(\ell', \zeta', \sigma') \models_\alpha Q$, and (ii) there is no (ℓ', ζ', σ') such that $(\ell, \zeta, \sigma) \succ_{sc \rightarrow} (\ell', \zeta', \sigma')$.*

Proof. By induction on the derivation of $\{P\} sc \{Q\}$. \square

To get completeness, we have to assume that the underlying logical language is *expressive*. For any assertion Q , we need an assertion $wlp(sc, Q)$ that, semantically, is its weakest precondition, i.e., for any state (ℓ, ζ, σ) and valuation α of free variables, we have $(\ell, \zeta, \sigma) \models wlp(sc, Q)$ iff $(\ell, \zeta, \sigma) \succ_{sc} (\ell', \zeta', \sigma')$ implies $(\ell', \zeta', \sigma') \models Q$ for any (ℓ', ζ', σ') . The wlp function is available for example when the underlying logical language has a greatest fixedpoint operator.

Lemma 1 $\{wlp(sc, Q)\} sc \{Q\}$.

Proof. By induction on the structure of sc . \square

Theorem 4 (Completeness of Hoare logic) *If, for any (ℓ, ζ, σ) and α such that $(\ell, \zeta, \sigma) \models_\alpha P$, it holds that (i) for any (ℓ', ζ', σ') such that $(\ell, \zeta, \sigma) \succ_{sc} (\ell', \zeta', \sigma')$, we have $(\ell', \zeta', \sigma') \models_\alpha Q$, and (ii) there is no (ℓ', ζ', σ') such that $(\ell, \zeta, \sigma) \succ_{sc \rightarrow} (\ell', \zeta', \sigma')$, then $\{P\} sc \{Q\}$.*

Proof. Immediate from the lemma using that any precondition of an assertion entails its wlp . \square

5 Abstract natural semantics

We now proceed to defining an abstract natural semantics for SPUSH that operates on type names as abstract values instead of concrete values. This allows us to later prove a type system for basic safety sound and complete. While soundness of the type system could also be shown wrt. the concrete natural semantics, completeness cannot.

The abstract semantics is defined in terms of abstract states, which are pairs of labels $\ell \in \mathbf{Label}$ and abstract stack contents $\psi \in \mathbf{AbsStack}$: $\mathbf{AbsState} =_{\text{df}} \mathbf{Label} \times \mathbf{AbsStack}$. Instead of values, abstract stacks stack names of value types: $\mathbf{AbsStack} =_{\text{df}} \{\text{int}, \text{bool}\}^*$. We do not have an abstract store component in an abstract state. Since variables can only be integers in a concrete store, there is no interesting information to record. To relate a concrete state to an abstract state, we have a function $\text{abs} \in \mathbf{State} \rightarrow \mathbf{AbsState}$, defined by $\text{abs}(\ell, \zeta, \sigma) =_{\text{df}} (\ell, \text{abs}(\zeta))$ where $\text{abs} \in \mathbf{Stack} \rightarrow \mathbf{AbsStack}$ replaces concrete values in a stack with the names of their types: $\text{abs}(\perp) =_{\text{df}} \perp$, $\text{abs}(n :: \zeta) =_{\text{df}} \text{int} :: \text{abs}(\zeta)$ for $n \in \mathbb{Z}$, and $\text{abs}(b :: \zeta) =_{\text{df}} \text{bool} :: \text{abs}(\zeta)$ for $b \in \mathbb{B}$.

The abstract semantics is a rather straightforward rewrite of the concrete semantics to work on abstract states, but it is important to notice that this makes evaluation nondeterministic. Just like in the concrete semantics, we need to distinguish between abnormal and normal evaluations, so there are two evaluation relations $\succ \rightarrow, \succ \rightarrow \rightarrow \subseteq \mathbf{AbsState} \times \mathbf{SCode} \times \mathbf{AbsState}$. The rules of the abstract natural semantics are given in Figure 4. Mimicking those of the concrete semantics from Figure 2, they should be self-explanatory. As before, we have spelled out the rules

for goto and gotoF in two alternative styles, recursive and direct. The nondeterminism stems from the non-exclusive rules of gotoF.

Concrete evaluations are preserved by abstraction.

Theorem 5 (Preservation of evaluations by abstraction) (i) *If $(\ell, \zeta, \sigma) \succ_{sc} (\ell', \zeta', \sigma')$, then $\text{abs}(\ell, \zeta, \sigma) \succ_{sc} \text{abs}(\ell', \zeta', \sigma')$.* (ii) *If $(\ell, \zeta, \sigma) \succ_{sc \rightarrow} (\ell', \zeta', \sigma')$, then $\text{abs}(\ell, \zeta, \sigma) \succ_{sc \rightarrow} \text{abs}(\ell', \zeta', \sigma')$.*

6 Type system from the Hoare logic

With the abstract semantics defined, we are now ready to show that the Hoare logic we have formulated for SPUSH can be weakened into a type system for establishing basic code safety—the absence of operand type and stack underflow errors in an PUSH program. The abstract semantics allows us to prove the type system not only sound, but also complete.

Instead of relating assertions as Hoare triples do, typings relate state types. The intuitive meaning of a typing is analogous to that of a Hoare triple: it says that if the given piece of code is run from an initial state in the given pretype, then if it terminates normally, the final state is in the posttype, and, moreover, it cannot terminate abnormally. Contrarily to assertions, state types are designed to record only that state information that is necessary for guaranteeing error-freedom.

The building blocks for state types are value types $\tau \in \mathbf{ValType}$ and stack types $\Psi \in \mathbf{StackType}$, defined by the grammars

$$\begin{aligned} \tau &::= \perp \mid \text{int} \mid \text{bool} \mid ? \\ \Psi &::= \perp \mid \square \mid \tau :: \Psi \mid * \end{aligned}$$

(note the overloading of the \perp sign). A state type $\Pi \in \mathbf{StateType}$ is a finite set of labelled stack types, i.e., pairs of a label and a stack type: $\mathbf{StateType} =_{\text{df}} \mathcal{P}_{\text{fin}}(\mathbf{Label} \times \mathbf{StackType})$. A state type Π is well-formed iff no label ℓ in it labels more than one stack type, i.e., $(\ell, \Psi) \in \Pi$ and $(\ell, \Psi') \in \Pi$ imply $\Psi = \Psi'$. The domain $\text{dom}(\Pi)$ of a state type is the set of labels appearing in it, i.e., $\text{dom}(\Pi) =_{\text{df}} \{\ell \mid (\ell, \Psi) \in \Pi\}$.

We will use the notation $\Pi|_L$ for the restriction of a state type Π to a domain $L \subseteq \mathbf{Label}$, i.e., $\Pi|_L =_{\text{df}} \{(\ell, \Psi) \mid (\ell, \Psi) \in \Pi, \ell \in L\}$, and write \bar{L} for the complement of L , i.e., $\bar{L} =_{\text{df}} \mathbf{Label} \setminus L$.

The meanings of value, stack and state types are set-theoretic, they denote sets of abstract values, abstract stacks and abstract states. The semantic functions $\langle \!| - \!| \rangle \in \mathbf{ValType} \rightarrow \mathcal{P}(\{\text{int}, \text{bool}\})$, $\langle \!| - \!| \rangle \in \mathbf{StackType} \rightarrow \mathcal{P}(\mathbf{AbsStack})$, $\langle \!| - \!| \rangle \in \mathbf{StateType} \rightarrow \mathcal{P}(\mathbf{AbsState})$ are defined as follows:

$$\begin{aligned} \langle \!| \perp \!| \rangle &=_{\text{df}} \emptyset \\ \langle \!| \text{int} \!| \rangle &=_{\text{df}} \{\text{int}\} \\ \langle \!| \text{bool} \!| \rangle &=_{\text{df}} \{\text{bool}\} \\ \langle \!| ? \!| \rangle &=_{\text{df}} \{\text{int}, \text{bool}\} \\ \langle \!| \perp \!| \rangle &=_{\text{df}} \emptyset \\ \langle \!| \square \!| \rangle &=_{\text{df}} \{\square\} \\ \langle \!| \tau :: \Psi \!| \rangle &=_{\text{df}} \{\delta :: \psi \mid \delta \in \langle \!| \tau \!| \rangle, \psi \in \langle \!| \Psi \!| \rangle\} \\ \langle \!| * \!| \rangle &=_{\text{df}} \{\text{int}, \text{bool}\}^* \\ \langle \!| \Pi \!| \rangle &=_{\text{df}} \{(\ell, \psi) \mid (\ell, \Psi) \in \Pi, \psi \in \langle \!| \Psi \!| \rangle\} \end{aligned}$$

On each of the three categories of types, we define a subtyping relation by the rules in Figure 5. These are relations $\leq \subseteq \mathbf{ValType} \times \mathbf{ValType}$, $\leq \subseteq \mathbf{StackType} \times \mathbf{StackType}$, $\leq \subseteq \mathbf{StateType} \times \mathbf{StateType}$.

$$\begin{array}{c}
\frac{}{\{(pc = \ell \wedge Q[pc, st \mapsto \ell + 1, x :: st]) \vee (pc \neq \ell \wedge Q)\} (\ell, \text{load } x) \{Q\}} \text{load}_{\text{hoa}} \\
\frac{}{\{(pc = \ell \wedge \exists z \in \mathbb{Z}, w \in (\mathbb{Z} \cup \mathbb{B})^*. st = z :: w \wedge Q[pc, st, x \mapsto \ell + 1, w, z]) \vee (pc \neq \ell \wedge Q)\} (\ell, \text{store } x) \{Q\}} \text{store}_{\text{hoa}} \\
\frac{}{\{(pc = \ell \wedge Q[pc, st \mapsto \ell + 1, n :: st]) \vee (pc \neq \ell \wedge Q)\} (\ell, \text{push } n) \{Q\}} \text{push}_{\text{hoa}} \\
\frac{}{\left\{ \begin{array}{l} (pc = \ell \wedge \exists z_0, z_1 \in \mathbb{Z}, w \in (\mathbb{Z} \cup \mathbb{B})^*. st = z_0 :: z_1 :: w \wedge Q[pc, st \mapsto \ell + 1, z_0 + z_1 :: w]) \\ \vee (pc \neq \ell \wedge Q) \end{array} \right\} (\ell, \text{add}) \{Q\}} \text{add}_{\text{hoa}} \\
\vdots \\
\frac{}{\left\{ \begin{array}{l} (pc = \ell \wedge ((m \neq \ell \wedge Q[pc \mapsto m]) \vee m = \ell)) \\ \vee (pc \neq \ell \wedge Q) \end{array} \right\} (\ell, \text{goto } m) \{Q\}} \text{goto}_{\text{hoa}} \\
\frac{}{\left\{ \begin{array}{l} (pc = \ell \wedge ((\exists w \in (\mathbb{Z} \cup \mathbb{B})^*. st = \text{tt} :: w \wedge Q[pc, st \mapsto \ell + 1, w]) \\ \vee (\exists w \in (\mathbb{Z} \cup \mathbb{B})^*. st = \text{ff} :: w \wedge Q[pc, st \mapsto m, w]))) \\ \vee (m = \ell \wedge \exists \text{ffs} \in \{\text{ff}\}^*, w \in (\mathbb{Z} \cup \mathbb{B})^*. st = \text{ffs} ++ \text{tt} :: w \wedge Q[pc, st \mapsto \ell + 1, w])) \\ \vee (pc \neq \ell \wedge Q) \end{array} \right\} (\ell, \text{gotoF } m) \{Q\}} \text{gotoF}_{\text{hoa}} \\
\frac{\overline{\{P\} \mathbf{0} \{P\}} \mathbf{0}_{\text{hoa}} \quad \frac{\{pc \in \text{dom}(sc_0) \wedge P\} sc_0 \{P\} \quad \{pc \in \text{dom}(sc_1) \wedge P\} sc_1 \{P\}}{\{P\} sc_0 \oplus sc_1 \{pc \notin \text{dom}(sc_0) \wedge pc \notin \text{dom}(sc_1) \wedge P\}} \oplus_{\text{hoa}}}{P \models P' \quad \{P'\} sc \{Q'\} \quad Q' \models Q}{\{P\} sc \{Q\}} \text{conseq}_{\text{hoa}}
\end{array}$$

Figure 3: Hoare rules of SPUSH

$$\begin{array}{c}
\frac{}{(\ell, \psi) \succ (\ell, \text{load } x) \rightarrow (\ell + 1, \text{int} :: \psi)} \text{load}_{\text{ans}} \\
\frac{}{(\ell, \text{int} :: \psi) \succ (\ell, \text{store } x) \rightarrow (\ell + 1, \psi)} \text{store}_{\text{ans}} \quad \frac{\forall \psi' \in \{\text{int}, \text{bool}\}^*. \psi \neq \text{int} :: \psi'}{(\ell, \psi) \succ (\ell, \text{store } x) \rightarrow (\ell, \psi)} \text{store}_{\text{ans}}^{ab} \\
\frac{}{(\ell, \psi) \succ (\ell, \text{push } n) \rightarrow (\ell + 1, \text{int} :: \psi)} \text{push}_{\text{ans}} \\
\frac{}{(\ell, \text{int} :: \text{int} :: \psi) \succ (\ell, \text{add}) \rightarrow (\ell + 1, \text{int} :: \psi)} \text{add}_{\text{ans}} \quad \frac{\forall \psi' \in \{\text{int}, \text{bool}\}^*. \psi \neq \text{int} :: \text{int} :: \psi'}{(\ell, \psi) \succ (\ell, \text{add}) \rightarrow (\ell, \psi)} \text{add}_{\text{ans}}^{ab} \\
\vdots \\
\left[\begin{array}{l} \frac{(m, \psi) \succ (\ell, \text{goto } m) \rightarrow (\ell', \psi')}{(\ell, \psi) \succ (\ell, \text{goto } m) \rightarrow (\ell', \psi')} \\ \frac{(m, \psi) \succ (\ell, \text{goto } m) \rightarrow (\ell', \psi')}{(\ell, \psi) \succ (\ell, \text{goto } m) \rightarrow (\ell', \psi')} \\ \frac{(m, \psi) \succ (\ell, \text{goto } m) \rightarrow (\ell', \psi')}{(\ell, \psi) \succ (\ell, \text{goto } m) \rightarrow (\ell', \psi')} \end{array} \right] \quad \frac{m \neq \ell}{(\ell, \psi) \succ (\ell, \text{goto } m) \rightarrow (m, \psi)} \text{goto}_{\text{ans}}^{\neq} \\
\left[\begin{array}{l} \frac{(\ell, \text{bool} :: \psi) \succ (\ell, \text{gotoF } m) \rightarrow (\ell + 1, \psi)}{(m, \psi) \succ (\ell, \text{gotoF } m) \rightarrow (\ell', \psi')} \\ \frac{(\ell, \text{bool} :: \psi) \succ (\ell, \text{gotoF } m) \rightarrow (\ell', \psi')}{(m, \psi) \succ (\ell, \text{gotoF } m) \rightarrow (\ell', \psi')} \\ \frac{(m, \psi) \succ (\ell, \text{gotoF } m) \rightarrow (\ell', \psi')}{(\ell, \text{bool} :: \psi) \succ (\ell, \text{gotoF } m) \rightarrow (\ell', \psi')} \\ \frac{(m, \psi) \succ (\ell, \text{gotoF } m) \rightarrow (\ell', \psi')}{(\ell, \text{bool} :: \psi) \succ (\ell, \text{gotoF } m) \rightarrow (\ell', \psi')} \\ \frac{\forall \psi' \in \{\text{int}, \text{bool}\}^*. \psi \neq \text{bool} :: \psi'}{(\ell, \psi) \succ (\ell, \text{gotoF } m) \rightarrow (\ell, \psi)} \end{array} \right] \quad \frac{m \neq \ell}{(\ell, \text{bool} :: \psi) \succ (\ell, \text{gotoF } m) \rightarrow (\ell + 1, \psi)} \text{gotoF}_{\text{ans}}^{\neq \text{tt}} \\
\frac{m \neq \ell}{(\ell, \text{bool} :: \psi) \succ (\ell, \text{gotoF } m) \rightarrow (m, \psi)} \text{gotoF}_{\text{ans}}^{\neq \text{ff}} \\
\frac{m \neq \ell \quad \forall \psi' \in \{\text{int}, \text{bool}\}^*. \psi \neq \text{bool} :: \psi'}{(\ell, \psi) \succ (\ell, \text{gotoF } m) \rightarrow (\ell, \psi)} \text{gotoF}_{\text{ans}}^{\neq ab} \\
\frac{\text{bools} \in \{\text{bool}\}^*}{(\ell, \text{bools} ++ \text{bool} :: \psi) \succ (\ell, \text{gotoF } \ell) \rightarrow (\ell + 1, \psi)} \text{gotoF}_{\text{ans}}^{=} \\
\frac{\text{bools} \in \{\text{bool}\}^* \quad \forall \psi' \in \{\text{int}, \text{bool}\}^*. \psi \neq \text{bool} :: \psi'}{(\ell, \text{bools} ++ \psi) \succ (\ell, \text{gotoF } \ell) \rightarrow (\ell, \psi)} \text{gotoF}_{\text{ans}}^{=ab} \\
\frac{\ell \in \text{dom}(sc_i) \quad (\ell, \psi) \succ sc_i \rightarrow (\ell'', \psi'') \quad (\ell'', \psi'') \succ sc_0 \oplus sc_1 \rightarrow (\ell', \psi')}{(\ell, \psi) \succ sc_0 \oplus sc_1 \rightarrow (\ell', \psi')} \oplus_{\text{ans}} \\
\frac{\ell \in \text{dom}(sc_i) \quad (\ell, \psi) \succ sc_i \rightarrow (\ell', \psi')}{(\ell, \psi) \succ sc_0 \oplus sc_1 \rightarrow (\ell', \psi')} \oplus_{\text{ans}}^{abn} \\
\frac{\ell \in \text{dom}(sc_i) \quad (\ell, \psi) \succ sc_i \rightarrow (\ell'', \psi'') \quad (\ell'', \psi'') \succ sc_0 \oplus sc_1 \rightarrow (\ell', \psi')}{(\ell, \psi) \succ sc_0 \oplus sc_1 \rightarrow (\ell', \psi')} \oplus_{\text{ans}}^{abl} \\
\frac{\ell \notin \text{dom}(sc)}{(\ell, \psi) \succ sc \rightarrow (\ell, \psi)} \text{ood}_{\text{ans}}
\end{array}$$

Figure 4: Abstract natural semantics rules of SPUSH

$$\begin{array}{c}
\overline{\tau \leq \tau} \quad \overline{\perp \leq \tau} \quad \overline{\tau \leq ?} \\
\hline
\overline{\Psi \leq \Psi} \quad \overline{\Psi \leq \Psi'' \quad \Psi'' \leq \Psi'} \quad \overline{\perp :: \Psi \leq \perp} \quad \overline{\tau :: \perp \leq \perp} \quad \overline{\perp \leq \Psi} \quad \overline{\Psi \leq *} \quad \overline{\tau \leq \tau' \quad \Psi \leq \Psi'} \\
\hline
\overline{\forall \ell, \Psi. (\ell, \Psi) \in \Pi \supset \Psi = \perp \vee \exists \Psi'. (\ell, \Psi') \in \Pi' \wedge \Psi \leq \Psi'} \\
\hline
\Pi \leq \Pi'
\end{array}$$

Figure 5: Subtyping rules of SPUSH

The subtyping relations thus introduced are sound and complete for the intended interpretation of subtyping as set inclusion.

Theorem 6 (Soundness and completeness of subtyping) (i) $\tau \leq \tau'$ iff $\langle \tau \rangle \subseteq \langle \tau' \rangle$. (ii) $\Psi \leq \Psi'$ iff $\langle \Psi \rangle \subseteq \langle \Psi' \rangle$. (iii) $\Pi \leq \Pi'$ iff $\langle \Pi \rangle \subseteq \langle \Pi' \rangle$.

Very pleasantly, the ranges $\mathcal{P}(\{\text{int}, \text{bool}\})$, $\{\langle \Psi \rangle \mid \Psi \in \mathbf{StackType}\}$, $\{\langle \Pi \rangle \mid \Pi \in \mathbf{StateType}\}$ of each of the three type interpretation functions are ω -complete lower semilattices with inclusion as the underlying partial order: set-theoretic binary intersections and intersections of nonincreasing ω -chains do not take us out of the range. (Note that the analogous statement about unions is not true, e.g., the set $\langle \perp \rangle \cup \langle \text{int} :: \perp \rangle$ has no type denotation. Note also that there are nonincreasing ω -chains that do not stabilize in a finite number of steps, e.g., $*$, $\text{int} :: *$, $\text{int} :: \text{int} :: *$, \dots , but all such chains have \perp as their glb.) Because of the soundness and completeness of subtyping, we can reflect this at the syntactic level: we can define a syntactic binary glb operator \wedge on types and a syntactic glb operator \bigwedge on deductively nonincreasing ω -sequences of types that are glb operators deductively (‘deductively’ meaning ‘in the sense of the subtyping relation’).

The typing relation $- : \longrightarrow \subseteq \mathbf{StateType} \times \mathbf{SCode} \times \mathbf{StateType}$ is defined by the rules in Figure 6. The typing rules for instructions are presented in a “weakest pretype” style, where the pretype is obtained by applying appropriate substitutions in the given posttype. For example the rule load_{ts} for $(\ell, \text{load } x)$ states that if stack type $\tau :: \Psi$ (where τ is int or $?$) or $*$ is required at label $\ell + 1$, then the suitable stack types for label ℓ are Ψ and $*$, respectively. Any other posttype at label $\ell + 1$ does not have a suitable pretype. At first sight, it might seem that wellformedness can be lost in the pretype by taking the union. This is in fact not the case: there is at most one stack type associated with label $\ell + 1$ in Π , hence both sets have at most one element and one of them must be empty. The rest of the non-jump instruction rules are defined in similar fashion.

The jump rules might need some explanation. The goto_{ts} rule allows to derive pretype $*$ for label ℓ : since the instruction does not terminate, any posttype will be satisfied by any pretype at label ℓ . The gotoF_{ts} rule combines two posttypes; since gotoF can branch, both posttypes must be satisfied at the entry, meaning that the pretype is the intersection of the posttypes. No pretype at ℓ can guarantee any posttype in the case of $(\ell, \text{gotoF } \ell)$, since such instruction could always terminate abnormally. The consequence rule could also be called subsumption, given that we are speaking about a type system: that is what it is really.

The type system is sound and complete wrt. the abstract natural semantics in the sense of error-free partial correctness.

Theorem 7 (Soundness of typing) If $sc : \Pi \longrightarrow \Pi'$ and $(\ell, \psi) \in \langle \Pi \rangle$, then (i) for any (ℓ', ψ') such that $(\ell, \psi) \succ_{sc} (\ell', \psi')$, we have $(\ell', \psi') \in \langle \Pi' \rangle$, and (ii) there is no (ℓ', ψ') such that $(\ell, \psi) \succ_{sc} (\ell', \psi')$.

Proof. By induction on the derivation of $sc : \Pi \longrightarrow \Pi'$, using that subtyping is sound. \square

From the preservation of evaluations by abstraction, it is immediate that therewith we also have soundness wrt. the concrete natural semantics.

Corollary 1 If $sc : \Pi \longrightarrow \Pi'$ and $\text{abs}(\ell, \zeta, \sigma) \in \langle \Pi \rangle$, then (i) for any (ℓ', ζ', σ') such that $(\ell, \zeta, \sigma) \succ_{sc} (\ell', \zeta', \sigma')$, we have $\text{abs}(\ell', \zeta', \sigma') \in \langle \Pi' \rangle$, and (ii) there is no (ℓ', ζ', σ') such that $(\ell, \zeta, \sigma) \succ_{sc} (\ell', \zeta', \sigma')$.

To prove completeness, we introduce a syntactic pretype function $\text{wpt} \in \mathbf{SCode} \times \mathbf{StateType} \rightarrow \mathbf{StateType}$. The definition is given in Figure 7. The ω -sequence glb in the clause for \oplus is welldefined because the operator S is monotone, making the sequence a nonincreasing chain. As S is also continuous, the glb is the greatest fixedpoint of S .

The following lemmata show that the wpt of a state type is semantically larger than any pretype and deductively (i.e., in the sense of typing) a pretype.

Lemma 2 If (i) for any (ℓ', ψ') such that $(\ell, \psi) \succ_{sc} (\ell', \psi')$, we have $(\ell', \psi') \in \langle \Pi' \rangle$, and (ii) there is no (ℓ', ψ') such that $(\ell, \psi) \succ_{sc} (\ell', \psi')$, then $(\ell, \psi) \in \langle \text{wpt}(sc, \Pi') \rangle$.

Lemma 3 $sc : \text{wpt}(sc, \Pi') \longrightarrow \Pi'$.

Proof. By induction on the structure of sc . \square

Theorem 8 (Completeness of typing) If, for any $(\ell, \psi) \in \langle \Pi \rangle$, it holds that (i) for any (ℓ', ψ') such that $(\ell, \psi) \succ_{sc} (\ell', \psi')$, we have $(\ell', \psi') \in \langle \Pi' \rangle$, and (ii) there is no (ℓ', ψ') such that $(\ell, \psi) \succ_{sc} (\ell', \psi')$, then $sc : \Pi \longrightarrow \Pi'$.

Proof. From the two lemmata, using that subtyping is complete. \square

It is fairly obvious that state types can be translated to assertions. We can define concretization functions $\text{conc} \in \mathbf{ValType} \rightarrow \mathcal{P}(\mathbb{Z} \cup \mathbb{B})$, $\text{conc} \in \mathbf{StackType} \rightarrow \mathcal{P}(\mathbf{Stack})$, $\text{conc} \in \mathbf{StateType} \rightarrow \mathbf{Assn}$, taking us from the language of the type system to the language of the logic, by

$$\begin{aligned}
\text{conc}(\perp) &=_{\text{df}} \emptyset \\
\text{conc}(\text{int}) &=_{\text{df}} \mathbb{Z} \\
\text{conc}(\text{bool}) &=_{\text{df}} \mathbb{B} \\
\text{conc}(?) &=_{\text{df}} \mathbb{Z} \cup \mathbb{B} \\
\text{conc}(\perp) &=_{\text{df}} \emptyset \\
\text{conc}(\perp) &=_{\text{df}} \emptyset \\
\text{conc}(\perp) &=_{\text{df}} \emptyset \\
\text{conc}(\tau :: \Psi) &=_{\text{df}} \{z :: w \mid z \in \text{conc}(\tau), w \in \text{conc}(\Psi)\} \\
\text{conc}(*) &=_{\text{df}} (\mathbb{Z} \cup \mathbb{B})^* \\
\text{conc}(\Pi) &=_{\text{df}} \bigvee \{pc = \ell \wedge st \in \text{conc}(\Psi) \mid (\ell, \Psi) \in \Pi\}
\end{aligned}$$

Concretization preserves and reflects derivable subtypings/entailments.

$$\begin{array}{c}
\frac{}{(\ell, \text{load } x) : \{(\ell, \Psi) \mid (\ell + 1, \tau :: \Psi) \in \Pi, \text{int} \leq \tau\} \cup \{(\ell, *) \mid (\ell + 1, *) \in \Pi\} \cup \Pi \upharpoonright_{\{\ell\}} \longrightarrow \Pi} \text{load}_{\text{ts}} \\
\frac{}{(\ell, \text{store } x) : \{(\ell, \text{int} :: \Psi) \mid (\ell + 1, \Psi) \in \Pi\} \cup \Pi \upharpoonright_{\{\ell\}} \longrightarrow \Pi} \text{store}_{\text{ts}} \\
\frac{}{(\ell, \text{push } n) : \{(\ell, \Psi) \mid (\ell + 1, \tau :: \Psi) \in \Pi, \text{int} \leq \tau\} \cup \{(\ell, *) \mid (\ell + 1, *) \in \Pi\} \cup \Pi \upharpoonright_{\{\ell\}} \longrightarrow \Pi} \text{push}_{\text{ts}} \\
\frac{}{(\ell, \text{add}) : \{(\ell, \text{int} :: \text{int} :: \Psi) \mid (\ell + 1, \tau :: \Psi) \in \Pi, \text{int} \leq \tau\} \cup \{(\ell, \text{int} :: \text{int} :: *) \mid (\ell + 1, *) \in \Pi\} \cup \Pi \upharpoonright_{\{\ell\}} \longrightarrow \Pi} \text{add}_{\text{ts}} \\
\vdots \\
\frac{m \neq \ell}{(\ell, \text{goto } m) : \{(\ell, \Psi) \mid (m, \Psi) \in \Pi\} \cup \Pi \upharpoonright_{\{\ell\}} \longrightarrow \Pi} \text{goto}_{\text{ts}}^{\neq} \quad \frac{}{(\ell, \text{goto } \ell) : \{(\ell, *)\} \cup \Pi \upharpoonright_{\{\ell\}} \longrightarrow \Pi} \text{goto}_{\text{ts}}^{\equiv} \\
\frac{m \neq \ell}{(\ell, \text{gotoF } m) : \{(\ell, \text{bool} :: (\Psi \wedge \Psi')) \mid (\ell + 1, \Psi), (m, \Psi') \in \Pi\} \cup \Pi \upharpoonright_{\{\ell\}} \longrightarrow \Pi} \text{gotoF}_{\text{ts}}^{\neq} \quad \frac{}{(\ell, \text{gotoF } \ell) : \Pi \upharpoonright_{\{\ell\}} \longrightarrow \Pi} \text{gotoF}_{\text{ts}}^{\equiv} \\
\frac{\mathbf{0} : \Pi \longrightarrow \Pi \quad \mathbf{0}_{\text{ts}} \quad \frac{sc_0 : \Pi \upharpoonright_{\text{dom}(sc_0)} \longrightarrow \Pi \quad sc_1 : \Pi \upharpoonright_{\text{dom}(sc_1)} \longrightarrow \Pi}{sc_0 \oplus sc_1 : \Pi \longrightarrow \Pi \upharpoonright_{\text{dom}(sc_0) \cup \text{dom}(sc_1)}} \oplus_{\text{ts}}}{\frac{\Pi'_0 \leq \Pi_0 \quad sc : \Pi_0 \longrightarrow \Pi_1 \quad \Pi_1 \leq \Pi'_1}{sc : \Pi'_0 \longrightarrow \Pi'_1} \text{conseq}_{\text{ts}}}
\end{array}$$

Figure 6: Typing rules of SPUSH

$$\begin{array}{ll}
\text{wpt}((\ell, \text{load } x), \Pi') & =_{\text{df}} \{(\ell, \Psi) \mid (\ell + 1, \tau :: \Psi) \in \Pi', \text{int} \leq \tau\} \cup \{(\ell, *) \mid (\ell + 1, *) \in \Pi'\} \cup \Pi' \upharpoonright_{\{\ell\}} \\
\text{wpt}((\ell, \text{store } x), \Pi') & =_{\text{df}} \{(\ell, \text{int} :: \Psi) \mid (\ell + 1, \Psi) \in \Pi'\} \cup \Pi' \upharpoonright_{\{\ell\}} \\
\text{wpt}((\ell, \text{push } n), \Pi') & =_{\text{df}} \{(\ell, \Psi) \mid (\ell + 1, \tau :: \Psi) \in \Pi', \text{int} \leq \tau\} \cup \{(\ell, *) \mid (\ell + 1, *) \in \Pi'\} \cup \Pi' \upharpoonright_{\{\ell\}} \\
\text{wpt}((\ell, \text{add}), \Pi') & =_{\text{df}} \{(\ell, \text{int} :: \text{int} :: \Psi) \mid (\ell + 1, \tau :: \Psi) \in \Pi', \text{int} \leq \tau\} \cup \{(\ell, \text{int} :: \text{int} :: *) \mid (\ell + 1, *) \in \Pi'\} \cup \Pi' \upharpoonright_{\{\ell\}} \\
\text{wpt}((\ell, \text{goto } m), \Pi') & =_{\text{df}} \begin{cases} \{(\ell, \Psi) \mid (m, \Psi) \in \Pi'\} \cup \Pi' \upharpoonright_{\{\ell\}} & \text{if } m \neq \ell \\ \{(\ell, *)\} \cup \Pi' \upharpoonright_{\{\ell\}} & \text{if } m = \ell \end{cases} \\
\text{wpt}((\ell, \text{gotoF } m), \Pi') & =_{\text{df}} \begin{cases} \{(\ell, \text{bool} :: (\Psi \wedge \Psi')) \mid (\ell + 1, \Psi), (m, \Psi') \in \Pi'\} \cup \Pi' \upharpoonright_{\{\ell\}} & \text{if } m \neq \ell \\ \Pi' \upharpoonright_{\{\ell\}} & \text{if } m = \ell \end{cases} \\
\text{wpt}(\mathbf{0}, \Pi') & =_{\text{df}} \Pi' \\
\text{wpt}(sc_0 \oplus sc_1, \Pi') & =_{\text{df}} \bigwedge_{i < \omega} \Pi_i \text{ where} \\
& \quad \Pi_0 =_{\text{df}} \{(\ell, *) \mid \ell \in \text{dom}(sc_0 \oplus sc_1) \cup \text{dom}(\Pi')\} \\
& \quad \Pi_{i+1} =_{\text{df}} S(\Pi_i) \\
& \quad S(\Pi) =_{\text{df}} \text{wpt}(sc_0, \Pi) \upharpoonright_{\text{dom}(sc_0)} \cup \text{wpt}(sc_1, \Pi) \upharpoonright_{\text{dom}(sc_1)} \cup \Pi' \upharpoonright_{\text{dom}(sc_0 \oplus sc_1)}
\end{array}$$

Figure 7: Weakest pretype calculus

Theorem 9 (Preservation of subtypings and reflection of entailments by concretization) (i) $\tau \leq \tau'$ iff $\text{conc}(\tau) \models \text{conc}(\tau')$. (ii) $\Psi \leq \Psi'$ iff $\text{conc}(\Psi) \models \text{conc}(\Psi')$. (iii) $\Pi \leq \Pi'$ iff $\text{conc}(\Pi) \models \text{conc}(\Pi')$.

Preservation holds also of typing.

Theorem 10 (Preservation of typings by concretization) If $sc : \Pi \longrightarrow \Pi'$, then $\{\text{conc}(\Pi)\} sc \{\text{conc}(\Pi')\}$.

We do not get reflection of Hoare triples by concretization, however. Consider, for example, the code $sc =_{\text{df}} (0, \text{push tt}) \oplus ((1, \text{gotoF } 3) \oplus (2, \text{push } 17))$. We have $\text{conc}((0, [])) = pc = 0 \wedge st = [], \text{conc}((3, [\text{int}])) = pc = 3 \wedge \exists z \in \mathbb{Z}. st = [z]$ and can derive $\{pc = 0 \wedge st = []\} sc \{pc = 3 \wedge \exists z \in \mathbb{Z}. st = [z]\}$, while we cannot derive $sc : \{(0, [])\} \longrightarrow \{(3, [\text{int}])\}$. The type system does not discover that the false branch will never be taken. The best posttype we can get for $\{(0, [])\}$ is $\{(3, *)\}$.

We finish the discussion of the type system by remarking that introducing the value type $?$ and the stack type $*$ was not inevitable. But a version without these constructs would only type pieces of code for which the operand stack has a definite depth and value type content for every label through which its evaluations may pass. More generally, there is a design issue here. We could, for example, introduce additional stack types int^* , bool^* for stacks of unspecified length, consisting of integers or booleans only. Yet another design choice would be to define **StackType** $=_{\text{df}} \mathcal{P}_{\text{fin}}(\text{AbsStack})$ instead. Under this discipline, some pieces of code with finitely unbalanced stack usage would receive more precise types, e.g., for the code

```

0 gotoF 3
1 push 17
2 goto 5
3 push tt
4 push ff

```

and pretype $\{(0, [\text{bool}])\}$, the best posttype we can get in our type system is $\{(5, ? :: *)\}$, but the alternative posttype $\{(5, [\text{int}], [\text{bool}, \text{bool}])\}$ is clearly more informative. On the other hand, a piece of code with infinite variation such as

```

0 load x
1 geq0
2 gotoF 8
3 push 17
4 load x
5 dec
6 store x
7 goto 0

```

and the pretype $\{(0, [])\}$ have $\{(8, *)\}$ as the strongest posttype in our type system but no posttype under the alternative approach.

7 Compilation

We shall now define a compilation function from WHILE programs to SPUSH pieces of code.

The compilation function is standard except that it produces structured code (we have chosen structures that are the most convenient for us) and is compositional. The compilation rules are given in Figure 8. The compilation relation for expressions $-\searrow- \subseteq \text{Label} \times (\text{AExp} \cup \text{BExp}) \times \text{SCode} \times \text{Label}$ relates a label and a WHILE expression to a piece of

code and another label. The relation for statements $-\searrow- \subseteq \text{Label} \times \text{Stm} \times \text{SCode} \times \text{Label}$ is similar. The idea is that the domain of a compiled expression or statement will be a left-closed, right-open interval. (It may be an empty interval, which does not even contain its beginning-point.) The first label is the beginning-point of the interval and the second is the corresponding end-point.

Compilation is total and deterministic, i.e., a function, and produces a piece of code whose support is exactly the desired interval.

Lemma 4 (Totality and determinacy of compilation) (i) For any ℓ, e , there exist sc, ℓ' such that $e \searrow_{\ell'} sc$. If $e \searrow_{\ell_0} sc_0$ and $e \searrow_{\ell_1} sc_1$, then $sc_0 = sc_1$ and $\ell_0 = \ell_1$. (ii) For any ℓ, s , there exist sc, ℓ' such that $s \searrow_{\ell'} sc$. If $s \searrow_{\ell_0} sc_0$ and $s \searrow_{\ell_1} sc_1$, then $sc_0 = sc_1$ and $\ell_0 = \ell_1$.

Lemma 5 (Domain of compiled code) (i) If $e \searrow_{\ell'} sc$, then $\text{dom}(sc) = [\ell, \ell')$. (ii) If $s \searrow_{\ell'} sc$, then $\text{dom}(sc) = [\ell, \ell')$.

That compilation does not alter the meaning of an expression or statement is demonstrated by the facts that WHILE evaluations are preserved and SPUSH evaluations are reflected by it. We must however take into account the fact a compiled WHILE expression or statement is intended to be entered from its beginning-point.

Theorem 11 (Preservation of evaluations) (i) If $e \searrow_{\ell'} sc$, then $(\ell, \zeta, \sigma) \succ_{sc} (\ell', \llbracket e \rrbracket \sigma :: \zeta, \sigma)$. (ii) If $s \searrow_{\ell'} sc$ and $\sigma \succ_s \sigma'$, then $(\ell, \zeta, \sigma) \succ_{sc} (\ell', \zeta, \sigma')$.

Proof. By induction on the structure of e or the derivation of $\sigma \succ_s \sigma'$. \square

Theorem 12 (Reflection of evaluations) (i) If $e \searrow_{\ell'} sc$ and $(\ell, \zeta, \sigma) \succ_{sc} (\ell'', \zeta', \sigma')$, then $\ell'' = \ell'$, $\zeta' = \llbracket e \rrbracket \sigma :: \zeta$ and $\sigma' = \sigma$. (ii) If $s \searrow_{\ell'} sc$ and $(\ell, \zeta, \sigma) \succ_{sc} (\ell'', \zeta', \sigma')$, then $\ell'' = \ell'$, $\zeta' = \zeta$ and $\sigma \succ_s \sigma'$.

Proof. By induction on the structure of sc and subordinate induction on the derivation of $(\ell, \zeta, \sigma) \succ_{sc} (\ell'', \zeta', \sigma')$. \square

It is easy to show that compilation preserves derivable WHILE Hoare triples (in a suitable format that takes into account that a WHILE statement proof assumes entry from the beginning-point and guarantees exit to the end-point). But one can also give a constructive proof: a proof by defining a compositional translation of WHILE program proofs to SPUSH program proofs, i.e., a proof compilation function.

Theorem 13 (Preservation of derivable Hoare triples) (i) If $e \searrow_{\ell'} sc$ and P is a WHILE assertion, then $\{pc = \ell \wedge st = \zeta \wedge P\} sc \{pc = \ell' \wedge st = e :: \zeta \wedge P\}$. (ii) If $s \searrow_{\ell'} sc$ and $\{P\} s \{Q\}$, then $\{pc = \ell \wedge st = \zeta \wedge P\} sc \{pc = \ell' \wedge st = \zeta \wedge Q\}$.

Proof. [Non-constructive proof] Straightforward from soundness of the Hoare logic of WHILE, reflection of evaluations by compilation and completeness of the Hoare logic of SPUSH. \square

Proof. [Constructive proof: Preservation Hoare triple derivations] By induction on the structure of e or the derivation of $\{P\} s \{Q\}$. \square

Reflection of derivable SPUSH Hoare triples by compilation can also be shown. As with preservation,

proving reflection non-constructively is a straightforward matter, but again there is also a constructive proof. Given a WHILE program, we can “decompile” the correctness proof of its compiled form (a SPUSH piece of code) into a correctness proof of the WHILE program. For the constructive proof, we have to use the fact that proofs of SPUSH programs admit a certain normal form.

Theorem 14 (Reflection of derivable Hoare triples) (i) If $e \stackrel{\ell}{\searrow}_{\ell'} sc$ and $\{P\} sc \{Q\}$, then $P[pc, st \mapsto \ell, \zeta] \models Q[pc, st \mapsto \ell', e :: \zeta]$. (ii) If $s \stackrel{\ell}{\searrow}_{\ell'} sc$ and $\{P\} sc \{Q\}$, then $\{P[pc, st \mapsto \ell, \zeta]\} s \{Q[pc, st \mapsto \ell', \zeta]\}$.

Proof. [Non-constructive proof] From soundness of the Hoare logic of SPUSH, preservation of evaluations by compilation and completeness of the Hoare logic of WHILE. \square

Proof. [Constructive proof: Reflection of Hoare triple derivations] By induction on the structure of sc , using the fact that any Hoare logic derivation can be normalized to a form where proper inferences come in strict alternation with consequence inferences. (Normalization is trivial: a sequence of several consecutive consequence inferences can be compressed into one and a missing consequence inference can be expanded into a trivial consequence inference.) \square

For the type system of SPUSH, we can prove the following analogous results. The first of them means that we can strengthen our compilation function to accompany the SPUSH code it produces from a WHILE-program with a typing derivation.

Theorem 15 (Typing from compilation) (i) If $a \stackrel{\ell}{\searrow}_{\ell'} sc$, then $sc : \{(\ell, \psi)\} \longrightarrow \{(\ell', \text{int} :: \psi)\}$. If $b \stackrel{\ell}{\searrow}_{\ell'} sc$, then $sc : \{(\ell, \psi)\} \longrightarrow \{(\ell', \text{bool} :: \psi)\}$. (ii) If $s \stackrel{\ell}{\searrow}_{\ell'} sc$, then $sc : \{(\ell, \psi)\} \longrightarrow \{(\ell', \psi)\}$.

Theorem 16 (Possible typings) (i) If $a \stackrel{\ell}{\searrow}_{\ell'} sc$ and $sc : \{(\ell, \psi)\} \longrightarrow \Pi$, then $\{(\ell', \text{int} :: \psi)\} \leq \Pi$. If $b \stackrel{\ell}{\searrow}_{\ell'} sc$ and $sc : \{(\ell, \psi)\} \longrightarrow \Pi$, then $\{(\ell', \text{bool} :: \psi)\} \leq \Pi$. (ii) If $s \stackrel{\ell}{\searrow}_{\ell'} sc$ and $sc : \{(\ell, \psi)\} \longrightarrow \Pi$, then $\{(\ell', \psi)\} \leq \Pi$.

8 Abstract natural semantics and type system for secure information flow

Besides stack-error freedom, it is possible to devise systems to present dataflow analyses. Here we sketch an abstract natural semantics and type system for secure information flow analysis. For space reasons, this description is very Spartan, but it should make sense to anyone familiar with secure information flow analyses for high-level imperative languages à la Denning & Denning (1977).

Central for both the abstract natural semantics and type system for secure information flow is a distributive lattice $(D, \leq, \wedge, \vee, L, H)$ of security levels for information flowing in the program (stack positions, variables and the pc). Abstract states are quadruples of a label $\ell \in \mathbf{Label}$, a security level $d \in D$ for the current pc value, and an abstract stack and an abstract store: $\mathbf{AbsState} =_{\text{df}} \mathbf{Label} \times D \times \mathbf{AbsStack} \times \mathbf{AbsStore}$. An abstract stack $\psi \in \mathbf{AbsStack}$ is a list over D corresponding to the security levels of the stack positions in the imaginable concrete state, an abstract store $\Sigma \in \mathbf{AbsStore}$ similarly records the security levels of the variables in the imaginable concrete state: $\mathbf{AbsStack} =_{\text{df}} D^*$, $\mathbf{AbsStore} =_{\text{df}} \mathbf{Var} \rightarrow D$.

The abstract semantics is sensitive to stack underflow, but ignores the possibility in the concrete semantics of operand type errors (confuses them with normal terminations). An important concept in the semantics is the notion of a single-exit piece of code: this is a piece of code sc for which one can single out a label ℓ^* such that every target label (successor label or jump target, depending on the kind of the instruction) of any labelled instruction in sc is in $\text{dom}(sc) \cup \{\ell^*\}$; we call ℓ^* the exit-point of sc . Single-exit unions are analogous to single-exit compound blocks in control-flow diagrams; compare these to if- or while-statements of WHILE, which are single-exit as all WHILE statements but special in that their control-flow diagrams enclose inner branchings. The rules of the semantics are presented in Figure 9. Because of the single-exit union rule, this abstract semantics is not neutral wrt. the structure imposed on an unstructured PUSH piece of code: depending on how small or large the smallest single-exit union enclosing a branching instruction gotoF is in the structure imposed on a code, a given initial security state can take us to a more or less optimistic terminal security state.

In the type system, the state types $\Pi \in \mathbf{StateType}$ are quadruples of a label, security level (for the pc), stack type and abstract store (there is no difference between an abstract store and a store type!): $\mathbf{StateType} =_{\text{df}} \mathcal{P}_{\text{fin}}(\mathbf{Label} \times D \times \mathbf{StackType} \times \mathbf{AbsStore})$ where no label may occur twice in a wellformed statetype. Stack types $\Psi \in \mathbf{StackType}$ are defined by the grammar

$$\Psi ::= \perp \mid [] \mid d :: \Psi \mid *$$

Stack types have a set-theoretic meaning defined as follows:

$$\begin{aligned} (\perp) &=_{\text{df}} \emptyset \\ ([]) &=_{\text{df}} \{\} \\ (d :: \Psi) &=_{\text{df}} \{d' :: \psi \mid d' \leq d, \psi \in (\Psi)\} \\ (*) &=_{\text{df}} D^* \end{aligned}$$

The type system is derived from the abstract natural semantics—the typing rules are in the weakest pretype style—and attests stack-underflow-error free information flow security. The type system may type pieces of code that can terminate abnormally due to wrong operand types. The subtyping rules are in Figure 10 while the typing rules appear in Figure 11. ($\psi \vee d$ denotes the list resulting from joining d to every element of ψ ; $\bigvee ds$ denotes the join of all elements of ds ; $\bigwedge \Psi$ denotes the meet of all elements of Ψ .)

9 Related work

In the young days of Hoare logic, quite some attention was paid to general and restricted jumps in high-level languages. Hoare’s original logic (1969) was for WHILE and characteristic to the various proposals that were made thereafter is that they deal with extensions of WHILE or a similar language. The logics of Clint & Hoare (1972), Kowaltowski (1977) and de Bruin (1981) use conditional Hoare triples (so the proof system is a natural deduction system) to be able to make and use assumptions about label invariants. In the solution of Arbib & Alagić (1979), Hoare triples have multiple postconditions, reflecting the fact that statements involving gotos are multiple-exit.

Logics for low-level languages without phrase structure have only become a topic of active research with the advent of PCC, with Java bytecode and .NET CIL being the main motivators. (There is

$$\begin{array}{c}
\frac{}{n^{\ell} \searrow_{\ell+1} (\ell, \text{push } n)} \quad \frac{}{x^{\ell} \searrow_{\ell+1} (\ell, \text{load } x)} \quad \frac{a_0^{\ell} \searrow_{\ell''} sc_0 \quad a_1^{\ell''} \searrow_{\ell'} sc_1}{a_0 + a_1^{\ell} \searrow_{\ell'+1} (sc_0 \oplus sc_1) \oplus \text{add}} \quad \frac{b_0^{\ell} \searrow_{\ell''} sc_0 \quad b_1^{\ell''} \searrow_{\ell'} sc_1}{b_0 = b_1^{\ell} \searrow_{\ell'+1} (sc_0 \oplus sc_1) \oplus \text{eq}} \\
\frac{a^{\ell} \searrow_{\ell'} sc}{x := a^{\ell} \searrow_{\ell'+1} (sc \oplus \text{store } x)} \quad \frac{}{\text{skip}^{\ell} \searrow_{\ell} 0} \quad \frac{s_0^{\ell} \searrow_{\ell''} sc_0 \quad s_1^{\ell''} \searrow_{\ell'} sc_1}{s_0; s_1^{\ell} \searrow_{\ell'} sc_0 \oplus sc_1} \\
\frac{b^{\ell} \searrow_{\ell''} sc_b \quad s_t^{\ell''+1} \searrow_{\ell'''} sc_t \quad s_f^{\ell'''+1} \searrow_{\ell'} sc_f}{\text{if } b \text{ then } s_t \text{ else } s_f^{\ell} \searrow_{\ell'} (sc_b \oplus (\ell'', \text{gotoF } \ell'''+1)) \oplus ((sc_t \oplus (\ell''', \text{goto } \ell')) \oplus sc_f)} \\
\frac{b^{\ell} \searrow_{\ell''} sc_b \quad s^{\ell''+1} \searrow_{\ell'} sc}{\text{while } b \text{ do } s^{\ell} \searrow_{\ell'+1} (sc_b \oplus (\ell'', \text{gotoF } \ell'+1)) \oplus (sc \oplus (\ell', \text{goto } \ell))}
\end{array}$$

Figure 8: Rules of compilation from WHILE to SPUSH

$$\begin{array}{c}
\frac{}{(\ell, d, \psi, \Sigma) \succ (\ell, \text{load } x) \rightarrow (\ell + 1, d, \Sigma(x) \vee d :: \psi, \Sigma)} \text{load}_{\text{ans}} \\
\frac{}{(\ell, d, d' :: \psi, \Sigma) \succ (\ell, \text{store } x) \rightarrow (\ell + 1, d, \psi, \Sigma[x \mapsto d' \vee d])} \text{store}_{\text{ans}} \quad \frac{\forall d \in D, \psi' \in D^*. \psi \neq d :: \psi'}{(\ell, d, \psi, \Sigma) \succ (\ell, \text{store } x) \rightarrow (\ell, d, \psi, \Sigma)} \text{store}_{\text{ans}}^{ab} \\
\frac{}{(\ell, d, \psi, \Sigma) \succ (\ell, \text{push } n) \rightarrow (\ell + 1, d, d :: \psi, \Sigma)} \text{push}_{\text{ans}} \\
\frac{}{(\ell, d, d_0 :: d_1 :: \psi, \Sigma) \succ (\ell, \text{add}) \rightarrow (\ell + 1, d, d_0 \vee d_1 \vee d :: \psi, \Sigma)} \text{add}_{\text{ans}} \quad \frac{\forall d_0, d_1 \in D, \psi' \in D^*. \psi \neq d_0 :: d_1 :: \psi'}{(\ell, d, \psi, \Sigma) \succ (\ell, \text{add}) \rightarrow (\ell, d, \psi, \Sigma)} \text{add}_{\text{ans}}^{ab} \\
\frac{m \neq \ell}{(\ell, d, \psi, \Sigma) \succ (\ell, \text{goto } m) \rightarrow (m, d, \psi, \Sigma)} \text{goto}_{\text{ans}}^{\neq} \\
\frac{m \neq \ell}{(\ell, d, d' :: \psi, \Sigma) \succ (\ell, \text{gotoF } m) \rightarrow (\ell + 1, d \vee d', \psi \vee (d \vee d'), \Sigma)} \text{gotoF}_{\text{ans}}^{\neq \text{tt}} \\
\frac{m \neq \ell}{(\ell, d, d' :: \psi, \Sigma) \succ (\ell, \text{gotoF } m) \rightarrow (m, d \vee d', \psi \vee (d \vee d'), \Sigma)} \text{gotoF}_{\text{ans}}^{\neq \text{ff}} \\
\frac{m \neq l \quad \forall d \in D, \psi' \in D^*. \psi \neq d :: \psi'}{(\ell, d, \psi, \Sigma) \succ (\ell, \text{gotoF } m) \rightarrow (\ell, d, \psi, \Sigma)} \text{gotoF}_{\text{ans}}^{\neq ab} \\
\frac{ds \in D^*}{(\ell, d, ds \dashv d' :: \psi, \Sigma) \succ (\ell, \text{gotoF } \ell) \rightarrow (\ell + 1, d, \psi \vee (d \vee \bigvee ds \vee d'), \Sigma)} \text{gotoF}_{\text{ans}}^{\neq} \quad \frac{ds \in D^*}{(\ell, d, ds, \Sigma) \succ (\ell, \text{gotoF } \ell) \rightarrow (\ell + 1, d, [], \Sigma)} \text{gotoF}_{\text{ans}}^{\neq ab} \\
\frac{\ell \in \text{dom}(sc_i) \quad (\ell, d, \psi, \Sigma) \succ sc_i \rightarrow (\ell'', d'', \psi'', \Sigma'') \quad (\ell'', d'', \psi'', \Sigma'') \succ sc_0 \oplus sc_1 \rightarrow (\ell', d', \psi', \Sigma') \quad sc_0 \oplus sc_1 \text{ multiple-exit}}{(\ell, d, \psi, \Sigma) \succ sc_0 \oplus sc_1 \rightarrow (\ell', d', \psi', \Sigma')} \oplus_{\text{ans}} \\
\frac{\ell \in \text{dom}(sc_i) \quad (\ell, d, \psi, \Sigma) \succ sc_i \rightarrow (\ell'', d'', \psi'', \Sigma'') \quad (\ell'', d'', \psi'', \Sigma'') \succ sc_0 \oplus sc_1 \rightarrow (\ell', d', \psi', \Sigma') \quad sc_0 \oplus sc_1 \text{ single-exit}}{(\ell, d, \psi, \Sigma) \succ sc_0 \oplus sc_1 \rightarrow (\ell', d', \psi', \Sigma')} \oplus_{\text{ans}} \\
\frac{\ell \in \text{dom}(sc_i) \quad (\ell, d, \psi, \Sigma) \succ sc_i \rightarrow (\ell'', d, \psi'', \Sigma'')}{(\ell, d, \psi, \Sigma) \succ sc_0 \oplus sc_1 \rightarrow (\ell', d', \psi', \Sigma')} \oplus_{\text{ans}}^{abn} \\
\frac{\ell \in \text{dom}(sc_i) \quad (\ell, d, \psi, \Sigma) \succ sc_i \rightarrow (\ell'', d, \psi'', \Sigma'') \quad (\ell'', d'', \psi'', \Sigma'') \succ sc_0 \oplus sc_1 \rightarrow (\ell', d', \psi', \Sigma')}{(\ell, d, \psi, \Sigma) \succ sc_0 \oplus sc_1 \rightarrow (\ell', d', \psi', \Sigma')} \oplus_{\text{ans}} \\
\frac{\ell \notin \text{dom}(sc)}{(\ell, d, \psi, \Sigma) \succ sc \rightarrow (\ell, d, \psi, \Sigma)} \text{ood}_{\text{ans}}
\end{array}$$

Figure 9: Abstract natural semantics rules of SPUSH for secure information flow

$$\begin{array}{c}
\frac{}{\Psi \leq \Psi} \quad \frac{\Psi \leq \Psi'' \quad \Psi'' \leq \Psi'}{\Psi \leq \Psi'} \quad \frac{}{\tau :: \perp \leq \perp} \quad \frac{}{\perp \leq \Psi} \quad \frac{}{\Psi \leq *} \quad \frac{\tau \leq \tau' \quad \Psi \leq \Psi'}{\tau :: \Psi \leq \tau' :: \Psi'} \\
\frac{\forall x. \Sigma(x) \leq \Sigma'(x)}{\Sigma \leq \Sigma'} \\
\frac{\forall \ell, d, \Psi, \Sigma. (\ell, d, \Psi, \Sigma) \in \Pi \supset \Psi = \perp \vee \exists \Psi'. (\ell, d', \Psi', \Sigma') \in \Pi' \wedge d \leq d' \wedge \Psi \leq \Psi' \wedge \Sigma \leq \Sigma'}{\Pi \leq \Pi'}
\end{array}$$

Figure 10: Subtyping rules of SPUSH for secure information flow

$$\begin{array}{c}
\frac{(\ell, \text{load } x) : \{(\ell, d' \wedge d, \Psi, \Sigma[x \mapsto d' \wedge \Sigma(x)]) \mid (\ell + 1, d, d' :: \Psi, \Sigma) \in \Pi\} \cup \{(\ell, d, *, \Sigma) \mid (\ell + 1, d, *, \Sigma) \in \Pi\} \cup \Pi_{\{\ell\}} \longrightarrow \Pi}{\text{load}_{\text{ts}}} \\
\frac{(\ell, \text{store } x) : \{(\ell, \Sigma(x) \wedge d, \Sigma(x) :: \Psi, \Sigma) \mid (\ell + 1, d, \Psi, \Sigma) \in \Pi\} \cup \Pi_{\{\ell\}} \longrightarrow \Pi}{\text{store}_{\text{ts}}} \\
\frac{(\ell, \text{push } n) : \{(\ell, d' \wedge d, \Psi, \Sigma) \mid (\ell + 1, d, d' :: \Psi, \Sigma) \in \Pi\} \cup \{(\ell, d, *, \Sigma) \mid (\ell + 1, d, *, \Sigma) \in \Pi\} \cup \Pi_{\{\ell\}} \longrightarrow \Pi}{\text{push}_{\text{ts}}} \\
\frac{(\ell, \text{add}) : \{(\ell, d' \wedge d, d' :: d' :: \Psi, \Sigma) \mid (\ell + 1, d, d' :: \Psi, \Sigma) \in \Pi\} \cup \{(\ell, d, H :: H :: *, \Sigma) \mid (\ell + 1, d, *, \Sigma) \in \Pi\} \cup \Pi_{\{\ell\}} \longrightarrow \Pi}{\text{add}_{\text{ts}}} \\
\vdots \\
\frac{m \neq \ell}{(\ell, \text{goto } m) : \{(\ell, d, \Psi, \Sigma) \mid (m, d, \Psi, \Sigma) \in \Pi\} \cup \Pi_{\{\ell\}} \longrightarrow \Pi} \text{goto}_{\text{ts}}^{\neq} \quad \frac{(\ell, \text{goto } \ell) : \{(\ell, H, *, \text{const } H)\} \cup \Pi_{\{\ell\}} \longrightarrow \Pi}{\text{goto}_{\text{ts}}^{\equiv}} \\
\frac{m \neq \ell}{(\ell, \text{gotoF } m) : \{(\ell, d_0, d_0 :: (\Psi \wedge \Psi'), \Sigma \wedge \Sigma') \mid (\ell + 1, d, \Psi, \Sigma), (m, d', \Psi', \Sigma') \in \Pi\} \cup \Pi_{\{\ell\}} \longrightarrow \Pi} \text{gotoF}_{\text{ts}}^{\neq} \\
\text{where } d_0 = d \wedge \Psi \wedge d' \wedge \Psi' \\
\frac{(\ell, \text{gotoF } \ell) : \Pi_{\{\ell\}} \longrightarrow \Pi}{\text{gotoF}_{\text{ts}}^{\equiv}} \\
\frac{\mathbf{0} : \Pi \longrightarrow \Pi \quad \mathbf{0}_{\text{ts}} \quad \frac{sc_0 : \Pi_{\text{dom}(sc_0)} \longrightarrow \Pi \quad sc_1 : \Pi_{\text{dom}(sc_1)} \longrightarrow \Pi \quad sc_0 \oplus sc_1 \text{ multiple-exit}}{sc_0 \oplus sc_1 : \Pi \longrightarrow \Pi_{\text{dom}(sc_0 \oplus sc_1)}} \oplus_{\text{ts}}}{\frac{sc_0 : \Pi_{\text{dom}(sc_0)} \longrightarrow \Pi \quad sc_1 : \Pi_{\text{dom}(sc_1)} \longrightarrow \Pi \quad sc_0 \oplus sc_1 \text{ single-exit with } \ell^* \text{ the exit-point}}{\Pi' \leq \Pi \quad \forall (\ell', d', \Psi', \Sigma') \in \Pi'_{\text{dom}(sc_0 \oplus sc_1)} \cdot d' \leq d^*} \oplus_{\text{ts}} \\
\frac{sc_0 \oplus sc_1 : \Pi' \longrightarrow \{(\ell^*, d^*, \Psi, \Sigma) \mid (\ell^*, d, \Psi, \Sigma) \in \Pi\} \cup \Pi_{\text{dom}(sc_0 \oplus sc_1) \cup \ell^*}}{\frac{\Pi'_0 \leq \Pi_0 \quad sc : \Pi_0 \longrightarrow \Pi_1 \quad \Pi_1 \leq \Pi'_1}{sc : \Pi'_0 \longrightarrow \Pi'_1} \text{conseq}_{\text{ts}}}
\end{array}$$

Figure 11: Typing rules of SPUSH for secure information flow

one very notable exception though: Floyd’s logic of control-flow graphs (1967).) The logic of Quigley (2003) for Java bytecode is based on decompilation, so it applies to pieces of code in the image of a fixed compiler. Benton’s (2004) logic for a PUSH-like stack-based language involves global contexts of label invariants as de Bruin’s logic. Bannwart & Müller’s (2005) logic extends it to a subset of Java bytecode, with both an operand stack and a call stack, leaving out exceptions.

The work of Huisman & Jacobs (2000) describes a Hoare logic for Java, incl. exceptions. Schröder & Mossakowski (2003) and Schröder & Mossakowski (2004) discuss a systematic method for designing Hoare logics for languages with monadic side-effects, in particular, exceptions.

The present paper builds upon our recent work (Saabas & Uustalu 2005), where a compositional natural semantics and Hoare logic based on the implicit finite unions structure are introduced for a simple low-level language GOTO with expressions. The same structure is used by Tan & Appel (2005) and Tan (2005), who study the same language. But instead of introducing a natural semantics for the structured version of the language, they proceed from a small-step ideology. As a result, they arrive at a continuation-style Hoare logic explainable by Appel & McAllester’s ‘indexed model’ (2001), with a rather convoluted interpretation of Hoare triples involving explicit fixedpoint approximations. Apparently unaware of Tan & Appel’s work, Benton (2005) defines a similar logic for a stack-based language with a typing component ensuring that the stack is used safely.

Presenting program analyses especially for functional languages in terms of type systems is a popular topic. Naik & Palsberg (2005) have related model checking and type systems for WHILE. A different general method to produce type systems for WHILE equivalent to dataflow analyses is described in the work of Laud et al. (2005). As for low-level languages, Morrisett et al. (1999) imposed a memory-safety type system on an assembly language and Morrisett et al. (2003) extended it for a stack-based language. Stata

& Abadi (1999) were the first to describe the Java bytecode verifier as a type system. All such systems are again non-compositional and make use of global contexts of label invariants (where an invariant is associated to every instruction or every basic block of the global piece of code), except for the type system component in Benton’s (2005) logic.

A static analysis for secure information flow was first described by Denning & Denning (1977). They worked with a WHILE-like language, but also proposed a way to handle languages with goto instructions. Kobayashi & Kirane (2002) and Barthe & Rezk (2005) use the same idea of control dependence regions in type systems equivalent to secure information flow analyses for sequential Java bytecode.

10 Conclusions and future work

We have shown that our original idea of structuring low-level languages with finite unions to obtain compositional natural semantics and Hoare logics (Saabas & Uustalu 2005) applies to stack-based languages just as well as to languages with store only. The possibility of abnormal terminations can be handled well, and the semantics and logics obtained are neat and enjoy every desirable metatheoretic property. Moreover, in the richer setting of a stack-based language, it is meaningful to consider abstracted semantics and type systems too. Notably, one can obtain a type system to attest safe stack usage, but also produce type systems for other purposes. We have demonstrated this on the example of a type system equivalent to a secure information flow analysis.

We plan to apply the method also to a language with both an operand stack and call stack, cf. (Benton 2005). We will also validate the practicality of our approach in realistic code and proof / type-derivation presentation (certified code formats). For proof compilation and generation of type derivations the approach seems just ideal and we intend to implement a proof compiler / type derivation generator.

On the theoretical side, we intend to carry out a detailed comparison of our natural-semantics based

direct approach to the continuation-style approach of Tan & Appel (2005) and Benton (2005) that relies on Appel & McAllester's (2001) 'indexed model'.

Acknowledgements This work was partially supported by the Estonian Science Foundation under grant No. 5567 and by the EU FP6 IST project MOBIUS.

References

- Appel, A. & McAllester, D. (2001), An indexed model of recursive types for foundational proof-carrying code, *ACM Trans. on Program. Lang. and Syst.* **23**(5), pp. 657–683.
- Arbib, M. A. & Alagić, S. (1979), Proof rules for **gotos**, *Acta Inform.* **11**, pp. 139–148.
- Bannwart, F. & Müller, P. (2005), A program logic for bytecode, to appear in 'Proc. of 1st Wksh. on Bytecode Semantics, Verification, Analysis and Transformation, BYTECODE 2005 (Edinburgh, UK, 9 Apr. 2005)', *Electr. Notes in Theor. Comput. Sci.*, Elsevier.
- Barthe, G. & Rezk, T. (2005), Non-interference for a JVM-like language, in G. Morrisett, M. Fähndrich, eds., 'Proc. of 2005 ACM SIGPLAN Int. Wksh. on Types in Languages Design and Implementation, TLDI '05 (Long Beach, CA, Jan. 2005)', ACM Press, pp. 103–112.
- Benton, N. (2004), A typed logic for stacks and jumps, draft.
- Benton, N. (2005), A typed, compositional logic for a stack-based abstract machine, Tech. report MSR-TR-2005-84, Microsoft Research, Cambridge; shorter version to appear in K. Yi, ed., 'Proc. of 3rd Asian Symp. on Programming Languages and Systems, APLAS 2005 (Tsukuba, Nov. 2005)', *Lect. Notes in Comput. Sci.* **3780**, Springer-Verlag.
- Clint, M. & Hoare, C. A. R. (1972), Program proving: Jumps and functions, *Acta Inform.* **1**, pp. 214–224.
- Cook, S. A. (1978), Soundness and completeness of an axiom system for verification, *SIAM J. of Comput.* **7**, pp. 70–90.
- de Bruin, A. (1981), Goto statements: Semantics and deduction systems, *Acta Inform.* **15**, pp. 385–424.
- Denning, D. E. & Denning, P. J. (1977), Certification of programs for secure information flow, *Commun. of ACM* **20**, pp. 504–513.
- Floyd, R. W. (1967), Assigning meanings to programs, in J. T. Schwartz, ed., 'Mathematical Aspects of Computer Science', *Proc. of Symp. in Appl. Math.* **19**, AMS, pp. 19–33.
- Hoare, C. A. R. (1969), An axiomatic basis for computer programming, *Commun. of ACM* **12**, pp. 576–583.
- Huisman, M. & Jacobs, B. (2000), Java program verification via a Hoare Logic with abrupt termination, in T. Maibaum, ed., 'Proc. of 3rd Int. Conf. on Fundamental Approaches to Software Engineering, FASE 2000 (Berlin, March/Apr. 2000)', *Lect. Notes in Comput. Sci.* **1783**, Springer-Verlag, pp. 284–303.
- Kobayashi, N. & Kirane, K. (2002), Type-based information analysis for low-level languages, in 'Proc. of 3rd Asian Wksh. on Programming Languages and Systems, APLAS'02 (Shanghai, Nov./Dec. 2002)', Shanghai Jiao Tong University, pp. 302–316.
- Kowaltowski, T. (1977), Axiomatic approach to side effects and general jumps, *Acta Inform.* **7**, pp. 357–360.
- Laud, P., Uustalu, T. & Vene, V. (2005), Type systems equivalent to dataflow analyses for imperative languages, in M. Hofmann & H.-W. Loidl, eds., 'Proc. of 3rd APPSEM II Wksh., APPSEM '05 (Frauenchiemsee, Sept. 2005)', 12 pp., Ludwig-Maximilians-Univ. München.
- Morrisett, J. G., Walker, D., Crary, K. & Glew, N. (1999), From system F to typed assembly language, *ACM Trans. on Program. Lang. and Syst.* **21**(3), pp. 527–568.
- Morrisett, J. G., Crary, K., Glew, N., & Walker, D. (2002), Stack-based typed assembly language, *J. of Funct. Program.* **12**(1), pp. 3–88. Correction, *ibid.* **13**(5) (2003), pp. 957–959.
- Naik, M. & Palsberg, J. (2005), in S. Sagiv, ed., 'Proc. of 14th European Symp. on Programming, ESOP 2005 (Edinburgh, Apr. 2005)', *Lect. Notes in Comput. Sci.* **3444**, Springer-Verlag, pp. 374–388.
- Quigley, C. L. (2003), A programming logic for Java bytecode programs, in D. A. Basin & B. Wolff, eds., 'Proc. of 16th Int. Conf. on Theorem Proving in Higher-Order Logics, TPHOLs 2003 (Rome, Italy, 8–12 Sept. 2003)', *Lect. Notes in Comput. Sci.* **2758**, Springer-Verlag, pp. 41–54.
- Saabas, A. & Uustalu, T. (2005), A compositional natural semantics and Hoare logic for low-level languages, to appear in P. Mosses & I. Ulidowski, eds., 'Proc. of 2nd Wksh. on Structured Operational Semantics, SOS 2005 (Lisbon, July 2005)', *Electr. Notes in Theor. Comput. Sci.*, Elsevier.
- Schröder, L. & Mossakowski, T. (2003), Monad-independent Hoare logic in HASCAL, in M. Pezzè, ed., 'Proc. of 6th Int. Conf. on Fundamental Approaches to Software Engineering, FASE 2003 (Warsaw, Apr. 2003)', *Lect. Notes in Comput. Sci.* **2621**, Springer-Verlag, pp. 261–277.
- Schröder, L. & Mossakowski, T. (2004), Generic exception handling and the Java monad, in C. Ratnay, S. Maharaj & C. Shankland, eds., 'Proc. of 10th Int. Conf. on Algebraic Methodology and Software Technology, AMAST 2004 (Stirling, July 2004)', *Lect. Notes in Comput. Sci.* **3116**, Springer-Verlag, pp. 443–459.
- Stata, R. & Abadi, M. (1999), A type system for Java bytecode subroutines, *ACM Trans. on Program. Lang. and Syst.* **21**(1), pp. 90–137.
- Tan, G. & Appel, A. W. (2005), A compositional logic for control flow, manuscript.
- Tan, G. (2005), A compositional logic for control flow and its application for proof-carrying code, PhD thesis, Princeton Univ.

Mechanically Verifying Correctness of CPS Compilation

Ye Henry Tian

School of Computer Science
McGill University,
Montreal, Quebec H3A 2A7, Canada,
Email: ytian8@cs.mcgill.ca

Abstract

In this paper, we study the formalization of one-pass call-by-value CPS compilation using higher-order abstract syntax. In particular, we verify mechanically that the source program and the CPS-transformed program have the same observable behavior. A key advantage of this approach is that it avoids any administrative redexes thereby simplifying the proofs about CPS-translations. The CPS translation together with its correctness proof is implemented and mechanically verified in the logical framework Twelf.

Keywords: Program transformation, correctness proofs, higher-order abstract syntax, logical framework

1 Introduction

Compilation is a program transformation from a source to a target language, each with a well-defined syntax and semantics. The problem is then to prove that the source and target program have the same observable behavior at execution time. Establishing the correctness of a compiler has been important from the beginning of compiler studies. However this is often found difficult since the semantics of a realistic programming language is often difficult to formalize and the execution models of source and target languages are often very different. Further more a (minor) change to the compiler (e.g. implementation of a new optimization strategy) which is likely to occur during the development cycle of a compiler, may require a renewed effort to reconstruct the correctness proof of the compiler. Mechanical verification can increase the confidence in the correctness proof, since they are often tedious and it is easy to make mistakes, especially if a language still evolves.

Continuation-passing style (CPS) is a program notation that makes every aspect of control flow and data flow explicit (Appel 1992). The original work of CPS study due to Plotkin goes back to the mid-70's (Plotkin 1975). In his work, he provided the first formalization of the CPS transformation which is first-order. However, this gives rise to annoying *administrative redexes* which are solely due to the CPS transformation and not corresponding to an actual reduction step in the original program (Danvy & Filinski 1992, Danvy 1991). These redexes are annoying because they interfere both with proving the

correctness and properties of a CPS transformation (Plotkin 1975, Danvy & Nielsen 2001) and with using it in a compiler (Guy L. Steele 1978).

Different studies of different versions of CPS transformations have been done since Plotkin's original work. To simplify the correctness proof of CPS transformation, and to simplify the reasoning of proving properties (CPS transformations preserve types) of CPS-transformed programs, O. Danvy and L. R. Nielsen presented a new first-order one-pass CPS transformation, which is compositional (Danvy & Nielsen 2002). They later presented a higher-order colon translation (Danvy & Nielsen 2001), which links higher-order CPS transformation with Plotkin's original proof technique. They extended Plotkin's original first-order colon translation to a higher-order version in order to prove the correctness of the higher-order CPS transformation. However, they still need a colon translation to handle administrative redexes. Olivier Danvy, Belmina Dzafic and Frank Pfenning showed their approach of formalizing a higher-order CPS transformation using logical relations in order to prove a syntactic property of the CPS program (Danvy, Dzafic & Pfenning 1999). They proved the proper occurrence of the formal parameters of the continuations. They formalized their proofs in Elf, but they did not show how to formalize the correctness proof of the CPS transformation. Hongwei Xi and Carsten Schürmann showed that typing derivations can be CPS-transformed, so that we can lift the CPS transformation from the level of expressions to the level of type derivations (Xi & Schürmann 2001). They studied a call-by-value CPS transformation for a core subset of the DML language. They proved they can lift the CPS transformation from the level of expressions to the level of type derivations, and they formalized the languages and transformation in LF. However, they did not prove nor verify the correctness of their CPS transformation for DML types. Yasuhiko Minamide and Koji Okuma have verified several versions of CPS transformation in the Isabelle/HOL theorem prover (Minamide & Okuma 2003). Since they chose to use first-order abstract syntax with variable names for their formalization, they had the problems of bound variables renaming and they needed to implement their own "substitution" programs.

None of the related work discussed above showed how to use higher-order abstract syntax to formalize and verify CPS translation, which is our goal here. In this paper, we presents a higher-order setting of CPS transformation as a rewriting system that directly produces a CPS program without administrative redexes. We also show how to formalize CPS translation using higher-order abstract syntax and verify its correctness proof using logical framework. This approach has several advantages compared to the ones above: 1) No administrative redexes. 2) Proof is a structural induction proof. 3) The formal-

Copyright copyright 2006, Australian Computer Society, Inc. This paper appeared at Computing: The Australasian Theory Symposium (CATS2006), Hobart, Australia. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 51. Barry Jay and Joachim Gudmundsson, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.

ization does not have the problems of fresh variables and α -equivalence.

The rest of this paper is organized as follows. Section 2 introduces the direct-style source language and the continuation-passing style target language we consider in our CPS translation and their operational semantics. The one-pass higher-order CPS transformation is presented at the end of this section. Section 3 shows how we prove the correctness of our CPS transformation by proving the “Soundness” and “Completeness” theorems. In Section 4, we present the formalization of the CPS translation and the correctness proof using higher-order abstract syntax in a logical framework. Section 5 concludes.

2 Languages and CPS Transformation

2.1 Operational semantics for Mini-ML

We will consider a fragment of a functional language as our source language. It serves as the jumping-off point for much of the studies of programming language concepts. In our CPS translation, we consider the Mini-ML expressions as the *direct-style* programs. We only consider a small subset of the language in this paper, the syntax of the Mini-ML language is defined as follows:

Expressions $e ::=$ $\text{zero} \mid (\text{succ } e) \mid (\text{case } e_1 \text{ of } \text{zero} \Rightarrow e_2 \mid \text{succ } x \Rightarrow e_3) \mid \text{pair}(e_1, e_2) \mid \text{fst } e \mid \text{snd } e \mid \text{app}(e_1, e_2) \mid \text{lam } x. e \mid \text{vl}(v)$

Values $v ::= x \mid \text{zero}^* \mid \text{succ}^* v \mid \text{pair}^*(v_1, v_2) \mid \text{lam}^* x. e$

Most of these constructs should be familiar from functional programming languages such as ML: **zero** stands for zero, **(succ e)** stands for the successor of e . A **case** expression chooses a branch based on whether the value of the first argument is zero or non-zero. Abstraction, **lam $x. e$** , forms functional expressions. Note that only value variables exist. The term **vl(v)** represents a coercion of a Mini-ML value to a Mini-ML expression. We again distinguish values from expressions since this will later simplify the formalization of the CPS translation.

$$\begin{array}{c}
\frac{}{\text{vl}(v) \hookrightarrow v} \text{ev_vl} \\
\frac{}{\text{zero} \hookrightarrow \text{zero}^*} \text{ev_z} \quad \frac{e \hookrightarrow v}{\text{succ } e \hookrightarrow \text{succ}^* v} \text{ev_s} \\
\frac{e_1 \hookrightarrow z \quad e_2 \hookrightarrow v}{(\text{case } e_1 \text{ of } z \Rightarrow e_2 \mid \text{succ } x \Rightarrow e_3) \hookrightarrow v} \text{ev_case_z} \\
\frac{e_1 \hookrightarrow \text{succ}^* v'_1 \quad [v'_1/x]e_3 \hookrightarrow v}{(\text{case } e_1 \text{ of } \text{zero} \Rightarrow e_2 \mid \text{succ } x \Rightarrow e_3) \hookrightarrow v} \text{ev_case_s} \\
\frac{e_1 \hookrightarrow v_1 \quad e_2 \hookrightarrow v_2}{\text{pair}(e_1, e_2) \hookrightarrow \text{pair}^*(v_1, v_2)} \text{ev_pair} \\
\frac{e \hookrightarrow \text{pair}^*(v_1, v_2)}{\text{fst } e \hookrightarrow v_1} \text{ev_fst} \quad \frac{e \hookrightarrow \text{pair}^*(v_1, v_2)}{\text{snd } e \hookrightarrow v_2} \text{ev_snd} \\
\frac{}{\text{lam } x. e \hookrightarrow \text{lam}^* x. e} \text{ev_lam} \\
\frac{e_1 \hookrightarrow \text{lam}^* x. e' \quad e_2 \hookrightarrow v_2 \quad [v_2/x]e' \hookrightarrow v}{\text{app}(e_1, e_2) \hookrightarrow v} \text{ev_app}
\end{array}$$

Figure 1: The Mini-ML big-step evaluation semantics.

Evaluation rules of the Mini-ML expressions are shown in Figure 1. They are given via a big-step

operational semantics and are described by the judgment $e \hookrightarrow v$ meaning the expression e evaluates to a value v . For example, to evaluate an application **app(e_1, e_2)**, we need to evaluate e_1 to some value **lam * $x. e'$** , e_2 to some value v_2 , and $[v_2/x]e'$ to the final value of the application. Note, the order of evaluation of these premises is left unspecified. The other rules are straightforward.

We will refer to the Mini-ML language as direct style to distinguish it from the continuation-passing style (CPS) calculus introduced below.

2.2 The continuation-passing style language

In this section, we define the target language of the CPS compilation, which describes programs in continuation-passing style.

CPS terms satisfy three properties (Plotkin 1975):

- Indifference: Evaluation order independent.
- Simulation: CPS encodes the evaluation order.
- Translation equational correspondence between direct-style and CPS-style.

The syntax of the CPS language we use to represent the continuation-passing style programs is defined as follows:

Expressions $E ::=$ **App(V_1, V_2, k)** | **Fst(V, k)** | **Snd(V, k)** | **(Case V of Zero $\Rightarrow E_2 \mid (\text{Succ } x) \Rightarrow E_3$)** | **vl(V)**

Values $V ::= x \mid \text{Zero} \mid \text{Succ } V \mid \text{Pair}(V_1, V_2) \mid \text{Lam } (x, k).E$

Continuations $k ::= \lambda x. E$

Evaluation contexts are modeled via CPS continuations. A CPS continuation is represented as a meta-level function $\lambda x. E$ which essentially describes a CPS expression which has a hole x , that can only be filled by a CPS value. The initial or empty evaluation context is represented by $\lambda x. \text{vl}(V)$. Substitution into a hole is modeled via meta-level application $((\lambda x. E) V)$ which beta-reduces to $[V/x]E$. An important property of CPS continuations is that they are in fact linear functions, i.e. the hole x in the CPS expression E occurs only once (Appel 1992, Danvy et al. 1999). We will not prove this property but it will be maintained as part of the invariants in this description.

The CPS expressions keep track of its evaluation context in which they will be executed using an extra parameter k . In the definition above, we separate cleanly CPS values from CPS expressions and **vl(V)** denotes the coercion of a CPS value to a CPS expression. Note that the arguments to CPS expressions must be CPS values and cannot be arbitrary subexpressions.

Figure 2 shows the operational semantics of the CPS language. The single-step reductions for the CPS expressions are defined first. The multi-step reduction relation \mapsto^* is the reflexive, transitive closure of single-step reduction \mapsto . Deductions of the judgment $E \mapsto^* E'$ have a very simple form: They all consist of a sequence of single steps terminated by an application of the reflexivity rule. We will follow standard practice and use a linear notation for sequences of steps:

$$E_1 \mapsto E_2 \mapsto E_3 \mapsto \dots \mapsto E_n$$

Similarly, we will mix multi-step and single-step transitions in sequences with the obvious meaning. Using the rules above we can reduce CPS expressions to CPS values (in the form of **vl(V)**). There are two important properties of this multi-reduction system.

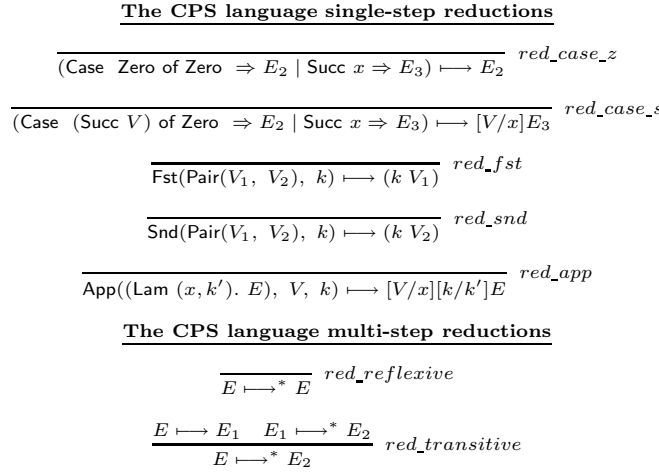


Figure 2: The CPS language operational semantics.

Theorem 1 (Uniqueness).

If $E \mapsto^* \text{vl}(V_1)$ and $E \mapsto^* \text{vl}(V_2)$, then $V_1 = V_2$.

Proof. Proof by structural induction on the reduction rules. \square

Theorem 2 (Termination).

For every CPS expression E there is some CPS value V such that $E \mapsto^* \text{vl}(V)$.

Proof. Note that no variable can be bound in more than one place within a continuation expression, and no variable can be mentioned outside its syntactic scope. So we can observe that each reduction step reduces the size of the term and that size is a termination measure because the usual order on the natural numbers is well founded. \square

2.3 The one-pass CPS transformation

In this section, we define two CPS-translations, one for Mini-ML values and one for Mini-ML expressions. The translations are mutual recursive. While the CPS-translation for values is essentially closed and independent of the CPS continuations, the CPS-translation for expressions must take into account the current evaluation state, which is captured by the CPS continuations. We are also very explicit about free variables which will all be bound in a context Γ , which is defined as follows:

$$\Gamma ::= . \mid \Gamma, x \mid \Gamma, k$$

The context Γ can be either empty or contain Mini-ML and CPS variables as well as continuation variable k .

The CPS transformation we present here is similar to an optimized version of CPS transformation by Danvy and Nielsen (Danvy & Nielsen 2002). The transformation operates in one pass and is both compositional and higher-order. Because it operates in one pass, it directly yields compact CPS programs that are comparable to what one would write by hand. Because it is compositional, it allows proofs by structural induction. Because it is higher-order, we avoid administrative redexes which in turn simplifies the proof about CPS-translation. We give the translation as an inference system, which may be slightly unusual but emphasizes the behavior of the CPS translation.

For the purpose of our formalization, Figure 3 expresses the one-pass CPS transformation as inference rules. It uses two judgments. A Mini-ML value v is

transformed into a CPS value V whenever the judgment

$$\Gamma \vdash v \overset{v}{\sim} V$$

is satisfied. Given a (higher-order) continuation k , a Mini-ML expression e is transformed into a CPS expression E whenever the judgment

$$\Gamma \vdash (e, k) \overset{e}{\sim} E$$

is satisfied.

These judgments can be interpreted operationally by assuming that v , e and k are given and V and E are to be constructed by building a CPS-translation derivation in a bottom-up fashion. Since the variables in the Mini-ML languages and the CPS language can only contain values, we say that a Mini-ML variable x is CPS-transformed into a CPS variable x , by $\Gamma \vdash x \overset{v}{\sim} x$, if x is in the context Γ . All the free variables are captured by the context Γ during the CPS-translation. The context Γ can actually contains both Mini-ML and CPS variables (one can think of that Γ contains implicit CPS-translation from Mini-ML variables to CPS variables). The free variables bound in Γ may occur in e , k or E . Note that in order to translate $\text{lam}^* x. e$ we need to recursively translate the function body e .

Transforming a Mini-ML expression into its corresponding CPS expression needs to represent continuation contexts as λ -abstractions. A Mini-ML expression e is transformed with a continuation context k , it ranges over meta-level functions, which map CPS values to CPS expressions. The result of transforming an expression e into CPS expression E in an empty context is given by

$$\Gamma \vdash (e, (\lambda x. \text{vl}(x))) \overset{e}{\sim} E$$

3 Correctness of the one-pass CPS transformation

Plotkin proved three properties of the CPS transformation: Indifference, Simulation, and Translation (Plotkin 1975). We prove the Simulation property here, since it formalizes the correctness of the call-by-value CPS transformation.

To prove correctness, we need to show the correspondence between the high-level language and the low-level abstract machine. In other words, we can translate programs written in the high-level language into programs which run on the abstract machine. Moreover, the source program will have the same observable behavior as the target program. The proof is constructive and constitutes a program which translates derivations in the source language into derivations of the target language and vice versa.

Theorem 3 (Correctness).**Soundness:**

If $\Gamma \vdash (e, (\lambda x. \text{vl}(x))) \overset{e}{\sim} E'$ and $E' \mapsto^* \text{vl}(V)$ then $e \hookrightarrow v$ and $\Gamma \vdash v \overset{v}{\sim} V$.

Completeness:

If $e \hookrightarrow v$ and $\Gamma \vdash (e, (\lambda x. \text{vl}(x))) \overset{e}{\sim} E'$ then $\Gamma \vdash v \overset{v}{\sim} V'$ and $E' \mapsto^* \text{vl}(V')$.

The theorem shows that the source program (Mini-ML program) and the target program (CPS-transformed program) will result in the same value. We will prove the correctness of our CPS-translation by proving the soundness and completeness of the translation.

CPS-translation for Mini-ML values

$$\begin{array}{c}
\frac{x \in \Gamma}{\Gamma \vdash x \overset{v}{\sim} x} \text{cps_var} \quad \frac{}{\Gamma \vdash \text{zero}^* \overset{v}{\sim} \text{Zero}} \text{cps_zero}^* \\
\\
\frac{\Gamma \vdash v \overset{v}{\sim} V}{\Gamma \vdash (\text{succ}^* v) \overset{v}{\sim} (\text{Succ } V)} \text{cps_succ}^* \quad \frac{\Gamma \vdash v_1 \overset{v}{\sim} V_1 \quad \Gamma \vdash v_2 \overset{v}{\sim} V_2}{\Gamma \vdash \text{pair}^*(v_1, v_2) \overset{v}{\sim} \text{Pair}(V_1, V_2)} \text{cps_pair}^* \\
\\
\frac{\Gamma, x, k' \vdash (e, k') \overset{e}{\sim} E' \quad \text{where } x \text{ and } k' \text{ are fresh}}{\Gamma \vdash (\text{lam}^* x. e) \overset{v}{\sim} \text{Lam } (x, k'). E'} \text{cps_lam}^*
\end{array}$$

CPS-translation for Mini-ML expressions

$$\begin{array}{c}
\frac{\Gamma \vdash v \overset{v}{\sim} V}{\Gamma \vdash (\text{v1}(v), k) \overset{e}{\sim} (k V)} \text{cps_v1} \quad \frac{}{\Gamma \vdash (\text{zero}, k) \overset{e}{\sim} (k \text{Zero})} \text{cps_zero} \\
\\
\frac{\Gamma \vdash (e, (\lambda x. k (\text{Succ } x))) \overset{e}{\sim} E'}{\Gamma \vdash ((\text{succ } e), k) \overset{e}{\sim} E'} \text{cps_succ} \\
\\
\frac{\Gamma \vdash (e_2, k) \overset{e}{\sim} E_2 \quad \Gamma, x \vdash (e_3, k) \overset{e}{\sim} E_3 \quad \Gamma \vdash (e_1, (\lambda x_1. \text{Case } x_1 \text{ of } z \Rightarrow E_2 \mid (\text{Succ } x) \Rightarrow E_3)) \overset{e}{\sim} E'}{\Gamma \vdash ((\text{case } e_1 \text{ of zero} \Rightarrow e_2 \mid (\text{succ } x) \Rightarrow e_3), k) \overset{e}{\sim} E'} \text{cps_case} \\
\\
\frac{\Gamma, x_1 \vdash (e_2, (\lambda x_2. k (\text{Pair}(x_1, x_2)))) \overset{e}{\sim} E_2 \quad \Gamma \vdash (e_1, (\lambda x_1. E_2)) \overset{e}{\sim} E'}{\Gamma \vdash (\text{pair}(e_1, e_2), k) \overset{e}{\sim} E'} \text{cps_pair} \\
\\
\frac{\Gamma \vdash (e, (\lambda x_1. \text{Fst}(x_1, k))) \overset{e}{\sim} E'}{\Gamma \vdash (\text{fst } e, k) \overset{e}{\sim} E'} \text{cps_fst} \quad \frac{\Gamma \vdash (e, (\lambda x_2. \text{Snd}(x_2, k))) \overset{e}{\sim} E'}{\Gamma \vdash (\text{snd } e, k) \overset{e}{\sim} E'} \text{cps_snd} \\
\\
\frac{\Gamma, x, k' \vdash (e, k') \overset{e}{\sim} E' \quad \text{where } x \text{ and } k' \text{ are fresh}}{\Gamma \vdash (\text{lam } x. e, k) \overset{e}{\sim} (k (\text{Lam } (x, k'). E'))} \text{cps_lam} \\
\\
\frac{\Gamma, x_1 \vdash (e_2, (\lambda x_2. \text{App}(x_1, x_2, k))) \overset{e}{\sim} E_2 \quad \Gamma \vdash (e_1, (\lambda x_1. E_2)) \overset{e}{\sim} E'}{\Gamma \vdash (\text{app}(e_1, e_2), k) \overset{e}{\sim} E'} \text{cps_app}
\end{array}$$

Figure 3: The one-pass CPS transformation formulated as inference rules.

To show that the abstract machine works correctly, we prove that all reduction sequences can be translated into evaluation derivations. To be more precise, we need to show that for all reduction sequences on the abstract machine starting with the empty context which evaluate some expression E ($\Gamma \vdash (e, (\lambda x. \text{vl}(x))) \stackrel{e}{\sim} E$) to some CPS value V in multiple steps, there exists an evaluation in the Mini-ML operational semantics s.t. expression e evaluates to value v and $\Gamma \vdash v \stackrel{v}{\sim} V$.

This guarantees that the translation is sound. However, we will not be able to prove this statement directly, since the evaluation context k will change, which will prevent the application of the induction hypothesis in the proof. Therefore we will prove the following generalized statement: if we start in an arbitrary evaluation context k , with the state $\Gamma \vdash (e, k) \stackrel{e}{\sim} E$ and a reduction sequence $E \mapsto^* \text{vl}(W)$ then there exists an intermediate state $(k \ V)$ such that $e \hookrightarrow v$ in the Mini-ML semantics and $\Gamma \vdash v \stackrel{v}{\sim} V$ and $(k \ V) \mapsto^* \text{vl}(W)$. The lemma states that a complete computation with an appropriate initial state can be translated into an evaluation followed by another complete computation. This proof follows similar proofs about the correctness of compilers which are discussed in more detail in (Pfenning 2001).

3.1 Substitution lemmas

Before we proceed to prove this soundness lemma, we will first show that CPS-translation is preserved across substitution. This is an important property needed in the proof for the Soundness and Completeness lemmas.

Lemma 1 (Substitution lemma).

1. For all Mini-ML value v and v' . If $\Gamma \vdash v' \stackrel{v}{\sim} V'$ and $\Gamma, x, \Gamma' \vdash v \stackrel{v}{\sim} V$ then $\Gamma, \Gamma' \vdash [v'/x]v \stackrel{v}{\sim} [V'/x]V$.
2. For all Mini-ML expressions e and values v' . If $\Gamma \vdash v' \stackrel{v}{\sim} V'$ and $\Gamma, x, \Gamma' \vdash (e, k) \stackrel{e}{\sim} E$ then $\Gamma, \Gamma' \vdash ([v'/x]e, [V'/x]k) \stackrel{e}{\sim} [V'/x]E$.

Proof. By mutual induction on v and e . We only show a few cases in the proof in detail; the remaining ones follow the same pattern.

Case: $v = x$

$\Gamma, x, \Gamma' \vdash x \stackrel{x}{\sim} V$ By ass.
 $V = x$ By cps_var
 $\Gamma \vdash v' \stackrel{v}{\sim} V'$ By ass.
 $\Gamma, \Gamma' \vdash [v'/x]x \stackrel{x}{\sim} [V'/x]x$ By subst. def.

Case: $v = \text{zero}^*$

$\Gamma, x, \Gamma' \vdash \text{zero}^* \stackrel{v}{\sim} V$ By ass.
 $V = \text{Zero}$ By cps_zero*
 $\Gamma \vdash v' \stackrel{v}{\sim} V'$ By ass.
 $\Gamma, \Gamma' \vdash [v'/x]\text{zero}^* \stackrel{v}{\sim} [V'/x]\text{Zero}$ By subst. def.

Case: $v = \text{pair}^*(v_1, v_2)$

$\Gamma, x, \Gamma' \vdash \text{pair}^*(v_1, v_2) \stackrel{v}{\sim} V$ By ass.
 $V = \text{Pair}(V_1, V_2)$,
 $\Gamma, x, \Gamma' \vdash v_1 \stackrel{v}{\sim} V_1$ and $\Gamma, x, \Gamma' \vdash v_2 \stackrel{v}{\sim} V_2$ By cps_pair*
 $\Gamma \vdash v' \stackrel{v}{\sim} V'$ By ass.
 $\Gamma, \Gamma' \vdash [v'/x]v_1 \stackrel{v}{\sim} [V'/x]V_1$ and
 $\Gamma, \Gamma' \vdash [v'/x]v_2 \stackrel{v}{\sim} [V'/x]V_2$ By I.H.
 $\Gamma, \Gamma' \vdash \text{pair}^*([v'/x]v_1, [v'/x]v_2) \stackrel{v}{\sim} \text{Pair}([V'/x]V_1, [V'/x]V_2)$

$\Gamma, \Gamma' \vdash [v'/x]\text{pair}^*(v_1, v_2) \stackrel{v}{\sim} [V'/x]\text{Pair}(V_1, V_2)$ By cps_pair*
 By subst. def.

Case: $v = \text{lam}^* y. e$ (where $y \neq x$)

$\Gamma, x, \Gamma' \vdash \text{lam}^* y. e \stackrel{v}{\sim} V$ By ass.
 $V = \text{Lam}(y, k'). E'$ and
 $\Gamma, y, k', x, \Gamma' \vdash (e, k') \stackrel{e}{\sim} E'$ By cps_lam*
 $\Gamma \vdash v' \stackrel{v}{\sim} V'$ By ass.
 $\Gamma, y, k', \Gamma' \vdash ([v'/x]e, k') \stackrel{e}{\sim} [V'/x]E'$ By I.H. on subst. lemma 2
 $\Gamma, \Gamma' \vdash \text{lam}^* y. [v'/x]e \stackrel{v}{\sim} \text{Lam}(y, k'). [V'/x]E'$ By cps_lam*
 $\Gamma, \Gamma' \vdash [v'/x]\text{lam}^* y. e \stackrel{v}{\sim} [V'/x]\text{Lam}(y, k'). E'$ By subst. def.

Case: $e = \text{vl}(v)$

$\Gamma, x, \Gamma' \vdash (\text{vl}(v), k) \stackrel{e}{\sim} E$ By ass.
 $E = (k \ V)$ and $\Gamma, x, \Gamma' \vdash v \stackrel{v}{\sim} V$ By cps_vl
 $\Gamma \vdash v' \stackrel{v}{\sim} V'$ By ass.
 $\Gamma, \Gamma' \vdash [v'/x]v \stackrel{v}{\sim} [V'/x]V$ By I.H. on subst. lemma 1
 $\Gamma, \Gamma' \vdash (\text{vl}([v'/x]v), [V'/x]k) \stackrel{e}{\sim} ([V'/x]k \ [V'/x]V)$ By cps_vl
 $\Gamma, \Gamma' \vdash ([v'/x]\text{vl}(v), [V'/x]k) \stackrel{e}{\sim} [V'/x](k \ V)$ By subst. def.

Case: $e = \text{zero}$

$\Gamma, x, \Gamma' \vdash (\text{zero}, k) \stackrel{e}{\sim} E$ By ass.
 $E = (k \ \text{Zero})$ By cps_zero
 $\Gamma \vdash v' \stackrel{v}{\sim} V'$ By ass.
 $\Gamma, \Gamma' \vdash ([v'/x]\text{zero}, [V'/x]k) \stackrel{e}{\sim} ([V'/x]k \ \text{Zero})$ By subst. def. and cps_zero
 $\Gamma, \Gamma' \vdash ([v'/x]\text{zero}, [V'/x]k) \stackrel{e}{\sim} [V'/x](k \ \text{Zero})$ By subst. def.

Case: $e = \text{pair}(e_1, e_2)$

$\Gamma, x, \Gamma' \vdash (\text{pair}(e_1, e_2), k) \stackrel{e}{\sim} E'$ By ass.
 $\Gamma, x_1, x, \Gamma' \vdash (e_2, (\lambda x_2. k \ (\text{Pair}(x_1, x_2)))) \stackrel{e}{\sim} E_2$ and
 $\Gamma, x, \Gamma' \vdash (e_1, (\lambda x_1. E_2)) \stackrel{e}{\sim} E'$ By cps_pair
 $\Gamma \vdash v' \stackrel{v}{\sim} V'$ By ass.
 $\Gamma, x_1, \Gamma' \vdash ([v'/x]e_2, [V'/x](\lambda x_2. k \ (\text{Pair}(x_1, x_2)))) \stackrel{e}{\sim} [V'/x]E_2$ By I.H.
 $\Gamma, x_1, \Gamma' \vdash ([v'/x]e_2, (\lambda x_2. [V'/x]k \ (\text{Pair}(x_1, x_2)))) \stackrel{e}{\sim} [V'/x]E_2$ By subst. def.
 $\Gamma, \Gamma' \vdash ([v'/x]e_1, [V'/x](\lambda x_1. E_2)) \stackrel{e}{\sim} [V'/x]E'$ By I.H.
 $\Gamma, \Gamma' \vdash ([v'/x]e_1, (\lambda x_1. [V'/x]E_2)) \stackrel{e}{\sim} [V'/x]E'$ By subst. def.
 $\Gamma, \Gamma' \vdash (\text{pair}([v'/x]e_1, [v'/x]e_2), k) \stackrel{e}{\sim} [V'/x]E'$ By cps_pair
 $\Gamma, \Gamma' \vdash ([v'/x]\text{pair}(e_1, e_2), k) \stackrel{e}{\sim} [V'/x]E'$ By subst. def.

Case: $e = \text{app}(e_1, e_2)$

$\Gamma, x, \Gamma' \vdash (\text{app}(e_1, e_2), k) \stackrel{e}{\sim} E'$ By ass.
 $\Gamma, x_1, x, \Gamma' \vdash (e_2, (\lambda x_2. \text{App}(x_1, x_2, k))) \stackrel{e}{\sim} E_2$ and
 $\Gamma, x, \Gamma' \vdash (e_1, (\lambda x_1. E_2)) \stackrel{e}{\sim} E'$ By cps_app
 $\Gamma \vdash v' \stackrel{v}{\sim} V'$ By ass.
 $\Gamma, x_1, \Gamma' \vdash ([v'/x]e_2, [V'/x](\lambda x_2. \text{App}(x_1, x_2, k))) \stackrel{e}{\sim} [V'/x]E_2$ By I.H.
 $\Gamma, x_1, \Gamma' \vdash ([v'/x]e_2, (\lambda x_2. \text{App}(x_1, x_2, [V'/x]k))) \stackrel{e}{\sim} [V'/x]E_2$ By subst. def.
 $\Gamma, \Gamma' \vdash ([v'/x]e_1, [V'/x](\lambda x_1. E_2)) \stackrel{e}{\sim} [V'/x]E'$ By I.H.
 $\Gamma, \Gamma' \vdash ([v'/x]e_1, (\lambda x_1. [V'/x]E_2)) \stackrel{e}{\sim} [V'/x]E'$ By subst. def.
 $\Gamma, \Gamma' \vdash (\text{app}([v'/x]e_1, [v'/x]e_2), [V'/x]k) \stackrel{e}{\sim} [V'/x]E'$ By cps_app
 $\Gamma, \Gamma' \vdash ([v'/x]\text{app}(e_1, e_2), [V'/x]k) \stackrel{e}{\sim} [V'/x]E'$ By subst. def.

□

3.2 Soundness

Lemma 2 (Soundness).

If $\mathcal{C} : \Gamma \vdash (e, k) \stackrel{e}{\sim} E$ and $\mathcal{S} : E \mapsto^* \text{vl}(W)$ then there exist derivations $\mathcal{D} : e \hookrightarrow v$, $\mathcal{C}' : \Gamma \vdash v \stackrel{v}{\sim} V$ and a rest computation $\mathcal{S}' : (k \ V) \mapsto^* \text{vl}(W)$, where \mathcal{S}' is a subsequence of \mathcal{S} .

Proof. By structural induction on the derivation $\mathcal{C} : \Gamma \vdash (e, k) \stackrel{\sim}{\sim} E$ and the CPS-reductions $\mathcal{S} : E \mapsto^* \text{vl}(W)$. We can apply the induction hypothesis if either the CPS-translation is applied to a sub-derivation of $\mathcal{C} : \Gamma \vdash (e, k) \stackrel{\sim}{\sim} E$ or we have a shorter sequence of CPS-reductions \mathcal{S}' where $\mathcal{S}' < \mathcal{S}$. We show most cases in the proof in detail; the remaining ones follow the same pattern. Throughout the proof we consider CPS expressions modulo $\stackrel{\beta}{\sim}$.

Case: $\mathcal{C} =$

$$\begin{array}{c}
\mathcal{C}_2 \\
\Gamma, x_1 \vdash (e_2, (\lambda x_2. k (\text{Pair}(x_1, x_2)))) \stackrel{\sim}{\sim} E_2 \\
\mathcal{C}_1 \\
\Gamma \vdash (e_1, (\lambda x_1. E_2)) \stackrel{\sim}{\sim} E' \\
\hline
\Gamma \vdash (\text{pair}(e_1, e_2), k) \stackrel{\sim}{\sim} E' \quad \text{cps_pair} \\
\\
\mathcal{S} : E' \mapsto^* \text{vl}(W) \quad \text{By ass.} \\
\mathcal{D}_1 : e_1 \hookrightarrow v_1, \\
\mathcal{C}'_1 : \Gamma \vdash v_1 \stackrel{\sim}{\sim} V_1, \text{ and} \\
\mathcal{S}'_1 : ((\lambda x_1. E_2) V_1) \mapsto^* \text{vl}(W) \quad \text{By I.H. on } \mathcal{C}_1 \\
\hline
\stackrel{\beta}{\sim} [V_1/x_1] E_2 \\
\Gamma \vdash ([v_1/x_1] e_2, (\lambda x_2. [V_1/x_1] k (\text{Pair}(x_1, x_2)))) \stackrel{\sim}{\sim} [V_1/x_1] E_2 \\
\text{By the Substitution lemma} \\
\Gamma \vdash (e_2, (\lambda x_2. k (\text{Pair}(V_1, x_2)))) \stackrel{\sim}{\sim} [V_1/x_1] E_2 \\
x_1 \text{ occurs only once in the continuation} \\
\mathcal{S}'_1 : [V_1/x_1] E_2 \mapsto^* \text{vl}(W) \quad \text{By previous lines} \\
\mathcal{D}_2 : e_2 \hookrightarrow v_2, \\
\mathcal{C}'_2 : \Gamma \vdash v_2 \stackrel{\sim}{\sim} V_2, \text{ and} \\
\mathcal{S}'_2 : ((\lambda x_2. k (\text{Pair}(V_1, x_2))) V_2) \mapsto^* \text{vl}(W) \quad \text{By I.H. } (\mathcal{S}'_1 < \mathcal{S}) \\
\hline
\stackrel{\beta}{\sim} (k \text{ Pair}(V_1, V_2)) \\
\mathcal{D} : \text{pair}(e_1, e_2) \hookrightarrow \text{pair}^*(v_1, v_2) \quad \text{By ev_pair} \\
\mathcal{C}' : \Gamma \vdash \text{pair}^*(v_1, v_2) \stackrel{\sim}{\sim} \text{Pair}(V_1, V_2) \quad \text{By cps_pair}^* \\
\mathcal{S}' : (k \text{ Pair}(V_1, V_2)) \mapsto^* \text{vl}(W) \quad \text{By } \mathcal{S}'_2
\end{array}$$

Case: $\mathcal{C} =$

$$\begin{array}{c}
\mathcal{C}_1 \\
\Gamma \vdash (e, (\lambda x_1. \text{Fst}(x_1, k))) \stackrel{\sim}{\sim} E' \\
\hline
\Gamma \vdash (\text{fst } e, k) \stackrel{\sim}{\sim} E' \quad \text{cps_fst} \\
\\
\mathcal{S} : E' \mapsto^* \text{vl}(W) \quad \text{By ass.} \\
\mathcal{D}_1 : e \hookrightarrow v, \\
\mathcal{C}'_1 : \Gamma \vdash v \stackrel{\sim}{\sim} V, \text{ and} \\
\mathcal{S}'_1 : ((\lambda x_1. \text{Fst}(x_1, k)) V) \mapsto^* \text{vl}(W) \quad \text{By I.H. on } \mathcal{C}_1 \\
\hline
\stackrel{\beta}{\sim} \text{Fst}(V, k) \\
V = \text{Pair}(V_1, V_2) \text{ and} \\
\mathcal{S}_1 : \text{Fst}(\text{Pair}(V_1, V_2), k) \mapsto^* \text{vl}(W) \quad \text{By red_fst} \\
\hline
\mathcal{S}' \\
v = \text{pair}^*(v_1, v_2), \Gamma \vdash v_1 \stackrel{\sim}{\sim} V_1, \text{ and } \Gamma \vdash v_2 \stackrel{\sim}{\sim} V_2 \\
\text{By inversion on cps_pair}^* \\
\mathcal{D}_1 : e \hookrightarrow \text{pair}^*(v_1, v_2) \quad \text{By previous lines} \\
\mathcal{D} : \text{fst } e \hookrightarrow v_1 \quad \text{By ev_fst} \\
\mathcal{S}' : (k V_1) \mapsto^* \text{vl}(W) \quad \text{By previous lines}
\end{array}$$

Case: $\mathcal{C} =$

$$\begin{array}{c}
\mathcal{C}_1 \\
\Gamma, x, k' \vdash (e, k') \stackrel{\sim}{\sim} E' \quad \text{where } x \text{ and } k' \text{ are fresh} \\
\hline
\Gamma \vdash (\text{lam } x. e, k) \stackrel{\sim}{\sim} (k (\text{Lam } (x, k'). E')) \quad \text{cps_lam} \\
\\
\mathcal{S} : (k (\text{Lam } (x, k'). E')) \mapsto^* \text{vl}(W) \quad \text{By ass.} \\
\mathcal{D} : \text{lam } x. e \hookrightarrow \text{lam}^* x. e \quad \text{By ev_lam} \\
\mathcal{C}' : \Gamma \vdash \text{lam}^* x. e \stackrel{\sim}{\sim} \text{Lam } (x, k'). E' \quad \text{By } \mathcal{C}_1 \text{ and cps_lam}^* \\
\mathcal{S}' : (k (\text{Lam } (x, k'). E')) \mapsto^* \text{vl}(W) \quad \text{By ass.}
\end{array}$$

Case: $\mathcal{C} =$

$$\begin{array}{c}
\mathcal{C}_2 \\
\Gamma, x_1 \vdash (e_2, (\lambda x_2. \text{App}(x_1, x_2, k))) \stackrel{\sim}{\sim} E_2 \\
\mathcal{C}_1 \\
\Gamma \vdash (e_1, (\lambda x_1. E_2)) \stackrel{\sim}{\sim} E' \\
\hline
\Gamma \vdash (\text{app}(e_1, e_2), k) \stackrel{\sim}{\sim} E' \quad \text{cps_app} \\
\\
\mathcal{S} : E' \mapsto^* \text{vl}(W) \quad \text{By ass.} \\
\mathcal{D}_1 : e_1 \hookrightarrow v_1, \\
\mathcal{C}'_1 : \Gamma \vdash v_1 \stackrel{\sim}{\sim} V_1, \text{ and} \\
\mathcal{S}'_1 : ((\lambda x_1. E_2) V_1) \mapsto^* \text{vl}(W) \quad \text{By I.H. on } \mathcal{C}_1 \\
\hline
\stackrel{\beta}{\sim} [V_1/x_1] E_2 \\
\Gamma \vdash ([v_1/x_1] e_2, (\lambda x_2. [V_1/x_1] \text{App}(x_1, x_2, k))) \stackrel{\sim}{\sim} [V_1/x_1] E_2 \\
\text{By the Substitution lemma} \\
\Gamma \vdash (e_2, (\lambda x_2. \text{App}(V_1, x_2, k))) \stackrel{\sim}{\sim} [V_1/x_1] E_2 \\
x_1 \text{ occurs only once in the continuation} \\
\mathcal{S}'_1 : [V_1/x_1] E_2 \mapsto^* \text{vl}(W) \quad \text{By previous lines} \\
\mathcal{D}_2 : e_2 \hookrightarrow v_2, \\
\mathcal{C}'_2 : \Gamma \vdash v_2 \stackrel{\sim}{\sim} V_2, \text{ and} \\
\mathcal{S}'_2 : ((\lambda x_2. \text{App}(V_1, x_2, k)) V_2) \mapsto^* \text{vl}(W) \quad \text{By I.H. } (\mathcal{S}'_1 < \mathcal{S}) \\
\hline
\stackrel{\beta}{\sim} \text{App}(V_1, V_2, k) \\
V_1 = \text{Lam } (x, k'). E \quad (\text{for some } E) \text{ and} \\
\mathcal{S}_2 : \text{App}((\text{Lam } (x, k'). E), V_2, k) \mapsto^* \text{vl}(W) \quad \text{By red_app} \\
\hline
v_1 = \text{lam}^* x. e \text{ and } \Gamma, x, k' \vdash (e, k') \stackrel{\sim}{\sim} E \\
\text{By inversion on cps_lam}^* \\
\Gamma \vdash ([v_2/x] e, k) \stackrel{\sim}{\sim} [V_2/x] [k/k'] E \quad \text{By the Substitution lemma} \\
\mathcal{S}_3 : [V_2/x] [k/k'] E \mapsto^* \text{vl}(W) \quad \text{By previous lines} \\
\mathcal{D}_3 : [v_2/x] e \hookrightarrow v, \mathcal{C}' : \Gamma \vdash v \stackrel{\sim}{\sim} V \quad (\text{for some } V), \text{ and} \\
\mathcal{S}' : (k V) \mapsto^* \text{vl}(W) \quad \text{By I.H. } (\mathcal{S}_3 < \mathcal{S}) \\
\mathcal{D} : \text{app}(e_1, e_2) \hookrightarrow v \quad \text{By ev_app}
\end{array}$$

□

It is important to realize that this lemma cannot be (automatically) proven only by applying structural induction on the length of reduction sequences $\mathcal{S} : E \mapsto^* \text{vl}(W)$ nor purely on the CPS-translation $\mathcal{C} : \Gamma \vdash (e, k) \stackrel{\sim}{\sim} E$. Clearly the CPS-reduction sequence is not decreasing in several cases. For the CPS-translation, the difficulty is to establish that we apply the CPS-translation always to a smaller expression. However this is non-trivial and relies essentially on the property that every variable in the CPS-continuation occurs only once, i.e. CPS-continuations as linear. An important observation is that we do not necessarily need linearity to justify the soundness proof, since in the cases where we cannot directly establish that the CPS-translation is applied to a smaller term, the CPS-reduction sequence is decreasing.

Theorem 4 (Soundness).

If $\mathcal{C} : \Gamma \vdash (e, \lambda x. \text{vl}(x)) \stackrel{\sim}{\sim} E$, and $\mathcal{S} : E \mapsto^* \text{vl}(V)$ then $\mathcal{D} : e \hookrightarrow v$, and $\mathcal{C}' : \Gamma \vdash v \stackrel{\sim}{\sim} V$.

Proof. By assumption we know that $\mathcal{C} : \Gamma \vdash (e, \lambda x. \text{vl}(x)) \stackrel{\sim}{\sim} E$ and $\mathcal{S} : E \mapsto^* \text{vl}(V)$. By the previous lemma, we know that $\mathcal{D} : e \hookrightarrow v$, $\mathcal{C}' : \Gamma \vdash v \stackrel{\sim}{\sim} V$ and $\mathcal{S}' : (\lambda x. \text{vl}(x)) V \mapsto^* \text{vl}(V)$. □

$$\stackrel{\beta}{\sim} \text{vl}(V)$$

3.3 Completeness

In this section, we prove the completeness of our CPS-translation. Similar to the soundness theorem, which we cannot prove directly, we will need to first prove the “Completeness” lemma. In order to prove the Completeness lemma, a utility lemma needs to be proven first.

Lemma 3 (Totality lemma).

1. For any Mini-ML value v , there exists some CPS value V such that $\Gamma \vdash v \overset{v}{\sim} V$ is derivable.
2. For any Mini-ML expression e and continuation k , there exists some CPS expression E such that $\Gamma \vdash (e, k) \overset{e}{\sim} E$ is derivable.

Proof. Proof by structural induction on the Mini-ML value v and Mini-ML expression e . For each Mini-ML value v and Mini-ML expression e , we have a corresponding CPS translation inference rule, which transforms the Mini-ML value v or Mini-ML expression e into their corresponding CPS value V or CPS expression E . \square

The totality lemma states that for any well-formed value v in the direct-style source program (the Mini-ML program), we can always CPS-transform it into a well-formed CPS value v by a CPS translation $\Gamma \vdash v \overset{v}{\sim} V$. Similar meaning as for the Mini-ML expressions. Note, that in order to prove the totality lemma for the case of Mini-ML value $\text{lam}^* x. e$, we need to use the totality lemma of Mini-ML expression for the body e .

Lemma 4 (Completeness).

If $\mathcal{D} : e \hookrightarrow v$ and $\mathcal{C}' : \Gamma \vdash v \overset{v}{\sim} V$, and $\mathcal{S}' : (k V) \mapsto^* \text{vl}(W)$ and $\mathcal{C} : \Gamma \vdash (e, k) \overset{e}{\sim} E$ then $\mathcal{S} : E \mapsto^* \text{vl}(W)$ is derivable.

Proof. By structural induction on the derivation $\mathcal{D} : e \hookrightarrow v$. We only show a few cases in the proof in detail; the remaining ones follow the same pattern.

Case: $\mathcal{D} =$

$$\frac{\frac{\mathcal{D}_1}{e_1 \hookrightarrow v_1} \quad \frac{\mathcal{D}_2}{e_2 \hookrightarrow v_2}}{\text{pair}(e_1, e_2) \hookrightarrow \text{pair}^*(v_1, v_2)} \text{ev_pair}$$

$\mathcal{C}' : \Gamma \vdash \text{pair}^*(v_1, v_2) \overset{v}{\sim} V$ By ass.
 $V = \text{Pair}(V_1, V_2)$ and $\Gamma \vdash v_1 \overset{v}{\sim} V_1, \Gamma \vdash v_2 \overset{v}{\sim} V_2$ By *cps_pair*
 $\mathcal{C} : \Gamma \vdash (\text{pair}(e_1, e_2), k) \overset{e}{\sim} E'$ By ass.
 $\mathcal{C}_2 : \Gamma, x_1 \vdash (e_2, (\lambda x_2. k (\text{Pair}(x_1, x_2)))) \overset{e}{\sim} E_2$, and
 $\mathcal{C}_1 : \Gamma \vdash (e_1, (\lambda x_1. E_2)) \overset{e}{\sim} E'$ By inversion on *cps_pair*
 $\mathcal{S}' : (k \text{Pair}(V_1, V_2)) \mapsto^* \text{vl}(W)$ By ass.
 $\mathcal{S}'_2 : ((\lambda x_2. k (\text{Pair}(V_1, x_2))) V_2) \mapsto^* \text{vl}(W)$ By β -reduction
 $\mathcal{C}_3 : \Gamma \vdash ([v_1/x_1]e_2, (\lambda x_2. [V_1/x_1]k (\text{Pair}(x_1, x_2)))) \overset{e}{\sim} [V_1/x_1]E_2$
 By the Substitution lemma
 $\mathcal{C}_3 : \Gamma \vdash (e_2, (\lambda x_2. k (\text{Pair}(V_1, x_2)))) \overset{e}{\sim} [V_1/x_1]E_2$
 x_1 occurs only once in the continuation
 $\mathcal{S}_3 : [V_1/x_1]E_2 \mapsto^* \text{vl}(W)$ By I.H.
 $\mathcal{S}'_1 : ((\lambda x_1. E_2) V_1) \mapsto^* \text{vl}(W)$ By previous lines
 $\mathcal{S} : E' \mapsto^* \text{vl}(W)$ By I.H.

Case: $\mathcal{D} =$

$$\frac{}{\text{lam } x. e \hookrightarrow \text{lam}^* x. e} \text{ev_lam}$$

$\mathcal{C}' : \Gamma \vdash \text{lam}^* x. e \overset{v}{\sim} V$ By ass.
 $V = \text{Lam}(x, k'). E'$ (for some E') By *cps_lam*
 $\mathcal{S}' : (k (\text{Lam}(x, k'). E')) \mapsto^* \text{vl}(W)$ By ass.
 $\mathcal{C} : \Gamma \vdash (\text{lam } x. e, k) \overset{e}{\sim} E$ By ass.
 $E = (k (\text{Lam}(x, k'). E'))$ By *cps_lam*
 $\mathcal{S} : (k (\text{Lam}(x, k'). E')) \mapsto^* \text{vl}(W)$ By ass.

Case: $\mathcal{D} =$

$$\frac{\frac{\mathcal{D}_1}{e_1 \hookrightarrow \text{lam}^* x. e'} \quad \frac{\mathcal{D}_2}{e_2 \hookrightarrow v'_2} \quad \frac{\mathcal{D}_3}{[v'_2/x]e' \hookrightarrow v}}{\text{app}(e_1, e_2) \hookrightarrow v} \text{ev_app}$$

$\mathcal{C}' : \Gamma \vdash v \overset{v}{\sim} V, \mathcal{S}' : (k V) \mapsto^* \text{vl}(W)$ and
 $\mathcal{C} : \Gamma \vdash (\text{app}(e_1, e_2), k) \overset{e}{\sim} E'$ By ass.
 $\mathcal{C}'_1 : \Gamma \vdash \text{lam}^* x. e' \overset{v}{\sim} \text{Lam}(x, k'). E$ (for some E), and
 $\Gamma, x, k' \vdash (e', k') \overset{e}{\sim} E$ By the Totality lemma
 $\mathcal{C}'_2 : \Gamma \vdash v'_2 \overset{v}{\sim} V'_2$ (for some V'_2) By the Totality lemma
 $\mathcal{C}_3 : \Gamma \vdash ([v'_2/x]e', k) \overset{e}{\sim} [V_2/x][k/k']E$ By the Substitution lemma
 $\mathcal{S}_3 : [V_2/x][k/k']E \mapsto^* \text{vl}(W)$ By I.H.
 $\mathcal{S}'_3 : \text{App}((\text{Lam}(x, k'). E), V_2, k) \mapsto [V_2/x][k/k']E$ By *red_app*
 $\mathcal{S}'_2 : ((\lambda x_2. \text{App}((\text{Lam}(x, k'). E), x_2, k)) V_2) \mapsto^* \text{vl}(W)$ By β -reduction
 $\mathcal{C}_2 : \Gamma, x_1 \vdash (e_2, (\lambda x_2. \text{App}(x_1, x_2, k))) \overset{e}{\sim} E_2$,
 $\mathcal{C}_1 : \Gamma \vdash (e_1, (\lambda x_1. E_2)) \overset{e}{\sim} E'$ By inversion on *cps_app*
 $\mathcal{C}_3 : \Gamma \vdash (e_2, (\lambda x_2. [\text{Lam}(x, k'). E/x_1] \text{App}(x_1, x_2, k))) \overset{e}{\sim} E'_2$
 and $E'_2 = [(\text{Lam}(x, k'). E)/x_1]E_2$ By the Substitution lemma
 $\mathcal{C}_3 : \Gamma \vdash (e_2, (\lambda x_2. \text{App}((\text{Lam}(x, k'). E), x_2, k))) \overset{e}{\sim} E'_2$
 x_1 occurs only once in the continuation
 $\mathcal{S}_2 : [(\text{Lam}(x, k'). E)/x_1]E_2 \mapsto^* \text{vl}(W)$ By I.H.
 $\mathcal{S}'_1 : ((\lambda x_1. E_2) (\text{Lam}(x, k'). E)) \mapsto^* \text{vl}(W)$ By previous lines
 $\mathcal{S} : E' \mapsto^* \text{vl}(W)$ By I.H.

\square

Theorem 5 (Completeness).

If $\mathcal{D} : e \hookrightarrow v$, and $\mathcal{C}' : \Gamma \vdash v \overset{v}{\sim} V$, and $\mathcal{C} : \Gamma \vdash (e, \lambda x. \text{vl}(x)) \overset{e}{\sim} E$ then $\mathcal{S} : E \mapsto^* \text{vl}(V)$.

Proof. By assumption we know that $\mathcal{D} : e \hookrightarrow v$, and $\mathcal{C}' : \Gamma \vdash v \overset{v}{\sim} V$, and $\mathcal{C} : \Gamma \vdash (e, \lambda x. \text{vl}(x)) \overset{e}{\sim} E$, and we also know that $\mathcal{S}' : (\underbrace{\lambda x. \text{vl}(x)}_{\overset{\beta}{\sim} \text{vl}(V)}) V \mapsto^* \text{vl}(V)$. By the

$\overset{\beta}{\sim} \text{vl}(V)$

previous lemma, we know that $\mathcal{S} : E \mapsto^* \text{vl}(V)$. \square

By proving the soundness and completeness of our CPS translation, we can show the correctness of our CPS transformation. We will present the formalization of our CPS translation and proofs in the next section.

4 Formalization in a meta-logical framework

From the previous section, one may have the feeling that, as transformations become more and more sophisticated, their hand-written correctness proofs become less and less readable and reliable. Thus, it is desirable to formalize and verify the correctness of program transformations such as CPS transformation automatically with theorem provers (Minamide & Okuma 2003) or other deductive systems.

We formalize our CPS translation in the meta-logical framework Twelf (Pfenning & Schürmann 1999, Pfenning & Schürmann 2002). It is a framework used for the specification, implementation, and meta-theory of deductive systems from the theory of programming languages and logics. The encoding style in Twelf is very intuitive and simple from a logic programming point of view, thus it is easier to understand. The formalization task consists of four stages:

1. The representation of the abstract syntax of the Mini-ML and CPS language.
2. The representation of the Mini-ML big-step operational semantics and the small-step reduction semantics of the CPS language.
3. The representation of the CPS transformation inference rules between the Mini-ML and CPS language.

4. The representation of meta-theory of the language (for example, the correctness proof of CPS transformation).

4.1 Formalization of the languages

Twelf employs the representation methodology and underlying type theory of the LF logical framework. The first task in the formalization of a language in a logical framework is the representation of its expressions. We base the representation on abstract (rather than concrete) syntax in order to expose the essential structure of the object language so we can concentrate on semantics and meta-theory, rather than details of lexical analysis and parsing. Expressions are represented as LF objects using the technique of *higher-order abstract syntax* whereby variables of an object language are mapped to variables in the meta-language. This means that common operations such as renaming of bound variables or capture-avoiding substitution are directly supported by the framework, so that we get bound variables for free, and do not need to implement any new capture-avoiding substitution for each object language.

The meta-language of LF is the $\lambda\Pi$ -calculus. It is a three-level hierarchical calculus for *objects*, *families* and *kinds*. Families are classified by kinds and objects are classified by types that are families of kind type (Pfenning 2001). So for each (abstract) syntactic category of the object language we introduce a new type constant in the meta-language via a declaration of the form `a:type`. Thus, in order to represent CPS language expressions and values we declare a type `cexp` and `cval` in the meta-language (the representation of the Mini-ML language is similar).

```
cexp : type.
cval : type.
```

We intend that every LF object M of type `cexp` represents a CPS language expression and vice versa. The zero constant `Zero` is now represented by an LF constant `Z` declared in the meta-language to be of type `cval`.

```
Z : cval.
```

The successor `Succ` is a value constructor. It is represented by a constant of functional type that maps CPS values to CPS values so that, for example, `Succ Zero` has type `cval`.

```
Succ : cval -> cval.
```

As discussed in Section 2 a continuation k is a function that takes in a CPS value and returns a CPS expression. So a continuation k has the type of `cval \rightarrow cexp`. Recall the application CPS expression `App(V_1 , V_2 , k)`, it has the type `cexp`.

```
App : cval -> cval -> (cval -> cexp) -> cexp.
```

Other constructs are defined in the same pattern. The definition of the abstract syntax of the CPS language is encoded in Twelf as follows:

```
cexp : type. %name cexp CE.
cval : type. %name cval CV.
```

```
% CPS expressions
```

```
app+ : cval -> cval -> (cval -> cexp) -> cexp.
fst+  : cval -> (cval -> cexp) -> cexp.
snd+  : cval -> (cval -> cexp) -> cexp.
case+ : cval -> cexp -> (cval -> cexp) -> cexp.
vl+   : cval -> cexp.
```

```
% CPS values
```

```
z+   : cval.
s+   : cval -> cval.
pair+ : cval -> cval -> cval.
lam+ : (cval -> (cval -> cexp) -> cexp) -> cval.
```

We annotate the CPS language terms with the symbol “+”, in order to distinguish them from the terms of the Mini-ML language.

4.2 Formalization of the operational semantics

For semantic specification, LF uses the *judgments-as-types* representation technique. This means that a derivation is coded as an object whose type represents the judgment it establishes. Checking the correctness of a derivation is thereby reduced to type-checking its representation in the logical framework (which is efficiently decidable) (Pfenning & Schürmann 1999, Pfenning & Schürmann 2002).

In the big-step operational semantics of Mini-ML, the evaluation judgment: $e \hookrightarrow v$ is defined in Twelf as the `eval` judgment.

```
eval : exp -> value -> type.
```

The `eval` predicate is very like a function, which takes in a Mini-ML expression and returns the value of evaluating this expression.

The following shows the Twelf encoding of the inference rules for `eval` of pairs and functions. The other cases follow the same pattern. Throughout the formalization in Twelf we reverse the function arrows writing $A_2 \leftarrow A_1$, instead of $A_1 \rightarrow A_2$ following logic programming notation. A more detailed discussion of this encoding style is given in (Pfenning 2001).

```
% Pairs
```

```
ev_pair : eval (pair E1 E2) (pair* V1 V2)
         <- eval E1 V1
         <- eval E2 V2.
ev_fst  : eval (fst E) V1
         <- eval E (pair* V1 V2).
ev_snd  : eval (snd E) V2
         <- eval E (pair* V1 V2).
```

```
% Functions
```

```
ev_lam : eval (lam E) (lam* E).
ev_app : eval (app E1 E2) V
         <- eval E1 (lam* E1')
         <- eval E2 V2
         <- eval (E1' V2) V.
```

The term `(E1' V2)` formalizes substitution by β -reduction. The expression `E1'` is of function type $(\lambda x.(E1'x))$, and we pass `V2` into the body of the function (`V2` substitutes the occurrence of variable x in `E1'`).

The single-step and multi-step reduction relations between two CPS expressions are formalized by two predicates in Twelf.

```
=> : cexp -> cexp -> type.
=>* : cexp -> cexp -> type.
```

All the reduction inference rules are expressed by:

```
%infix none 6 =>.
%infix none 5 =>*.
```

```
stop : E =>* E. % reflexivity
<< : E =>* E' % transitivity
    <- E => E1
    <- E1 =>* E'.
```

```
cred_case+_z : (case+ z+ E2 E3) => E2.
cred_case+_s : (case+ (s+ V1') E2 E3) => (E3 V1').
cred_fst+ : (fst+ (pair+ V1 V2) K) => (K V1).
cred_snd+ : (snd+ (pair+ V1 V2) K) => (K V2).
cred_app+ : (app+ (lam+ E1') V2 K) => (E1' V2 K).
```

The predicates `=>` and `=>*` are set as *infix* operators. The reflexivity rule is indicated as a stopped state.

4.3 Formalization of CPS transformation

The CPS transformation is formulated by two judgments: $\Gamma \vdash v \overset{v}{\sim} V$ and $\Gamma \vdash (e, k) \overset{e}{\sim} E$. We can interpret these two judgments by assuming that v , e and k are given as inputs and V , E are to be constructed as

outputs. The two judgments are formalized in Twelf by two predicates: `cpsExp` and `cpsValue`. The Twelf definitions of these two judgments follow the same as their type definitions:

```
cpsValue : value -> cval -> type.
cpsExp   : exp -> (cval -> cexp) -> cexp -> type.
```

The CPS transformation rules for Mini-ML values are expressed by:

```
cpsV_z*   : cpsValue z* z+.
cpsV_s*   : cpsValue (s* V) (s+ V+)
           <- cpsValue V V+.
cpsV_pair* : cpsValue (pair* V1 V2) (pair+ V1+ V2+)
           <- cpsValue V1 V1+
           <- cpsValue V2 V2+.
cpsV_lam* : cpsValue (lam* E) (lam+ E')
           <- ({x:value} {x':cval}
              cpsValue x x'
              -> {k:cval -> cexp}
              cpsExp (E x) k (E' x' k)).
```

In the case of CPS transformation for `lam*`, we implicitly create a new continuation `k`, which we use it to transform the function body `E` to its corresponding CPS expression, by plugging in a value into `E`. The term `{x:value}` means that for all `x` that is of type `value`. The big-brackets `{ }` in Twelf has the similar meaning to the universal quantifier (\forall) in first-order logic.

The following shows some cases of the formalization of CPS transformation rules for Mini-ML expressions in Twelf. The other cases follow the same pattern.

```
cps_pair: cpsExp (pair E1 E2) K E'
         <- ({x1':cval}
            cpsExp E2 ([x2':cval] K (pair+ x1' x2'))
            (E2' x1'))
         <- cpsExp E1 E2' E'.

cps_lam: cpsExp (lam E) K (K (lam+ E'))
         <- ({x:value} {x':cval}
            cpsValue x x'
            -> {k:cval -> cexp}
            cpsExp (E x) k (E' x' k)).

cps_app: cpsExp (app E1 E2) K E'
         <- ({x1:cval}
            cpsExp E2 ([x2:cval] app+ x1 x2 K) (E2' x1))
         <- cpsExp E1 ([x1:cval] E2' x1) E'.
```

The term `[x:cval]` is equal to λx , where x is of type `cval`. The predicate `cpsExp` can be viewed as a function, where its inputs are a Mini-ML expression e and a continuation (K), and its output is a CPS expression (of type `cexp`).

4.4 Formalization of the correctness proof

In Twelf we can express the meta-theory of deductive systems using *higher-level judgments* (Pfenning & Schürmann 1999). A higher-level judgment describes a relation between derivations inherent in a (constructive) meta-theoretic proof. So we can execute a meta-theoretic proof using the operational semantics for LF. Twelf checks the proof by type-checking the judgments. However, type-checking a higher-level judgment does not by itself guarantee that it correctly implements a proof.

Recall the Soundness lemma in Section 2. It is represented in Twelf as follows:

```
cpsd : cpsExp E K M
     -> M =>* (v1+ W)
     -> {V:value}eval E V
     -> {V+:cval}cpsValue V V+
     -> (K V+) =>* (v1+ W)
     -> type.
%name cpsd CS.
%mode cpsd +C +S -V -D -V+ -C' -S'.
```

The mode declaration:

```
%mode cpsd +C +S -V -D -V+ -C' -S'.
```

specifies the inputs and outputs of the predicate `cpsd`. The “+” mode indicates inputs and the “-” mode indicates outputs. This follows exactly the statements of the lemma, where all the assumptions are indicated as inputs whereas the conclusions are indicated as outputs. The mode checker verifies that all inputs are known when the predicate is called and all output arguments are known after successful execution of the predicate (Pfenning & Schürmann 2002). The predicate `cpsd` is defined just like what the Soundness lemma is stated. The inputs of the predicate assume that $C : \Gamma \vdash (e, k) \sim E$ and $S : E \mapsto^* \forall(W)$ exist, and the outputs are to be $D : e \hookrightarrow v$, $C' : \Gamma \vdash v \sim V$ and $S' : (k V) \mapsto^* \forall(W)$.

The following shows the formalization of the Soundness lemma proof for the cases of `pair`, `lam` and `app`. The other cases follow the same pattern.

```
% Pairs
cpsd_pair: cpsd (cps_pair ES1 ES2) C
           (pair* V1 V2) (ev_pair D2 D1)
           (pair+ V1+ V2+) (cpsV_pair* T2 T1) C2
           <- cpsd ES1 C V1 D1 V1+ T1 C1
           <- cpsd (ES2 V1+) C1 V2 D2 V2+ T2 C2.

% Functions
cpsd_lam: cpsd (cps_lam Es') C1 (lam* E) ev_lam
           (lam+ E') (cpsV_lam* Es') C1.
cpsd_app: cpsd (cps_app ES1 ES2) C
           V (ev_app D3 D2 D1) V+ T3 C3
           <- cpsd ES1 C
           (lam* E) D1 (lam+ E') (cpsV_lam* Es3) C1
           <- cpsd (ES2 (lam+ E')) C1
           V2 D2 V2+ T2 (C2 <- cred_app)
           <- cpsd (Es3 V2 V2+ T2 K) C2 V D3 V+ T3 C3.
```

To check the Twelf program actually constitutes a proof, meta-theoretic properties such as *coverage* and *termination* need to be established. Termination guarantees that the input of each recursive call (induction hypothesis) is smaller than the input of the original call (induction conclusion) (Pientka 2001, Pientka & Pfenning 2000). For termination checking the program needs to be well-modeled. As discussed in Section 3, we specify that the predicate `cpsd` should terminate in the arguments `S` and `C` by

```
%terminates {S C} (cpsd C S V D V+ C' S').
```

Atomic, lexicographic subterm ordering is indicated by `{S C}`. For reduction checking we specify an explicit order relation between input and output elements. In the Soundness lemma proof, we declare

```
%reduces S' <= S (cpsd C S V D V+ C' S').
```

to verify that S' is a rest computation of S .

Coverage says that the execution will always make progress. In order to correctly coverage check the proof, we need to give correct specification of relation `cpsd`'s mode, and the specification of the applicable *world*. Worlds in Twelf work very much like modes. When specifying a world for a relation, it does two things. First, it specifies the appropriate world for that relation; and second, it checks that the entire computation rooted at that relation (ie, that relation, and every other one it calls) is well-worlded (Harper & Crary 2005, Pfenning & Schürmann 2002). The world declaration is defined for `cpsd` as follows:

```
%worlds () (cpsd C S V D V+ C' S').
```

this is actually a special case of regular world declaration, which declares that the type families in `cpsd C S V D V+ C' S'` do not introduce any new parameters or hypotheses. After specifying the mode

and world declaration, we can then activate the coverage checker by specifying the coverage declaration:

```
%covers cpsd +C +S -V -D -V' -C' -S'.
```

The coverage checker only checks for “input” coverage, that is if all the possible cases for a given collection of input arguments are covered (Schürmann & Pfenning 2003). In order to check “output” coverage (the output arguments to the subgoal cover all the possible values that may be returned in these positions) (Pfenning & Schürmann 2002), we need to specify the “totality” checker as well:

```
%total {S C} (cpsd C S V D V+ C' S').
```

The Soundness theorem is expressed by:

```
sound: cpsExp E ([x:cval] vl+ x) E+
  -> {V+:cval}E+ =>* vl+(V+)
  -> {V:value}eval E V
  -> cpsValue V V+
  -> type.
%mode sound +C +V+ +S -V -D -T.
```

The proof is simple, since it is the corollary of the Soundness lemma.

```
sound_cps: sound C V+ S V D T
  <- cpsd C S V D V+ T stop.
```

The `stop` state indicates the reflexivity rule in the CPS expression reductions. In the CPS language small-step reduction semantics, we define that the CPS expression `vl(V)` cannot be reduced. However, in the formalization of the proof, we say that `vl(V)` can be reduced in one step to itself as the reduction stops.

The formalizations of the proofs for the Completeness lemma and theorem follow the same pattern as the Soundness ones, except we need to formalize the Totality lemma.

```
tcps: {E}{K}cpsExp E K E' -> type.
%mode tcps +E +K -D.

tcpsv: {V}cpsValue V V+ -> type.
%mode tcpsv +V -D'.
```

The predicate `tcpsv` says that for any Mini-ML value `V`, there exists a derivation `D'`, where `V` can be CPS transformed into a CPS value `V+`. The predicate `tcps` represents the same meaning for Mini-ML expressions.

The following shows two cases in the formalization of the proof for the Totality lemma, the other cases follow the same pattern.

```
tcps_pair: tcps (pair E1 E2) K (cps_pair D1 D2)
  <- tcps E1 K D1
  <- {x1':cval}
    tcps E2 ([x2':cval] K (pair+ x1' x2'))
    (D2 x1').

tcpsv_lam*: tcpsv (lam* E) (cpsV_lam* D')
  <- ({x:value}{x':cval}{u:cpsValue x x'}
    {k:cval -> cexp}
    tcps (E x) k (D' x x' u k)).
```

The Twelf code of the CPS translation (including all the formalization of the four stages) can be found at <http://www.cs.mcgill.ca/~ytian8/CPS/>.

5 Conclusions and Future Work

People from the research community of the programming languages are now interested in a question (Aydemir, Bohannon, Fairbairn, Foster, Pierce, Sewell, Vytiniotis, Washburn, Weirich & Zdanczewicz 2005): How close are we to a world where programming language papers are routinely supported

by machine-checked metatheory proofs, where full-scale language definitions are expressed in machine-processed mathematics, and where language implementations are directly tested against those definitions? This paper is intended to be a guide to a formal development of CPS compilation techniques as well as a case study of the question above.

We have considered a typical example from verifying the correctness of abstract machines (Hannan & Pfenning 1992). To prove correctness, we need to show the correspondence between the high-level language and the low-level abstract machine. In other words, we can translate programs written in the high-level language into programs which run on the abstract machine. Moreover, the source program always has the same observable behavior as the target program. The proof is constructive and constitutes a program which translates derivations in the source language into derivations of the target language and vice versa. We consider Mini-ML as the source language, and the terms in continuation-passing style as the low-level target language.

We have presented a higher-order setting of CPS transformation, which operates in one-pass and directly produces compact CPS programs without administrative redexes (Danvy 1991, Danvy & Nielsen 2002, Danvy & Nielsen 2001). This higher-order CPS transformation also simplified the process of proving the correctness of CPS transformation, as we do not need to use colon-translation to handle those administrative redexes. We have encoded the CPS translation and the correctness proofs in the metalogical framework Twelf using higher-order abstract syntax. We have mechanically verified the correctness of our CPS transformation and also other properties like “termination” of our CPS transformation and “uniqueness” of the CPS expression reductions.

This paper has showed a lot of benefits of using meta-logical approach of formalizing metatheory proofs of programming languages and program compilations. People have also demonstrated that using meta-logical approach enables relatively rapid development of foundational certified code (Crary & Sarkar 2003). We are interested in exploring the power of using meta-logical approach of formalizing and verifying theory foundations of programming languages in the future.

6 Acknowledgements

This work could not have been accomplished without the insights, persistence and critical feedback of my advisor Brigitte Pientka.

References

- Appel, A. W. (1992), *Compiling with Continuations*, Cambridge University Press.
- Aydemir, B. E., Bohannon, A., Fairbairn, M., Foster, J. N., Pierce, B. C., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S. & Zdanczewicz, S. (2005), Mechanized metatheory for the masses: The POPLmark Challenge, in ‘Proceedings of TPHOLs 2005: the 18th International Conference on Theorem Proving in Higher Order Logics (Oxford)’.
- Crary, K. & Sarkar, S. (2003), Foundational certified code in a metalogical framework., in ‘CADE’, pp. 106–120.
- Danvy, O. (1991), Three steps for the cps transformation, Technical Report CIS-92-2, Department

- of Computing and Information Sciences, Kansas State University.
- Danvy, O., Dzafic, B. & Pfenning, F. (1999), On proving syntactic properties of CPS programs, in 'Proceedings of the Third International Workshop on Higher Order Operational Techniques in Semantics (HOOTS'99)', Paris, pp. 19–31. Electronic Notes in Theoretical Computer Science, Volume 26.
- Danvy, O. & Filinski, A. (1992), 'Representing control: A study of the CPS transformation', *Mathematical Structures in Computer Science* **2**(4), 361–391.
- Danvy, O. & Nielsen, L. R. (2002), A first-order one-pass cps transformation, in 'Proceedings of Foundations of Software Science and Computation Structures, FOSSACS', 2303 of LNCS, pp. 98–113.
- Danvy, O. & Nielsen, L. R. (2001), A higher-order colon translation, in 'Proceedings of International Symposium on Functional and Logic Programming, FLOPS', pp. 78–91.
- Guy L. Steele, J. (1978), Rabbit: A compiler for scheme, Technical report, Cambridge, MA, USA.
- Hannan, J. & Pfenning, F. (1992), Compiler verification in LF, in A. Scedrov, ed., 'Seventh Annual IEEE Symposium on Logic in Computer Science', Santa Cruz, California, pp. 407–418.
- Harper, R. & Crary, K. (2005), How to believe a twelf proof.
*<http://www-2.cs.cmu.edu/~rwh/papers/how/believe-twelf.pdf>
- Minamide, Y. & Okuma, K. (2003), Verifying cps transformations in isabelle/hol, in 'Proceedings of the 2003 Workshop on Mechanized Reasoning about Languages with Variable Binding', pp. 1–8.
- Pfenning, F. (2001), *Computation and Deduction*, Cambridge University Press. In preparation. Draft from April 1997 available electronically.
- Pfenning, F. & Schürmann, C. (1999), System description: Twelf — a meta-logical framework for deductive systems, in 'Proceedings of the 16th International Conference on Automated Deduction (CADE-16)', Springer-Verlag LNAI 1632, Trento, Italy, pp. 202–206.
- Pfenning, F. & Schürmann, C. (2002), *Twelf User's Guide*, 1.4 edn. Available as Technical Report CMU-CS-98-173, Carnegie Mellon University.
- Pientka, B. (2001), Termination and reduction checking for higher-order logic programs, in 'IJCAR '01: Proceedings of the First International Joint Conference on Automated Reasoning', Springer-Verlag, London, UK, pp. 401–415.
- Pientka, B. & Pfenning, F. (2000), Termination and reduction checking in the logical framework, in C. Schürmann, ed., 'Workshop on Automation of Proofs by Mathematical Induction', Pittsburgh, Pennsylvania.
- Plotkin, G. (1975), 'Call-by-name, call-by-value and the λ -calculus', *Theoretical Computer Science* **1**, 125–159.
- Schürmann, C. & Pfenning, F. (2003), A coverage checking algorithm for lf, in 'TPHOLS', pp. 120–135.
- Xi, H. & Schürmann, C. (2001), 'CPS Transform for Dependent ML (abstract)', *Logic Journal of IGPL* **9**(5), 739–754.

Formalising the L4 microkernel API

Rafal Kolanski

Gerwin Klein

National ICT Australia Ltd. (NICTA)
Locked Bag 6016
The University of New South Wales
Sydney NSW 1466
Australia

Email: {rafal.kolanski|gerwin.klein}@nicta.com.au

Abstract

This paper gives an overview of a pilot project on the specification and verification of the L4 high-performance microkernel. Of the three aspects examined in the project, we describe one in more detail: the formalisation of the kernel's Application Programming Interface using the B Method. We conclude that machine-supported formal verification of software is at a turning point; that it is now feasible, and desirable, to formally verify production-quality operating systems.

Keywords: B Method, Operating System Specification, Software Verification

1 Introduction

The operating system (OS) kernel is defined to be the part of the OS that runs in the privileged mode of the hardware and thus is able to bypass hardware protection mechanisms. A microkernel is a kernel designed to be minimal in code size and concepts.

L4 is a second generation microkernel [14]. It provides the traditional advantages of the microkernel approach to system structure, namely improved reliability and flexibility, while overcoming the performance limitations of the previous generation of microkernels. With implementation sizes in the order of 10,000 lines of C++ and assembler code it is an order of magnitude smaller than Mach and two orders of magnitude smaller than Linux.

The correctness and reliability of any nontrivial system clearly critically depends on the operating system and its kernel. In terms of security, the OS is part of the trusted computing base, that is, the hardware and software necessary for the enforcement of a system's security policy. It has been repeatedly demonstrated that current operating systems fail at correctness, reliability, and security. Microkernels address the problem by applying the principles of minimality and least privilege to OS architecture. To gain confidence in the overall system, it is therefore highly desirable to formally verify the correctness of this design and its implementation.

Copyright ©2006, Australian Computer Society, Inc. This paper appeared at Computing: The Australasian Theory Symposium (CATS2006), Hobart, Australia. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 51. Barry Jay and Joachim Gudmundsson, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.

National ICT Australia is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through *Backing Australia's Ability* and the ICT Research Centre of Excellence programs.

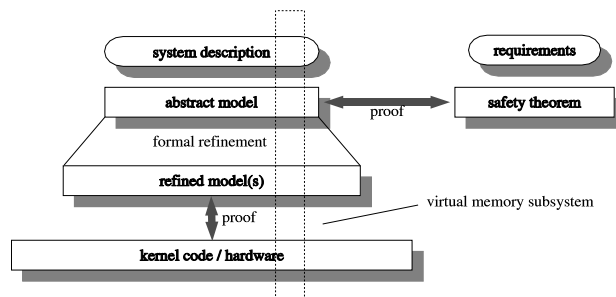


Figure 1: Overview

The L4 kernel is of a size which makes formalisation and verification feasible. Compared to other OS kernels, L4 is very small; compared to the size of other verification efforts, 10,000 lines of code is still considered a very large and complex system. Our methodology for solving this verification problem is shown in figure 1. It is a classic refinement strategy. We start out from an abstract model of the kernel that is phrased in terms of user concepts as they are explained in the L4 reference manual [13]. This is the level at which most of the safety and security theorems will be shown. We then formally refine this abstract model in multiple property preserving steps towards the implementation of L4. The last step consists of verifying that the C++ and assembler source code of the kernel correctly implements the most concrete refinement level. At the end of this process, we will have shown that the kernel source code satisfies the safety and security properties we have proved about the abstract model.

We conducted a pilot project to judge the feasibility of this verification task. The project investigated three main aspects: a formalisation of the kernel's Application Programming Interface (API) using the B Method (the first horizontal formalisation layer in figure 1), a full refinement proof for a non-trivial subsystem of the kernel using Isabelle/HOL (the vertical slice in figure 1), and a literature survey on formalising safety and security properties on the design level (the right-hand side of figure 1).

In this paper we give an overview of the first of these aspects: the API formalisation using the B Method, depicted as *abstract model* in figure 1. The L4 API provides three basic abstractions: threads, synchronous inter process communication (IPC), and virtual memory management (VMM). Our formalisation covers threads and IPC in detail and contains the basic structure for VMM. The latter has been formalised in depth in the vertical slice part of the project and is already described in earlier publications [20, 10]. Our formalisation is based on release version 0.3 of the L4Ka::Pistachio implementa-

tion [12].

We chose the B Method [1], because there existed a significant amount of experience with this approach among our student population and we wanted to compare at least two different formalisms before embarking on the full verification task. The B Method is a formal development methodology based on set theory with first-order logic. It allows progress from an initial high-level specification all the way to implementation via formal refinement. In this part of the project we have not done any formal refinement, but used the B Method and tool for formalisation only. The B Toolkit [2] allows for animation of the top-level specification which makes validating the specification more convenient. In this mode, the user becomes the implementation of all non-deterministic or undefined aspects.

After reviewing related work in section 2 and introducing B concepts and notation in section 3, we describe our formalisation of the L4 API in section 4. Section 5 gives pointers to further work and concludes.

2 Related Work

Earlier work on operating system kernel formalisation and verification includes PSOS [15] and UCLA Secure Unix [22]. The focus of this work was on capability-based security kernels, allowing security policies such as multi-level security to be enforced. These efforts were hampered by the lack of mechanisation and appropriate tools available at the time and so while the designs were formalised, the full verification proofs were not practical. Later work, such as KIT [3], describes verification of properties such as process isolation to source or object level but with kernels providing far simpler and less general abstractions than modern microkernels. There exists some work in the literature on the modelling of microkernels at the abstract level with varying degrees of completeness. Bevier and Smith [4] specify legal Mach states and describe Mach system calls using temporal logic. Shapiro and Weber [18] give an operational semantics for EROS and prove a confinement security policy. A number of case studies [6, 5, 21] in the literature describe the IPC and scheduling subsystems of microkernels in PROMELA and verify the formal descriptions with the SPIN model checker. These abstractions were not necessarily sound, having been manually constructed from the implementations, and so while useful for discovering concurrency bugs do not provide guarantees of correctness.

The VFiasco project, working with the Fiasco implementation of L4, has published exploratory work on the issues involved in C++ verification at the source level [9]. The VeriSoft project [8] is attempting to verify a whole system stack, including hardware, compiler, applications, and a simplified microkernel called VAMOS that is inspired by, but not very close to L4. While the simplifications are appropriate for the goals of VeriSoft, it is doubtful that the VAMOS kernel will show the necessary performance to be relevant for industrial use.

Spivey uses Z, a predecessor formalism of B, to specify a simple kernel for a safety-critical X-ray diagnostic machine [19]. In abstracting the kernel from its implementation and documenting it for future reimplementations (possibly on different architectures), he finds a flaw in the system that could potentially have caused the X-ray machine to inflict damage.

The more academic approach of using formal design and specification in the kernel development process up front and *then* proceeding with the implementation is utilised to good effect by Fowler and

Wellings [7] for an Ada95 runtime support system in a hard real-time environment. From the verification perspective, this approach is more efficient than the post-hoc formalisation that is commonly found and which we are presenting here. The drawback is that while this process ensures a correct kernel, it is hard to get the runtime performance that separates practical microkernels from impractical ones.

In fact, we propose to do post-hoc formalisation of the existing L4 microkernel whose architecture has proven to deliver the required performance, but for the verification task itself we reserve the freedom to change details in the code base that make the verification process easier.

3 Notation

At the top specification level, the B Method uses *machines*, which represent finite state automata. *Refinements* further refine these, and *implementations* are the most concrete in the chain. Since our formalisation is entirely contained at the top level, we will only describe machine notation. A machine consists of the following sections:

DEFINITIONS They are purely syntactic translations. Any single-letter token counts as a so-called joker and can represent any set of tokens, similar to `#define` in C and C++. One definition cannot use another one within the same machine.

VARIABLES A comma separated list of variables.

SETS Enumerations and abstract sets.

CONSTANTS Declares constant sets, members of sets, or functions (which are also represented as sets).

PROPERTIES Restrictions on sets and constants.

INVARIANT The invariants of the machine, used to define variable types, properties, and relationships.

INITIALISATION Initial values for variables.

OPERATIONS The state transitions of the machine. At the abstract machine level, only *parallel* composition is allowed, i.e. all statements in the operation (including invoking other operations) occur at the same time; the operation itself is instantaneous. An operation may only invoke operations in other machines, and only when permitted by inter-machine relationships. Operations can have preconditions.

The relationship between machines is restricted. A machine may *INCLUDE* (read-access to everything plus invoking operations), or *SEE* (read-access only to sets and constants) other machines. Write-access is only permitted through operations. If machine X includes Y, it can select which of Y's operations are visible when another machine includes X with *PRO-MOTES*.

Since there is only one name space in B, we use naming conventions to avoid collisions. We use prefixes for all enumerations and a 'd' prefix for most definitions, as well as long classifying names such as *thread_ipc_waiting_timeout*.

Robinson provides a good reference to B syntax [17]. We define a few core notational concepts here and explain other non-standard notation as it occurs.

B supplies built-in sets such as the natural numbers, *NAT* and *NAT1* ($\mathbb{N} - \{0\}$). The library also

provides machines defining sets such as *INTEGER* and *BOOL* = {*FALSE*, *TRUE*}.

Two frequently used operations are the relational image and domain restriction. The relational image of set *S* under relation *r* is defined as:

$$r[S] = \{ y \mid \exists x \cdot x \in S \wedge x \mapsto y \in r \}$$

If *r* is a function, this means all *y* such that *r*(*x*) = *y*. The pair (*x*, *y*) is denoted *x* \mapsto *y*.

For a relation *r* and set *S*, the domain restriction operator (\triangleright) is defined as follows:

$$r \triangleright S = \{ x \mapsto y \mid x \mapsto y \in r \wedge y \in S \}$$

The predefined functions **dom** and **ran** return the domain and range of relations and functions; **card** is the the cardinality of a set.

4 The Formalisation

This section describes the formalisation of the L4 microkernel API. As far as possible, we will introduce the kernel concepts together with their formal counterparts. The goals of the formalisation were the following.

Learning Animation of the model to improve and accelerate understanding of L4 internals for new users and developers.

Documentation L4 is continually developed and improved for efficiency. Boiling down the system to its essentials in the form of an abstract model will help to document and clarify the initial intentions and underlying basic mechanisms.

Experimentation The current version of the L4 API lacks an efficient communication restriction for information flow and also is vulnerable to denial of service attacks on kernel memory resources. One of the follow-up projects to this formalisation is revising the L4 API to fix these shortcomings. It is one of the goals of the formalisation presented here to serve as an experiment in kernel modelling to find out which methods work well.

The API is the boundary between user space and kernel space. When building a model, the question is *Whose viewpoint do we model?*

From the perspective of a thread running in the system, kernel operations are system calls. They return values and they return them immediatly.

From the kernel's perspective, however, internal state changes are visible. For instance, the kernel might pick out the parameters from the thread's registers and memory, then pass them to an internal operation which implements the required functionality. The operation does not have to return immediately. The kernel can freeze the thread, change its state, put it on a waiting queue and so forth. The system call also does not simply return a value internally, but instead copies return values into the thread's registers and memory.

To document the kernel behaviour in detail, we chose the second viewpoint. It allows, for example, modelling of thread state transitions in a natural way. Taking the inside view does not mean that we are exposing implementation details of the API — the formalisation remains at the conceptual level of a reference manual. The API for instance already implies that the kernel manages thread control states and the formalisation describes how these states are affected by operations. The formalisation does not describe which data structures are used to implement thread states in the kernel.

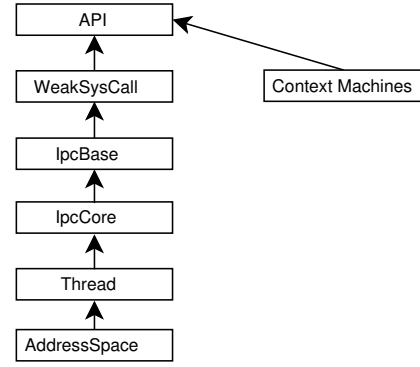


Figure 2: Inclusion diagram for the B development.

Figure 2 shows the module structure of the formalisation. Microkernels strive for the minimal set of functionality that is sufficient to build an OS. This means that the L4 kernel is effectively a single module in which everything is intertwined. We were able to separate out B modules for each of the major subsystems (address spaces, threads, IPC), but as figure 2 shows, they still depend on each other.

In the formalisation, we have placed all types and constants in separate context machines. There is one such context machine for each machine containing operations. In this presentation, we summarise these under the label *ContextMachines* and describe them in section 4.1. The rest of the presentation follows figure 2 bottom up. In section 4.2 we describe the address space stub (this subsystem has already been formalised separately), in section 4.3 the concept of threads and in section 4.4 the inter process communication subsystem. We leave out the description of *WeakSysCall* which collects these together into the L4 API functions, but does not contain any precondition checking yet. Section 4.5 describes the final interface available to the user. Due to space constraints, our description does not cover all of the formalisation at the same level of detail. The full formalisation is available elsewhere [11].

4.1 Context Machines

This section defines the basic types and constants used in the rest of the formalisation.

In addition to type information, all systems manage a finite set of resources. By defining abstract sets of *things* (such as thread numbers) and restricting their cardinality, we implicitly define an upper limit on the number of such things in the system.

In L4, a structure called the *Kernel Information Page* (KIP) contains all the constant values in the system (how many interrupts, first id of a user thread, etc.) The context machines serve a similar purpose.

We start off with the three main limiting aspects of the kernel: the number of threads in the system (*kMaxThreads*), the number of address spaces in the system (*kMaxAddressSpaces*), and the number of threads in an address space (*kMaxThreadsPerSpace*). These three constants have the following properties:

PROPERTIES

$$\begin{aligned}
 &kMaxThreads \in \mathbb{N}_1 \wedge \\
 &3 \leq kMaxThreads \wedge \\
 &kMaxThreadsPerSpace \in \mathbb{N}_1 \wedge \\
 &kMaxAddressSpaces \in \mathbb{N}_1 \wedge \\
 &3 \leq kMaxAddressSpaces
 \end{aligned}$$

Each thread must have an address space; an address space can only be created by also creating a

thread [13, section 2.4]. There are three address spaces initially in the system: the sigma0 space, the root server space and the kernel space. The minimum number of address spaces is therefore 3, and the same goes for threads. Hence, the maxima must be at least 3, too.

In order to talk about address spaces within the model, we define the abstract set of all possible address spaces and restrict them to the maximum number of address spaces in the system:

SETS

ADDRESS_SPACE

PROPERTIES

```
card ( ADDRESS_SPACE ) =
  kMaxAddressSpaces ∧
  kRootServerSpace ∈ ADDRESS_SPACE ∧
  kSigma0Space ∈ ADDRESS_SPACE ∧
  kKernelSpace ∈ ADDRESS_SPACE ∧
  kRootServerSpace ≠ kSigma0Space ∧
  kSigma0Space ≠ kKernelSpace ∧
  kRootServerSpace ≠ kKernelSpace
```

In the above, we define three new constants. Their function is to reserve three arbitrary members of *ADDRESS_SPACE* for the three core address spaces mentioned before: *kSigma0Space*, *kRootServerSpace*, and *kKernelSpace*.

These address spaces have special status in L4, they are privileged:

DEFINITIONS

```
dIsPrivilegedSpace ( s ) ≡
  s ∈ { kSigma0Space , kRootServerSpace ,
        kKernelSpace }
```

The following constants describe the control states that threads can experience in L4:

tsAborted the thread exists, but has not been initialised

tsRunning the thread has been initialised and if scheduled, can run

tsPolling thread is waiting on an IPC send to another thread

tsWaitingTimeout thread is waiting for incoming IPCs from one or more threads, with a finite time-out

tsWaitingForever as above, but the time-out is infinite

Figure 3 presents an overview of the possible transitions between these states. We show a complete diagram below in figure 4, section 4.4.3.

These states differ from the ones in the L4 implementation in following ways:

- Multiprocessing-related states are missing since our model is too abstract to demonstrate effects of multiple-CPU interaction;
- The halted state is missing. According to discussions with the L4 developers, this state is better modelled by a flag. As defined in [13, section 2.3], halting a thread prevents it from executing in user mode, while ongoing IPC is not affected. This means that it simply prevents the thread from being scheduled. Furthermore, the *EXCHANGEREGISTERS* system call needs to resume halted threads, creating the need for another (saved) thread state. This preserves functionality, but makes for a simpler model;

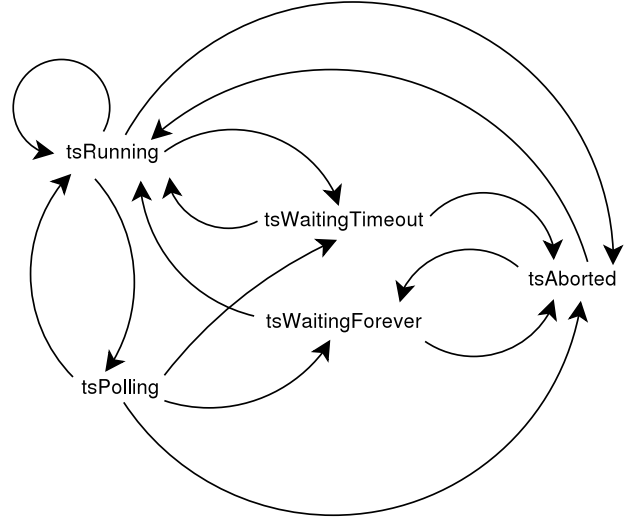


Figure 3: A simplified diagram of possible thread state transitions.

- The aborted state has a slightly different meaning than in the L4 implementation. In L4, all kernel thread control blocks are preallocated and their initial state is aborted. When a thread gets created inactive, the state *remains* aborted. The actual existence of a thread is defined as the thread having been assigned to an address space. Deleting a thread involves deleting this assignment. In our model, the non-existence of a thread is marked by its absence from the set of existing threads, so the threads do not have any actual state. Once the thread is created inactive, the two viewpoints merge.

We now come to the IPC related constants:

DEFINITIONS

```
canSend ( t ) ≡
  thread_state ( t ) ∈ { tsRunning , tsPolling } ;
canReceive ( t ) ≡
  thread_state ( t ) ∈ { tsWaitingTimeout ,
                        tsWaitingForever }
```

To send an IPC, a thread must either be running (it invokes the IPC) or polling (the kernel invokes the IPC on behalf of the thread). To receive one, it must be waiting.

The set *TCB* represents all possible threads creatable in the system. We have chosen this name due to its similarity to the pre allocated Kernel Thread Control Blocks in the system. The constants *kSigma0* and *kRootServer* reserve two distinct members of *TCB* for sigma0 and the root server threads, respectively.

Additionally, the constant *kIntThreads* reserves a subset of *TCB* for interrupt threads as follows:

```
kIntThreads ⊂ TCB ∧
kIntThreads ≠ {} ∧
card ( kIntThreads ) ≤ kMaxThreadsPerSpace ∧
kSigma0 ∉ kIntThreads ∧
kRootServer ∉ kIntThreads
```

The constant *kIntThreads* is a proper subset of *TCB*, of which *kRootServer* and *kSigma0* are not members. Since interrupt threads go in the kernel address space, there must not be more than *kMaxThreadsPerSpace* of them. There must be at least one interrupt thread in the system.

The set *EXREGS_FLAGS* defines the various options that can be passed into the *EXCHANGEREGISTERS* system call [13, section 2.3]: *ex_h* represents *h*,

ex. R represents R and so on for all the flags: *hpu-fisSRH*. We explain the meaning of these flags below when we introduce the corresponding operations.

Now that the thread context is defined, we can model thread identifiers, the user's view of threads. We are leaving out local thread identifiers and thread versions, as they both are mainly a performance optimisation and do not extend the behaviour of the kernel. They can be added as a separate concept during refinement later. Thus, the set *GLOBAL.TNO* represents all possible global thread identifiers. The constants *kAnyGNo* and *kNilGNo* represent *anythread* and *nilthread* respectively. There must be enough thread numbers for all threads plus two for the aforementioned constants, making the set cardinality $kMaxThreads + 2$. We omit the obvious definition in B.

The last set of constants concerns time-outs. Since the actual values of time-outs are irrelevant at this level of abstraction, we only define three predicates:

isNoTimeout requests an action be taken (or it will fail) immediately

isFiniteTimeout means that the thread will wait or poll for some time until timed out by the kernel, or cancelled by another thread

isInfiniteTimeout indicates that unless the operation is cancelled, the wait will go on indefinitely

We leave out the enumeration of error messages. It suffices to know that there is a set *ERROR* listing all of them. Also, *dIpcFailures* lists the failures during IPC that are beyond the deterministic control of the abstract model. If IPC fails non-deterministically, one of these will occur.

4.2 Address Spaces

Once the context is set up, the first important aspect of L4, on which all other aspects are based, is address spaces. Since this model does not go into the details of memory management, it suffices to model which spaces are used by the system and which of those have been initialised. The model mainly consists of three operations *CreateAddressSpace*, *InitialiseAddressSpace*, and *DeleteAddressSpace*, which we describe below.

The *AddressSpaces* machine *SEES* the context machines described above, importing their abstract sets and constants and introduces two new variables:

spaces representing the address spaces that have been created, and

initialised_spaces representing the address spaces that are created and initialised

Their relationship is defined as follows:

INVARIANT

$$\begin{aligned} spaces &\subseteq ADDRESS_SPACE \wedge \\ initialised_spaces &\subseteq spaces \end{aligned}$$

There cannot be more address spaces created in the system than the system can hold, nor can more be initialised than have been created. Being initialised implies being created.

Every variable in B must be initialised in a manner that establishes the invariant. In the context machines three address spaces were reserved: *kSigma0Space*, *kRootServerSpace*, and *kKernelSpace*. These are the spaces created and initialised by the root task on start up:

INITIALISATION

```
spaces := { kSigma0Space,
            kRootServerSpace, kKernelSpace } ||
initialised_spaces := { kSigma0Space,
                       kRootServerSpace, kKernelSpace }
```

The operator \parallel denotes parallel composition.

Next, we define the three operations that modify the state (variables) of this machine in the *OPERATIONS* clause. Since the operations are designed in such a way that satisfying their preconditions guarantees success, they do not return any values for error reporting.

The three operations are creating an address space, initialising it, and deleting it. Once an address space is initialised it cannot be uninitialised.

```
CreateAddressSpace ( space )  $\hat{=}$ 
  PRE  space  $\in$  ADDRESS_SPACE - spaces
  THEN
    spaces := spaces  $\cup$  { space }
  END
```

To guarantee the success of this operation, the address space identifier passed in must be one of those not yet created. This becomes the precondition. Once the precondition is satisfied, the new identifier is added to the set of created address spaces. In the user visible L4 API there do not exist any real address space identifiers. Address spaces are referred to implicitly by the ID of any thread running in the address space. More than one thread ID can refer to the same address space. Internally, address spaces are identified by just pointers to address space structures which is what the space identifiers in this formalisation correspond to.

Initialisation is easy as well. If the space identifier is one of those already created, the operation succeeds and adds the identifier to the set of initialised address spaces:

```
InitialiseAddressSpace ( space )  $\hat{=}$ 
  PRE  space  $\in$  spaces THEN
    initialised_spaces := initialised_spaces  $\cup$  { space }
  END
```

The final operation is deletion of an address space. To satisfy the invariant, it suffices that any member of *ADDRESS_SPACE* be passed in. For the operation to make sense, however, invoking it should only have meaning for an existing address space. Additionally, L4 does not allow deletion of privileged threads [12, SYS.THREAD_CONTROL in thread.cc], which means it does also not allow deletion of privileged address spaces.

```
DeleteAddressSpace ( space )  $\hat{=}$ 
  PRE  space  $\in$  spaces  $\wedge$ 
         $\neg$  ( dIsPrivilegedSpace ( space ) )
  THEN
    spaces := spaces - { space } ||
    initialised_spaces :=
      initialised_spaces - { space }
  END
```

As address space identifiers are not visible on the user level, address spaces are deleted implicitly when the last thread running in an address space is deleted. That means *DeleteAddressSpace* (as the other operations in this machine) are not visible at the API top level, but rather provide functionality for the rest of the formalisation.

4.3 Threads

Thread functionality is divided into three machines:

Thread contains all aspects of threads not directly related to IPC (such as state, pagers, schedulers, etc.)

IpcCore contains the place holder for an operation copying one thread's virtual registers onto another

IpcBase contains IPC-related aspects of threads, such as which thread is waiting on another.

This section describes the *Thread* machine. Due to the kernel being extremely intertwined, the layered approach imposed by B caused a more complex structure than what would be expected from a greatly simplified specification.

We abstract away from the concept of processors and a currently running thread. One can look at it as a magical machine on which each thread has its own processor to execute on. That means, from a thread's perspective, suspension from execution is essentially transparent in our model.

The machine *INCLUDES* all the functionality in *AddressSpace*, and *PROMOTES* the *InitialiseAddressSpace* operation so that higher-level machines can call it whenever an address space needs to be created.

4.3.1 Variables

In the invariant, we proceed to define the meaning of the machine and its variables. We begin with threads and their subsets:

$$\begin{aligned} threads &\subseteq TCB \wedge \\ halted_threads &\subseteq threads \wedge \\ active_threads &\subseteq threads \wedge \\ kSigma0 &\in active_threads \wedge \\ kRootServer &\in active_threads \wedge \\ kIntThreads &\subseteq active_threads \end{aligned}$$

Where *threads* are the threads that have been created, *active_threads* are those that have been activated. The privileged threads are implicitly initialised when the kernel starts and cannot be uninitialised. Halted threads may not enter user mode. Interrupt threads have been assigned different halting semantics by the L4 kernel designers. See *thread_state* below.

All threads in L4 are uniquely identified by their thread numbers, with two arbitrarily reserved to represent any thread and no thread respectively:

$$\begin{aligned} thread_gno &\in threads \mapsto GLOBAL_TNO \wedge \\ kAnyGNo &\notin ran (thread_gno) \wedge \\ kNilGNo &\notin ran (thread_gno) \end{aligned}$$

We define a total injective function mapping threads to thread numbers, but excluding reserved numbers from its range. Note that in B, instead of a type declaration, we say that a function is a member of the set of all functions meeting given constraints.

Next we define the relationship between the existing threads and address spaces:

$$\begin{aligned} thread_space &\in threads \mapsto spaces \wedge \\ thread_space (kSigma0) &= kSigma0Space \wedge \\ thread_space (kRootServer) &= kRootServerSpace \wedge \\ thread_space [active_threads] &\subseteq initialised_spaces \wedge \\ thread_space [kIntThreads] &= \{ kKernelSpace \} \wedge \\ thread_space^{-1} [\{ kKernelSpace \}] &= kIntThreads \end{aligned}$$

In L4, a created thread must have an address space. In fact, the address space pointer in the TCB is what defines whether a thread exists or not. Furthermore, there are no explicit address space identifiers; one specifies a thread in the address space instead. Hence, address spaces cannot be empty and *thread_space* is total as well as surjective (denoted \twoheadrightarrow). For a thread to be active it must reside in an initialised address space.

Interrupt threads are only an abstraction of the underlying hardware interrupts, and cannot actually run or have an implementation. The kernel space is therefore allocated to them.

Each thread has two other threads associated with it. These are the thread's scheduler and pager. The former is permitted to change the thread's scheduling-related properties, while the latter is invoked if the thread causes a page fault.

Let us look at the scheduler first:

$$\begin{aligned} thread_scheduler &\in threads - kIntThreads \rightarrow TCB \wedge \\ thread_scheduler (kSigma0) &= kRootServer \wedge \\ thread_scheduler (kRootServer) &= kRootServer \end{aligned}$$

Interrupt threads, naturally, cannot be scheduled. Since L4 does not keep track of schedulers, the range of *thread_scheduler* is not the set of existing threads, but can in fact be any TCB. The root server is traditionally the scheduler for sigma0 and itself. We believe that this situation should be maintained at all times, since otherwise the privileged threads could lose control of the system. The L4 source code does not, at this time, contain checks for this.

In L4, the process of page faults is resolved via IPC, i.e. a faulting thread needs a target to 'send' to (it is the kernel, however, which really performs the action on the thread's behalf). This target is known as the thread's *pager*:

$$\begin{aligned} thread_pager &\in threads \twoheadrightarrow TCB \wedge \\ kSigma0 &\notin dom (thread_pager) \wedge \\ \forall kk . (kk \in kIntThreads \wedge kk \notin halted_threads \Rightarrow \\ &\quad thread_pager (kk) = kk) \wedge \\ \forall kk . (kk \in kIntThreads \wedge kk \in halted_threads \Rightarrow \\ &\quad thread_pager (kk) \neq kk) \end{aligned}$$

The function is partial (\twoheadrightarrow) since sigma0, being the initial system pager, does not have another pager to fall back on. Additionally, until the thread is activated, the pager field in its TCB is meaningless (in fact, setting a valid pager constitutes activation). Similar to *thread_scheduler* the range of *thread_pager* cannot be enforced, since the thread's pager may have been deleted and is not necessarily valid. Interrupts are enabled by setting the corresponding interrupt thread's *halted* flag and setting its pager to something other than itself. When disabled, the thread's pager must be the thread itself.

All threads in the system must be in one of the known states:

$$\begin{aligned} thread_state &\in threads \rightarrow THREAD_STATE \wedge \\ active_threads \cap thread_state^{-1} [\{ tsAborted \}] &\subseteq \\ &\quad kIntThreads \wedge \\ tsRunning &\notin thread_state [kIntThreads] \wedge \end{aligned}$$

The *aborted* state and a thread being active are mutually exclusive, with the exception of interrupt threads, which do not achieve a *running* state under any circumstances. Since they participate in IPC, they can assume waiting and polling states, but once IPC is resolved they return to *aborted*. A probable reason for this is efficiency: since the scheduler only looks for *running* threads to execute, it will automatically overlook interrupt threads, at the price of making the interrupts-as-threads abstraction less

complete. See figure 3 for a diagram of possible state transitions.

Finally, to make the specification simpler to read, we have a variable *threads.in.space* keeping a separate count of how many threads are in each address space. Its range is $0 \dots kMaxThreadsPerSpace$.

4.3.2 Initialisation

We begin initialisation by creating *sigma0*, the root server and the interrupt threads. They are created active:

$$\begin{aligned} threads &:= \{ kSigma0, kRootServer \} \cup kIntThreads \parallel \\ active_threads &:= \{ kSigma0, kRootServer \} \\ &\quad \cup kIntThreads \end{aligned}$$

They will be in the address spaces *kSigma0Space*, *kRootServerSpace* and *kKernelSpace* respectively:

$$\begin{aligned} thread_space &:= \{ kSigma0 \mapsto kSigma0Space, \\ &\quad kRootServer \mapsto kRootServerSpace \} \\ &\quad \cup kIntThreads \times \{ kKernelSpace \} \end{aligned}$$

Note that the Cartesian product of *kIntThreads* and the singleton set $\{kKernelSpace\}$ is a function mapping all the interrupt threads to that space.

L4 initialises interrupt threads on first activation transparently to the user. Therefore it is not possible to tell whether an inactive interrupt thread has been initialised. In light of this, all interrupt threads in our model start as existing, but disabled:

$$halted_threads := \{ \}$$

Since interrupt threads start out disabled, they are by definition (see section 4.3.1) their own pagers. Additionally, *sigma0* is the root server's pager:

$$\begin{aligned} thread_pager &:= \{ kRootServer \mapsto kSigma0 \} \cup \\ &\quad id (kIntThreads) \end{aligned}$$

where *id* is the identity relation.

The root server starts up as the scheduler for *sigma0* and for itself [12, thread.cc]:

$$\begin{aligned} thread_scheduler &:= \{ kSigma0 \mapsto kRootServer, \\ &\quad kRootServer \mapsto kRootServer \} \end{aligned}$$

The root server and *sigma0* start with a *running* state, while interrupt threads start out as *aborted*:

$$\begin{aligned} thread_state &:= \{ kSigma0 \mapsto tsRunning, \\ &\quad kRootServer \mapsto tsRunning \} \cup \\ &\quad kIntThreads \times \{ tsAborted \} \end{aligned}$$

Finally, we set the thread counters in the respective address spaces:

$$\begin{aligned} threads_in_space &:= \{ kSigma0Space \mapsto 1, \\ &\quad kRootServerSpace \mapsto 1, \\ &\quad kKernelSpace \mapsto card (kIntThreads) \} \end{aligned}$$

4.3.3 Operations

As the *Thread* machine contains the core functionality in the formalisation, we will describe its operations in some detail. The machine corresponds to the THREADCONTROL subsystem of L4. The key operations at this level are *CreateThread*, *ActivateThread*, and *DeleteThread*.

To create a thread we need a free TCB, a thread number, an address space and a scheduler:

CreateThread (*tcb*, *global_tno*, *space*, *scheduler*)

CreateThread creates inactive threads. In order to succeed, the thread must not already exist, the supplied thread number must not be reserved or used for any existing thread. The address space the thread is to be created in, does not need to exist, but it cannot be the kernel space (which is reserved for interrupts):

PRE

$$\begin{aligned} tcb &\in TCB - threads \\ global_tno &\in GLOBAL_TNO \wedge \\ global_tno &\notin ran (thread_gno) \wedge \\ global_tno &\neq kNilGNo \wedge \\ global_tno &\neq kAnyGNo \wedge space \in ADDRESS_SPACE \wedge \\ space &\neq kKernelSpace \end{aligned}$$

Additionally, when no address space is supplied during thread creation, L4 creates a new address space for the new thread. At the level of the thread machine, we model this with the *CreateAddressSpace* operation if the space passed in does not exist. If it does exist, the total number of threads in it must not exceed the limit once this thread is added:

$$\begin{aligned} space \in spaces &\Rightarrow \\ threads_in_space (space) &< kMaxThreadsPerSpace \end{aligned}$$

Since an inactive thread is being created, no restriction is placed on *scheduler*. If these conditions are satisfied, the operation is guaranteed to succeed. In order to actually create the thread, we either create a new address space with the thread in it, or add one to the address space, and simultaneously set all properties for the new thread:

$$\begin{aligned} \text{IF } space \notin spaces \quad \text{THEN} \\ &\quad CreateAddressSpace (space) \parallel \\ &\quad threads_in_space (space) := 1 \\ \text{ELSE} \\ &\quad threads_in_space (space) := threads_in_space (space) + 1 \\ \text{END} \parallel \\ threads &:= threads \cup \{ tcb \} \parallel \\ thread_gno (tcb) &:= global_tno \parallel \\ thread_space (tcb) &:= space \parallel \\ thread_scheduler (tcb) &:= scheduler \end{aligned}$$

In the description of the following operations, trivial typing preconditions (such as $tcb \in TCB$) will be omitted. This is an area where we would have found a type system like in Isabelle/HOL useful where type inference takes care of such trivial conditions automatically.

In order for a thread to be able to do anything in the system, it must first be activated. This can be done as part of creation, or as an *ActivateThread* operation on an inactive thread:

ActivateThread(*tcb*, *space*, *pager*, *scheduler*)

In order for the operation to succeed exactly when activation in L4 succeeds, *tcb* must be an existing but inactive thread and *pager* must exist and be running when the thread starts executing [13, section 2.4]:

PRE

$$\begin{aligned} pager &\in threads \wedge \\ scheduler &\in active_threads \end{aligned}$$

L4 allows to migrate threads into new address spaces on activation. If this occurs, we must make sure that the thread fits into the new space:

$$\begin{aligned} space \in initialised_spaces \wedge \\ (space \neq thread_space (tcb) \Rightarrow \\ threads_in_space (space) < kMaxThreadsPerSpace) \end{aligned}$$

The operation itself updates the pager and the scheduler, adds *tcb* to active threads, sets its state to *tsWaitingForever*, and migrates the thread if necessary. In L4, a thread will begin waiting for an IPC from its pager straight after activation. This is why its state begins as waiting forever. The IPC component will be initialised in the operation *ActivateThread2* in section 4.4.3 below. The migration is performed as follows:

```

IF   space  $\neq$  thread_space ( tcb )   THEN
  thread_space ( tcb ) := space ||
  threads_in_space := threads_in_space  $\Leftarrow$ 
    { space  $\mapsto$  threads_in_space ( space ) + 1 ,
      thread_space ( tcb )  $\mapsto$ 
        threads_in_space ( thread_space ( tcb ) ) - 1 }
END

```

The thread counters for the two address spaces (current and target) are updated using *right overriding* (denoted \Leftarrow). Its definition is:

$$r_1 \Leftarrow r_2 = r_2 \cup (\text{dom}(r_2) \Leftarrow r_1)$$

The definition uses another of B's operators, *domain subtraction* (\Leftarrow), defined as:

$$S \Leftarrow r = \{ x \mapsto y \mid x \mapsto y \in r \wedge x \notin S \}$$

We define *CreateActiveThread* as a merger of *CreateThread* and *ActivateThread*. The only difference is that migrating the thread is not possible as it does not exist yet. Note that a higher-level operation cannot combine the those two operations due to B's restrictions.

The operation *DeleteThread*, given an existing thread *tcb*, removes it from the set of known, active and halted threads provided the *tcb* is not in one of the privileged address spaces. We also remove it from all thread-related functions in the machine:

```

DeleteThread(tcb)
  thread_space := { tcb }  $\Leftarrow$  thread_space ||
  thread_state := { tcb }  $\Leftarrow$  thread_state ||
  thread_pager := { tcb }  $\Leftarrow$  thread_pager ||
  thread_scheduler := { tcb }  $\Leftarrow$  thread_scheduler ||
  thread_gno := { tcb }  $\Leftarrow$  thread_gno

```

Furthermore, if the thread is the only one left in the address space, we delete the address space, otherwise we decrement the thread counter.

```

IF   tcb = thread_space-1 [ { thread_space ( tcb ) } ]
THEN
  DeleteAddressSpace ( thread_space ( tcb ) ) ||
  threads_in_space :=
    { thread_space ( tcb ) }  $\Leftarrow$  threads_in_space
ELSE
  threads_in_space ( thread_space ( tcb ) ) :=
    threads_in_space ( thread_space ( tcb ) ) - 1
END

```

Apart from creating, deleting and activating, the THREADCONTROL API section in L4 contains a number of further operations on threads which we explain below.

Modifying the thread's scheduler is one of them. The *SetScheduler* operation is trivial, assuming only that the thread and the scheduler exist, and updating the thread's scheduler. We omit its definition in B here.

We also omit the *Migrate* operation. It performs the same task as the migration in *ActivateThread*.

As our description of the formalisation progresses it becomes obvious that there is much statements duplication. Indeed, the next operation in the machine is *MigrateAndSetScheduler* which demonstrates the problem again. As mentioned before, this is a restriction of the B system. Writing *Migrate* and *SetScheduler* does not mean a higher-level machine can combine them to get *MigrateAndSetScheduler*. If they are to be combined, statements must be duplicated. We believe this to be a shortcoming of the B Method as used by the B Toolkit [2]. One solution to this is to defer any specific actions to refined machines, then use sequential composition in the refinements.

The problem with this is that the resulting top-level machines cannot be meaningfully animated, making validation of the formal model more difficult.

We next define the *SetState* operation, taking *tcb* and *state*, with the restrictions that *tcb* is active, not an interrupt thread (since they have different state semantics) and *state* cannot be *tsAborted* (transition to an aborted state would imply thread deactivation, which L4 does not provide):

```

SetState (tcb, state)  $\hat{=}$ 
PRE   state  $\in$  THREAD_STATE  $\wedge$ 
        state  $\neq$  tsAborted  $\wedge$ 
        tcb  $\in$  active_threads  $\wedge$ 
        tcb  $\notin$  kIntThreads
THEN
  thread_state ( tcb ) := state
END

```

To implement the EXCHANGEREGISTERS [13, section 2.3] system call, we need to specify its semantics with respect to the variables in the *Thread* machine. Its parameters are as follows:

tcb the thread to act on

control a subset of *EXREGS_FLAGS*, representing the set of actions the operation is to take (see section 4.1)

pager the pager to set the thread's pager to, if indicated by control

unwait should the target thread be woken; this is to correctly set the thread state if a machine including this one, such as *IpcBase*, uses its equivalent of *ExchangeRegisters* to cancel waiting or polling IPC states (see section 4.4.3).

Since there is also no specification of user-level registers saved in the kernel, IP, SP and FLAGS are not passed in. The semantics of *ExchangeRegisters* means a thread can only invoke it on another thread in its address space. Since no thread can be in the reserved kernel address space, interrupt threads are excluded. The operation sets the pager (if *ex.p* \in *control*), halts the thread (if *ex.H* \in *control*, resumes if not) and resets any waiting states (*unwait* = TRUE)

```

ThreadExchangeRegisters(tcb, control, pager, unwait)  $\hat{=}$ 
PRE   tcb  $\in$  threads  $\wedge$  control  $\subseteq$  EXREGS_FLAGS  $\wedge$ 
        pager  $\in$  TCB  $\wedge$  tcb  $\notin$  kIntThreads  $\wedge$  unwait  $\in$  BOOL
THEN
IF   ex.p  $\in$  control THEN
  thread_pager ( tcb ) := pager
END ||
IF   ex.h  $\in$  control THEN
IF   ex.H  $\in$  control THEN
  halted_threads := halted_threads - { tcb }
ELSE
  halted_threads := halted_threads  $\cup$  { tcb }
END
END ||
IF   unwait = TRUE THEN
IF   tcb  $\in$  active_threads THEN
  thread_state ( tcb ) := tsRunning
ELSE
  thread_state ( tcb ) := tsAborted
END
END
END

```

There are two special operations for interrupt threads: *ActivateInterrupt* and *DeactivateInterrupt*. In L4, activating an interrupt thread means setting it to *halted* and setting its pager to a value other than itself:

ActivateInterrupt(*tcb*, *handler*) \triangleq
PRE $tcb \in kIntThreads \wedge handler \in TCB \wedge$
 $handler \neq tcb$
THEN
 $halted_threads := halted_threads \cup \{ tcb \} \parallel$
 $thread_pager (tcb) := handler$
END

To deactivate an interrupt thread, we perform the opposite: the pager is set to itself and the *halted* flag is reset. The definition is analogous to *ActivateInterrupt* and we omit it here.

We will now describe helper-operations enabling IPC-related state transitions. The IPC state transitions themselves are the subject of the next section. The first operation is *Unwait*, which reverts a waiting or polling (waiting to send) thread to its normal state. For normal threads this is *running*; for inactive and interrupt threads it is *aborted*.

UnWait(*tcb*) \triangleq
PRE $tcb \in threads$ **THEN**
SELECT
 $tcb \in active_threads \wedge tcb \notin kIntThreads$
THEN
 $thread_state (tcb) := tsRunning$
WHEN
 $tcb \in active_threads \wedge tcb \in kIntThreads$
THEN
 $thread_state (tcb) := tsAborted$
ELSE
 $thread_state (tcb) := tsAborted$
END
END

A *SELECT* statement non-deterministically selects one of the cases whose condition is true and evaluates the statement contained in the *THEN* clause. If no condition is true, the *ELSE* clause is evaluated.

When a running thread attempts to send an IPC to another thread, one of three things happens:

- the other thread is not waiting: the running thread polls — *SetState* is used
- the other thread is waiting, no receive phase is included: the IPC occurs, the remote thread is woken with *UnWait*
- as above, but a non-trivial receive phase is included: the IPC occurs, the remote thread is woken, but the current thread starts waiting — *WakeUpAndWait* is used

The *WakeUpAndWait* operation takes a running thread, a waiting thread and a the wait state the sending thread is to assume. Both threads must be active, the first must be running (*isRunning* tests for equality with *tsRunning*), the second must be waiting (*tsWaitingForever* or *tsWaitingTimeout*):

WakeUpAndWait(*running_tcb*, *waiting_tcb*, *wait_state*) \triangleq
PRE $running_tcb \in active_threads \wedge$
 $waiting_tcb \in active_threads \wedge$
 $isWaiting (wait_state) \wedge$
 $isRunning (thread_state (running_tcb)) \wedge$
 $isWaiting (thread_state (waiting_tcb))$
THEN
IF $waiting_tcb \in kIntThreads$ **THEN**
 $thread_state := thread_state \Leftarrow$
 $\{ running_tcb \mapsto wait_state ,$
 $waiting_tcb \mapsto tsAborted \}$
ELSE
 $thread_state := thread_state \Leftarrow$
 $\{ running_tcb \mapsto wait_state ,$

$waiting_tcb \mapsto tsRunning \}$

END
END

Threads are not the only cause of IPC happening. When an IPC cannot be resolved immediately, the situation may arise that two threads, one polling on the second and the second waiting on the first, might be inside the system. It is then up to the scheduler to cause the IPC to happen. When it does, both threads need to be woken up and resume running. This is also true if the IPC fails. To handle this case, we use the *DualWakeUp* operation, which simply takes a polling and waiting thread and essentially performs an *Unwait* on each. We omit the obvious formal definition.

4.4 Inter Process Communication

This section describes the machines *IpcCore* and *IpcBase* which cover the IPC-related operations in L4.

Inter process communication is the core component of L4. Nearly all aspects of the system are abstracted by IPC when possible, including donation and leasing of memory to other processes. IPC is *synchronous*; for a successful transfer to occur, the sender must be sending or polling while the receiver is waiting (or running, if the sender is polling). What is more, the receiver must be waiting for the sender for this to work. The special thread identifiers *anythread* and *anylocalthread* also declare who a thread is willing to receive from. When the sender tries sending an IPC but the receiver is not ready or currently willing to receive, it goes into a *polling* state and is placed in the receiver's incoming queue. There is only one *polling* state, regardless of the timeout. Polling may include an additional receive phase, which means that should the send succeed, the kernel immediately places it into a *waiting* state with the receive phase parameters.

At first glance, the *IpcCore* machine does not seem very useful, as it has no state and contains only a single place-holder operation *PerformIpc*, which does nothing. This is because the *IpcCore* operation represents the transfer of information contained in Message Registers (MRs) from the sending to the receiving thread, but MRs do not exist in the specification. They could be added in a later refinement step.

We decided to leave out MRs at this level of the specification, because they contain too much implementation detail. When an IPC is performed, MRs do not merely get transferred, but can also contain information on memory maps and grants which would have duplicated the efforts of the VM subsystem part of the verification pilot project.

The machine contains a useful definition *canIPC* representing whether a thread can invoke the IPC system call (for interrupt threads, this means whether the kernel can perform the IPC on behalf of the thread): the thread must be active, running and not halted (except for interrupt threads, which must be halted to be enabled):

canIPC(*t*) \triangleq
 $t \in active_threads \wedge$
 $(t \in kIntThreads \Rightarrow t \in halted_threads) \wedge$
 $(t \notin kIntThreads \Rightarrow thread_state (t) = tsRunning \wedge$
 $t \notin halted_threads)$

The *IpcBase* machine is the basis for all state transitions during IPC. It *INCLUDES* *IpcCore* and so builds on all machines described so far. It does not promote any operations. The IPC operations are non-deterministic, i.e. there are situations which might cause them to fail which are not explicitly contained in this specification. This means that the first possibility of failure is in this machine. The operations

in the previous machines always succeeded given the preconditions. In L4, the error condition is stored in the Error Thread Control Register, which means we have to specify some form of this TCR in the *IpcBase* machine.

Unfortunately, B's inability to perform two operations from the same machine in parallel means that the promoted operation cannot clear the Error TCR themselves. As a work-around we introduce new local versions of these operations, which only add that error functionality and shadow all operations which might normally just be promoted.

4.4.1 Variables

The *IpcBase* machine uses information on which thread is waiting/polling for which other thread to check when to allow the IPC to occur, and handles invoking the proper state-transition operations from the *Thread* machine.

In L4, threads which are in a *waiting* state must be waiting for a specific thread number, or *anythread*. They cannot wait for *nilthread*. If a thread wants to make sure the waiting operation times out, it should wait for itself [13, section 5.6]:

$thread_ipc_waiting_for \in$
 $active_threads \leftrightarrow GLOBAL_TNO \wedge$
 $kNilGNo \notin ran (thread_ipc_waiting_for)$

Only active threads may participate in IPC, but they may wait for any thread number. The reference manual states that if the partner does not exist, the IPC operation will fail. However, the thread might exist when IPC is invoked, but be deleted before IPC completes, so *thread_ipc_waiting_for* cannot have *active_threads*, or even *threads* as its permitted range.

We not only need to know *that* a thread is waiting, but also for *how long* it is waiting. We therefore define the *thread_ipc_waiting_timeout* function. Its range is identical to the one of *thread_ipc_waiting_for*, but it specifies the timeout for waiting threads:

$thread_ipc_waiting_timeout \in$
 $active_threads \leftrightarrow TIMEOUT \wedge$
 $eZeroTimeout \notin ran (thread_ipc_waiting_timeout) \wedge$
 $dom (thread_ipc_waiting_timeout) =$
 $dom (thread_ipc_waiting_for) \wedge$
 $dom (thread_ipc_waiting_timeout) =$
 $thread_state^{-1} [\{ tsWaitingTimeout ,$
 $tsWaitingForever \}]$

All the threads in the domain must either be waiting with a timeout, or waiting forever. No thread with a waiting state may be absent, and no thread in the function's domain may have a different state. Since the domain is the same as that for *thread_ipc_waiting_for*, the constraint applies there as well. Zero-timeout is not permitted, since those calls are resolved immediately without forcing the thread to wait.

Similar to the two functions above, we define polling semantics for threads:

$thread_ipc_polling_on \in active_threads \leftrightarrow threads \wedge$
 $thread_ipc_polling_timeout \in$
 $active_threads \leftrightarrow TIMEOUT \wedge$
 $dom (thread_ipc_polling_timeout) =$
 $thread_state^{-1} [\{ tsPolling \}] \wedge$
 $eZeroTimeout \notin ran (thread_ipc_polling_timeout) \wedge$
 $dom (thread_ipc_polling_on) \subseteq$
 $dom (thread_ipc_polling_timeout)$

In L4, each thread keeps track of which thread it is polling on (if any), as well as keeping a list of threads

which are polling on it. The deletion of a thread which another thread is polling on is not well defined (neither in manual nor source code). In our formalisation, we only remove the target thread from the range of *thread_ipc_polling_on* and let the IPC time out. For this reason, there can be some threads in a polling state which are not actually polling for any thread. All polling threads must have a timeout, even if the thread they are polling on was deleted, otherwise they will never return to running.

The convenience function *thread_incoming*, given a thread, yields the threads that are polling on it. It represents each thread's incoming IPC buffer and is just the relational inverse of all threads on *thread_ipc_polling_on*:

$\forall tt . (tt \in active_threads \Rightarrow thread_incoming (tt) =$
 $thread_ipc_polling_on^{-1} [\{ tt \}])$

For some operations, we need the thread numbers (IDs) of incoming threads. Again, to simplify the formalisation we define *thread_incoming_gnos* as:

$\forall tt . (tt \in active_threads \Rightarrow$
 $thread_incoming_gnos (tt) =$
 $thread_gno [thread_incoming (tt)])$

In addition to the above, when the send phase of an IPC succeeds, the receive phase is invoked. If no receive was requested, the thread goes directly back to running. Otherwise the receive is performed, which means if no candidates are available, the thread must wait. We store a timeout for each polling thread. The *thread_recv_waiting* functions only differ from the *thread_ipc_waiting* functions in that not all polling threads will have a receive phase, and no waiting thread may have a future receive phase:

$dom (thread_recv_waiting_for) \subseteq$
 $dom (thread_ipc_polling_timeout) \wedge$
 $dom (thread_recv_waiting_timeout) \cap$
 $dom (thread_ipc_waiting_timeout) = \{ \}$

The variable *thread_error* represents the concept of each thread having some error condition which resulted from a previous operation:

$thread_error \in active_threads \rightarrow ERROR$

For inactive threads (which cannot execute), the mapping has no meaning and so does not exist. Note that *thread_error* is not the same as the Error TCR [13], since *ERROR* contains *eNoError*, a condition to signify success. L4 would put the success/failure result into a register instead.

4.4.2 Initialisation

We now describe, how the variables defined in the section before are initialised. Initially, no thread is waiting for any other thread or engaged in IPC in any way, meaning that all the *thread_ipc_** variables as well as the *thread_recv_** variables are initialised to empty sets.

Since the interrupt threads, sigma0 and the root server exist on start up, we want their incoming sets to be present, but empty:

$thread_incoming \in$
 $\{ kSigma0 , kRootServer \} \cup kIntThreads \rightarrow \{ \{ \} \} \parallel$
 $thread_incoming_gnos \in$
 $\{ kSigma0 , kRootServer \} \cup kIntThreads \rightarrow \{ \{ \} \}$

Any function mapping those threads to the the empty set (there is only one) will satisfy that requirement (\in denotes the choice operator).

As for the error condition, all existing threads (the same ones as above) start out with the *eNoError* condition:

$thread_error := \{ kSigma0 \mapsto eNoError ,$
 $kRootServer \mapsto eNoError \} \cup$
 $kIntThreads \times \{ eNoError \}$

4.4.3 Operations

The new variables introduced in this machine, together with the invariant of the included machines produce a new, larger invariant. Promotion of some operations causes the local invariant to be violated as the operations in lower machines know nothing about it. Operations which introduce handling of *IpcBase*'s variables to operations from *Thread* or *AddressSpace* have 2 appended to their name.

The first of these is *ActivateThread2*, which sets up the local variables when the thread is activated, and invokes the original *ActivateThread*. Their preconditions and parameters are the same, except that we now need to specify who a freshly activated thread must wait for an IPC from. In L4, that is its pager. Since it cannot run until the message is received, it will wait forever:

```

ActivateThread2(tcb, space, pager, scheduler)  $\hat{=}$ 
  PRE   tcb  $\in$  threads  $\wedge$  tcb  $\notin$  active_threads  $\wedge$ 
        tcb  $\neq$  pager  $\wedge$ 
        thread_space ( tcb )  $\in$  initialised_spaces  $\wedge$ 
        pager  $\in$  active_threads  $\wedge$ 
        scheduler  $\in$  active_threads  $\wedge$ 
        space  $\in$  initialised_spaces  $\wedge$ 
        ( space  $\neq$  thread_space ( tcb )  $\Rightarrow$ 
          threads_in_space ( space )  $<$  kMaxThreadsPerSpace )
  THEN
    ActivateThread ( tcb , space , pager , scheduler ) ||
    thread_ipc_waiting_timeout ( tcb ) :=
      eInfiniteTimeout ||
    thread_ipc_waiting_for ( tcb ) := thread_gno ( pager ) ||
    thread_error ( tcb ) := eNoError ||
    thread_incoming ( tcb ) :=
      thread_ipc_polling_on-1 [ { tcb } ] ||
    thread_incoming_gnos ( tcb ) :=
      thread_gno [ thread_ipc_polling_on-1 [ { tcb } ] ]
  END

```

We need to set the thread's error condition to some value and since no error has occurred, that is *eNoError*. Additionally, some threads may already be polling for this thread ID, so the *thread_incoming* and *thread_incoming_gnos* functions are updated appropriately.

The *CreateActiveThread2* operation is augmented analogously and we omit its definition here.

Next we add necessary statements to *DeleteThread* which clean up the affected variables in this machine. The preconditions and parameters do not change. Apart from the obvious domain subtraction of {*tcb*} from *thread_ipc_waiting_**, *thread_ipc_polling_timeout*, *thread_rcv_** and *thread_error*, we need to remove the thread from both the range and domain of *thread_ipc_polling_on*:

```

thread_ipc_polling_on :=
  { tcb }  $\triangleleft$  thread_ipc_polling_on  $\triangleright$  { tcb }

```

The application of the domain subtraction (\triangleleft , precedence is left-to-right) removes all mappings denoting that this thread is polling on another one (there is only one). Then, the application of range subtraction (\triangleright) removes all mappings denoting that another thread is polling on this one. Those threads that were polling on the one being deleted are now stranded until their IPCs time out. Range subtraction is defined canonically:

$$r \triangleright S = \{x \mapsto y \mid x \mapsto y \in r \wedge y \notin S\}$$

This resolves the situation of who is polling on whom. The thread still needs to be removed from the incoming sets of other threads:

```

thread_incoming :=
  { aa , bb |
    aa  $\in$  dom ( thread_incoming ) - { tcb }  $\wedge$ 
    bb = thread_incoming ( aa ) - { tcb } }

```

We do the removal via a set comprehension which keeps only other threads' incoming sets, but also removes the deleted thread from them. We apply the same technique to modify *thread_incoming_gnos*.

Operations from previous machines that complete successfully need to clear the error attribute of the thread they are operating on. Hence, we extended these operations with that functionality at the *IpcBase* machine level. Their preconditions are almost the same as their *Thread* counterparts', and the operation body invokes them directly. The only difference is that they take an extra parameter (*itcb*) that says which thread's error attribute should be cleared. We omit the B definition of these machines. They are: *InitialiseAddressSpace2*, *CreateThread2*, *SetScheduler2*, *Migrate2*, *MigrateAndSetScheduler2*, *ActivateInterrupt2* and *DeactivateInterrupt2*.

We now cover the operations directly related to IPC. In all operations, the invoking thread is checked with the *canIpc* definition of the *IpcCore* machine.

The first of the operations enabling IPC is *JustWait*. It is invoked when a thread requests an IPC operation consisting of a receive phase only, but no thread in its incoming sets is available to receive from, thus causing the receiver thread to wait. Given the three parameters *tcb* (the thread wishing to receive), *timeout* and *fromSpecifier* (who it is willing to receive from), the preconditions are as follows: the thread *canIpc*; *timeout* is either finite or infinite, but not zero (instant time-out); *fromSpecifier* is not *nilthread* and is either *anythread* or a known thread. Also, we exclude all conditions which would cause an immediate IPC reception to occur: if *fromSpecifier* is *anythread*, the requester must not have incoming threads; for other specifiers, they must not be in the incoming thread numbers. The work done by the operation is minimal. It updates *thread_ipc_waiting_for* and its time-out equivalent to indicate the thread is waiting and who it is waiting for. It also uses *SetState* to set the thread's state to *tsWaitingForever* or *tsWaitingTimeout* depending on the value of *timeout*.

```

JustWait(tcb, timeout, fromSpecifier)  $\hat{=}$ 
  PRE   canIPC ( tcb )  $\wedge$  timeout  $\in$  TIMEOUT  $\wedge$ 
         $\neg$  ( isNoTimeout ( timeout ) )  $\wedge$ 
        fromSpecifier  $\in$  GLOBAL_TNO  $\wedge$ 
        fromSpecifier  $\neq$  kNilGNo  $\wedge$ 
        fromSpecifier  $\in$ 
          thread_gno [ threads ]  $\cup$  { kAnyGNo }  $\wedge$ 
        ( fromSpecifier = kAnyGNo  $\Rightarrow$ 
          thread_incoming ( tcb ) = { } )  $\wedge$ 
        ( fromSpecifier  $\neq$  kAnyGNo  $\Rightarrow$ 
          fromSpecifier  $\notin$  thread_incoming_gnos ( tcb ) )
  THEN
    thread_ipc_waiting_for ( tcb ) := fromSpecifier ||
    thread_ipc_waiting_timeout ( tcb ) := timeout ||
    IF isInfinite ( timeout ) THEN
      SetState ( tcb , tsWaitingForever )
    ELSE
      SetState ( tcb , tsWaitingTimeout )
    END
  END

```

We have covered threads that want to receive, but cannot. The *SetUpReceivePhaseAndPoll* operation handles the case of when the operation wants to send but cannot (either the remote thread is not waiting, or it is not waiting for the sending thread). The operation takes *tcb_from* and *tcb_to* (sending and target

threads), *poll.timeout*, *recv.timeout* (time-out for the future receive phase) and a *fromSpecifier* (who the thread is willing to receive from in the receive phase, or *nilthread* if there is no receive phase).

As in *JustWait*, *tcb.from* must be able to perform IPC. The target must be an existing thread. While the poll time-out must not be zero, the receive time-out is only restricted if *fromSpecifier* is not *nilthread*. The actual *fromSpecifier* must be a thread number of an existing thread.

```

SetUpReceivePhaseAndPoll(tcb.from, tcb.to, poll.timeout,
  recv.timeout, fromSpecifier)  $\hat{=}$ 
PRE   canIPC ( tcb.from )  $\wedge$  tcb.to  $\in$  threads  $\wedge$ 
  ( tcb.to  $\in$  dom ( thread_ipc.waiting_for )  $\Rightarrow$ 
    thread_ipc.waiting_for ( tcb.to )  $\neq$ 
    thread_gno ( tcb.from )  $\wedge$ 
    thread_ipc.waiting_for ( tcb.to )  $\neq$  kAnyGNo )  $\wedge$ 
    fromSpecifier  $\in$  GLOBAL_TNO  $\wedge$ 
    poll.timeout  $\in$  TIMEOUT  $\wedge$ 
     $\neg$  ( isNoTimeout ( poll.timeout ) )  $\wedge$ 
    recv.timeout  $\in$  TIMEOUT  $\wedge$ 
    ( fromSpecifier  $\neq$  kNilGNo  $\Rightarrow$ 
       $\neg$  ( isNoTimeout ( recv.timeout ) ) )  $\wedge$ 
    fromSpecifier  $\in$  thread_gno [ threads ]  $\cup$ 
      { kAnyGNo , kNilGNo }

THEN
  thread_ipc.polling_on ( tcb.from ) := tcb.to ||
  thread_ipc.polling_timeout ( tcb.from ) := poll.timeout ||
  thread_incoming ( tcb.to ) :=
    thread_incoming ( tcb.to )  $\cup$  { tcb.from } ||
  thread_incoming_gnos ( tcb.to ) :=
    thread_incoming_gnos ( tcb.to )  $\cup$ 
    { thread_gno ( tcb.from ) } ||
  SetState ( tcb.from , tsPolling ) ||
IF   fromSpecifier  $\neq$  kNilGNo THEN
  thread_recv.waiting_for ( tcb.from ) :=
    fromSpecifier ||
  thread_recv.waiting_timeout ( tcb.from ) :=
    recv.timeout
END
END

```

To verify that the operation happens in the aforementioned circumstances, if the target thread is in a waiting state, then it must not be waiting for *anythread* (since this one will fulfil the criterion) and it must not be waiting for *from_tcb*'s number:

```

( tcb.to  $\in$  dom ( thread_ipc.waiting_for )  $\Rightarrow$ 
  thread_ipc.waiting_for ( tcb.to )  $\neq$ 
  thread_gno ( tcb.from )  $\wedge$ 
  thread_ipc.waiting_for ( tcb.to )  $\neq$  kAnyGNo )

```

Once that is established, the operation can proceed successfully by updating *thread_ipc.polling_** to reflect the thread's polling information, and also adds *from_tcb* and its thread number to the incoming sets of *tcb.to*. If *fromSpecifier* is not *nilthread*, *thread_recv_** is updated with the future waiting information.

Having covered the cases where IPC cannot happen, let us look at the simplest case of IPC occurring: the thread requests an IPC with only a receive phase, and a suitable thread is in its incoming set. Like *JustWait*, *JustReceive* takes two parameters (whose meaning is the same): *itcb* and *fromSpecifier*.

JustReceive(*itcb*, *fromSpecifier*)

It does not need a time-out as the operation will go ahead immediately. The value of *fromSpecifier* must not be *nilthread*, and must either be *anythread* (in which case the incoming set must not be empty) or a

thread number already in the incoming set. The operation may then go ahead. However, it might not succeed due to aspects beyond the control of the current model (such as the operation being aborted halfway by another thread). We model this failure by non-determinism.

```

canIPC ( itcb )  $\wedge$  fromSpecifier  $\in$  GLOBAL_TNO  $\wedge$ 
fromSpecifier  $\neq$  kNilGNo  $\wedge$ 
( fromSpecifier  $\neq$  kAnyGNo  $\Rightarrow$ 
  fromSpecifier  $\in$  thread_incoming_gnos ( itcb ) )  $\wedge$ 
( fromSpecifier = kAnyGNo  $\Rightarrow$ 
  thread_incoming ( itcb )  $\neq$  { } )

```

The polling thread which is allowed to send is chosen non-deterministically (since sets have no implicit ordering):

```

ANY   tcb.from
WHERE tcb.from  $\in$  thread_incoming ( itcb )  $\wedge$ 
  ( fromSpecifier  $\neq$  kAnyGNo  $\Rightarrow$ 
    thread_gno ( tcb.from )  $\in$ 
    thread_incoming_gnos ( itcb ) )

```

In other words, choose any of the threads in the incoming set, with the extra constraint that if *fromSpecifier* is not *anythread*, that thread's number must be in the set of incoming thread numbers for the receiving thread. The preconditions guarantee that a thread that satisfies this constraint actually exists.

Regardless of the IPC succeeding or failing, the sending thread will no longer be polling at the end of the operation:

```

thread_ipc.polling_on :=
  { tcb.from }  $\triangleleft$  thread_ipc.polling_on ||
thread_ipc.polling_timeout :=
  { tcb.from }  $\triangleleft$  thread_ipc.polling_timeout
thread_incoming ( itcb ) :=
  thread_incoming ( itcb ) - { tcb.from } ||
thread_incoming_gnos ( itcb ) :=
  thread_incoming_gnos ( itcb ) -
  { thread_gno ( tcb.from ) }

```

Since it will no longer be polling, the future settings for its receive phase will no longer be applicable. If the IPC succeeds, they will be used to set up the new receive phase for the thread. If IPC fails, they will be discarded:

```

thread_recv.waiting_timeout := { tcb.from }  $\triangleleft$ 
  thread_recv.waiting_timeout ||
thread_recv.waiting_for := { tcb.from }  $\triangleleft$ 
  thread_recv.waiting_for

```

We now reach the point where IPC either succeeds or fails. This is performed using the non-deterministic CHOICE *path1* OR *path2* END construct. During animation, the user is asked to choose the path.

On the IPC success path, the IPC transfer is performed and the error fields for both threads are cleared:

```

PerformIPC ( tcb.from , itcb ) ||
  thread_error := thread_error  $\triangleleft$ 
  { itcb  $\mapsto$  eNoError , tcb.from  $\mapsto$  eNoError }

```

Then, if the sender had a receive phase waiting, it is set up (using identical statements to those in *JustWait*). The receiving thread's state does not change. It was either running or an activated interrupt thread, and remains so. If the sender does not have a receive phase waiting, its waiting state is cancelled using *UnWait*.

The IPC failure path consists of cancelling the sender's waiting state with *UnWait* and picking an

error non-deterministically among the possible unpredictable IPC errors (see section 4.1) and assigned as an error indicator for both threads.

WakeDestThenWait covers the opposite direction to *JustReceive*: a thread wishes to send and the second thread is waiting, the IPC occurs immediately, the destination thread is woken up, while the source thread starts waiting if a receive phase was specified.

WakeDestThenWait(*tcb_from*, *tcb_to*, *recv_timeout*,
fromSpecifier)

The precondition combines aspects of the previous IPC operations: the destination must be waiting for either the source's thread number or *anythread*; *fromSpecifier* must be that of an existing thread, *nilthread* or *anythread*; a non-*nilthread fromSpecifier* indicates a receive phase for the source and so *recv_timeout* must not be zero; there is no polling timeout, since the operation goes ahead immediately.

This time there are no common items between the success and failure paths.

The success path begins as for *JustReceive* by performing the IPC transfer and clearing the error indicators for both threads. If *tcb_from* (the source thread) did not request a receive phase, the operation can be quickly finished by domain subtraction of *tcb_to* from *thread_ipc_waiting_** and using *UnWait* to cancel its waiting state. If it *did* request a receive phase, then the situation is more complicated. The destination still has to be removed from *thread_ipc_waiting_**, but now the source thread must also be inserted. The first half of the IF statement is presented below, for when the time-out is infinite. The second half is analogous, but the timeout is finite and so the state will be *tsWaitingTimeout*:

```
thread_ipc_waiting_for :=
  { tcb_to }  $\triangleleft$  thread_ipc_waiting_for  $\cup$ 
  { tcb_from  $\mapsto$  fromSpecifier } ||
IF isInfinite ( recv_timeout ) THEN
  thread_ipc_waiting_timeout :=
  { tcb_to }  $\triangleleft$  thread_ipc_waiting_timeout  $\cup$ 
  { tcb_from  $\mapsto$  eInfiniteTimeout } ||
  WakeUpAndWait ( tcb_from , tcb_to ,
    tsWaitingForever )
ELSE ...
```

WakeUpAndWait is used to wake up *tcb_to*, and make *tcb_from* wait with one of the time-outs.

We leave out the formal definition of the failure path here. It just removes the destination from *thread_ipc_waiting_**, picks an error, sets it as the error attribute for both threads, and uses *UnWait* on the destination thread.

The *ResolveIPC* operation covers the situation where one thread₁ is polling on thread₂, while the latter is waiting for the former. Since neither of them is executing, the kernel will perform the IPC. We omit the formal definition of **ResolveIPC**(*tcb_from*, *tcb_to*). It is a combination of *JustReceive* and *WakeDestThenWait*. The main difference is the precondition. It requires that both threads are active, the sender is polling and the receiver is waiting, the sender is polling on the receiver and the receiver either accepts *anythread* or the receiver's thread number.

When the kernel finds a thread that has been polling for longer than its time-out value, a time-out occurs. Since the model abstracts from exact values for time-outs, we let the time-out occur non-deterministically. The *TimeoutPoll* operation picks any thread which is polling with a non-infinite time-out and times it out. If such a thread does not exist, it does nothing. Of course, non-determinism is not random, it just states that the decision algorithm is

not specified at this level. During animation, the user is asked to be that algorithm. The *TimeoutPoll* operation removes the thread from the state variables and sets its error attribute to *eSendTimeout*. The *TimeoutWait* operation is the equivalent of *TimeoutPoll*, but times out a thread which is waiting with a finite time-out.

TimeoutPoll \triangleq

BEGIN

IF *thread_ipc_polling_timeout* \triangleright
{ *eFiniteTimeout* } = { }

THEN

skip

ELSE

ANY *tcb*

WHERE

tcb \in dom (*thread_ipc_polling_timeout* \triangleright
{ *eFiniteTimeout* })

THEN

UnWait (*tcb*) ||

thread_ipc_polling_on :=

{ *tcb* } \triangleleft *thread_ipc_polling_on* ||

thread_ipc_polling_timeout :=

{ *tcb* } \triangleleft *thread_ipc_polling_timeout* ||

thread_incoming (

thread_ipc_polling_on (*tcb*)) :=

thread_incoming (

thread_ipc_polling_on (*tcb*)) - { *tcb* } ||

thread_incoming_gnos (

thread_ipc_polling_on (*tcb*)) :=

thread_incoming_gnos (

thread_ipc_polling_on (*tcb*)) -

{ *thread_gno* (*tcb*) } ||

thread_recv_waiting_timeout :=

{ *tcb* } \triangleleft *thread_recv_waiting_timeout* ||

thread_recv_waiting_for :=

{ *tcb* } \triangleleft *thread_recv_waiting_for* ||

thread_error (*tcb*) := *eSendTimeout*

END

END

END

The *IpcBase* machine also provides an operation *SetError* as a way for operations in higher-level machines to set the error attribute for an active thread. This is used for example, to signal that a thread lacks necessary privileges to perform an operation.

When we described the EXCHANGEREGISTERS functionality in section 4.3.3, we only covered the functionality pertaining directly to threads and their control state. As the reference manual [13, section 2.3] states, EXCHANGEREGISTERS can be used to cancel or abort ongoing IPCs. Now that the IPC state transitions are available, we can model the IPC functionality in EXCHANGEREGISTERS. *IpcBaseExchangeRegisters* takes one fewer parameter than *ThreadExchangeRegisters*. It is the one that decides whether a waiting/polling thread is to be woken up. The preconditions, with the exception of the *unwait* flag are identical.

IpcBaseExchangeRegisters(*tcb*, *control*, *pager*) \triangleq

PRE *tcb* \in *threads* \wedge *control* \subseteq *EXREGS_FLAGS* \wedge
pager \in *TCB* \wedge *tcb* \notin *kIntThreads*

THEN

IF

ex_S \in *control* \wedge

tcb \in dom (*thread_ipc_polling_on*)

THEN

ThreadExchangeRegisters(*tcb*, *control*, *pager*,
TRUE) ||

```

thread_ipc_polling_on :=
  { tcb }  $\triangleleft$  thread_ipc_polling_on ||
thread_ipc_polling_timeout :=
  { tcb }  $\triangleleft$  thread_ipc_polling_timeout ||
thread_incoming (
  thread_ipc_polling_on ( tcb ) ) :=
  thread_incoming (
    thread_ipc_polling_on ( tcb ) ) - { tcb } ||
thread_incoming_gnos (
  thread_ipc_polling_on ( tcb ) ) :=
  thread_incoming_gnos (
    thread_ipc_polling_on ( tcb ) ) -
  { thread_gno ( tcb ) } ||
thread_rcv_waiting_timeout :=
  { tcb }  $\triangleleft$  thread_rcv_waiting_timeout ||
thread_rcv_waiting_for :=
  { tcb }  $\triangleleft$  thread_rcv_waiting_for ||
ANY err WHERE
  err  $\in$  { eSendCancelled , eAborted }
THEN
  thread_error ( tcb ) := err
END
ELSIF
  ex_R  $\in$  control  $\wedge$ 
  tcb  $\in$  dom ( thread_ipc_waiting_for )
THEN
  ThreadExchangeRegisters(tcb, control, pager,
    TRUE) ||
  thread_ipc_waiting_for :=
    { tcb }  $\triangleleft$  thread_ipc_waiting_for ||
  thread_ipc_waiting_timeout :=
    { tcb }  $\triangleleft$  thread_ipc_waiting_timeout ||
  ANY err WHERE
    err  $\in$  { eRecvCancelled , eAborted }
  THEN
    thread_error ( tcb ) := err
  END
ELSE
  ThreadExchangeRegisters (tcb, control, pager,
    FALSE) ||
  thread_error ( tcb ) := eNoError
END
END

```

The functionality at the IPC level consists of the following bits in *control*: if $S = 1$, a currently ongoing send IPC operation will be *aborted*, while an IPC send operation waiting to happen will be *cancelled*; if $R = 1$, likewise, but for receiving IPC. In the current model, bits are not used. Instead, the bits are represented by set membership of ex_S and ex_R in *control*. If neither are present, the operation invokes *ThreadExchangeRegisters* with *unwait* set to *FALSE* (do not change the state) and clears the error attribute.

If ex_S is present, the operation is removed from the state variables to do with polling, as well as from the incoming set of the thread it is polling on. Parallel composition means it is impossible to determine whether the IPC operation was *cancelled* or *aborted*, so a non-deterministic choice is made and becomes the value of the thread's error attribute. *ThreadExchangeRegisters* is invoked with the *unwait* flag equal to *TRUE*, forcing the function to be awakened.

If ex_R is present, events proceed as above, except the thread is removed from state variables related to *waiting*.

This concludes the operations of *IpcBase*. Having defined all the core functionality present in the model, we can now show an accurate view of state transitions in figure 4.

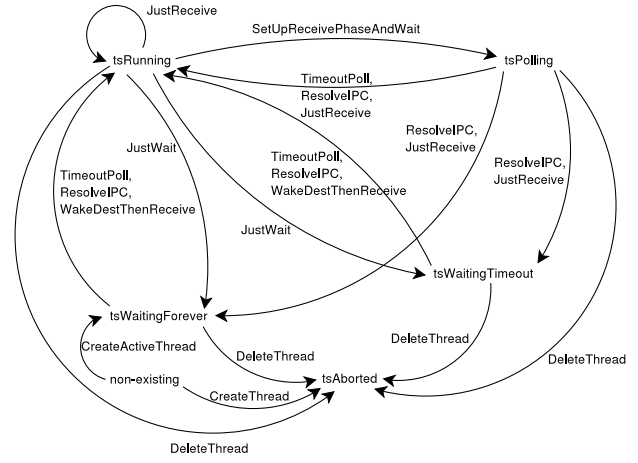


Figure 4: Possible state transitions in the model and operations which cause them.

4.5 API

This is the topmost machine in the specification. It *INCLUDES* *WeakSysCall* and all context machines.

Operations in *API* are either direct equivalents of L4 system calls, or operations representing system internals for use in animation. Their only real task at this level is to provide precondition support to lower-level operations (such as those in *WeakSyscall*) and pick which of these operations to invoke. They are very simple, if sometimes long, and are better examined directly.

It is worth noting that the top-level system-call operations still have preconditions: the invoking thread must be active and running, otherwise the system scheduler is fundamentally broken.

To give the reader an idea of what such an operation looks like, we present the final version of *ExchangeRegisters*. It augments the *IpcBaseExchangeRegisters* operation with the error-checking necessary to make it succeed, as well as non-deterministically modelling the system call components not within the scope of the formalisation:

```

ExchangeRegisters ( itcb , tcb , control , sp , ip , flags ,
  pager , handle )  $\triangleq$ 
PRE   itcb  $\in$  active_threads  $\wedge$ 
  thread_state ( itcb ) = tsRunning  $\wedge$ 
  tcb  $\in$  TCB  $\wedge$  control  $\subseteq$  EXREGS_FLAGS  $\wedge$ 
  sp  $\in$   $\mathbb{N}$   $\wedge$  ip  $\in$   $\mathbb{N}$   $\wedge$ 
  pager  $\in$  TCB  $\wedge$  flags  $\in$   $\mathbb{N}$   $\wedge$  handle  $\in$   $\mathbb{N}$ 
THEN
  SELECT   tcb  $\notin$  threads THEN
    SetError ( itcb , eInvalidThread )
  WHEN   tcb  $\in$  threads  $\wedge$ 
    thread_space ( tcb )  $\neq$  thread_space ( itcb )
  THEN
    SetError ( itcb , eInvalidThread )
  ELSE
    CHOICE
      IpcBaseExchangeRegisters ( tcb , control , pager )
    OR
      ANY error
      WHERE   error  $\in$  { eOutOfMemory ,
        eInvalidUtlbLocation }
      THEN
        SetError ( itcb , error )
      END
    END
  END

```

As mentioned before, the two remaining non-trivial assumptions are that the invoking thread *itcb* must be active and running (otherwise it cannot perform a system call). Via the non-deterministic *SELECT* statement with exclusive conditions, we enforce the preconditions of *IpcBaseExchangeRegisters*: the target thread *tcb* must exist and be in the caller's address space. If the preconditions are met the operation will succeed, but this is not necessarily true of the system call [13, section 2.3]: we may be out of memory or point to a bad memory location. Since the virtual memory subsystem is outside the scope of this formalisation, we model these failures via non-deterministic choice. The instruction and stack pointers, due to no knowledge of memory layout, are ignored; so is the user-defined *handle*, since it has no effect on actions performed by the kernel.

5 Conclusion

In this paper we have described our formalisation of the L4 high-performance microkernel in the B method. The main work on the formalisation was done as the honours thesis project of the first author which equates to an investment of roughly 5 person months. The final formalisation extends to about 2000 lines of B specification.

The goals of the formalisation effort were reached. The model is animatable and can be used as a learning tool. During the project it became apparent that in spite of detailed, good quality documentation, there were a number of ambiguities in the description of the L4 API and even inconsistent expectations towards its behaviour. Using code inspection and discussions with L4 developers those could be resolved, made precise and documented in the model. We are confident that the formalisation provides a good basis for the planned revision of the L4 API that involves formal modelling from the start.

The level of detail that was achieved during the available time frame suggests that formal specification of real-world operating system kernels is entirely feasible, a good opportunity for documentation, and a good starting point for verification of the system.

The context of this work is a pilot project on exactly that: the verification of the L4 microkernel. In other work [20] we have demonstrated the feasibility of this as well. With an investment of about 1.5 person years we were able to specify a significant part of the L4 virtual memory subsystem and fully verify it down to C code, integrated into the kernel, running on real hardware.

Despite the good results and progress we achieved using the B method, we found that a number of restrictions were hindering our work. They are not directly a fault of the B method itself, but more of an incompatibility with our goals. The requirement for animation for instance, precluded some more convenient formalisation mechanisms. Furthermore, the B method is geared towards refinement proofs in multiple steps with code generation at the end. We do agree in principle with this technique, but although the code generation step from B to the C programming language seems appropriate for application code, it bridges too large a gap to effectively control performance critical sections in operating systems code.

We therefore decided to use the other formalism that was successfully applied in the pilot project, Isabelle/HOL [16], for the future, full verification. The main concepts of the B formalisation — a state based description of the L4 API functions will stay the same, only the notation will be different (higher order logic instead of set theory).

In fact, we are not performing a translation from B to Isabelle/HOL, but we are first developing a new version of the L4 API that introduces efficient and flexible security mechanisms. As most of the API will stay unchanged, the hope is that the experience gained in this formalisation will significantly speed up the formal specification process. Moreover, the formal specification this time is integrated with the API design from the start.

We estimate that the full verification of L4 will take about 20 person years, including verification tool development. This effort must be seen in relation to the cost of developing the kernel in the first place, and the potential benefits of verification. The present kernel was written by a three-person team over a period of 8–12 months, with significant improvements since. Furthermore, for most of the developers it was the third in a series of similar kernels they had written, which meant that when starting they had a considerable amount of experience. A realistic estimate of the cost of developing a high-performance implementation of L4 is probably at least 5–10 person years.

Under those circumstances, we argue that the full verification of L4 is highly desirable and provides a good return of investment. The kernel is the lowest and most critical part of any software stack, and any assurances on system behaviour are built on sand as long as the kernel is not shown to behave as expected. Furthermore, formal verification puts pressure on kernel designers to simplify their systems, which has obvious benefits for maintainability and robustness even when not yet formally verified.

Acknowledgements We thank Ken Robinson who supervised the honours thesis, Kevin Elphinstone who assessed it, and the L4 development team for their help and useful discussions. We also thank Ansgar Fehnker for reading drafts of this paper.

References

- [1] J.-R. Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
- [2] B-Core. The B-Toolkit. <http://www.b-core.com/btoolkit.html>, 2002.
- [3] W. R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.
- [4] W. R. Bevier and L. M. Smith. A mathematical model of the Mach kernel. Technical Report 102, Computational Logic, Inc., 1994.
- [5] T. Cattel. Modelization and verification of a multiprocessor realtime OS kernel. In *Proceedings of FORTE '94, Bern, Switzerland*, 1994.
- [6] G. Duval and J. Julliand. Modelling and verification of the RUBIS μ -kernel with SPIN. In *SPIN'95, Workshop on Model Checking of Software*, 1995.
- [7] S. Fowler and A. Wellings. Formal analysis of a real-time kernel specification. In B. Jonsson and J. Parrow, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1135, pages 440–458, Uppsala, Sweden, 1996. Springer-Verlag.
- [8] M. Gargano, M. Hillebrand, D. Leinenbach, and W. Paul. On the correctness of operating system kernels. In *Proc. 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, Oxford, UK, 2005. to appear.

- [9] M. Hohmuth, H. Tews, and S. G. Stephens. Applying source-code verification to a microkernel — the VFiasco project. Technical Report TUD-FI02-03-März, TU Dresden, 2002.
- [10] G. Klein and H. Tuch. Towards verified virtual memory in L4. In K. Slind, editor, *TPHOLs Emerging Trends '04*, Park City, Utah, USA, 2004.
- [11] R. Kolanski. A formal model of the μ -kernel api using the B method. Honours Thesis, School of Computer Science and Engineering, The University of New South Wales, Sydney, 2004.
- [12] L4 development team. L4ka::pistachio source code v0.3. <http://www.l4ka.org/download/>, 2004.
- [13] L4Ka Team. *L4 eXperimental Kernel Reference Manual Version X.2r3*, 2004. <http://www.l4ka.org>.
- [14] J. Liedtke. Towards real μ -kernels. *Communications of the ACM*, 39(9):70–77, 1996.
- [15] P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson. A provably secure operating system: The system, its applications, and proofs. Technical Report CSL-116, SRI International, 1980.
- [16] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCs*. Springer, 2002.
- [17] K. Robinson. A concise summary of the B mathematical toolkit. <http://www.cse.unsw.edu.au/~cs2110/B-Summary/>, 2005.
- [18] J. Shapiro and S. Weber. Verifying operating system security. Technical Report MS-CIS97-26, University of Pennsylvania, Philadelphia, PA, USA, 1997.
- [19] J. M. Spivey. Specifying a real-time kernel. *IEEE Software*, 7(5):21–28, September 1990.
- [20] H. Tuch and G. Klein. Verifying the L4 virtual memory subsystem. In G. Klein, editor, *Proc. NICTA FM Workshop on OS Verification*, pages 73–97. Technical Report 0401005T-1, National ICT Australia, 2004.
- [21] P. Tullmann, J. Turner, J. McCorquodale, J. Lepreau, A. Chitturi, and G. Back. Formal methods: a practical tool for OS implementors. In *HotOS-VI*, 1997.
- [22] B. J. Walker, R. A. Kemmerer, and G. J. Popek. Specification and verification of the UCLA Unix security kernel. *Communications of the ACM*, 23(2):118–131, 1980.

Combinatorial Generation by Fusing Loopless Algorithms

Tadao Takaoka

Stephen Violich

Department of Computer Science and Software Engineering
University of Canterbury
Christchurch, New Zealand
{tad, ssv10}@cosc.canterbury.ac.nz

Abstract

Some combinatorial generation problems can be broken into subproblems for which loopless algorithms already exist. We discuss means by which loopless algorithms can be fused to produce a new loopless algorithm that solves the original problem. We demonstrate this method with two new loopless algorithms, MIXPAR and MULTPERM. MIXPAR generates well-formed parenthesis strings containing two different types of parentheses. MULTPERM generates multiset permutations in linear space using only arrays; it is simpler and more efficient than the recent algorithm of Korsh and LaFollette (2004).

1 Introduction

The generation of combinatorial objects, such as combinations, permutations and parenthesis strings, is a well studied area, covered by Nijenhuis and Wilf (1975), Reingold, Nievergelt and Deo (1977), Wilf (1989) and Savage (1997).

Loopless algorithms for combinatorial generation were introduced by Ehrlich (1973). These algorithms generate each combinatorial object from its predecessor using no more than a constant number of instructions, thus they are ‘loop-free’. It follows that it should be possible to combine loopless algorithms in such a way that the resulting algorithm still satisfies this property. If a combinatorial generation problem can be broken down into subproblems for which loopless algorithms already exist, then combining those algorithms might lead to a loopless algorithm for the original problem.

This idea is not new, for example Korsh and Lipschutz (1997) and Korsh and LaFollette (2004) give loopless algorithms for multiset permutations that combine existing loopless algorithms for element selection and combination movement. We believe, however, that combining loopless algorithms has not been discussed in general before. We refer to the combining of algorithms as *fusing* because this does not limit us to any particular structures or patterns.

We introduce general program structures for fused loopless algorithms and discuss implementation issues in Section 2. We then cover Williamson’s (1985) algorithm for variations in Gray code order in Section 3, as it is the basis for many of the subsequent algorithms we discuss. We use fusing to produce MIXPAR, an algorithm for generating mixed parenthesis strings, which comprise parentheses of different types,

Copyright ©2006, Australian Computer Society, Inc. This paper appeared at Computing: The Australasian Theory Symposium (CATS2006), Hobart, Australia. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 51. Barry Jay and Joachim Gudmundsson, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.

in Section 4. A second new algorithm, MULTPERM, is presented in Section 5, and experimentally evaluated against the algorithm recently published by Korsh and LaFollette. Finally, we draw some conclusions in Section 6.

2 Fusing Loopless Algorithms

A generalised a loopless algorithm is shown in Figure 1(a). Function *init* initialises the algorithm and generates the first object, *next* generates each successive object, while *last* returns whether this current object is the final one in the sequence. Functions *next* and *last* run in $O(1)$ time, while *init* is allowed $O(n)$ time. ‘Loopless’ may seem a misnomer, since a control loop is required, but it is the generation of *each* object that is loop-free.

Two loopless algorithms can be nested so that a complete cycle of the inner algorithm runs during each iteration of the outer algorithm, as shown in Figure 1(b). Functions *next_1* and *isnext_1* belong to the inner algorithm, while *next_2* and *isnext_2* belong to the outer. Because the initial and final states of a loopless algorithm differ, a new function, *reinit_1*, is required to reinitialise the inner algorithm before it begins a new cycle. There are two ways an algorithm can be reinitialised: *refreshing* means to reset an al-

```
1.  init
2.  while not last do
3.      next
```

(a) Single loopless algorithm

```
1.  init
2.  while not last_2 do
3.      while not last_1 do
4.          next_1
5.          reinit_1
6.          next_2
```

(b) Two loopless algorithms, nested

```
1.  init
2.  while not last_2 do
3.      if not last_1 then
4.          next_1
5.      else
6.          reinit_1
7.          next_2
```

(c) Two loopless algorithms, un-nested

Figure 1: Program structures for loopless algorithms.

Algorithm 1 Williamson’s (1985) loopless algorithm for variations in Gray code order.

```

/* Initialise */
1. procedure init_Wil
2.   read  $n$ 
3.   for  $i = 1$  to  $n$  do read  $r[i]$ 
4.   for  $i = 1$  to  $n$  do  $v[i] = 1$ 
5.   for  $i = 1$  to  $n$  do  $d[i] = 1$ 
6.   for  $i = 0$  to  $n$  do  $e[i] = i$ 
7.    $j = n$ ;

/* Generate */
8. procedure next_Wil
9.    $e[n] = n$ 
10.  add  $d[j]$  to  $v[j]$ 
11.  if  $v[j]$  is either 1 or  $r[j]$  then
12.     $e[j] = e[j - 1]$ 
13.     $e[j - 1] = j - 1$ 
14.     $d[j] = -d[j]$ 
15.     $j = e[n]$ 

/* Main */
16. init_Wil
17. print  $v$ 
18. while  $j$  is not 0 do
19.   next_Wil
20. print  $v$ 

```

algorithm to its initial state; *reversing* means to alter the algorithm so it will run from its final state back to its initial state over a cycle. Since reinitialisation occurs between objects, *reinit* is only allowed $O(1)$ time. Although these nested loopless algorithms contain an extra while loop, successive objects are still generated in no more than a constant number of instructions.

For greater clarity, the nested structure can be modified into an *un-nested* structure by replacing the second while loop with an if-then-else statement, as shown in Figure 1(c). This un-nested configuration executes the functions in the same order as the nested configuration, but now a single loop-free algorithm that generates exactly one object per iteration can be isolated within the program. The new algorithms that we develop in Sections 4 and 5 adhere to this un-nested structure.

Although *reinit_1* is limited to $O(1)$ time, there are a couple of tricks for fitting $O(n)$ -time reinitialisation into this framework. For example, the final state of an algorithm might include some array $a_{1..n}$ that has $O(n)$ points of difference from its initial state. Supposing the algorithm is irreversible, then it requires $O(n)$ time to reinitialise. One option, available if the algorithm finishes with different a_i at different stages during its cycle, is to reinitialise each a_i as soon as it becomes obsolete, during iterations of *next_1*. In this way, $O(n)$ reinitialising steps can be executed in $O(1)$ time per object, a technique we call *time-stealing*. In the best case, this algorithm would give cues as to exactly when each a_i becomes obsolete; in the worst, a for-loop would be simulated, using a counter variable and an arbitrary start cue. We use this time-stealing technique to iteratively re-initialise array s in algorithm MULTPERM in Section 5. A second option is less elegant and much less efficient, although it seems universally applicable: maintain two separate versions of the troublesome arrays or variables. Then, in any given cycle of the inner algorithm, one version can be used while the other is reinitialised as per time-stealing.

	$v_{1..3}$	$e_{0..3}$	j		$v_{1..3}$	$e_{0..3}$	j
1.	1 1 1	0 1 2 3	3	10.	<u>2</u> 3 3	<u>0</u> 0 2 3	3
2.	1 1 2	0 1 2 3	3	11.	2 3 <u>2</u>	0 0 2 3	3
3.	1 1 <u>3</u>	0 1 <u>2</u> 2	2	12.	2 3 <u>1</u>	0 0 <u>2</u> 2	2
4.	1 2 3	0 1 2 3	3	13.	2 2 1	0 0 2 3	3
5.	1 2 2	0 1 2 3	3	14.	2 2 2	0 0 2 3	3
6.	1 2 <u>1</u>	0 1 <u>2</u> 2	2	15.	2 2 <u>3</u>	0 0 <u>2</u> 2	2
7.	1 <u>3</u> 1	0 <u>1</u> 1 3	3	16.	2 <u>1</u> 3	0 <u>1</u> 0 3	3
8.	1 3 2	0 1 1 3	3	17.	2 1 2	0 1 0 3	3
9.	1 3 <u>2</u>	0 1 <u>2</u> 1	1	18.	2 1 <u>1</u>	0 1 <u>2</u> 0	0

Figure 2: Output for Williamson’s algorithm for inputs $n = 3$, $r = \{2, 3, 3\}$. Each $v[i]$ varies between 1 and $r[i]$ inclusive. Underlines indicate when $v[j]$ becomes extremal, and the corresponding conveying from $e[j - 1]$ to $e[j]$ and resetting of $e[j - 1]$. Note that j changes at the end of each iteration, so the value of j used to generate any v and e is on the preceding line.

3 Williamson’s Algorithm

We include a discussion of Williamson’s (1985, p.112) loopless algorithm for generating variations in Gray code order because its recursion-simulation technique is used by three out of the four subsequent algorithms in this paper. The algorithm generates elements of the product space $S = S_1 \times S_2 \times \dots \times S_n$, with $S_i = 0, 1, \dots, r_i - 1$ for $i = 1, 2, \dots, n$. Williamson’s algorithm is shown in Algorithm 1.

The variables in Williamson’s algorithm are: $v_{1..n}$, the current variation; j , the current position in v to change; $d_{1..n}$, the current increment (1 or -1) for each position in v ; and $e_{0..n}$, which determines the order in which positions in v should be selected as values for j . Values for n and all $r[i]$ are read from the user. The remaining variables are initialised as follows: all v_i are set to 0; all d_i are set to 1; all e_i are set to i ; and j is set to n . Array e is used to looplessly simulate a recursive tree traversal. Though this technique is well known and comprises only a few lines of code, it is nontrivial and rarely explained.

When e_i is set to i , we say that e_i is *reset*, since i was the initialised value of e_i . When v_j becomes extremal, the value at e_{j-1} is passed along one place to e_j , then e_{j-1} is reset. Referring to the coding tree in Figure 2, this can be seen when $v = \{1, 3, 3\}$, for example. Because v_3 has become a last child, e_3 inherits the value 1 from e_2 , while e_2 is reset to 2.

A similar pass-reset pattern occurs between e_n and variable j . At the end of every iteration of *next* the value at e_n is passed along to variable j ; at the start of the next iteration, e_n is reset. Referring again to Figure 2, the resetting of e_3 is visible when $v = \{1, 1, 3\}$, $\{1, 2, 1\}$, and so on. It *happens* on every line, of course, but can only be seen when e_3 was not already 3 and was not subsequently changed.

In effect, e can be thought of as a conveyor belt that passes information along towards variable j . It is helpful to picture variable j as positioned immediately after e_n , since information flows along array e and into j . Whenever information is passed along, the source of that information is reset.

Any value i can only enter the array by resetting e_i . When e_i inherits a value from e_{i-1} , that value instead of i will be carried towards variable j . That means that v_i will be skipped over on the next occasion that would have otherwise been its turn to be changed.

When v_i is skipped, and one of its ancestors is changed, v_i becomes a first child, so it should not be skipped again. Thus, as soon the value of e_i is passed on, e_i is reset. This means the *subsequent* value to be passed from e_i will be i again, making v_i available

<i>par</i>	<i>mix</i>	<i>mixpar</i>
(() ())	(((((((()))))))	(((((((()))))))
	(([((((([]))))))	(((((([] [])))))
	([(((([] [])))))	([((([] [])))))
	[[(([[] []))]]	[[([[] [])]]]
	[([[([] [])]]]	[([[([] [])]]]
((()))	[((([())]))	[((([())]))
	...	

(a) Par-outside-mix (par-mix)

<i>mix</i>	<i>par</i>	<i>mixpar</i>
([[(() ())]]	(((((((()))))))	(((((((()))))))
	(((((([] [])))))	((((([] [])))))
	((((([] [])))))	((([[] []))))
	((()) ())	([]) []
([((()))])	...	

(b) Mix-outside-par (mix-par)

Figure 3: Sample outputs for mixpar algorithms with opposite nesting configurations.

for change. Note that if the value of e_i was already i before it was passed along then resetting e_i has no effect.

Both of our new algorithms, MIXPAR and MULTPERM, in Sections 4 and 5 respectively, use the Williamson's variables j , d and e to select elements for change. MIXPAR uses a second set, labelled jj , dd and ee , since both of its component algorithms follow the Williamson model.

4 Mixed Parenthesis Strings

The first combinatorial generation problem we apply our fusing framework to is in the area of parenthesis strings. A well-formed parenthesis string, or *par* for short, can be derived from the grammar $P \rightarrow \epsilon \mid (P) \mid PP$. A *par* has n pairs, and so its size is $2n$.

We introduce a new combinatorial object: *mixed parenthesis strings*, or *mixpars* for short, which comprise parentheses of different types. In this paper we limit the number of types to two, but it is trivial to extend the ideas beyond binary. The grammar for a mixpar is a modification of that for a *par*, in this case $M \rightarrow \epsilon \mid (M) \mid [M] \mid MM$. Thus, a mixpar is well-formed if its parentheses are arranged as per an ordinary *par*, and if both parentheses in each pair share the same type. For example, $() []$ and $([])$ are a valid mixpars, while $(] [)$ and $([)]$ are not.

A mixpar can be thought of as a *par* with a certain *mix* of types. For example, the mixpar $() []$ can be described as the *par* $(())$ with the *mix* $([]$. Note that with only two types, a *mix* corresponds to a binary string. It follows that generating all mixpars for some n is a matter of generating either all mixes for each *par* or all *pars* for each *mix*. Thus, an algorithm for generating mixpars nests algorithms for generating *pars* and mixes in some way. Because loopless algorithms for *pars* and binary strings exist, we hypothesized that a loopless algorithm for generating mixpars could be fused from these. This fusion is carried out within the framework discussed in Section 2.

<i>par</i>	<i>mix</i>	<i>mixpar</i>
(() ())	(((((((()))))))	(((((((()))))))
	[(((([())]))]	((((([())])))
((()))	(((((((()))))))	(((((((()))))))
	[(((([())]))]	((((([())])))
(()) ()	(((((((()))))))	(((((((()))))))
	...	

(a) Refreshing.

<i>par</i>	<i>mix</i>	<i>mixpar</i>
(() ())	(((((((()))))))	(((((((()))))))
	[(((([())]))]	((((([())])))
((()))	(((((((()))))))	(((((((()))))))
	[(((([())]))]	((((([())])))
(()) ()	(((((((()))))))	(((((((()))))))
	...	

(b) Reversing.

Figure 4: Sample outputs for mixpar algorithms refreshing and reversing the inner mix algorithm respectively.

The way in which the two algorithms are nested affects the modifications required to make each algorithm operate directly on mixpars. Figure 3 shows output for mixpar algorithms with the two possible nesting configurations, *par*-outside-*mix* and *mix*-outside-*par*. (The *par* algorithm used is that of Xiang and Ushijima (2001), which is the one we ultimately chose and will discuss later; the *mix* algorithm is simply a Gray code generator.) From Figure 3(a) it can be seen that each iteration of the *mix* algorithm must change the type of one pair in the mixpar. Figure 3(b) shows that each iteration of the *par* algorithm must change the places or types of two to four parentheses. The *mix*-*par* configuration seemed to require more difficult modification to its inner algorithm, so we opted for the *par*-*mix* arrangement.

The method of reinitialising the inner algorithm also has an impact on the difficulty of fusing the algorithms. Recalling Section 2, inner algorithms can be reinitialised by either refreshing or reversing. Figure 4 shows output for mixpar algorithms that refresh and reverse their inner algorithms respectively. From Figure 4(a) it can be seen that refreshing the *mix* algorithm means that all parentheses are round whenever it is the *par* algorithm's turn to operate. (This takes advantage of the fact that the last object in a Gray code has only one point of difference to the first object.) Figure 4(b) shows that reversing the *mix* algorithm means the *par* algorithm will frequently have to cope with one pair of an alternate type. Again, we opted for the simpler option, that of refreshing rather than reversing the *mix* algorithm.

In order to change the types of pairs, the positions of the parentheses in each pair must be known. Let l_i be the position of the i th left parenthesis, and let r_i be the position of the *partner* of the i th left parenthesis (that is, not simply the i th right parenthesis as counted from the start). For example, for the mixpar $((()))$, $l_2 = 2$ and $r_2 = 5$.

Although we do not know of a loopless *par* algorithm that correctly maintains all l_i and r_i , Xiang and Ushijima's (2001) algorithm does correctly maintain all l_i . We now present a method for finding all r_i in

Algorithm 2 Xiang and Ushijima's (2001) loopless algorithm for parenthesis strings.

```

/* Initialise */
1. procedure init_XU
2.   read  $n$ ;
3.   for  $i = 1$  to  $2n$  by  $2$  do
4.     set  $par[i]$  to '(',  $par[i + 1]$  to ')'
5.   for  $i = 1$  to  $n$  do  $l[i] = 2i - 1$ 
6.   for  $i = 1$  to  $n$  do  $d[i] = 1$ 
7.   for  $i = 0$  to  $n$  do  $e[i] = i$ 
8.    $j = n$ 

/* Generate */
9. procedure next_XU
10.   $e[n] = n$ 
11.   $i = l[j]$ 
12.  if  $d[j]$  is 1 then
13.    if  $l[j]$  is  $2j - 1$  then
14.       $l[j] = l[j - 1] + 1$ 
15.    else
16.      add 1 to  $l[j]$ 
17.    else
18.      if  $l[j]$  is  $l[j - 1] + 1$  then
19.         $l[j] = 2j - 1$ 
20.      else
21.        subtract 1 from  $l[j]$ 
22.      swap  $par[i]$  and  $par[l[j]]$ 
23.      if  $l[j] \geq 2j - 2$  then
24.         $d[j] = -d[j]$ 
25.         $e[j] = e[j - 1]$ 
26.         $e[j - 1] = j - 1$ 
27.       $j = e[n]$ 

/* Main */
28. init_XU
29. print  $par$ 
30. while  $j$  is not 1 do
31.   next_XU
32.   print  $par$ 

```

constant time per object. The entire mixpar cannot be scanned after every iteration of the par algorithm, as that would require $O(n)$ time, so the solution is to use the time-stealing technique mentioned in Sect. 2, finding each r_i in $O(1)$ time during iterations of the mix algorithm.

We say a parenthesis pair is *empty* if no pairs are nested inside it. Recalling the grammar for a par, the n th pair must be empty, since no subsequent pairs exist. Thus:

$$r_n = l_n + 1 \quad (1)$$

It follows that the $(n - 1)$ th pair must be empty or nested around the n th pair. Our algorithm is based on the idea that, if we start from the n th pair and work backwards to the first, each pair must be either empty or nested around some substring comprising pairs we have already encountered. Thus, information about substrings must be stored. Let s_{l_i} be the position after the longest well-formed substring beginning at l_i . For example, for the mixpar $((()))$, $l_2 = 2$ and $s_{l_2} = s_2 = 6$. Because we cannot know all s_i immediately, our algorithm initialises array $s_{1..2n}$ such that all $s_i = i$. Equation (1) is the base step of our induction. We now show how each successive s_{l_i} and r_i can be found in constant time by working backwards from $i = n$.

If there is no j th left parenthesis immediately after r_i , then the substring beginning at l_i ends at r_i , and s_{r_i+1} will not have changed since initialisation.

	<i>par</i>	<i>l</i>
1.	() () () ()	1 3 5 7
2.	() () () ()	1 3 5 6
3.	() () () ()	1 3 4 6
4.	() () () ()	1 3 4 5
5.	() () () ()	1 3 4 7
6.	() () () ()	1 2 4 7
7.	() () () ()	1 2 4 5
8.	() () () ()	1 2 4 6
9.	() () () ()	1 2 3 6
10.	() () () ()	1 2 3 5
11.	() () () ()	1 2 3 4
12.	() () () ()	1 2 3 7
13.	() () () ()	1 2 5 7
14.	() () () ()	1 2 5 6

Figure 5: Xiang and Ushijima's algorithm output for $n = 4$.

On the other hand, if r_i is adjacent to some l_j , then the substrings beginning at l_i and l_j end in the same position, and because we are working backwards from the n th pair, s_{l_j} will already have been set correctly. Thus, we derive an unconditional equation for s_{l_i} independent of j :

$$s_{l_i} = \begin{cases} r_i + 1 & = s_{r_i+1} \text{ iff } r_i + 1 \neq l_j \\ s_{l_j} & = s_{r_i+1} \text{ iff } r_i + 1 = l_j \end{cases} \quad (2)$$

Similarly for r_i , if the $(i + 1)$ th left parenthesis is not immediately after l_i , then r_i must be, and $s_{l_{i+1}}$ will not have changed since initialisation. Conversely, if the i th and $(i + 1)$ th left parentheses are adjacent, then r_i must be immediately after the substring starting at l_{i+1} . Because we are working backwards from the n th pair, $s_{l_{i+1}}$ will already have been set correctly. Thus, we derive an unconditional equation for r_i :

$$r_i = \begin{cases} l_i + 1 & = s_{l_{i+1}} \text{ iff } l_i + 1 \neq l_{i+1} \\ s_{l_{i+1}} & = s_{l_i+1} \text{ iff } l_i + 1 = l_{i+1} \end{cases} \quad (3)$$

Thus, using (1), (2) and (3), right parentheses from n th to first can be found in $O(1)$ time each, during iterations of the first half of the Gray cycle. As we finish with each r_i during the second half of the Gray cycle, we reset each s_{l_i} .

We now cover Xiang and Ushijima's par algorithm. In addition to correctly maintaining all l_i , it is a very efficient loopless par algorithm in terms of time and space. It is also very simple, which helped keep our final MIXPAR algorithm simple. Xiang and Ushijima's algorithm is shown in Algorithm 2 (note that we have renamed their array for the positions of the left parentheses from p to l for consistency with our right-finding approach). Its element-selection mechanism is familiar from Williamson's algorithm in Section 3, although its element-change code is a little more complex.

Xiang and Ushijima's algorithm introduces several new variables. As mentioned, the number of parenthesis pairs is n , which is read from the user. The par is stored in $par_{1..2n}$, while the left parentheses positions are stored in $l_{1..n}$. These are initialised to () () ... () and $1, 3, \dots, 2n - 1$ respectively. Finally, i and c are temporary variables used to facilitate an array swap, storing an integer and character respectively. Variables j , $d_{1..n}$ and $e_{0..n}$ are inherited from Williamson's algorithm, and relate to the left parentheses; initialisations remain the same.

It works in the same way as their combinations algorithm from the same paper; both are variations

Algorithm 3 MIXPAR, a new, loopless algorithm for mixed parenthesis strings

```

/* Initialise */
1. procedure init_Mix
2.   init_XU /* From Alg. 2 */
3.   for  $i = 1$  to  $n$  do  $dd[i] = 1$ 
4.   for  $i = 0$  to  $n$  do  $ee[i] = i$ 
5.   for  $i = 1$  to  $2n$  do  $s[i] = i$ 
6.    $jj = n$ 
7.    $t = n$ 

/* Find right parenthesis */
8. procedure find
9.   if  $dd[1]$  is 1 then
10.     $r[jj] = s[l[jj] + 1]$ 
11.    if  $jj$  is not 1 then
12.       $s[l[jj]] = s[r[jj] + 1]$ 
13.      subtract 1 from  $t$ 
14.    else
15.       $s[l[jj]] = l[jj]$ 
16.      add 1 to  $t$ 

/* Generate by Gray */
17. procedure next_Gray
18.    $ee[n] = n$ 
19.   if  $jj$  is  $t$  then find
20.   change  $par[l[jj]]$  and  $par[r[jj]]$  from
   round to square or vice versa
21.    $ee[jj] = ee[jj - 1]$ 
22.    $ee[jj - 1] = jj - 1$ 
23.    $dd[jj] = -dd[jj]$ 
24.    $jj = ee[n]$ 

/* Re-initialise Gray */
25. procedure reinit_Gray
26.   change  $par[l[1]]$  and  $par[r[1]]$  to round
27.    $dd[1] = 1$ 
28.    $jj = n$ 
29.    $t = n$ 

/* Main */
30. init_Mix
31. print  $par$ 
32. while  $j$  is not 1 or  $jj$  is not 0 do
33.   if  $jj$  is not 0 then
34.     next_Gray
35.   else
36.     reinit_Gray
37.   next_XU /* From Alg. 2 */
38.   print  $par$ 

```

on Williamson's algorithm in which no two elements in the same object can have the same value. Xiang and Ushijima noted that parentheses maintain a relative order, that is $l_1 < l_2 < \dots < l_n$, and that well-formedness dictates how far to the right each left parenthesis can travel, that is $l_i \leq 2i - 1$ for $1 \leq i \leq n$. At any time, these principles determine the upper and lower bounds for left parenthesis travel.

Xiang and Ushijima extended Williamson's algorithm to have four patterns of change: O^+ , $O^{+'}$, O^- and $O^{-'}$. The regular positive direction, O^+ , causes a parenthesis to move steadily right between its current bounds. The prime positive direction, $O^{+'}$, causes a parenthesis to jump from its lower bound to its upper bound, then move steadily left through all remaining values. The negative directions have the opposite effects. These jumps in the prime directions allow the algorithm to avoid clashes (different elements sharing

1. () () ()	25. ((()))
2. () () []	26. (([]))
3. () [] []	27. ([[]])
4. () [] ()	28. ([()])
5. [] [] ()	29. [[()]]
6. [] [] []	30. [[[]]]
7. [] () []	31. [([])]
8. [] () ()	32. [(())]
9. () (())	33. (()) ()
10. () ([])	34. (()) []
11. () [[]]	35. ([]) []
12. () [()]	36. ([]) ()
13. [] [()]	37. [[]] ()
14. [] [[]]	38. [[]] []
15. [] ([])	39. [()] []
16. [] (())	40. [()] ()
17. (() ())	
18. (() [])	
19. ([] [])	
20. ([] ())	
21. [[] ()]	
22. [[] []]	
23. [() []]	
24. [() ()]	

Figure 6: MIXPAR algorithm output for $n = 3$. Line-breaks have been inserted to highlight when the par is changed by the outer algorithm.

the same value) while generating all combinations of left parenthesis positions.

Output for Xiang and Ushijima's algorithm for $n = 4$ is shown in Figure 5. All l_i begin maximally, and increment or decrement in a pattern similar, at first glance, to that of Williamson's algorithm. Closer examination of lines 2–5, however, reveals the effect of a prime direction jump. On line 2, l_4 is minimal, so in Williamson's algorithm you would expect it to reverse direction next time it moved. But on line 3, the change to l_3 means that l_4 is no longer minimal. On line 4, a prime jump is employed so that l_4 can take the newly available minimum value before ascending as per usual to the maximum on line 5.

Algorithm MIXPAR, our new mixed parenthesis strings algorithm, is given in Algorithm 3. A complete C++ program is given in Appendix A. It's main statements (lines 30–38) reveal that it fits exactly into the un-nested structure outlined in Section 2. The initialisation and next methods belonging to Xiang and Ushijima's algorithm are able to be incorporated verbatim.

Most of the variables in MIXPAR are inherited from its constituent algorithms. From Xiang and Ushijima's algorithm come the variables n , $par_{1..2n}$, $l_{1..n}$, j , $d_{1..n}$, $e_{0..n}$, i and c . From Williamson's algorithm, to run our mix (Gray code) algorithm, come the variables jj , $dd_{1..n}$ and $ee_{0..n}$. All initialisations are as previously described.

Three new variables are introduced. Finding right parentheses requires arrays $r_{1..n}$ and $s_{1..2n}$, of which r is not initialised and the initialisation of s has already been covered. Finally, to keep track of which right parenthesis is due to be found during the first half of the Gray cycle, and which value of s is due to be refreshed during the second half, we use variable t ; initially $t = n$.

A sample output of MIXPAR for $n = 3$ is shown in Figure 6. The output is displayed in columns separated by newlines, where each column begins with a par generated by Xiang and Ushijima's algorithm. The remaining lines in each column show complete Gray code cycles of mixes for that column's par.

Algorithm 4 Chase's (1989) loopless algorithm for combinations by $O(1)$ -distance transpositions.

```

/* Initialise */
1. procedure init_Chase
2.   read  $n$  and  $r$ 
3.   for  $i = 1$  to  $n$  do  $comb[i] = i$ 
4.    $comb[i] = 2r - 1$ 
5.    $z = n + 1$ 
6.   Set  $b$  to 1 if  $r$  is even, else 2

/* Next */
7. procedure next_Chase
8.   if  $z$  is 1 then
9.     if  $inc(1)$  then
10.      if  $adj(1)$  then
11.        if  $inc(2)$  then  $move(1, 1, 2)$ 
12.        else  $move(2, -1, 2)$ 
13.        else  $move(1, 1, 1)$ 
14.        else  $move(1, -1, 1)$ 
15.      else
16.        if  $inc(z - 1)$  then
17.          if  $z > 2$  and  $inc(z - 2)$  then
18.             $move(z - 2, 1, 2)$ 
19.            else  $move(z - 1, 1, 1)$ 
20.          else
21.            if not  $adj(z)$  then
22.              if  $inc(z)$  then  $move(z, 1, 1)$ 
23.              else  $move(z, -1, 1)$ 
24.            else
25.              if  $inc(z + 1)$  then
26.                 $move(z, 1, 2)$ 
27.                else  $move(z + 1, -1, 2)$ 

/* Move comb elements */
28. procedure move( $p, d, s$ )
29.    $x = comb[p]$ 
30.    $y = x + s \times d$ 
31.    $comb[p] = x + d$ 
32.    $comb[p + d(s - 1)] = y$ 
33.   if  $comb[z]$  is  $z$  then
34.     add  $s$  to  $z$ 
35.   if  $comb[z]$  is  $z$  then add  $s$  to  $z$ 
36.   else if  $comb[z - 1]$  is not  $z - 1$  then
37.     subtract  $s$  from  $z$ 

/* Returns comb[i] increasing? */
38. function inc( $i$ )
39.   return  $comb[i + 1]$  is odd

/* Returns comb[i] and [i+1] adjacent? */
40. function adj( $i$ )
41.   return  $comb[i] + 1$  is  $comb[i + 1]$ 

/* Main */
42. init_Chase
43. print  $comb$ 
44. while  $comb[n - b]$  is not minimal or
    $comb[n - b + 1]$  is not maximal do
45.   next_Chase
46.   print  $comb$ 

```

5 Multiset Permutations

The second combinatorial generation problem we apply our fusing framework to is that of multiset permutations. A multiset, or set with repetitions, has k distinct elements, which we assume without loss of generality to be the integers $[1, k]$. Each distinct el-

	<i>comb</i>	bit vector	z	case
1.	1 2 3 4	1 1 1 1 0 0	5	6
2.	1 2 3 5	1 1 1 0 1 0	4	5
3.	1 3 4 5	1 0 1 1 1 0	2	6
4.	2 3 4 5	0 1 1 1 1 0	1	2
5.	1 2 4 5	1 1 0 1 1 0	3	9
6.	1 2 5 6	1 1 0 0 1 1	3	6
7.	1 3 5 6	1 0 1 0 1 1	2	6
8.	2 3 5 6	0 1 1 0 1 1	1	1
9.	3 4 5 6	0 0 1 1 1 1	1	4
10.	2 4 5 6	0 1 0 1 1 1	1	4
11.	1 4 5 6	1 0 0 1 1 1	2	10
13.	2 3 4 6	0 1 1 1 0 1	1	2
14.	1 2 4 6	1 1 0 1 0 1	3	8
15.	1 2 3 6	1 1 1 0 0 1	4	

Figure 7: Chase's algorithm output for $n = 4$, $r = 6$. The case column identifies which of the ten move calls is used to generate each object.

ement i has a multiplicity m_i , which is the number of times it appears in the multiset. The size n of the multiset is the sum of all multiplicities. For example, the multiset $\{1, 1, 1, 2, 2, 3\}$ has $k = 3$, $m = \{3, 2, 1\}$ and $n = 6$. Indistinguishable elements are called *similar*.

Our approach to generating multiset permutations is based on the Johnson (1963) and Trotter (1962) algorithms for set permutations, which work by iteratively moving single elements through subpermutations. We reasoned that a modified algorithm could iteratively move groups of similar elements through subpermutations, thereby generating multiset permutations, and that this grouped element movement could be achieved using a combinations algorithm. A similar approach was taken by Korsh and LaFollette (2004) to develop the first linear-space loopless multiset permutations algorithm using only arrays. We subsequently draw attention to several important design differences that led us to choose a more advantageous combinations algorithm than Korsh and LaFollette, and ultimately develop a simpler and more efficient algorithm. Other multiset permutations algorithms based on combining algorithms include Korsh and Lipschutz (1997) and Vajnowski (2003).

A recursive algorithm for multiset permutations is as follows. Let $perm$ be a multiset permutation of n integers. Let $subp_i$ be a subpermutation of $perm$ comprising all elements *greater than* i . Initially $perm$ is the lexicographically least permutation. If $k = 1$ then $perm$ is the only permutation. Otherwise, the 1s are placed among $subp_1$ in all remaining distinct ways such that the relative order of elements of $subp_1$ is maintained, and $subp_1$ is contiguous in the final permutation. This generates all permutations containing $subp_1$. If there is another $subp_1$ of $perm$, it is generated recursively, and the next $perm$ becomes this next $subp_1$ bounded by the 1s. The 1s are now placed among this next $subp_1$ in all remaining distinct ways, subject to the same conditions as before. This generates all permutations containing this next $subp_1$. This process of moving 1s through $subp_1$ s continues until they have appeared in all distinct ways in the last $subp_1$. When the k integers are distinct this algorithm mimics the Johnson-Trotter.

The recursive algorithm we describe is similar to that described by Korsh and LaFollette, with one important difference: when the similar elements finish moving through a subpermutation, Korsh and LaFollette require that they all be at one end (left or right) of the subpermutation; we require only that the subpermutation be contiguous, meaning the similar elements may finish distributed across both ends. This

Algorithm 5 MULTPERM, a new multiset permutations algorithm.

```

1.  /* Initialise */
2.  procedure init_Mul
3.    read  $k$ 
4.    for  $i = 1$  to  $k$  do read  $m[i]$ 
5.    set  $n$  to the sum of all  $m$ 
6.    for  $i = 1$  to  $k$  do
7.      set  $o[i]$  to the sum of  $m[1]$  to  $[i - 1]$ 
8.      for  $i = 1$  to  $k$  do
9.        set  $r[i]$  to the sum of  $m[i]$  to  $[k]$ 
10.     for  $i = 1$  to  $k$  do
11.       for  $j = 1$  to  $m[i]$  do
12.          $perm[j + o[i]] = i$ 
13.       for  $i = 1$  to  $k$  do  $d[i] = 1$ 
14.       for  $i = 1$  to  $k$  do  $e[i] = i$ 
15.       for  $i = 1$  to  $k$  do
16.         for  $j = 1$  to  $m[i]$  do  $comb[i][j] = j$ 
17.          $comb[i][m[i] + 1] = 2r[i] + 1$ 
18.          $z[i] = m[i] + 1$ 
19.         for  $i = 1$  to  $k - 1$  do
20.            $a[i] = i + 1$ 
21.           set  $b[i]$  to 1 if  $m[i]$  is 1 or  $r[i]$  is even,
22.           else 2
23.            $j = 1$ 
24.
25.  /* Generate */
26.  procedure next_Mul
27.     $e[1] = 1$ 
28.    determine  $x$  and  $y$  as per Chase, but
29.    using  $comb[j]$  and  $z[j]$ 
30.     $perm[x + o[j]] = perm[y + o[j]]$ 
31.     $perm[y + o[j]] = j$ 
32.    if  $a[j] < k$  then
33.       $o[a[j]] = o[a[j]] - b[j] \times d[j]$ 
34.      add 1 to  $a[j]$ 
35.    if ( $comb[j][m[j] - b[j]]$  is minimal
36.    and  $comb[j][m[j] - b[j] + 1]$  is maximal)
37.    or  $comb[j][m[j] - b[j]]$  is minimal then
38.       $e[j] = e[j + 1]$ 
39.       $e[j + 1] = j + 1$ 
40.       $d[j] = -d[j]$ 
41.       $a[j] = j + 1$ 
42.       $j = e[1]$ 
43.
44.  /* Main */
45.  init_Mul
46.  print  $perm$ 
47.  while  $j$  is not  $k$  do
48.    next_Mul
49.    print  $perm$ 

```

more relaxed requirement meant we had more combinations algorithms to choose from than Korsh and LaFollette. Besides requiring that the 0s (in terms of bit vector notation) finish as a contiguous substring, we also required, as per our recursive algorithm, that the relative order of 0s be maintained, and that the algorithm be reversible in $O(1)$ time. We preferred that the algorithm's transpositions be limited to $O(1)$ distance, as this would avoid significant extra book-keeping.

The combinations algorithm we chose was that of Chase (1989), shown in Algorithm 4. We regret that a full explanation of Chase's algorithm is outside the scope of this paper, but we hope that our overview will satisfy the reader's curiosity enough to accept Chase's algorithm as component for use in our MULTPERM algorithm.

1. 1 1 2 2 3	11. 2 3 2 1 1	21. 1 1 3 2 2
2. 1 2 1 2 3	12. 2 3 1 2 1	22. 1 3 1 2 2
3. 2 1 1 2 3	13. 2 1 3 2 1	23. 3 1 1 2 2
4. 2 2 1 1 3	14. 1 2 3 2 1	24. 3 2 1 1 2
5. 2 1 2 1 3	15. 1 2 3 1 2	25. 3 1 2 1 2
6. 1 2 2 1 3	16. 2 1 3 1 2	26. 1 3 2 1 2
7. 1 2 2 3 1	17. 2 3 1 1 2	27. 1 3 2 2 1
8. 2 1 2 3 1	18. 2 1 1 3 2	28. 3 1 2 2 1
9. 2 2 1 3 1	19. 1 2 1 3 2	29. 3 2 1 2 1
10. 2 2 3 1 1	20. 1 1 2 3 2	30. 3 2 2 1 1

Figure 8: MULTPERM output for $k = 3$, $m = 2, 2, 1$.

We have altered the algorithm so that all decision making is clear (optimised shortcuts have been replaced with assumed original conditional statements) and so that the algorithm can run both forwards and backwards. Its 1- or 2-apart transpositions means the relative order 0s is easily maintained. It is easily reversible, requiring only the inversion of one boolean function. It starts with 1s *all-left* ($1^n 0^k$) and finishes in one of two easily recognisable arrangements: *one-right* ($1^{n-1} 0^{k-n} 1$) iff $n = 1$ or k is even; or *two-right* ($1^{n-2} 0^{k-n} 11$) iff $n > 1$ and k is odd. Another benefit is that it uses very few variables.

The variables in Chase's algorithm are: $comb_{1..n+1}$, the current combination; z , the position in $comb$ of the first non-minimal element, that is the lowest i such that $comb_i > i$; and x and y , the values exiting and entering $comb$ respectively. Values for n and r are read from the user. All $comb_i$ are set to i , except $comb_{n+1}$ which is initialised to $2r + 1$. Variable z is set to $n + 1$.

The functions in Chase's algorithm are: $adj(i)$, which returns whether $comb_i$ and $comb_{i+1}$ are adjacent, that is whether $comb_i + 1 = comb_{i+1}$; and $inc(i)$, which returns whether $comb_i$ is increasing or not, which is equivalent to $comb_{i+1} \bmod 2$. In Chase's algorithm, each position's direction is determined by the next position's parity; inverting function inc makes the algorithm run in reverse.

The many nested if-then-else statements evaluate directions and adjacencies of certain elements within one or two positions of $comb_z$, the first non-minimal element. These classify the current state of $comb$ and z into one of ten cases, which determine which transposition to make. We have isolated this transposition in procedure *move*, whose parameters are the position, direction and span (distance) of the transposition. Output for Chase's algorithm is shown in Figure 7.

To fuse a loopless multiset permutations algorithm from Williamson's and Chase's algorithms required surprisingly few modifications. Each of the k groups of similar elements moves as a combination through its subpermutation, requiring its own Chase data. Thus, Chase's variable z and array $comb_{1..n}$ were extended by one dimension each to $z_{1..k}$ and $comb_{1..k, 1..m_i}$ respectively. Each $comb_i$ is of length m_i . An extra terminating condition was added, since Chase's algorithm would now be running backwards as well as forwards. Williamson's algorithm was altered to start with $j = 1$ instead of n , and its second (incrementing/decrementing) step was replaced with the modified Chase's algorithm. Thus Williamson's algorithm selects the similar elements j to move, and Chase's algorithm moves them among $subp_j$ in combination fashion, using $comb_j$ and z_j .

Algorithm MULTPERM, our new multiset permutations algorithm, is given in Algorithm 5; a complete C++ program is given in Appendix B. The appendix was written to match the style of Korsh and LaFollette's algorithm, for a more accurate comparison.

	Uniform	
	KL04	MULTPERM
Permutations	168,168,000	168,168,000
Mean Time (s)	31.3	21.5
	Varied	
	KL04	MULTPERM
Permutations	75,675,600	75,675,600
Mean Time (s)	14.2	9.4

Table 1: Results from experimental evaluation showing that MULTPERM runs 31–34% faster than KL04. Evaluation was over two multisets with many million permutations; multiplicities were uniform $\{3, 3, 3, 3, 3\}$ and varied $\{2, 3, 5, 2, 3\}$ respectively. Both algorithms generated the expected numbers of permutations.

To translate the relative transpositions of elements in Chase combinations to absolute transpositions in the multiset permutation, $perm_{1\dots n}$, required several new variables: $o_{1\dots k}$, the absolute offsets for each combination; $a_{1\dots k}$, which keeps track of the offsets that have been updated for the current j 's Chase cycle; and $b_{1\dots k}$, the number (one or two) of elements that finish right for each combination. For any selected group of similar elements j , each complete Chase cycle displaces subsequent subpermutations by b_j (reverse cycle) or $-b_j$ (forward cycle) positions. Thus all o_i for $i > j$ must be updated during the Chase cycle for j . This is achieved using the time-stealing method mentioned in Section 2, in which what would be a for-loop is distributed over subsequent calls to function *next*. In this case, over several calls to *next*, a_j counts from $j + 1$ to $k - 1$, and each o_{a_j} is incremented or decremented by b_j . To recognise when forward Chase cycles are complete, that is when combinations are one-right or two-right, array $r_{1\dots k}$ stores the maximum value that may appear in each of the combinations.

Reversing Chase's algorithm requires no re-initialisation. We have tied function *inc* to Williamson's array d , so changing the sign of d_j inverts *inc*, reversing the algorithm.

MULTPERM runs in constant time per object and requires linear space. Referring to Algorithm 5, lines 24, 31–35, and 36 correspond to the first, third and fourth steps of Williamson's algorithm respectively. Line 25 is where Chase's algorithm is used, while lines 26–30 translate Chase's transpositions to the multiset permutation; these steps together correspond to the second step of Williamson's algorithm. A sample output of MULTPERM for $k = 3$, $m = \{3, 2, 1\}$ is shown in Figure 8.

We experimentally evaluated MULTPERM against Korsh and LaFollette's algorithm, which we label KL04. Both programs were implemented in C++, and the structure, procedure calls, and I/O were made as similar as possible; this is evident in Appendix B. Timing included the initialisation and memory-clearing procedures. By convention, output statements were replaced by statements incrementing a counter, whose final value was output to verify that the correct number of objects were generated.

We ran the experiment over two multisets, each with millions of distinct permutations, but with *uniform* and *varied* multiplicities respectively: both multisets had $k = 5$ distinct integers, but the uniform had $m = \{3, 3, 3, 3, 3\}$ and the varied had $m = \{2, 3, 5, 2, 3\}$. Our mean times and standard deviation were produced over 10 iterations.

As can be seen from Table 1, MULTPERM runs 31–34% faster than KL04 across both multisets.

MULTPERM generated the 168 million permutations of the uniform multiset in an average of 21.5s ($\sigma = 0.11$) to KL04's 31.3s ($\sigma = 0.11$), and the 75 million permutations of the varied multiset in 9.4s ($\sigma = 0.05$) to KL04's 14.2s ($\sigma = 0.05$). We attribute the extra speed and simplicity of MULTPERM over KL04 to the advantages of our component algorithm for combinations over that used by Korsh and LaFollette.

6 Conclusion

There is room for further investigation and improvement in both of the problems we applied our framework to. Algorithm MIXPAR could be modified to allow a variable number of parenthesis types, and nesting a second Gray coder could allow it to cycle through another property of parentheses, e.g. colour. Regarding MULTPERM, there may yet be more advantageous loopless combinations algorithms than Chase's.

More interesting would be investigating which other combinatorial generation problems can be solved looplessly by fusion. Both of the problems we addressed quite obviously comprise two combinatorial subproblems, and therefore were conducive to this approach. We wonder:

- Can fusion be used for more complicated combinatorial generation problems?
- Can fusion be used where the decomposition into subproblems is not so obvious?

Acknowledgements

The authors would like to thank the referees for their constructive criticism.

References

- Chase, P. J. (1989), 'Combination generation and graylex ordering.', *Congressus Numerantium* **69**, 215–242.
- Ehrlich, G. (1973), 'Loopless algorithms for generating permutations, combinations, and other combinatorial configurations.', *J. ACM* **20**(3), 500–513.
- Johnson, S. M. (1963), 'The generation of permutations by adjacent transpositions.', *Math. Comp.* **17**, 282–285.
- Korsh, J. F. & LaFollette, P. S. (2004), 'Loopless array generation of multiset permutations.', *The Computer Journal* **47**(5), 612–621.
- Korsh, J. & Lipschutz, S. (1997), 'Generating multiset permutations in constant time.', *J. Alg.* **25**(2), 321–335.
- Nijenhuis, A. & Wilf, H. S. (1975), *Combinatorial Algorithms*, Academic Press.
- Reingold, E. M., Nievergelt, J. & Deo, N. (1977), *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall.
- Savage, C. (1997), 'A survey of combinatorial gray codes', *SIAM Review* **39**(4), 605–629.
- Trotter, H. F. (1962), 'Perm (algorithm 115)', *Commun. ACM* **5**(8), 434–435.

Vajnovszki, V. (2003), 'A loopless algorithm for generating the permutations of a multiset.', *Theor. Comput. Sci.* **307**(2), 415–431.

Wilf, H. S. (1989), *Combinatorial Algorithms: An Update*, SIAM.

Williamson, S. G. (1985), *Combinatorics for Computer Science*, Computer Science Press.

Xiang, L. & Ushijima, K. (2001), 'On $O(1)$ time algorithms for combinatorial generation.', *Comput. J.* **44**(4), 292–302.

A mixpar.cpp

```
/* Same style as Appendix B for consistency. */
#include <iostream>

using namespace std;

int n, j, *d, *e, jj, *dd, *ee, *l, *r, *s, t, i, num;
char *par, c;

void init() {
    cin>>n;
    par = new char[2*n+1]; d = new int[n+1]; e = new int[n+2];
    dd = new int[n+1]; ee = new int[n+2]; l = new int[n+1];
    r = new int[n+1]; s = new int[2*n+1];
    for (i=1; i<=n; i++) { par[2*i-1] = '('; par[2*i] = ')'; }
    for (i=1; i<=n; i++) { d[i] = 1; dd[i] = 1; }
    for (i=1; i<=n+1; i++) { e[i] = i-1; ee[i] = i-1; }
    for (i=1; i<=n; i++) { l[i] = 2*i-1; }
    for (i=1; i<=2*n; i++) { s[i] = i; }
    j = n; jj = n; t = n; num = 1;
}

void output() {
    cout<<num<<" ";
    for (i=1; i<=2*n; i++) { cout<<par[i]<<" "; }
    cout<<endl;
}

void next() {
    if (jj > 0) {
        ee[n+1] = n;
        if (dd[1] > 0 && jj == t) {
            r[jj] = s[l[jj]+1];
            if (jj > 1) { s[l[jj]] = s[r[jj]+1]; t = t-1; }
        }
        if (par[l[jj]] == '(')
            { par[l[jj]] = '['; par[r[jj]] = ']'; }
        else { par[l[jj]] = '('; par[r[jj]] = ')'; }
        ee[jj+1] = ee[jj]; ee[jj] = jj-1; dd[jj] = -dd[jj];
        if (dd[1] < 0 && jj == t) { s[l[jj]] = l[jj]; t = t+1; }
        jj = ee[n+1];
    } else {
        par[l[1]] = '('; par[r[1]] = ')';
        jj = n; t = n;
        dd[1] = 1; ee[n] = n-1;
        e[n+1] = n; i = l[j];
        if (d[j] > 0) {
            if (l[j] == 2*j-1) { l[j] = l[j-1]+1; }
            else { l[j] = l[j]+1; }
        } else {
            if (l[j] == l[j-1]+1) { l[j] = 2*j-1; }
            else { l[j] = l[j]-1; }
        }
        c = par[i]; par[i] = par[l[j]]; par[l[j]] = c;
        if (l[j] > 2*j-3)
            { e[j+1] = e[j]; e[j] = j-1; d[j] = -d[j]; }
        j = e[n+1];
    }
    num++;
}

void clean() {
    delete[] par; delete[] d; delete[] dd; delete[] e;
    delete[] ee; delete[] l; delete[] r; delete[] s;
}

int main() {
    init();
    output();
    while (j != 1 || jj != 0) {
        next();
        output();
    }
    clean();
}
```

B multperm.cpp

```
/* Same style as Korsh and LaFollette 2004 for comparison. */
#include <iostream>
using namespace std;

int k, n, j, x, y, i, u, v, w, num, *perm, **comb, *m, *d,
    *e, *o, *r, *z, *a, *b;

void init() {
    cin>>k; n = 0; m = new int[k+1];
    for (i=1; i<=k; i++) { cin>>m[i]; n += m[i]; }
    perm = new int[n+1];
    comb = new int*[k+1];
    for (i=1; i<=k; i++) { comb[i] = new int[m[i]+2]; }
    d = new int[k+1]; e = new int[k+1]; o = new int[k+1];
    r = new int[k+1]; z = new int[k+1]; a = new int[k+1];
    b = new int[k+1];
    o[1] = 0; for (i=2; i<=k; i++) { o[i] = o[i-1]+m[i-1]; }
    r[k] = m[k]; for (i=k-1; i>=1; i--) { r[i] = r[i+1]+m[i]; }
    for (i=1; i<=k; i++)
        { for (j=1; j<=m[i]; j++) { perm[j+o[i]] = i; } }
    for (i=1; i<=k; i++) { d[i] = 1; }
    for (i=0; i<=k+1; i++) { e[i] = i; }
    for (i=1; i<=k; i++) {
        for (j=1; j<=m[i]; j++) { comb[i][j] = j; }
        comb[i][m[i]+1] = 2*r[i]+1;
        z[i] = m[i]+1;
    }
    for (i=1; i<=k-1; i++)
        { a[i] = i+1; b[i] = 1+(m[i]>1 && r[i]%2); }
    j = 1; num = 1;
}

int adj(int i) {
    return comb[j][i]+1 == comb[j][i+1];
}

int inc(int i) {
    return comb[j][i+1]%2 == d[j]>0;
}

void output() {
    cout<<num<<" ";
    for (i=1; i<=n; i++) { cout<<perm[i]<<" "; } cout<<endl;
}

void next() {
    e[1] = 1;
    if (z[j] == 1) {
        v = 1;
        if (inc(1)) {
            if (adj(1)) { u = 2; w = 2*inc(2)-1; }
            else { u = 1; w = 1; }
        } else { u = 1; w = -1; };
    } else {
        if (inc(z[j]-1))
            { u = (z[j]>2 && inc(z[j]-2))+1; v = z[j]-u; w = 1; }
        else { v = z[j]; u = 1+adj(v); w = 2*inc(v-1+u)-1; }
    }
    i = v+(w-1)*(u-1)/-2;
    x = comb[j][i]; y = x+u*w;
    comb[j][i] = x+w; comb[j][i+(u-1)*w] = y;
    z[j] = z[j]-(comb[j][v] == v)*u*w-(v<z[j])*u;
    perm[x+o[j]] = perm[y+o[j]]; perm[y+o[j]] = j;
    if (a[j]<k) { o[a[j]] = o[a[j]]-b[j]*d[j]; a[j] = a[j]+1; };
    if (comb[j][m[j]-b[j]+1] == r[j]-b[j]+1
        && comb[j][m[j]-b[j]] == m[j]-b[j] || comb[j][m[j]] == m[j])
        { d[j] = -d[j]; e[j] = e[j+1]; e[j+1] = j+1; a[j] = j+1; }
    j = e[1]; num++;
}

void clean() {
    for (i=1; i<=k; i++) { delete[] comb[i]; }
    delete[] perm; delete[] comb; delete[] a; delete[] b;
    delete[] d; delete[] e; delete[] m; delete[] o; delete[] r;
    delete[] z;
}

int main() {
    init();
    output();
    while (j != k) {
        next();
        output();
    }
    clean();
}
```


The Busy Beaver, the Placid Platypus and other Crazy Creatures

James Harland

School of Computer Science and Information Technology
RMIT University
GPO Box 2476V
Melbourne, 3001
Australia
jah@cs.rmit.edu.au

Abstract

The busy beaver is an example of a function which is not computable. It is based on a particular class of Turing machines, and is defined as the largest number of 1's that can be printed by a terminating machine with n states. Whilst there have been various quests to determine the precise value of this function (which is known precisely only for $n \leq 4$), our aim is not to determine this value per se, but to investigate the properties of this class of machines. On the one hand, these are remarkably simple (and, intuitively, form perhaps the simplest class of computationally complete machines); on the other hand, as some of the machines for $n = 6$ show, they are capable of representing phenomenally large numbers. We describe our quest to better understand these machines, including the *placid platypus problem*, ie. to determine the minimum number of states needed by a machine of this type to print a given number of 1's.

1 Introduction

The busy beaver function (Rado 1963) was introduced by Rado in the 1960's as an example of a non-computable function, and is defined in terms of a particular class of Turing machines (Sudkamp 2005). This class is one in which there is a single tape which is infinite in both directions, the machine is deterministic, the tape is initially blank and the tape alphabet consists of only two symbols (traditionally blank (B) and one (1)). For each machine there is a distinguished halt state, from which there are no transitions. An n -state machine of this form is one that contains one halt state and n further non-halt states. The *busy beaver* function is the maximum number of 1's that is printed by a terminating n -state Turing machine. This function is often denoted as $\Sigma(n)$; in this paper we will use the more intuitive notation of $bb(n)$. The number of 1's printed by the machine is known as its *productivity*.

This function can be shown to be non-computable by proving that it grows faster than any computable function. Hence this function holds a special interest as one with an intuitively simple definition but which is capable of some phenomenally fast growth.

Because of this rate of growth, it has been historically quite difficult to evaluate precise values for this function. Its value for $n = 1, 2, 3$ were determined by Lin and Rado in the 1960's (Lin & Rado 1964), and the case for $n = 4$ by Brady in the 1970's (Brady 1983).

Copyright ©2006, Australian Computer Society, Inc. This paper appeared at Twelfth Computing: Australasian Theory Symposium (CATS2006), Hobart. Conferences in Research and Practice in Information Technology, Vol. 51. Barry Jay and Joachim Gudmundsson, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

However, establishing $bb(5)$ and $bb(6)$ has been more problematic, particularly due to some spectacularly large lower bounds for these values (Marxen and Buntrock 1990, Marxen 2005). The only claim, to the author's knowledge, for a precise evaluation for $n = 5$ is Master's thesis published in August, 2005 (Kellett 2005), which uses a mixture of machine evaluation and human analysis (although there are some others performing similar searches, such as Georgi Georgiev (Skelet 2005)). There are some interesting analyses of the current 5-state and 6-state champions (Munafo 2005, Michel 2005), but some speculate that due to the sheer size of the numbers involved, $bb(7)$ may never be known, despite some promising techniques for evaluating machines with extremely large productivities (Holkner 2004).

A recent variation on this problem is to consider machines with more than two tape symbols, and there have been various spectacular examples of large computations with such machines (such as a 3-state machine with 3 tape symbols that prints 544,884,219 symbols before halting (Brady 2005)). The relationship between the busy beaver function and the $3n+1$ sequence has been investigated by Pascal Michel (Michel 2005), which has resulted in an elegant analysis of the most complex machines currently known. Some analysis has also been done on Turing machines in which one can either move the tape head or write a new symbol on the tape, but not both in one atomic action (Ross 2003). There has also been an investigation of busy beaver functions on a one-way tape (Walsh 1982) rather than a two-way one.

The main use of this function has been as a simple example of non-computability, especially as there are also larger functions which can be defined similarly. One such function, often denoted $\mathcal{S}(n)$ in the literature, the maximum number of state transitions made by a terminating Turing machine of the above form. We denote this function as $ff(n)$.¹ A notable use of this function is given in Boolos and Jeffrey (Boolos & Jeffrey 1980), in which the busy beaver function is the subject of the first undecidability result established, rather than the more usual choice of the halting problem for Turing machines.

The relationship between $bb(n)$ and $ff(n)$ has been investigated (Julstrom 1992), and it is known that $ff(n) < bb(3n + c)$ for a constant c (Yang, Ding & Xu 1997). However, this is still rather loose, and does not give us much insight into the relationship between $bb(n)$ and $ff(n)$. In a similar manner, lower bounds on $bb(n)$ have been known for some time (Green 1964); however, those given for $n \leq 6$ have been far surpassed already.

In this paper, we introduce some new perspectives on the busy beaver function. In essence, our main interest is not so much the ability to define non-computable functions, but the properties of the un-

¹We call this function the *frantic frog*.

derlying class of Turing machines.

To motivate this point of view, consider the table below (drawn from (Lin & Rado 1964), (Brady 1983), (Marxen and Buntrock 1990), (Marxen 2005)).

n	$bb(n)$	$ff(n)$
1	1	1
2	4	6
3	6	21
4	13	107
5	≥ 4098	$\geq 47,176,870$
6	$\geq 1.29 * 10^{865}$	$\geq 3 * 10^{1730}$

The vast increase from $n = 5$ to $n = 6$ seems to suggest that 6 states is the minimum for some particular functionality, such as multiplication or exponentiation, or a universal Turing machine. Moreover, it is counterintuitive (to say the least!) to be able to print out a sequence with as many as 10^{865} elements in it via a Turing machine with only 6 states.

This makes it clear that the range of productivities for 6-state machines is astronomical, and leads us to a problem which is in some sense dual to the busy beaver problem. The busy beaver problem may be considered as finding the maximum number of 1's that can be printed by a Turing machine with a given number of states. The dual problem is to find the minimum number of states needed in order to print a given number of 1's. Hence an n -state machine of productivity m may be considered as evidence that $bb(n) \geq m$ as well as evidence that it requires at most n states to print m 1's. A natural question is whether there is a 5-state machine of productivity m for every $bb(4) < m \leq bb(5)$. Because of the duality with the busy beaver, we denote this as the *placid platypus problem*.²

In order to answer questions such as these, it seems appropriate to prepare the groundwork by searching amongst the 5-state machines, but in such a way that more than just the champion machine (i.e. the one of highest productivity) is retained. It should be stressed that this investigation is very much a means to determine better analysis techniques, which will hopefully make such searching partially (or perhaps totally) redundant. We thus commence with an empirical investigation, which will hopefully lead us into a more analytic one. For example, whilst it is well known that the lower bound for $bb(5)$ is 4,098, there are actually 6 machines with productivities equal to or very close to this value, including two machines which produce 4,098 ones. However, as one of them takes 47,176,870 transitions and the other 11,798,826, one can argue that the second is actually a "better" machine (at least in terms of the effort required to generate the 1's that are output). In addition, the first machine at one point has more than 12,000 1's on the tape, but just before terminating, it deletes two-thirds of these 1's. The first though, is usually considered the 5-state champion, as the larger value provides a faster-growing function than then small one. In addition, it is known that there are two 5-state machines with productivity 4097, and a further two with productivity 4096. However, the next highest known productivity is 1471, which suggests that these six machines are rather unusual. Analysis of this phenomenon has concentrated on determining the productivity of a particular machine; however, it would seem that it is at least as interesting to determine why only this small number of machines has this behaviour.

²What could be the opposite to a busy beaver? A platypus is an Australian monotreme, and hence is native to the southern hemisphere (as the beaver is to the north). Platypuses are shy and retiring by nature, making this name a natural choice.

In this paper we explore some of these issues, with a particular focus on the placid platypus. Accordingly, the main contribution of this paper is to identify and discuss some new aspects of this well-known problem, rather than to report on a particular technical development. In Section 2 we discuss some basic concepts, and in Section 3 we report on the state of our (as yet incomplete) search of the 5-state machines. Section 4 presents some details which emerge from the search and Section 5 introduces the placid platypus in some detail. Section 6 discusses some further ideas of interest.

2 Definitions

We use the following definition of a Turing machine.

Definition 1 A Turing machine is a quadruple $(Q \cup \{h\}, \Gamma, \delta, q_0)$ where

- h is a distinguished state called a halting state
- Γ is the tape alphabet
- δ is a partial function from $Q \times \Gamma$ to $Q \times \Gamma \times \{l, r\}$ called the transition function
- $q_0 \in Q$ is a distinguished state called the start state

Note that there are no transitions for state h , and that as δ is a partial function, there is at most one transition for a given pair of a state and a character in the tape alphabet.

Note that this is the so-called quintuple transition variation of Turing machines, in that a transition must specify for a given input state and input character, a new state, an output character and a direction for the tape in which to move. Hence a transition can be specified by a quintuple of the form

(State, Input, Output, Direction, NewState)

Some varieties of Turing machines allow only one of the latter two possibilities, i.e. either to write a new character on the tape or to move, and not both; for such machines, clearly only a tuple of 4 elements is required. Note that our machines are deterministic (due to δ being a function rather than a relation).

Our class of machines will be a particular type of this variety of Turing machines.

Definition 2 A unary Turing machine is a Turing machine containing a single two-way infinite tape, and whose tape alphabet includes only blanks and 1.

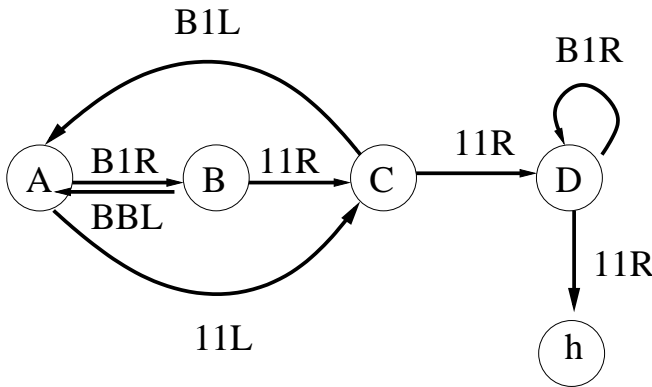
A beaver machine is a unary Turing machine whose tape is entirely blank on input.

An animal machine is a unary Turing machine whose tape contains a finite number of 1's on input.

We denote by an n -state Turing machine one in which $|Q| = n$. In other words, an n -state Turing machine has n "real" states and a halt state.

Clearly a crucial issue in the evaluation of the busy beaver function is to determine whether or not a given machine terminates. Whilst in general this is undecidable, it is still an open problem, to the author's knowledge, whether the $n = 5$ case is decidable. In practice, this will not be a great problem if the number of machines whose status cannot be determined (often referred to as *holdouts*) is small enough for hand analysis (say under 100).

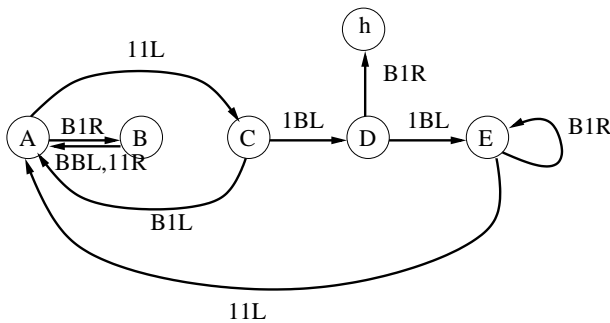
There are various ways in which a beaver machine may fail to terminate. One such way is to move endlessly in one direction, as the machine below does.



This machine prints three 1's before endlessly moving towards the right.

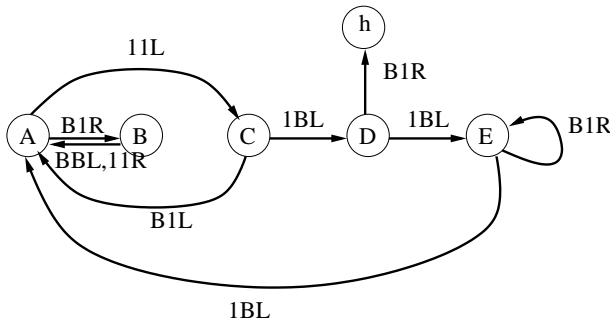
We call such machines *escapees*.

Another way is to return to exactly the same configuration as an earlier one. For example, consider the machine below.



This continuously returns to a configuration in state E with a single 1 on the tape. We call such machines *cyclers*.

A more subtle way is to reproduce the same context further along the tape. For example consider the machine below.



This machine does not repeat the overall state of the tape, but only because it is repeatedly shifting the repeated pattern along to the left. We call such machines *dizzy ducks*. Note also that this machine only differs from the previous one by changing the output of the transition for state E with a one from a one to a blank.

Each of these three cases is relatively easy to detect, and relatively common in the search conducted thus far. A further useful observation is that if the tape becomes all blank during computation, then we may immediately discard the machine from consideration. This is not because the machine will necessary loop (although it may, of course), but because there will be some other machine that produces the same eventual outcome without the tape going blank. To see this, let us assume that the tape becomes blank in state 4, and the computation continues from there. Then there is another machine which has the same

computation behaviour as the original machine does from state 4, but commencing from the second machine's initial state. Hence we can immediately disregard any machine which makes the tape return to being all blank. We call such machines *blankers*.

3 Searching for busy beavers

Searching for machines of a particular type (such as the busy beaver) is a matter of enumerating all possible transitions, and then performing various tests. As the number of n -state machines is exponential, it is impractical to do so for every machine. For an n -state machine, there are $2n$ transitions, each with 2 possible outputs, 2 possible directions and $n+1$ possible new states, leading to a total of $(4(n+1))^{2n}$ machines. However, as the halt state will only be used once in each machine, this naive maximum can be reduced to $2n \times (4n)^{2n-1}$. In addition, by fixing the first transition (Lin & Rado 1964), we can reduce this further to $(2n-1) \times (4n)^{2n-2}$.

This number remains formidable. For $n=5$, this still leaves around 2.3×10^{11} machines to evaluate. In practice, though, this figure can be reduced significantly.

The standard way in which to search for busy beaver machines is to use the *tree normal form* method of Lin & Rado (Lin & Rado 1964). In essence, this involves emulating the machine as transitions are generated. By running an incomplete machine until an unspecified transition is required, we can then ensure that the new transition obeys certain constraints, such as not generating a blank tape, and not creating a cyclor or an escapee. In addition, this method ensures that the halt transition is added to the machine only when all other transitions have been allocated. In other words, we ensure that the halt transition will be the last one used.

We execute the machine until the halt transition has been allocated, or until some specified maximum number of transitions has been reached. As we are dealing with 5-state machines, we used a maximum of 110, i.e. three more than $ff(4)$, thus ensuring a "genuine" 5-state machine.

At this point, we could store the machine and proceed to analysis. This reduces the number of machines that must be stored to a little under 10^8 . This is still non-trivial, but much more in line with what can be achieved by commodity hardware (such as a typical desktop PC). However, our analysis of such machines showed the presence of a surprisingly large number of cyclers and blankers, whose behaviour would have been noticed earlier if we had run all generated machines for 110 transitions, rather than halting once the final transition was decided. Hence we executed all machines thus generated for 110 transitions, removing those which were found to be cyclers or blankers in this time. This drastically reduced the search space even further (at a significant increase in the time required to perform the search), leaving "only" 15,595,622 machines to be stored and further analysed.

Note also that when searching for busy beavers, it is sensible to allocate the output of the halt transition to be always 1. This clearly cannot decrease the productivity, and may in fact increase it. This also helps, in a minor way, to reduce the search space. We will denote such machines as *halting up* machines; we will return to this point later.

We have a prototype implementation of this approach, but both the development of this program and the search itself are still ongoing. The current implementation consists of about 1,800 lines of Ciao

Prolog (Ciao Manual 2005) (around half of which is redundant). A feature of this implementation, when compared to some others, is that we are interested in keeping the evaluation results for all generated machines, and not just those which halt. Whilst this may seem overly pedantic, and wasteful of storage space, it seems to be a useful resource for the desired further analysis of this class of machines. The storage requirements themselves are not outrageous by modern standards; even using a naive method of storage takes no more than 3 Gigabytes without any compression, which is well within the storage capacity of an average PC. An intended side-effect of our investigation is a publicly available database of these machines; clearly it is important for this database to be as irredundant as possible, and hence we hope to eventually provide a more analytically compact representation of this information than a mere collection of all instances of the generated machines.³

For 4 states, there are in principle 117,440,512 machines to be evaluated. The above techniques reduced this to “only” 444,481, of which 208,011 did not terminate, and a further 115,750 made the tape blank during computation. This leaves 120,720 which terminated (or around 0.1% of the original number). Of these, only 1,939 had productivity of 7 or more (i.e. an improvement on the 3-state busy beaver), and only 22,283 had productivity of 5 or more. A more detailed analysis is given below.

≤ 4	5	6	7	8	9	10	11+
98,437	15,089	5,255	1,487	357	74	11	10

Hence the maximal machines are very rare in the general population, even when non-terminating machines are removed. This strongly suggests that an analytical criterion is possible.

Our statistics are less complete for the $n = 5$ case. However a preliminary analysis of the search results to date is given below.

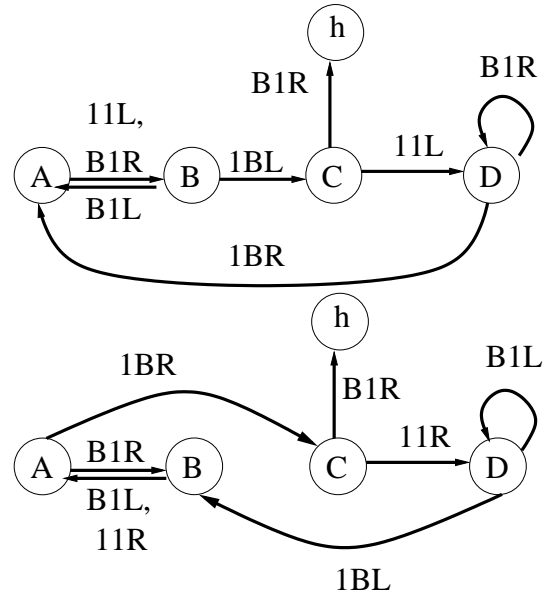
Type	Percentage
halts	69.2
cyclers + escapees	0.001
repeaters	30.1
blankers	0.0005
unclassified	0.6

Hence classifying 99.4% of the cases leaves “only” 106,379 machines still to be classified. Such machines will require more sophisticated analyses, such as inductive methods of showing that a machine does not terminate (as suggested in (Brady 1983)). In addition, there are various termination analysis techniques that are used in analysing constraint logic programs and other declarative programming paradigms which may be directly applicable here. Incorporating such techniques into the implementation is work in progress.

4 “Maximal” machines

Consider the two machines below.

³This collection will be made available at <http://www.cs.rmit.edu.au/~jah/busybeaver>.



The first one is the 4-state busy beaver, with productivity 13 and taking 107 steps to terminate, thus showing that $bb(4) \geq 13$ and $ff(4) \geq 107$. However, the second hand machine also has productivity 13, but only takes 96 steps to terminate. Hence it is tempting to consider the right hand machine as a more optimal one than the left hand one, in that it has the same productivity for fewer steps of execution.

This situation is reflected in the 5-state case. Consider the six machines in Figure 4.

The machine m_1 is the 5-state busy beaver candidate. However, as in the case for $n = 4$, there is a machine (m_2) of the same productivity which takes fewer steps (denoted hops in the table above), and it is arguably a better machine for this reason. This pairwise behaviour recurs for productivities of 4,097 and 4,096 as well. In addition, m_4 and m_6 are remarkably similar; they differ only on two transitions (state B with input 1 and state D with input 1). A more in-depth understanding of the relationships between these machines is clearly crucial to an understanding of the busy beaver function.

5 The Placid Playtpus

It is trivial to show that $bb(n) \geq n$, as it is trivial to find an n -state machine that prints n 1's and then halts. Boolos and Jeffrey show how it is possible to take a given beaver machine, and construct another which produces doubles the length of the string of 1's output by the original machine. In our framework, this takes an extra states, thus showing that $bb(n + 7) \geq 2n$. Hence given an n -state machine which prints m 1's, we can repeatedly apply these additional states to get a $n + 7k$ -state machine which prints $m \times 2^k$ 1's.

A result claimed in (Dewdney 1993), but left as an exercise, is that $bb(n) \geq 2^n$. In other words, to print m 1's, a machine that has no more than $\log_2 m$ states is required.

This leads to the following question: given a string of n 1's, what is the minimum size of a beaver machine which prints it? We will refer to this function as the *placid platypus* function, and denote it by $pp(n)$. This is in some sense a dual to the busy beaver function, in that if $bb(n) \geq m$, we have that $pp(m) \leq n$. In other words, an n -state beaver machine which prints m 1's shows both that $bb(n) \geq m$ and $pp(m) \leq n$.

Dually to the frantic frog function, we also define the *weary wombat* function, which is the minimum number of state transitions for an n -state machine necessary to print $pp(n)$ 1's. We denote this function

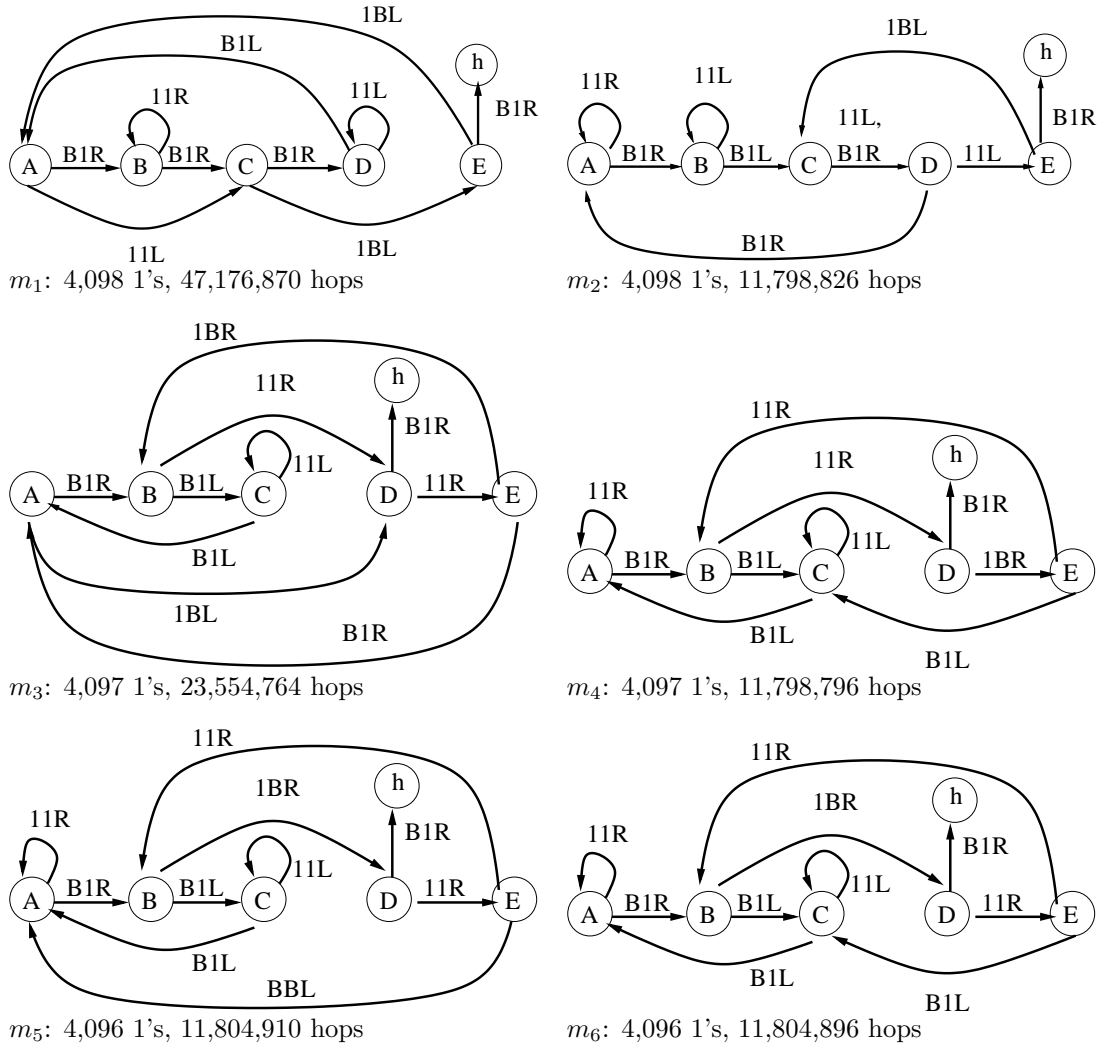


Figure 1: Monster 5-state machines

by $w(n)$.

It is not hard to find some early values of $pp(n)$ and $ww(n)$, as shown in Figure 5.

Above the productivity of 20, our search is much less complete (possibly due to our implementation capping the maximum number of hops initially at 110). As can be seen in Figure 3 below, it is as yet unknown whether there is a 5-state machine of productivity 29, 30, 31, 34 or 37.

In fact, it is possible to obtain a machine of productivity 31 from the one of productivity 32, by changing the halt transition. In other words, as the machine for productivity 32 is a halting up machine, this shows that there is a halting down machine of productivity 31.

This suggests that an interesting question is whether for every $bb(4) < m \leq bb(5)$ there is a machine of corresponding productivity. Certainly this is true for the range $bb(3) < m \leq bb(4)$, and must be false for the range $bb(5) < m \leq bb(6)$. The former case we have shown by construction, and in the latter one, there are simply too few machines to cover a range of more than 10^{800} numbers. Another way to put the question is to determine the minimum value m such that there is no 5-state machine of productivity m . Certainly $bb(5) + 1$ is an upper bound on this value; the question is whether it is a lower bound as well. A related question is that of the largest continuous range of representable numbers: in other words, what is the largest value of m such that for every $bb(4) < k < m$ there is a 5-state machine of productivity k ? In general, the distribution of platypus machines for 5-state and 6-state machines remains an intriguing question.

A further aspect of this question is the ability to produce an analytical answer. Certainly we can settle the question of the distribution of 5-state machines by enumerating all of them and examining the results. However, we see this as a necessary first step towards a systematic method by which one can construct an m -state machine of productivity n , where m satisfies a constraint such as $m \leq \log_2 n$. For example, if there is no 5-state machine with productivity 29, can we construct a 6-state or 7-state machine by some algorithmic process?

An extension of this line of thought is to consider equivalences across varying numbers of states. For example, consider the four machines in Figure 6, all of which have productivity 6.

In a sense, these machines are all equivalent. In addition, the similarity between the 4-state and 5-state machines suggests that it may be possible to reduce the 5-state machine to a 4-state one (and possibly further still to the 3-state one) by a process of state elimination. Such techniques are known for finite-state automata, and are not generally applicable to Turing machines. However, for this restricted class and in this particular context, it seems reasonable to investigate an approach along these lines.

6 Discussion

We have seen how examining the machines which lead to the busy beaver function have led to some further interesting questions. A similar one that can be asked is what are the maximum productivities for some sub-classes of machines. In particular, what is the maximum productivity of a 5-state machine in which there is no transition with input 1 and output blank? (We call these machines *monotonic*, in that no 1 is ever erased from the tape). What is the maximum productivity for *eager* machines? (i.e. those with transitions whose output is always 1)? What is the maximum productivity for *contiguous* machines?

(i.e. those which always maintain the 1's in a single string)? This latter problem is called the little busy beaver problem in (Yang, Ding & Xu 1997), but no values are given for it. All of these are seemingly natural classes for humans attempting to construct busy beaver machines by hand, but none of these properties holds for the six 5-state machines with the largest known productivities.

For that matter, more sophisticated measurements of the tape usage may be used, such as the largest string of contiguous 1's, the amount of tape used, or the distance moved from the point of origin. Holkner (Holkner 2004) has observed that several of the 6-state monster machines use no more than 10 distinct 'blocks', i.e. repetitions of short strings. Hence whilst the overall length is very large, it has a very regular structure. Finding an appropriate metric for this structure is an intriguing question.

Whilst there have been some investigations of the busy beaver function for one-way tapes, other constraints on the use of the tape may be interesting. One possibility is to constrain the amount of tape that may be used, such as by using a circular tape, or by bounding the number of tape cells that can be written to, analogous to the difference between linear-bounded automata and Turing machines (Sudkamp 2005).

As noted above, there are some "maximising" strategies built into the search methods for busy beaver machines, such as searching only for halting up machines. A stronger restriction is the one that ensures that the halt transition can only be used once all other transitions have been used. Whilst this is appropriate for the busy beaver function, in that it seems reasonable to require that all possible transitions be used before terminating, it is possible that the lack of necessity to use every transition may lead to some simpler placid platypus machines. The cost is, of course, a huge increase in the search space, making an analytical criterion for the detection of platypus machines more critical.

A final point to note is that the machines which do not terminate may also be of interest. In Conway's game of life, for example, some of the most interesting configurations are those which lead to self-reproducing behaviour, which, in our context, correspond to particular kinds of non-terminating machines. Hence dizzy ducks, which reproduce the same context but further along the tape, may be of interest in this sense. Potentially interesting also are *phoenix* machines, i.e. those which continually print some number of 1's, and then return the tape to all blanks in the initial state. Such a machine will eternally convert the blank tape into a fixed number of 1's and back again.

7 Conclusions and Further Work

We have seen how a quest for determining the value of $bb(5)$ can lead to a number of new questions about this class of Turing machines. In this sense, the main contribution of this paper is not some particular piece of technical progress, but to consider various new questions, which, we hope, are of more general interest. It is difficult to conceive of a practical application of busy beaver machines⁴, but the fundamental simplicity of the machines together with the astounding sizes of some of the numbers involved suggest that there is something of significant interest under the surface. Once we have achieved a better analytical understanding of these 2-symbol machines, we may be able to better understand the even more astounding

⁴Except, perhaps as a particularly bizarre form of a screen saver!

n	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
$pp(n)$	2	3	3	4	4	4	4	4	4	4	5	5	5	5	5	5	5
$ww(n)$	4	7	11	12	14	19	30	40	53	96	≤ 41	≤ 45	≤ 50	≤ 57	≤ 63	≤ 43	≤ 62

Figure 2: Lower Placid Platypus values

n	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	
$pp(n)$	5	5	5	5	5	5	5	5			5	5	5		5	5	(* these
$ww(n)^*$	72	98	69	223	155	298	343	102			512	427	559		691	808	are upper bounds only)

Figure 3: Higher Placid Platypus values

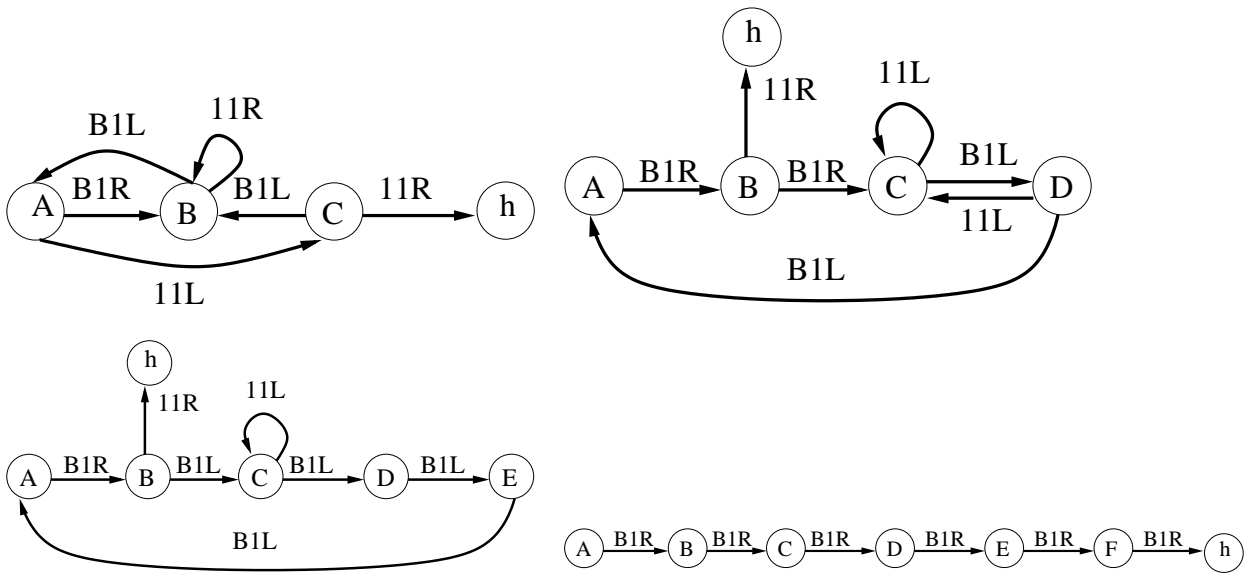


Figure 4: Machines of productivity 6

ing numbers being generated by the search for 3-, 4-, 5- and 6-symbol busy beavers.

Acknowledgements

The author is grateful to Jeanette Holkner, Michael Winikoff and Sandra Uitdenbogerd for assistance with zoological names and descriptions.

References

- George Boolos and Richard Jeffrey, *Computability and Logic*, 2nd edition, Cambridge University Press, 1980.
- Allen Brady, Busy Beaver Problem of Tibor Rado, <http://www.cse.unr.edu/~al/BusyBeaver.html>.
- Allen Brady, *The Determination of the value of Rado's noncomputable function $\Sigma(k)$ for four-state Turing machines*, Mathematics of Computation 40(162): 647-665, 1983.
- The Ciao Prolog Development System WWW Site, <http://clip.dia.fi.upm.es/Software/Ciao>.
- A. Dewdney, *The (New) Turing Omnibus*, Computer Science Press, 1993.
- Milton Green, A lower bound on Rado's sigma function for binary Turing machines, Proceedings of the Fifth Annual IEEE Symposium on Switching Circuit Theory and Logical Design 91-94, Princeton, November, 1964.
- Heiner Marxen and Jürgen Buntrock, *Attacking the Busy Beaver 5*, Bulletin of the EATCS 40:247-251, February 1990.
- Alex Holkner, *Acceleration Techniques for Busy Beaver Candidates*, in Gad Abraham and Benjamin I.P. Rubenstein (eds.), *Proceedings of the Second Australian Undergraduate Students' Computing Conference* 75-80, December, 2004. ISBN 0-975-71730-8. Available from <http://www.cs.berkeley.edu/~benr/publications/auscc04>.
- B. Julstrom, *A Bound on the Shift Function in terms of the Busy Beaver Function*, ACM SIGACT News 23(3):100-106, 1992.
- Owen Kellett, *A Multi-Faceted Attack on the Busy Beaver Problem*, Master's Thesis, Rensselaer Polytechnic Institute, August, 2005.
- Wang Kewen and Shurun Xu, *New Relation Between The Shift Function and the Busy Beaver Function*, Chinese Journal of Advanced Software Research, 2(2):192-197, 1995.
- Shen Lin and Tibor Rado, *Computer Studies of Turing Machine Problems*, Journal of the Association for Computing Machinery 12(2):196-212, 1964.
- Heiner Marxen, Busy Beaver web page, <http://www.dr.b.insel.de/~heiner/BB/index.html>.
- Pascal Michel, Behavior of Busy Beavers, <http://www.logique.jussieu.fr/~michel/beh.html>.
- R. Munafo, Large Numbers – Notes, <http://home.earthlink.net/~mrob/pub/math/ln-notes1.html>.
- Tibor Rado, *On non-computable functions*, Bell System Technical Journal 41: 877-884, 1963.
- Kyle Ross, *Use of Optimisation Techniques in Determining Values for the Quadruplorum Variants of Rado's Busy Beaver Function*, Masters thesis, Rensselaer Polytechnic Institute, 2003.
- Georgi Georgiev, Busy Beaver Prover, <http://skelet.ludost.net/bb/index.html>.
- Thomas Sudkamp, *Languages and Machines: An Introduction to the Theory of Computer Science*, (3rd ed.), Addison Wesley, 2005.
- T. Walsh, *The Busy Beaver on a One-Way Infinite Tape*, ACM SIGACT News 14(1):38-43, 1982.
- Ruiguang Yang, Longyun Ding and Shurun Xu, *Some Better Results Estimating the Shift Function in Terms of Busy Beaver Function*, ACM SIGACT News 28(1): 43-48, March, 1997.

A Polynomial Algorithm for Codes Based on Directed Graphs

A.V. Kelarev

School of Computing
University of Tasmania, Private Bag 100
Hobart, Tasmania 7001, Australia
Email: Andrei.Kelarev@utas.edu.au
www.comp.utas.edu.au/users/kelarev

Abstract

A complete description and proof of correctness are given for a new polynomial time algorithm for a class of codes based on directed graphs and involving construction well known in system theory. Our construction has already been considered in the literature in relation to other questions. The investigation of codes in this graph-based construction is inspired by analogy with classical cyclic codes that are defined in a similar way in polynomial rings. We show that all cyclic codes can be embedded in this construction. For each graph, the algorithm computes the largest number of errors which can be corrected by codes defined with this graph. In addition, it finds a generator of a code with this optimum value.

Keywords: algorithms, coding, directed graphs.

1 Introduction

It has been established by Downey, Fellows, Whittle, and Vardy that several fundamental problems concerning linear codes are NP-complete and W[1]-hard, see (Downey & Fellows 1999, Downey & Fellows 1999b, Downey, Fellows, Whittle & Vardy 2001). The aim of this paper is to develop a polynomial time algorithm for a class of codes inspired by analogy with classical cyclic codes. We are applying directed graphs to define a class of error-correcting codes and develop a polynomial algorithm for computing the number of errors these codes can correct. Our first main theorem proves the correctness and evaluates the running time of the algorithm.

As a guide we are motivated by analogy with the fact that all cyclic codes, including various efficient codes used in practice, are specified in polynomial rings. We are going to use another important construction defined in terms of directed graphs and considered by many authors, see (Kelarev 2002) for references. It is shown that all cyclic codes can be embedded in it. This construction enables us to define a class of error-correcting codes specified in terms of directed graphs.

It is natural to investigate how properties of the code depend on the properties of the graph that defines it. We develop a polynomial algorithm for computing the number of errors which can be corrected by codes defined with any given graph. In addition,

our algorithm finds a generator of a code with this optimum value.

We use standard concepts concerning codes following (Lidl & Niederreiter 1994), (Lidl & Niederreiter 1997), (Lidl & Wiesenbauer 1980), (Moffat & Turpin 2002), (Pless, Huffman & Brualdi 1998) and include all the necessary information for convenience of the readers, see also (Asano, Wada & Masuzawa 2003), (Cormen, Leiserson, Rivest & Stein 2001), (Kelarev 2001), (Kelarev 2002), (Kelarev 2003), (Kelarev 2004), (Kelarev & Sokratova 2001), (Kelarev & Solé 2000), (Cazaran, Kelarev, Quinn & Vertigan 2006). As it is customary in the literature, by a polynomial algorithm we mean an algorithm with polynomial running time. It is worth mentioning that in fact our algorithm solves a special case of the more general Problem 10.1 recorded in (Kelarev 2002).

There are many known cyclic and linear codes used in practice. For example, the BCH codes are used in CDs, DVDs, mobile phones and digital television, see (Stallings 2002). The second main theorem of this paper demonstrates that our construction is general enough to incorporate all cyclic codes providing additional structure for them.

It is important to emphasise that the research on algorithmic aspects of coding theory brings practical benefits as a result of cumulative combined effect of many incremental steps in the investigation conducted by many researchers throughout the world. For example, it took decades from the invention of the BCH codes to their practical uses in CDs, DVDs, mobile phones and digital television.

The author is grateful to Mike Fellows for permission to use his idea of simplifying the main algorithm that has substantially reduced its complexity. It has lead to a much simpler polynomial algorithm and was suggested to the author during Mike's visit to the School of Computing at the University of Tasmania.

2 Main results

Throughout the word 'graph' will mean a directed graph without multiple edges but possibly with loops. Let $D = (V, E)$ be a graph with the set $V = \{v_1, \dots, v_n\}$ of vertices and a set $E \subseteq V \times V$ of edges. Following standard conventions of coding theory, let us denote by $F = F_q$ an encoding alphabet regarded as a finite field. The incidence ring of D is denoted by $I_D(F)$ and is defined as the set of all formal finite sums of edges in E with coefficients in F . It is endowed with multiplication defined by the distributive laws and the rule

$$(x, y) \cdot (z, w) = \begin{cases} (x, w) & \text{if } y = z, (x, w) \in E, \\ 0 & \text{otherwise,} \end{cases} \quad (1)$$

for $x, y, z, w \in V$, see, for example, (Kelarev 2002), §3.15. The graph D is said to be *balanced* if, for

Copyright ©2006, Australian Computer Society, Inc. This paper appeared at Computing: The Australasian Theory Symposium (CATS2006), Hobart, Australia. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 51. Barry Jay and Joachim Gudmundsson, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.

This research has been supported by ARC Discovery grant DP0449469.

all $x_1, x_2, x_3, x_4 \in V$ with $(x_1, x_2), (x_2, x_3), (x_3, x_4), (x_1, x_4) \in E, (x_1, x_3) \in E \Leftrightarrow (x_2, x_4) \in E$. It is well known and fairly easy to verify that the above definition correctly defines multiplication as an associative operation on $I_D(F)$ if and only if D is balanced.

Thus, every element x of $I_D(F)$ is uniquely represented as a sum of the form $x = \sum_{(u,v) \in E} x_{(u,v)}(u, v)$, where $x_{(u,v)} \in F$ and only a finite number of the coefficients $f_{(u,v)}$ are nonzero.

As a motivating example, let us recall that every cyclic code can be defined with a generator polynomial as follows. Recall that the polynomial quotient ring

$$Q_n = F[x]/(1 - x^n) \quad (2)$$

consists of all polynomials with one variable x and degree $\leq n$, where addition of polynomials is defined as usual, and multiplication is performed modulo $1 - x^n$. This means that the product of two polynomials in $F[x]/(1 - x^n)$ is equal to the remainder of their product in $F[x]$ upon division by $1 - x^n$. For any $g(x) \in F[x]/(1 - x^n)$, the cyclic code $(g(x))$ generated by $g(x)$ is the set of all multiples of $g(x)$, i.e., all elements of the form

$$(f_0 + f_1x + \dots + f_{n-1}x^{n-1})g(x),$$

where $f_0, f_1, \dots, f_{n-1} \in F$. Similarly, we say that the error-correcting code generated by the elements g_1, \dots, g_k in $I_D(F)$ is the set

$$\begin{aligned} C_D(g_1, \dots, g_k) &= \\ &= \{(f_1 + h_1)g_1 + \dots + (f_k + h_k)g_k \mid \text{where} \\ &\quad f_1, \dots, f_k \in F, h_1, \dots, h_k \in I_D(F)\}. \end{aligned} \quad (3)$$

Theorem 1 *For each balanced directed graph $D = (V, E)$, Algorithm 1 finds the number of errors that error-correcting codes $C_D(g_1, \dots, g_k)$ can correct and returns a generator g of the optimal code $C_D(g)$ which achieves this error-correcting capability. The running time of the algorithm is $O(n^3)$.*

A nice and unexpected conclusion is that it turns out possible to generate an optimal code with just one generator. Thus, a perfect analogy with cyclic codes occurs, despite the fact that not all codes can be generated by one element in $I_D(F)$.

It is worth noting that a brute force algorithm for solving the same task would be exponential, because it is fairly easy to show that the number of codes of the form $C_D(g_1, \dots, g_k)$, and even the number of pairwise incomparable with respect to inclusion codes of the form $C_D(g)$, intricately depends on the graph D and can grow exponentially with n . Together with NP-completeness and W[1] hardness results due to (Downey et al. 2001), see also (Downey & Fellows 1999, Downey & Fellows 1999b), it shows that devising a polynomial algorithm in this situation was not easy.

Theorem 2 *Every cyclic code C can be embedded in $I_D(F)$ for some D and F so that C is generated by one element.*

Theorem 3 *Every linear code can be embedded in $I_D(F)$ for some D and F .*

3 Main algorithm

A pseudocode with concise description of the main algorithm is given in Figure 1. This section supplies additional intuitive explanation of the steps of the algorithm. Algorithm 1 utilizes only properties of the

Algorithm 1 *Given a balanced directed graph $D = (V, E)$, returns the largest number of errors that codes in $I_D(F)$ can correct and an element generating an optimal code of this sort.*

```

1. int i, j, b = 0, c = n*n;
2. L = a set of triplets represented as
   a red-black tree, initially empty;
3. for ( i = 1; i <= n; i++ )
4.   find In( $v_i$ ) = { $x \in V \mid (x, v_i) \in E$ };
5. for ( j = 1; j <= n; j++ ) {
6.   for ( v ∈ In( $v_j$ ) ) {
7.     S = In(v) ∩ In( $v_j$ );
8.     if ( ∃T : (v, S, T) ∈ L )
9.       { t++; insert  $v_j$  in T; }
10.    else
11.      insert (v, S, { $v_j$ }) in L;
12.   }
13. }
14. Find ( $v', S', T'$ ) ∈ L with maximum |T'| = b;
15.  $g_b = \sum_{y \in T'} (v', y)$ ;  $g_c = \sum_{w \in E} w$ ;
16. for ( i = 1; i <= n; i++ ) {
17.   for ( j = 1; j <= n; j++ ) {
18.     for ( t = 1; t <= n; t++ ) {
19.       if ( (t,  $v_i$ ), (t,  $v_j$ ) ∈ E ) {
20.         c--;
21.          $g_c = g_c - (v_i, v_j)$ ;
22.         break;
23.       }
24.     }
25.   }
26. }
27. if ( b ≥ c )
28.   return ⌊(b - 1)/2⌋,  $g_b$ ;
29. else
30.   return ⌊(c - 1)/2⌋,  $g_c$ ;

```

Figure 1: Main Algorithm

graphs. The notation used in the algorithm is illustrated in Figure 2. It is well known that

$$Q_n = F[x]/(1 - x^n)$$

always contains F , but $I_D(F)$ does not have to contain F . Besides, every code can be generated with just one polynomial in Q_n , whereas in $I_D(F)$ several generators can define larger classes of codes compared to just one generator.

The algorithm calculates and returns the optimal value which can be defined as follows. Let $D = (V, E)$ be a transitive graph. Recall that the *in-degree* and *out-degree* of a vertex $v \in V$ are defined by

$$\text{indeg}(v) = |\{w \in V \mid (w, v) \in E\}|, \quad (4)$$

$$\text{outdeg}(v) = |\{w \in V \mid (v, w) \in E\}|. \quad (5)$$

A vertex of D is called a *source* (*sink*) if $\text{indeg}(v) = 0$ and $\text{outdeg}(v) > 0$ (respectively, $\text{indeg}(v) > 0$, $\text{outdeg}(v) = 0$). Denote by $\text{sources}(D)$ and $\text{in}(D)$ the sets of all sources and sinks of D , respectively. For each vertex $v \in V$, put

$$\text{sources}(v) = \{u \in \text{out}(D) \mid (u, v) \in E\}, \quad (6)$$

$$\text{sinks}(v) = \{u \in \text{sinks}(D) \mid (v, u) \in E\}. \quad (7)$$

For every transitive graph it is easy to see that $\text{sources}(v)$ is equal to the set of all vertices u in $\text{sources}(D)$ such that there exists a directed path from u to v . Similarly, $\text{sinks}(v)$ coincides with the set of all vertices u in $\text{sinks}(D)$ such that there exists a directed path from v to u .

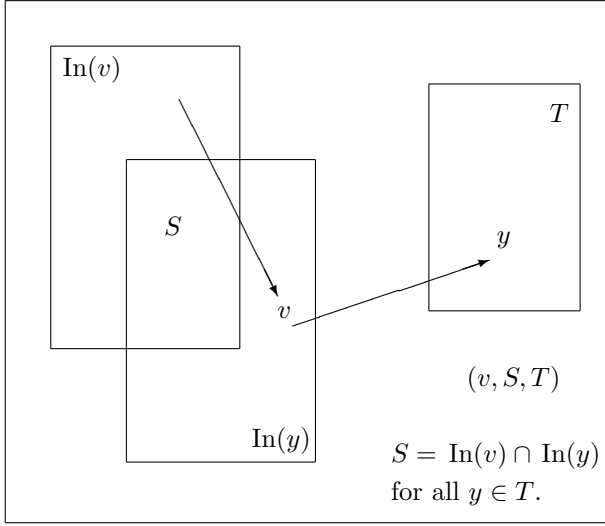


Figure 2: Steps of Algorithm 1

As we can see from Figure 1, the two key values computed by the algorithm can be defined as follows. Take a vertex $v \in V$. For each vertex $v \in V$, we introduce two special sets of vertices:

$$\text{In}(v) = \{x \in V \mid (x, v) \in E\},$$

$$\text{Out}(v) = \{x \in V \mid (v, x) \in E\}.$$

The first value b is found as the largest number of elements y in $\text{Out}(v)$ such that

$$\text{In}(y) \cap \text{In}(v) = S$$

that can be achieved for all v in V and all subsets S of $\text{In}(v)$. The second value c is the number of all edges (x, y) such that there does not exist any vertex z satisfying $(z, x), (z, y) \in E$.

Both of these values are quite sophisticated, and this is what makes the proof of our main theorem given in the next section rather nontrivial. Note that a brute force exhaustive search through all the elements v and all subsets S of $\text{In}(v)$ above requires exponential time.

4 Proofs of the main theorems

Proof of Theorem 1. In the first part of the proof we are going to show that the code $C_D(g_b)$ can correct $\lfloor (b-1)/2 \rfloor$ errors and the code $C_D(g_c)$ can correct $\lfloor (c-1)/2 \rfloor$ errors.

Minimal prerequisites on coding theory are involved in a few steps of the proof of correctness of our algorithm. Recall that the *weight* $\text{wt}_H(x)$ of the element x in $I_D(F)$ is the number of edges that occur with nonzero coefficients in x . The minimum distance of a code is the minimum weight of a difference of two distinct elements in the code. The *weight* $\text{wt}_H(C)$ of an error-correcting code C is the minimum weight of a nonzero element in C . If a code is linear, then its weight is equal to its minimum distance. A code with minimum distance d can correct $\lfloor (d-1)/2 \rfloor$ errors. Conversely, a code correcting e errors has minimum distance at least $2e+1$. Therefore, all we have to verify in the first part is that $C_D(g_b)$ has minimum distance b and $C_D(g_c)$ has minimum distance c .

First, consider the code $C_D(g_b)$. It is easily seen from line 14 in Figure 1 that Algorithm 1 ensures that

$$g_b = \sum_{y \in T'} (v', y),$$

where $(v', S', T') \in L$ and $b = |T'|$. It follows from the definition (3) that

$$C_D(g_b) = \{fg_b + hg_b \mid f \in F, h \in I_D(F)\}. \quad (8)$$

Choose a nonzero element

$$c_{\min} = fg_b + hg_b$$

with minimum weight in $C_D(g_b)$. Since the element h belongs to $I_D(F)$, the definition of $I_D(F)$ given above implies that this element can be represented in the form

$$h = \sum_{(u,v) \in E} h_{(u,v)}(u, v).$$

In view of (1) all edges (u, v) with $v \neq v'$ produce zero products in the summand hg_b of c_{\min} . Hence we can rewrite c_{\min} as

$$c_{\min} = fg_b + \sum_{(u,v') \in E} h_{(u,v')}(u, v') \sum_{y \in T'} (v', y), \quad (9)$$

where $f, h_{(u,v')} \in F$. Further, consider two possible cases.

Case 1. There exists $u \neq v'$ such that $h_{(u,v')} \neq 0$ and $(u, y) \in E$. In this case the first term

$$f \sum_{y \in T'} (v', y)$$

of c_{\min} in (9) has no summands with edges which begin in u . Therefore, if we look at the sum of the edges in c_{\min} which begin in u , then it follows from (1) that we get the expression

$$h_{(u,v')}(u, v') \sum_{y \in T'} (v', y), \quad (10)$$

which is a part of c_{\min} and does not cancel with the remaining summands of c_{\min} .

For any vertex $v \in V$, define the set

$$\text{In}(v) = \{x \in V \mid (x, v) \in E\}.$$

Notice that Algorithm 1 ensures that

$$S = \text{In}(v) \cap \text{In}(y)$$

in line 7 before it inserts y in T for $(v, S, T) \in L$ in line 9. It follows that our triple (v', S', T') satisfies $|T'| = b$ and

$$(\forall y \in T') \quad S' = \text{In}(v') \cap \text{In}(y) \quad (11)$$

Comparing (10) with (11) we see that the following two subcases are possible.

Subcase 1.1. $u \notin S'$. Then $(u, v')(v', y) = 0$ for all $y \in T'$. In this case (10) could have been eliminated from (9) which would only make c_{\min} expressed in a simpler form in (9). Since eliminations like this cannot continue indefinitely, we may assume that this subcase does not occur.

Subcase 1.2. $u \in S'$. Then $(u, v')(v', y) = (u, y)$ for all $y \in T'$, and (10) becomes

$$h_{(u,v')} \sum_{y \in T'} (u, y).$$

In this case the expression (10) contributes $|T'| = b$ to the weight of c_{\min} . Hence $w_H(c_{\min}) \geq b$.

Case 2. There does not exist $u \neq v'$ such that $h_{(u,v')} \neq 0$ and $(u, y) \in E$. In this case it follows from (1) that

$$c_{\min} = f \sum_{y \in T'} (v', y) + h_{(v',v')} \sum_{y \in T'} (v', y). \quad (12)$$

If $(v', v') \notin E$, then (10) implies

$$c_{\min} = f \sum_{y \in T'} (v', y)$$

and we get

$$w_h(c_{\min}) = |T'|.$$

If, however, $(v', v') \in E$, then (10) yields

$$c_{\min} = (f + h_{(v',v')}) \sum_{y \in T'} (v', y). \quad (13)$$

Given that c_{\min} is nonzero, we get

$$f + h_{(v',v')} \neq 0.$$

Therefore $w_h(c_{\min}) = |T'| = b$, again.

Thus, in all cases we have $w_H(c_{\min}) \geq b$. Clearly, $C_D(g_b)$ contains elements with weight b . By the choice of c_{\min} , it follows that the minimum distance of $C_D(g_b)$ is b .

Second, consider the code $C_D(g_c)$. Lines 15 to 24 of Algorithm 1 guarantee that $g_c =$

$$= \sum \{(u, v) \in E \mid (\forall t \in V)(t, u) \notin E \text{ or } (t, v) \notin E\}. \quad (14)$$

It follows from (1) that $h_{g_c} = 0$ for all $h \in I_D(F)$. Therefore (3) implies that

$$C_D(g_c) = \{fg_b \mid f \in F\}. \quad (15)$$

Hence the minimum distance of $C_D(g_c)$ is c indeed.

It remains to show that $I_D(F)$ never has codes of the form $C_D(g_1, \dots, g_k)$ which can correct more errors than the best of the codes $C_D(g_b)$ and $C_D(g_c)$. (Note that some of the codes $C_D(g_1, \dots, g_k)$ may have larger information rates.) Let us consider an arbitrary code $C = C_D(g_1, \dots, g_k)$. We have to verify that the number of errors in correct does not exceed

$$\max\{\lfloor (b-1)/2 \rfloor, \lfloor (c-1)/2 \rfloor\}.$$

To this end we'll show that the minimum distance of C is at most $\max\{b, c\}$.

Let us start with a nonzero element $x = \sum_{i=1}^m f_i(x_i, y_i)$ which has minimum weight d in C , where $(x_i, y_i) \in E$, $f_i \in F$ and $f_i \neq 0$ for all i . We are to check that $\text{wt}_H(x) \leq \max\{b, c\}$. Consider several possible cases.

Case 1. There exists $u \in E$ such that $(u, x_i), (u, y_i) \in E$ for some i . Put $v = x_i$. Then it follows from (1) that $(u, v)x \neq 0$ and that all edges occurring in $(u, v)x$ begin in u . By the minimality of $w_H(x)$, we see that $w_H((u, v)x) = w_H(x)$, and so $(u, v)x = x$. Therefore, by (1) we have $x_1 = \dots = x_m = v$ and

$$x = \sum_{i=1}^m f_i(v, y_i), \quad (16)$$

where $(u, y_i) \in E$ for all i .

Suppose that there exist $1 \leq i, j \leq m$ such that

$$\text{In}(v) \cap \text{In}(y_i) \neq \text{In}(v) \cap \text{In}(y_j).$$

Without loss of generality we may assume that

$$\text{In}(v) \cap \text{In}(y_i) \subset \text{In}(v) \cap \text{In}(y_j).$$

Choose any

$$w \in \text{In}(v) \cap \text{In}(y_j) \setminus \text{In}(v) \cap \text{In}(y_i).$$

By the choice of w and (1), we get

$$(w, v)(v, y_i) = 0, \quad (17)$$

$$(w, v)(v, y_j) = (w, y_j), \quad (18)$$

because $(w, y_i) \notin E$ and $(w, y_j) \in E$. Since x is in $C_D(g_1, \dots, g_k)$, it follows from (3) that $(w, v)x$ belongs to $C_D(g_1, \dots, g_k)$, too. Besides, (18) implies that $(w, v)x \neq 0$. However, (17) shows us that $w_H((w, v)x) < w_H(x)$. This contradicts the minimality of $w_H(x)$, and demonstrates that, for all $1 \leq i, j \leq m$,

$$\text{In}(v) \cap \text{In}(y_i) = \text{In}(v) \cap \text{In}(y_j) = S. \quad (19)$$

Here we have denoted the intersection which occurs in (19) by S .

Let i be the smallest positive integer such that $1 \leq i \leq n$ and $\text{In}(v) \cap \text{In}(y_i) = S$. (Obviously, $i \leq b$.) It is straightforward that Algorithm 1 inserts $(v, S, \{v_i\})$ in L in line 11. Hence, when L is complete and b is being found, there will always exist T and t such that (v, S, T) belongs to L .

It follows from (19) that $y_1, \dots, y_m \in T$. Therefore $m \leq |T| = t \leq b$ by the choice of b in line 14 of the algorithm. However, $d = m$. Thus $d \leq b$ in this case.

Case 2. For all $u \in V$ and all $1 \leq i \leq m$, if $(u, x_i) \in E$ then $(u, y_i) \notin E$. Then the containment condition in line 19 of Algorithm 1 never holds true for $v = x_i, y = y_i$. Therefore line 21 is never executed for edges (x_i, y_i) which occur in x . Hence all of these edges remain in g_c , and in this case we get

$$w_H(x) \leq w_H(g_c).$$

This demonstrates that Algorithm 1 indeed returns the correct largest number of errors that a code of the form $C_D(g_1, \dots, g_k)$ can correct.

Let us now evaluate the running time of Algorithm 1. It is clear that lines 3, 4 execute in $O(n^2)$ time.

The loops in lines 5, 6 have n^2 iterations. Each iteration requires $O(n)$ to find the intersection in line 7. Given the upper bound on $|L|$, the containment condition in line 8 can be verified in $O(\lg(n^2)) = O(\lg(n))$ time, see (Asano et al. 2003) and (Cormen et al. 2001), Chapter 13. Each insertion in T in line 9 takes $O(n)$, and could even be reduced to $O(\lg(n))$, but this is not necessary. Similarly, each insertion in L in line 11 can be done in $O(\lg(n))$. Therefore the nested loops in lines 5 to 11 run in $O(n^3)$ time.

Line 14 executes in $O(\lg(n))$, and line 15 takes $O(n^2)$ to compute. The running time of lines 16 to 24 is $O(n^3)$. Therefore, the total running time of Algorithm 1 is $O(n^3)$. \square

Proof of Theorem 2. Suppose that the cyclic code C consists of codewords

$$(a_0, \dots, a_{n-1})$$

over the finite field F as encoding alphabet, where

$$a_0, \dots, a_{n-1} \in F.$$

If we identify each codeword (a_0, \dots, a_{n-1}) with the polynomial

$$a_0 + a_1x + \dots + a_{n-1}x^{n-1} \in Q_n,$$

then the code C becomes embedded in Q_n . It is well known that then there exists one polynomial $g(x)$ in Q_n such that C coincides with the set of all multiples of $g(x)$ in Q_n , i.e.,

$$C = \{f(x) \cdot g(x) \mid f(x) \in Q_n\}. \quad (20)$$

Let $K_n = (V_n, E_n)$ be the complete graph with the set

$$V_n = \{v_1, \dots, v_n\}$$

of vertices and E containing all edges including loops. For any subgraph $D = (V, E)$ of K_n , denote by $A = A_D$ the following element of $I_{K_n}(F_q)$

$$A_D = \sum_{(i,j) \in E} (v_i, v_j) \in I_{K_n}(F_q).$$

For example, if

$$L_n = (V, \{(1,1), \dots, (n,n)\})$$

is the set of all loops, then A_L is the identity element I_n of $I_{K_n}(F_q)$. Denote by C_n the cycle

$$C_n = (V, \{(v_1, v_2), \dots, (v_{n-1}, v_n), (v_n, v_1)\})$$

regarded as a subgraph of $I_{K_n}(F_q)$, and let $y = A_{C_n}$. The modulo operator gives the remainder of m on division by n and is denoted by $m \% n = m \bmod n$. It is easy to verify that the following equalities hold in $I_{K_n}(F)$, for all nonnegative integers k, m and all powers of y ,

$$y^k = \sum_{i=1}^n e_{i, (i+k) \bmod n}, \quad (21)$$

$$\begin{aligned} y^k y^m &= \sum_{i=1}^n e_{i, (i+k+m) \bmod n} \\ &= y^{(k+m) \bmod n}. \end{aligned} \quad (22)$$

The edge sets of the graphs of

$$y, y^2, \dots, y^{n-1}$$

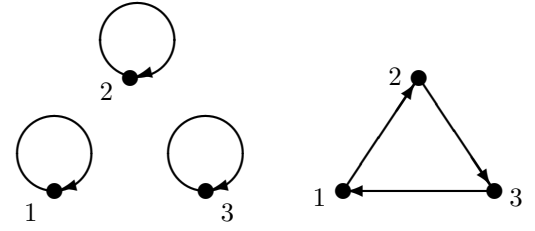
partition the set of edges of complete graph K_n . These graphs and their adjacency matrices are illustrated in Figure 3.

The equalities (22) for $1, y, \dots, y^{n-1}$ above are precisely those that are satisfied for $1, x, \dots, x^{n-1}$ according to the definition of multiplication in the quotient ring Q_n . Therefore the linear space $F[Y]$ spanned by the set

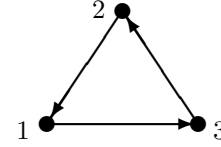
$$Y = \{1, y, y^2, \dots, y^{n-1}\} \quad (23)$$

in $I_{K_n}(F_q)$ is isomorphic to Q_n , i.e., it has the same elements and operations up to notation used for elements. If we identify the elements x^k of Q_n with elements y^k , then the parity-check code turns into the set generated in $F[Y]$ as the set of all multiples of the element

$$1 - y = \begin{bmatrix} 1 & -1 & 0 & \dots & 0 \\ 0 & 1 & -1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -1 & 0 & 0 & \dots & 1 \end{bmatrix}$$



$$1 = y^0 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad y = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$



$$y^2 = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

Figure 3: Adjacency matrices $1, y, y^2 \in M_3[F]$.

This example illustrates the fact that every cyclic code can be defined as a set of all multiples of one element g of the ring $I_{K_n}(F)$ multiplied by all elements of a subring of the form

$$F[Y] = Fy^0 + Fy + Fy^2 + \dots + Fy^{n-1},$$

where Y is given by (23). \square

Proof of Theorem 3. Let C be a linear (n, m) code over a finite field F , and let D be the graph with the set of vertices $V = \{1, \dots, n+1\}$ and the set of edges $\{(1, 2), (1, 3), \dots, (1, n+1)\}$. It is easily seen that the set of all elements of the form

$$c_0(1, 2) + c_1(1, 3) + \dots + c_n(1, n+1)$$

such that

$$(c_0, c_1, \dots, c_n) \in C$$

is equivalent to C regarded as a linear code. It is also easily seen that this set is closed with respect to the multiplication by arbitrary elements of $I_D(F)$. \square

5 Examples

Example 1 Let D be the graph in Figure 4. Then it is not hard to check that all sets T have cardinality at most one, for all $(v, S, T) \in L$, and so g_b generates the code which cannot correct any errors. However, the maximum number of errors which can be corrected is n with the optimal code generated by

$$g_c = (x_n, y_n) + (v, y_1) + \sum_{i=1}^n (x_i, v) + \sum_{i=1}^{n-1} (x_i, y_{i+1})$$

Example 2 Let D be the graph in Figure 5. Then the maximum number of errors which can be corrected by codes $C_D(g_1, \dots, g_k)$ is $\lfloor (n-1)/2 \rfloor$ with the optimal code generated by

$$g_b = \sum_{i=1}^n (v, y_i)$$

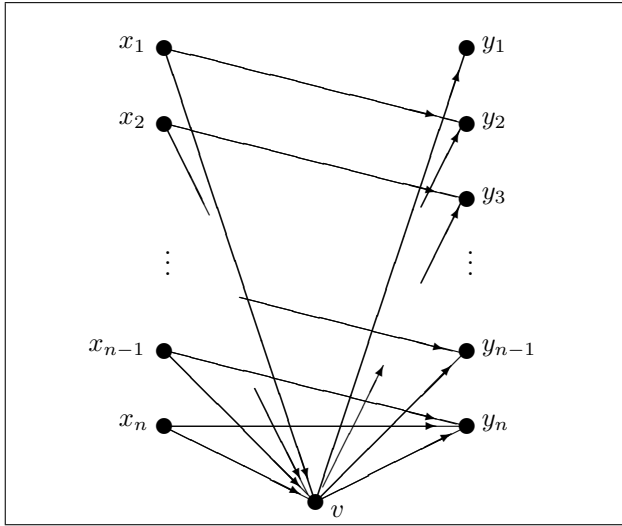


Figure 4: Graph in Example 1

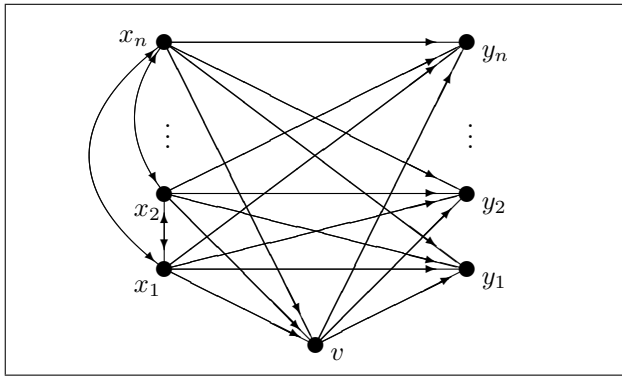


Figure 5: Graph in Example 2

Example 3 Let D be the graph in Figure 6. Then the maximum number of errors which can be corrected by codes $C_D(g_1, \dots, g_k)$ is $\lfloor n^2 - 1/2 \rfloor$ with the optimal code generated by

$$g_c = \sum_{i=1}^n \sum_{j=1}^n (x_i, y_j)$$

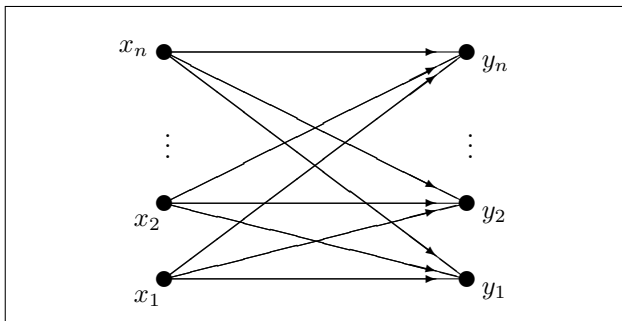


Figure 6: Graph in Example 3

The author is grateful to three referees for detailed corrections and comments that have helped to improve the exposition.

References

- Asano, T., Wada, K. & Masuzawa, T. (2003), *Theory of Algorithms*, Ohm Publishing.
- Cazaran, J., Kelarev, A.V., Quinn, S.J. Vertigan, D. (2006), An algorithm for computing the minimum distances of extensions of BCH codes, in preparation.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. (2001), *Introduction to Algorithms*, The MIT Press, Cambridge.
- Downey, R. G. & Fellows, M. R. (1999), *Parameterized Complexity*, Monographs in Computer Science, Springer, New York.
- Downey, R. G. & Fellows, M. R. (1999), Parameterized complexity after (almost) ten years: review and open questions, in 'Combinatorics, Computation & Logic '99' (Auckland), *Aust. Comput. Sci. Commun.* **21**(3), 1–33.
- Downey, R., Fellows, M. R., Whittle, G. & Vardy, A. (1999), 'The parametrized complexity of some fundamental problems in coding theory', *SIAM J. Comput.* **29**(2), 545–570.
- Kelarev, A. V. (2001), On a theorem of Cohen and Montgomery for graded rings, *Proc. Royal Soc. Edinburgh A*, 131, 1163–1166.
- Kelarev, A. V. (2002), *Ring Constructions and Applications*, World Scientific.
- Kelarev, A. V. (2003), *Graph Algebras and Automata*, Marcel Dekker, New York.
- Kelarev, A. V. (2004), Minimum distances and information rates for matrix extensions of BCH codes, in 'The 3rd Workshop on the Internet, Telecommunications and Signal Processing', WITSP 2004, (Adelaide, 20–22 December 2004), pp. 1–6.
- Kelarev, A. V. & Sokratova, O. V. (2001), Information rates and weights of codes in structural matrix rings, *Lecture Notes in Computer Science* **2227**, 151–158.
- Kelarev, A. V. & Solé, P. (2000), Error-correcting codes as ideals in group rings, in 'Proc. Internat. Conf. AGRAM 2000', (Perth, Australia, September 2000), pp. 11–18.
- Lidl, R. & Niederreiter, H. (1994), *Introductions to Finite Fields and Their Applications*, Cambridge University Press, Cambridge.
- Lidl, R. & Niederreiter, H. (1997), *Finite Fields*, Cambridge University Press, Cambridge.
- Lidl, R. & Wiesenbauer, J. (1980), *Ring Theory and Applications*, Wiesbaden (in German).
- Moffat, A. & Turpin, A. (2002), *Compression and Coding Algorithms*, Kluwer.
- Pless, V. S., Huffman, W. C. & Brualdi, R. A., eds., (1998), *Handbook of Coding Theory*, Elsevier.
- Stallings, W. (2002), *Wireless Communications and Networking*, London, Prentice Hall.

On the complexity of the DNA Simplified Partial Digest Problem

Jacek Blazewicz

Marta Kasprzak

Institute of Computing Science, Poznan University of Technology, Poznan,
and Institute of Bioorganic Chemistry, Polish Academy of Sciences, Poznan, Poland.
Emails: jblazewicz@cs.put.poznan.pl, marta@cs.put.poznan.pl

Abstract

The problem to be addressed is one of the genome mapping of DNA molecules. The new approach — the Simplified Partial Digest Problem (SPDP), is analyzed. This approach is easy in laboratory implementation and robust with respect to measurement errors. In the paper, it is formulated in terms of a combinatorial search problem and proved to be strongly NP-hard for the general error-free case. For a subproblem of the SPDP, a simple $O(n \log n)$ -time algorithm is given, where n is a number of restriction sites.

Keywords: combinatorial optimization, DNA restriction mapping, partial digest, computational complexity.

1 Introduction

A construction of a DNA physical map is an important step in the genome sequencing process (cf. Waterman (1995), Setubal & Meidanis (1997), Pevzner (2000), and Fogel & Corne (2003) as excellent sources of algorithmic ideas used in computational biology). This map of a DNA molecule contains the information about locations of short, specific subsequences called markers and in turn places longer DNA subchains on the chromosome. One approach to create the map is based upon splitting the molecule into many shorter ones and *hybridizing* them with the *markers* (known very short DNA sequences). This approach results in the *interval graph model* used successfully in the past (Benzer 1959).

A more recent approach to the genome map construction relies on a digestion of a DNA molecule with restriction enzymes (Waterman 1995). These enzymes cut DNA molecules within specific, short patterns of nucleotides called restriction sites. After the digestion, the lengths of obtained fragments are measured and the original ordering of these fragments must be reconstructed, and this is the place where combinatorial optimization methods come to the effect. In practice, several variants of this approach are used. Two of the best known are *the double digest* and *the partial digest* (with its many variants).

The research has been supported by KBN grant 3T11F00227. The research of the first author was sponsored by the ESPRC Fellowship at the University of Nottingham.

Corresponding author: Marta Kasprzak, Institute of Computing Science, Poznan University of Technology, Piotrowo 3A, 60-965 Poznan, Poland.

Copyright ©2006, Australian Computer Society, Inc. This paper appeared at Computing: The Australasian Theory Symposium (CATS2006), Hobart, Australia. Conferences in Research and Practice in Information Technology, Vol. 51. Barry Jay and Joachim Gudmundsson, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

In the double digest approach two restriction enzymes are used. A target DNA is amplified, e.g. using a PCR reaction, and the copies are divided into three sets. Molecules from the first set are digested by one enzyme, molecules from the second set are digested by the other enzyme and molecules from the third set are cut by both enzymes. All digestions are complete for the time span of each reaction is long enough to allow the enzyme to cut the target strand at each occurrence of the restriction site. As the result one obtains three collections of short DNA fragments that correspond to three digestion processes. The lengths of these fragments are measured during a gel electrophoresis process and recorded as three multisets. On the basis of this data locations of restriction sites in the target DNA are reconstructed. Unfortunately, from the combinatorial point of view the *Double Digest Problem* (DDP) is NP-hard even in an ideal case involving no errors (Goldstein & Waterman 1987), thus unlikely to admit a polynomial time algorithm. Another difficulty which must be taken into account when dealing with this approach is an exponential number of possible solutions (as measured with respect to a number of restriction sites — n) (Schmitt & Waterman 1991). As an alternative *the partial digest approach* has been proposed (Skiena, Smith & Lemke 1990, Skiena & Sundaram 1994). Here, one enzyme only is applied to cut the DNA molecule into fragments of different lengths by using the enzyme for different time periods. In the error-free case one gets here all $\binom{n}{2}$ fragment lengths between all pairs of cuts (including two ends of the DNA molecule). The resulting *Partial Digest Problem* (PDP) consists in reconstructing the original positions of the cuts. This problem is known also in discrete geometry (Skiena, Smith & Lemke 1990, Skiena 1997), where having all interpoint distances, one reconstructs the positions of a set of points on a line. For PDP two backtracking algorithms with an exponential worst case complexity have been proposed by Skiena and co-workers (Skiena, Smith & Lemke 1990, Skiena & Sundaram 1994). In general, the complexity of the PDP remains an open question, although measurement errors and noisy data result in the strong NP-hardness of the problem ((Cieliebak & Eidenbenz 2004) and (Cieliebak, Eidenbenz & Penna 2003), respectively). It is worth stressing that the number of possible solutions in the PDP is bounded from above by a polynomial in the number of restriction sites (Skiena, Smith & Lemke 1990). Other known approaches in that area include *optical mapping* (Schwartz et al. 1993, Karp & Shamir 1998), *probed partial digest mapping* (Newberg & Naor 1993), and *labeled partial digest* (Pandurangan & Ramesh 2002). Let us especially comment on the labeled partial digest approach, which although more complicated from the implementation point of view (labeling the ends of a DNA molecule by using radioactive labeling), results in a

polynomial algorithm for the related variant of the PDP and produces a unique solution (Pandurangan & Ramesh 2002).

In this paper, we study yet another variant of the PDP, called the *Simplified Partial Digest Problem* (SPDP), which is very simple for the laboratory implementation (Blazewicz et al. 2001). In this approach only two digestions are performed. We will call them, respectively, a short digestion and a complete (long) digestion. After amplification, the copies of a target strand are splitted into two sets. The goal of a short digestion is to have all molecules from one of the sets cut in at most one occurrence of the restriction site. This is assured by properly chosen time span of the reaction. Molecules from the other set are cut in all occurrences of the restriction site due to the long reaction time span (a complete digestion). Then, as in other methods, the lengths of restriction fragments obtained, are measured during a gel electrophoresis process. Although, the algorithms proposed for the SPDP (Blazewicz et al. 2001, Blazewicz & Jaroszewski 2003) are very efficient on the average and robust with respect to measurement errors, the complexity of the problem in the error-free case was open for several years.

In this paper, we present the proof that the general error-free variant of the SPDP is strongly NP-hard (of course in its *search version*). On the other hand, if the complete digestion yields a multiset composed of 1s and 2s only, the problem is solvable in $O(n \log n)$ time, where n is a number of digestion sites. An organization of the paper is as follows. Section 2 contains the formal definition of the SPDP and the proof of its strong NP-hardness. Section 3 presents a polynomial time algorithm for the restricted case of the problem. Conclusions in Section 4 summarize the work and indicate further research problems.

2 Computational complexity of the SPDP

In this section we prove that the combinatorial search version of the Simplified Partial Digest Problem without errors is strongly NP-hard. It is worth stressing that we do not do it in a standard way by a transformation from the decision version of a known intractable problem to the decision version of our problem, since the decision counterpart of the SPDP is polynomially solvable. This is because — on the assumption that the instance of our problem has no errors — the restriction fragments from both digestion reactions exactly match those existing in the real restriction map, therefore the solution always exists. However, the process of finding this solution (i.e. the map) is not easy from the computational complexity point of view.

A similar situation — where the decision version of a problem is easy but its search version is hard — is encountered in the DNA Sequencing problem (Blazewicz & Kasprzak 2003). In this problem, as well as in SPDP, the additional information about instances — the lack of errors or some type of errors — results in the answer “yes” for every instance of the decision version. Therefore, the process of proving strong NP-hardness of the search problem must go through an artificial decision problem, allowing also instances with the answer “no”.

We based our reasoning on a deduction formed in Johnson (1985) for the Hamiltonian Circuit Problem (being NP-complete in its decision version). As it has been stated there, even if we knew a graph contains a Hamiltonian circuit (in this case the decision version is, of course, trivially easy), we could not find it in polynomial time unless $P=NP$. For “if we had such an algorithm A , we could use it to tell in polynomial

time whether an arbitrary graph G has a Hamiltonian circuit. Let p be the polynomial that bounds A ’s running time on graphs with Hamiltonian circuits. Apply A to G . If G has a Hamiltonian circuit, A will find one in time $p(|G|)$. If G does not have such a circuit, then after $p(|G|)$ steps A could not have found one, and we will know that none exists”.

2.1 Formulations of related problems

The innovation of the well-known Partial Digest Problem, studied here, was based on a simplified biochemical experiment and proposed for the first time by Blazewicz et al. (2001). Its search version without any error in instances can be formulated as below.

Problem 1 *Simplified Partial Digest Problem* (Π_{SPDS}) — *search version*.

Instance: Multiset $A = \{a_1, a_2, \dots, a_{2n}\}$ of lengths of restriction fragments coming from short digestion and multiset $B = \{b_1, b_2, \dots, b_{n+1}\}$ of lengths of restriction fragments coming from long digestion, A and B containing no errors.

Answer: A map of n restriction sites of a DNA chain consistent with multisets A and B , i.e. such an order of elements of B which allows to cover the indicated cuts by some order of elements of A .

(See Appendix for original formulation from Blazewicz et al. (2001).) We see that $D_{\Pi_{\text{SPDS}}} = Y_{\Pi_{\text{SPDS}}}$, where $D_{\Pi_{\text{SPDS}}}$ is the set of all instances of Π_{SPDS} and $Y_{\Pi_{\text{SPDS}}}$ is the set of all instances of Π_{SPDS} for which such a solution exists. (This follows from the fact, that both multisets come from a real error-free digestion experiment.) Example 1 shows an instance of this problem together with a possible solution.

Example 1 Let the output of the digestion reactions performed without errors be: $A = \{3, 6, 8, 9, 10, 11, 13, 16\}$ from the short reaction and $B = \{2, 3, 3, 5, 6\}$ from the long one. The feasible solution for the problem is presented in Figure 1 and can be written as the following ordered list of B : $[3, 5, 2, 3, 6]$. \square

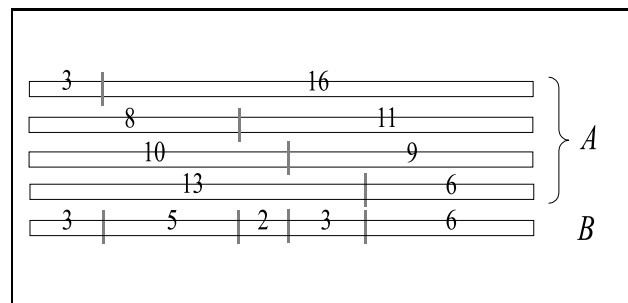


Figure 1: An example for the Simplified Partial Digest Problem without errors.

To study the computational complexity of problem Π_{SPDS} , we must introduce an additional decision *quasi-mapping* problem Π_{QMD} . In the quasi-mapping problem the multisets contain arbitrary positive integers instead of the ones coming from an errorless experiment, thus the answer to the new problem is not always “yes”.

Problem 2 *Quasi-Mapping Problem* (Π_{QMD}) — *decision version*.

Instance: Multisets $A = \{a_1, a_2, \dots, a_{2n}\}$ and $B = \{b_1, b_2, \dots, b_{n+1}\}$ of positive integers.

Question: Does there exist a map of n “restriction

sites" consistent with multisets A and B , i.e. such an ordering of elements of B which allows to cover the indicated "cuts" by some order of elements of A ?

The two problems Π_{SMs} and Π_{QMd} have the same sets of instances with the positive answer: $Y_{\Pi_{\text{SMs}}} = Y_{\Pi_{\text{QMd}}}$, because any instance of Π_{QMd} resulting in the "yes" answer belongs also to $D_{\Pi_{\text{SMs}}}$. However, problem Π_{QMd} contains also instances with the negative answer ($D_{\Pi_{\text{SMs}}} \subset D_{\Pi_{\text{QMd}}}$).

In the next subsection we construct a pseudo-polynomial transformation from the problem Numerical Matching With Target Sums, described below, to the considered SPDP.

Problem 3 Numerical Matching With Target Sums (Π_{NMd}) — decision version (Garey & Johnson 1979).

Instance: Disjoint sets X and Y , each containing m elements, sizes $s(x_i)$ and $s(y_i)$ for every $x_i \in X$ and $y_i \in Y$, and a target vector $[z_1, z_2, \dots, z_m]$; $s(x_i)$, $s(y_i)$, and z_i being positive integers, $i = 1..m$.

Question: Can $X \cup Y$ be partitioned into m disjoint sets W_1, W_2, \dots, W_m , each containing exactly one element from X and one element from Y , such that $\sum_{w \in W_i} s(w) = z_i$, $i = 1..m$?

This problem is strongly NP-complete (Garey & Johnson 1979).

2.2 The transformation

The first stage of proving strong NP-hardness of the Simplified Partial Digest Problem without errors consists in a simple modification of problem Π_{NMd} . We are interested in a variant with the ranges of variables shifted by some added values. The modified ranges will have several properties, in particular, they will be disjoint. Initially, the ranges of values of variables $s(x_i)$, $s(y_i)$, and z_i , $i = 1..m$, can be written as follows:

$$\begin{aligned} s(x_i) &\in \langle x_L, x_R \rangle, \\ s(y_i) &\in \langle y_L, y_R \rangle, \\ z_i &\in \langle z_L, z_R \rangle, \end{aligned}$$

where the variables with index L mean the smallest values and the ones with index R mean the largest values in the respective collections. These ranges are arbitrary, however, we assume here that they satisfy few obvious conditions:

$$\begin{aligned} z_R &\leq x_R + y_R, \\ z_L &\geq x_L + y_L, \\ z_R &> x_R, \\ z_R &> y_R. \end{aligned}$$

The above assumptions do not change the computational complexity of problem Π_{NMd} — if one of them is not satisfied, the problem becomes easy because the answer is obviously "no".

We modify the initial ranges of the variables in the following way (where ":= " assigns the right-hand side value to the left-hand side variable):

$$\begin{aligned} s(x_i) &:= s(x_i) + z_R, & i &= 1..m \\ s(y_i) &:= s(y_i) + 2x_R + 2z_R, & i &= 1..m \\ z_i &:= z_i + 2x_R + 3z_R, & i &= 1..m. \end{aligned}$$

The new ranges have the following form ($i = 1..m$):

$$\begin{aligned} s(x_i) &\in \langle x_L + z_R, x_R + z_R \rangle, \\ s(y_i) &\in \langle y_L + 2x_R + 2z_R, y_R + 2x_R + 2z_R \rangle, \\ z_i &\in \langle z_L + 2x_R + 3z_R, 4z_R + 2x_R \rangle. \end{aligned}$$

They are visualized in Figure 2.

Lemma 1 Problem Π_{NMd} and its version with the ranges of variable values shifted as above, are equivalent.

PROOF

Both problems have the same sets of instances (at the beginning). The only difference is in the equation from the question, where, instead of $s(x_i) + s(y_j) = z_k$ in problem Π_{NMd} , we have $s(x_i) + z_R + s(y_j) + 2x_R + 2z_R = z_k + 2x_R + 3z_R$ in the new problem with the shifted ranges. We see that the two problems are the same. \square

The variables from problem Π_{NMd} increased by the proposed values have several useful properties.

Lemma 2 None of the modified z_i , $i = 1..m$, can be equal to some $s(x_j)$ or to a sum of any $s(x_j)$ and $s(x_k)$.

PROOF

The first statement is obvious, see Figure 2. As to the second one, the sum of two largest possible values of $s(x_j)$ and $s(x_k)$, i.e. $2x_R + 2z_R$, is still smaller than $z_L + 2x_R + 3z_R$ being the smallest z_i . \square

Lemma 3 None of the modified z_i , $i = 1..m$, can be equal to some $s(y_j)$ or to a sum of any $s(y_j)$ and $s(y_k)$.

PROOF

Also here the first statement is obvious, see Figure 2. On the other hand, the sum of two smallest possible values of $s(y_j)$ and $s(y_k)$, i.e. $2y_L + 4x_R + 4z_R$, is larger than $2x_R + 4z_R$ being the largest possible z_i . \square

Lemma 4 None of the modified z_i , $i = 1..m$, can be equal to a sum of any $s(y_j)$, $s(x_k)$, and $s(x_l)$.

PROOF

The sum of one smallest $s(y_j)$ and two smallest $s(x_k)$ and $s(x_l)$, i.e. $y_L + 2x_L + 2x_R + 4z_R$, is larger than the largest $z_i = 2x_R + 4z_R$. \square

Now, we can define a transformation from problem Π_{NMd} to problem Π_{QMd} , which is given below.

The transformation

Given an instance of problem Π_{NMd} , the corresponding instance of Π_{QMd} is constructed as follows.

- (1) Shift the ranges of numbers in the problem Π_{NMd} as specified above, i.e.

$$\begin{aligned} s(x_i) &:= s(x_i) + z_R, & i &= 1..m \\ s(y_i) &:= s(y_i) + 2x_R + 2z_R, & i &= 1..m \\ z_i &:= z_i + 2x_R + 3z_R, & i &= 1..m. \end{aligned}$$

From now on all the variables have these modified values, if not stated otherwise.

- (2) Add values $s(x_i)$ and $s(y_i)$, $i = 1..m$, to (initially empty) multiset B . Also add to B ($\sum_{i=1..m} z_i - 2m$) times value 1.
- (3) Set L to $2 \sum_{i=1..m} z_i - 2m$ and n to $\sum_{i=1..m} z_i - 1$.
- (4) For all $i = 1..\frac{L}{2}$ add values i and $L - i$ to (initially empty) multiset A . Also add to A values $\sum_{j=1..i} z_j - i$ and $L - \sum_{j=1..i} z_j - i$ for all $i = 1..m - 1$. \square

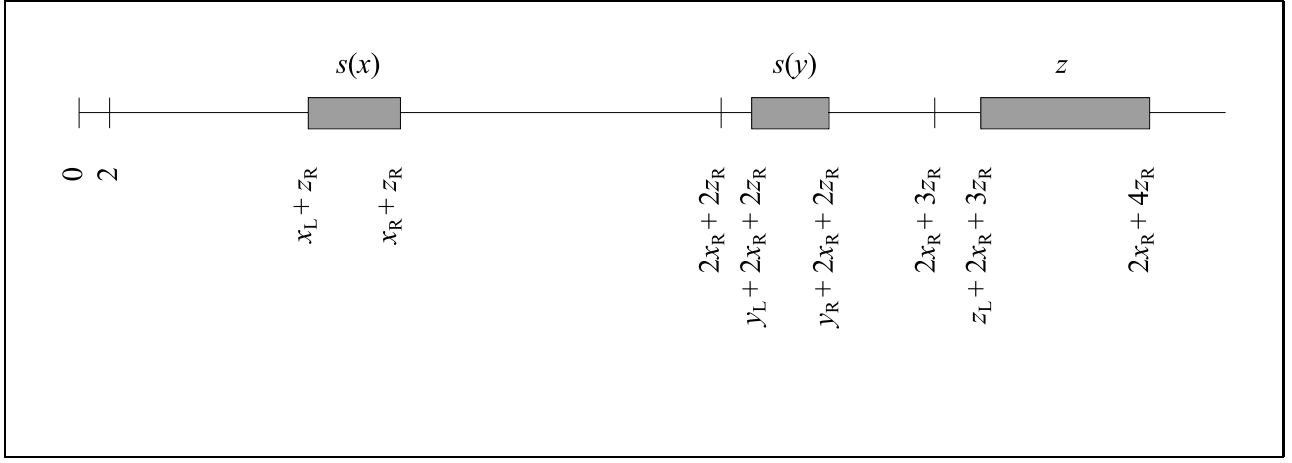


Figure 2: The ranges of values of variables $s(x_i)$, $s(y_i)$, and z_i , $i = 1..m$, after the modification.

Lemma 5 *The proposed transformation can be computed in time bounded by a polynomial in the length of the instance of Π_{NMd} (Len_{NMd}) and the maximal number appearing in this instance (Max_{NMd}).*

PROOF

Len_{NMd} is $O(m[\log \text{Max}_{\text{NMd}}])$. In the first step of the transformation we make $O(\text{Len}_{\text{NMd}})$ operations. New values of the variables do not change Len_{NMd} and Max_{NMd} substantially: m is not changed, new Max_{NMd} is up to 6 times larger than previously.

Filling multiset B requires $O(\text{Len}_{\text{NMd}} \text{Max}_{\text{NMd}})$ operations. Step (3) is $O(\text{Len}_{\text{NMd}})$ and filling A takes $O(\text{Len}_{\text{NMd}}[\log \text{Len}_{\text{NMd}}] \text{Max}_{\text{NMd}})$ operations. Taking the above functions together, we have $O(\text{Len}_{\text{NMd}}[\log \text{Len}_{\text{NMd}}] \text{Max}_{\text{NMd}})$ as the complexity of the proposed transformation. Thus, it is pseudo-polynomial in time. \square

We can now prove the following main theorem.

Theorem 1 *Quasi-mapping problem Π_{QMd} is strongly NP-complete.*

PROOF

The proof uses the proposed transformation, which is pseudo-polynomial one (see Lemma 5). Below we prove that the transformation is correct. For all $I \in D_{\Pi_{\text{NMd}}}$, $I \in Y_{\Pi_{\text{NMd}}}$ if and only if $t(I) \in Y_{\Pi_{\text{QMd}}}$, where t means the transformation.

The first step of the transformation slightly modifies problem Π_{NMd} , but both versions are equivalent (see Lemma 1). Thus, in the following the shifted ranges of variable values are used.

Let us assume that there exists a solution for problem Π_{NMd} . It means that there is a partition of $X \cup Y$ such that every disjoint subset W_i , $i = 1..m$, contains one x_j and one y_k and $s(x_j) + s(y_k) = z_i$, for some $j, k \in \langle 1, m \rangle$. Then, the solution of problem Π_{QMd} can be constructed by ordering elements of B in the following way.

```

for  $i := 1$  to  $m$  with step 1
begin
  take  $s(x_j) : x_j \in W_i$ ;
  for  $j := 1$  to  $z_i - s(x_j) - 1$  with step 1
    take 1;
end
for  $i := m$  to 1 with step -1
begin
  take  $s(y_k) : y_k \in W_i$ ;
  for  $j := 1$  to  $z_i - s(y_k) - 1$  with step 1
    take 1;
end

```

We take to the solution m times some $s(x_j)$, m times some $s(y_k)$, and $\sum_{i=1..m} (z_i - s(x_j) - 1 + z_i - s(y_k) - 1)$ times element 1, $x_j, y_k \in W_i$. All x_j and y_k , $j, k = 1..m$, are in the partitioning and the sums of sizes of these pairs cover the whole vector $[z_1, z_2, \dots, z_m]$. Thus, all the elements of B are used after finishing the above procedure.

Moreover, we can cover n indicated “cuts” by an order of pairs of all elements of A . The pairs can be easily determined by summing the elements up to L (which is the length of the solution of Π_{QMd}). The procedure of ordering them (placing an element on the left or on the right side of the solution, i.e. calculating the distance from the left or from the right end of the restriction map, respectively), is shown below.

```

for  $i := 1$  to  $m$  with step 1
  for  $j := 1$  to  $z_i - 1$  with step 1
    begin
      if  $j < s(x_k) : x_k \in W_i$  then
        begin
          place  $\sum_{l=1..i-1} (z_l - 1) + j$  on the
            right side;
          place  $L - \sum_{l=1..i-1} (z_l - 1) + j$  on
            the left side;
        end
      else
        begin
          place  $\sum_{l=1..i-1} (z_l - 1) + j$  on the
            left side;
          place  $L - \sum_{l=1..i-1} (z_l - 1) + j$  on
            the right side;
        end
      end
    end
  end
for  $i := 1$  to  $m - 1$  with step 1
begin
  place  $\sum_{j=1..i} (z_j - 1)$  on the right side;
  place  $L - \sum_{j=1..i} (z_j - 1)$  on the left
    side;
end

```

We use in the procedure all elements of A : values increasing from 1 to $\sum_{i=1..m} (z_i - 1)$ with step 1, as well as the elements $\sum_{j=1..i} (z_j - 1)$, $i = 1..m-1$, together with their complements. Moreover, their ordering agrees with the ordering within B in the sense of “cuts”, what follows the procedures. Therefore, a feasible solution for problem Π_{QMd} exists.

Now, let us assume, that there exists a solution for problem Π_{QMd} . Thus, there is an order of $n + 1$ elements of B which allows to cover the indicated “cuts”

by some order of n pairs of elements of A . The solution for the corresponding instance of problem Π_{NMD} can be then constructed as follows.

Having the solution for problem Π_{QMD} , we know that several “cuts” appear for sure. Those are the ones corresponding to duplicated pairs of complements: if we have in A value d twice (together with $L-d$ twice, of course), then in any solution to problem Π_{QMD} one d will be placed on the left and the second d will be placed on the right. (Two “cuts” cannot appear on the same side, because we assume the lack of any errors in the data of the problem. Duplication of a “cut” would be such an error.) The guaranteed “cuts” are determined by pairs $\sum_{j=1..i} z_j - i$ and $L - \sum_{j=1..i} z_j - i$ for all $i = 1..m-1$, together with their mirrors. Also the “cut” in the middle of L is guaranteed by the pair of two $\frac{L}{2}$. These “cuts” become ends of fragments, which correspond to disjoint subsets from a solution of problem Π_{NMD} . Every pair of mirroring fragments F_{iL} and F_{iR} of length $z_i - 1$ corresponds to set W_i of size z_i .

How are elements of B arranged in the solution? Except for many 1s, B contains also all $s(x_i)$ and $s(y_i)$, $i = 1..m$. These sizes are significant, especially for Y . Summing up Lemmae 2, 3, and 4, which are still true if values z_i will be decreased by 1, we deduce that every pair F_{iL} , F_{iR} can contain at most one $s(y_j)$ of any value. Because B includes m sizes of elements of Y and there are m pairs of fragments, all $s(y_i)$, $i = 1..m$, must be placed in separate pairs. In addition, none two (or more) sizes of elements of X can be added to a pair of fragments (already containing some $s(y_j)$), so all $s(x_i)$, $i = 1..m$, must be also placed in separate pairs.

At first glance, it is not obvious why two mirroring fragments F_{iL} and F_{iR} cannot include two sizes of elements of Y . It is displayed in Figure 3. If some size is assigned to one fragment, the corresponding place in the second fragment must be filled by “cuts”. Because sizes of elements of Y are greater than halves of lengths of fragments plus 1, none two such sizes can be inserted into a mirroring pair.

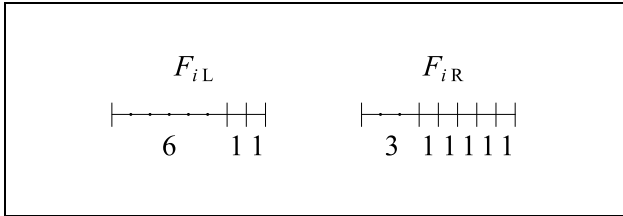


Figure 3: Two mirroring fragments F_{iL} and F_{iR} with some components.

If we have exactly one $s(x_j)$ and exactly one $s(y_k)$ assigned to a pair F_{iL} , F_{iR} , for some $i, j, k \in \langle 1, m \rangle$, plus in addition only 1s, there is only one possible placement. One of the longer sizes is shifted to the left (or right) end of F_{iL} and the second one is shifted to the left (or right, respectively) end of F_{iR} . The sum of both sizes is equal to z_i . The remaining places are filled by 1s (see Figure 3). Another placement would result in the occurrence of an additional element of a value greater than 1, what is in contradiction to previous proofs.

Having the solution of problem Π_{QMD} we can immediately read the solution of problem Π_{NMD} : those elements x_j and y_k , which sizes appear in pair F_{iL} , F_{iR} , will compose set W_i . All the requirements for a solution of problem Π_{NMD} are satisfied. This ends the proof. \square

The proposed transformation of an instance of problem Π_{NMD} to an instance of problem Π_{QMD} is illustrated by the following example.

Example 2 Let the example instance of problem Π_{NMD} be:

$$\begin{aligned} m &= 3, \\ X &= \{x_1, x_2, x_3\}, \\ Y &= \{y_1, y_2, y_3\}, \\ s(x_1) &= 2, s(x_2) = 3, s(x_3) = 5, \\ s(y_1) &= 4, s(y_2) = 5, s(y_3) = 6, \\ z_1 &= 7, z_2 = 9, z_3 = 9. \end{aligned}$$

After shifting the ranges of the variables we get the values:

$$\begin{aligned} s(x_1) &= 11, s(x_2) = 12, s(x_3) = 14, \\ s(y_1) &= 32, s(y_2) = 33, s(y_3) = 34, \\ z_1 &= 44, z_2 = 46, z_3 = 46. \end{aligned}$$

The construction of the instance of problem Π_{QMD} ends with the following result:

$$\begin{aligned} n &= 135, L = 266, \\ A &= \bigcup_{i=1..133} \{i, 266 - i\} \cup \{43, 223, 88, 178\}, \\ B &= \bigcup_{i=1..130} \{1\} \cup \{11, 12, 14, 32, 33, 34\}. \end{aligned}$$

The feasible solution for the instance after the transformation (i.e. of problem Π_{QMD}) is shown in Figure 4.

This solution can be easily translated to a feasible solution of problem Π_{NMD} :

$$\begin{aligned} W_1 &= \{x_1 + y_2\} \quad (11 + 33 = 44 \rightarrow 2 + 5 = 7), \\ W_2 &= \{x_2 + y_3\} \quad (12 + 34 = 46 \rightarrow 3 + 6 = 9), \\ W_3 &= \{x_3 + y_1\} \quad (14 + 32 = 46 \rightarrow 5 + 4 = 9). \quad \square \end{aligned}$$

Finally, we prove the computational hardness of the Simplified Partial Digest Problem without errors in the search version Π_{SMs} .

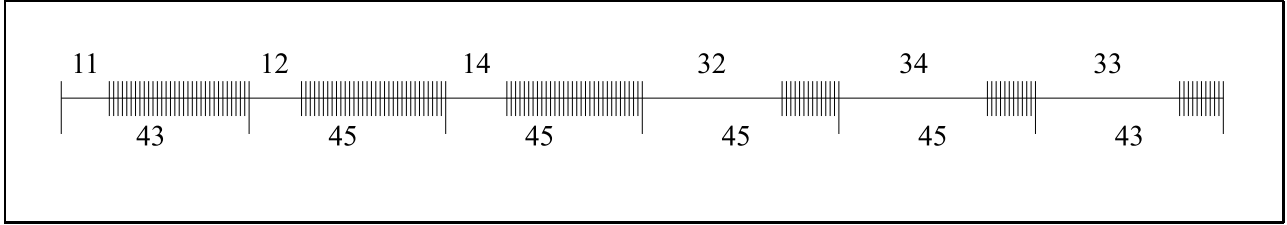
Theorem 2 The Simplified Partial Digest Problem without errors Π_{SMs} (search version) is strongly NP-hard.

PROOF

Proving strong NP-completeness of quasi-mapping problem Π_{QMD} (Theorem 1) directly leads to proving strong NP-hardness of the corresponding problem Π_{SMs} . For, if we had an algorithm solving Π_{SMs} in polynomial time, we could use it to solve problem Π_{QMD} in polynomial time in the following way. We could apply the algorithm to the instance of Π_{QMD} , and after a number of steps bounded by the polynomial function known for the algorithm we could have the answer for Π_{QMD} . Either the algorithm would find the solution and the answer would be “yes”, or the algorithm would not find one and the answer would be “no”. As pointed out earlier, a similar reasoning has been used in Johnson (1985), where the problem of looking for a Hamiltonian circuit in a graph has been considered. \square

3 A polynomially solvable subproblem of SPDP

On the other hand, a special version of the Simplified Partial Digest Problem without errors (a combinatorial search problem), in which multiset B is composed of only the elements of values 1 and 2, is polynomially solvable. In this section we present the algorithm solving the subproblem, defined below.

Figure 4: An example solution of problem Π_{QMd} .

Problem 4 *Simplified Partial Digest Problem with long-digestion fragments of lengths only 1 or 2 (Π_{SM12s}) — search version.*

Instance: Multiset $A = \{a_1, a_2, \dots, a_{2n}\}$ of lengths of restriction fragments coming from short digestion and multiset $B = \{b_1, b_2, \dots, b_{n+1}\}$ of lengths of restriction fragments coming from long digestion, A and B containing no errors and $b_i \in \{1, 2\}$, $i = 1..n+1$.

Answer: A map of n restriction sites of a DNA chain consistent with multisets A and B , i.e. such an order of elements of B which allows to cover the indicated cuts by some order of the elements of A .

The algorithm solving Π_{SM12s}

- (1) Calculate $L = \sum_{i=1..n+1} b_i$. Initially the solution is one fragment of length L without cuts.
- (2) Match complementary pairs of lengths in A , i.e. the ones summing up to L . Let the pairs be denoted by A_i , $i = 1..n$, and satisfying $\bigcup_{i=1..n} A_i = A$ and $\bigcap_{i=1..n} A_i = \emptyset$.
- (3) Select all identical pairs A_i and A_j , $i, j = 1..n$, and mark the cuts corresponding to them in the solution symmetrically (from both sides). Remove all selected A_i and A_j from A .
- (4) If A is empty, go to step (7). Otherwise, select such A_i in A , that contains the shortest length l in current A . Mark the cut placed l units from the left end of the solution and remove A_i from A .
- (5) If A is empty, go to step (7). Otherwise, increase l by 1. If some cut already exists in the solution l units from the right end, go to step (4). Otherwise, mark the cut placed l units from the right end of the solution and remove A_i which contains l from A .
- (6) If A is empty, go to step (7). Otherwise, increase l by 1. If some cut already exists in the solution l units from the left end, go to step (4). Otherwise, mark the cut placed l units from the left end of the solution, remove A_i which contains l from A , and go to step (5).
- (7) The current solution contains all cuts, i.e. it corresponds to an order of elements of multiset B .

Let us note that the algorithm uses only information coming from multiset A and does not check it with multiset B . This is because the applied moves are the only possible ones, and in the case where we have no errors they have to be correct. This is demonstrated in the following proof.

Theorem 3 *Problem Π_{SM12s} is solvable in time $O(n \log n)$.*

PROOF

Let us analyze the consecutive steps of the algorithm.

The first two steps are obvious. The third one consists in placing the symmetric cuts indicated by

pairs from A . Of course, if we have in A two identical subsets $\{x, L-x\}$, then in the problem without errors both cuts must appear. Thus, they must be placed in opposite parts of the solution, i.e. one cut x units from the left and the second one x units from the right end.

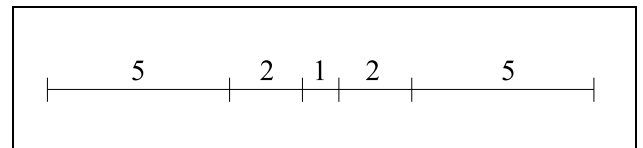
All remaining cuts are assymmetric. Because the current solution is symmetric, no matter what side we place the next cut. So, the left side can be chosen, as well as the strategy of taking the cut nearest the left end first. Therefore, the fourth step is correct. It also concerns further calling of step (4), when we fill a part between symmetric cuts and go to the next slot (i.e. a fragment between two pairs of symmetric cuts from step (3)) — then there is no difference from what side we start to place next cuts.

After placing the previous cut in step (4) we must look at the opposite side of the solution. In the symmetric place we cannot have a cut, since all cuts added after step (3) are assymmetric. And because in multiset B there are only lengths equal to 1 or 2, we must place a cut to get a fragment of length at most 2 after one unit on the right side without a cut. So, step (5) is correct as well as step (6) with a similar reasoning. The only exception, when we reach an existing cut, concerns the situation when we fill the whole current slot and jump to the next one (the slot closer to the middle of the solution).

Summing up the above reasoning, we may conclude that the algorithm is correct. As to its complexity, if the elements of A are sorted in step (2), the steps can be done in time, respectively, $O(n \log n)$, $O(n \log n)$, $O(n \log n)$, $O(\log n)$, $O(\log n)$, and $O(\log n)$. Steps (4)–(6) are performed up to n times, so the whole algorithm is of the complexity $O(n \log n)$. \square

Example 3 presents in detail the proposed algorithm.

Example 3 *Let our instance of problem Π_{SM12s} be: $A = \{1, 2, 3, 4, 5, 5, 6, 7, 7, 8, 8, 9, 10, 10, 11, 12, 13, 14\}$ and $B = \{1, 1, 1, 1, 1, 2, 2, 2, 2\}$. Thus, we have 9 restriction places within the solution of length 15. The complementary pairs of lengths in A are: $\{1, 14\}$, $\{2, 13\}$, $\{3, 12\}$, $\{4, 11\}$, $\{5, 10\}$, $\{5, 10\}$, $\{6, 9\}$, $\{7, 8\}$, and $\{7, 8\}$. To the initially empty solution we add all symmetric cuts determined by the instance and we get the sequence of fragments as in Figure 5.*

Figure 5: A symmetric part of an example solution of problem Π_{SM12s} .

After that $A = \{\{1, 14\}, \{2, 13\}, \{3, 12\}, \{4, 11\}, \{6, 9\}\}$. We add the remaining cuts starting from the

one nearest the left end of the solution, the result is shown in Figure 6.

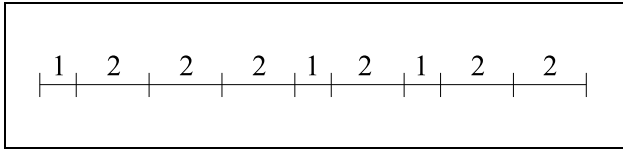


Figure 6: An example solution of problem Π_{SM12s} with the first slot filled by cuts.

Finally, we jump to the next slot and place the only remaining cut $\{6,9\}$. The whole solution is in Figure 7. \square

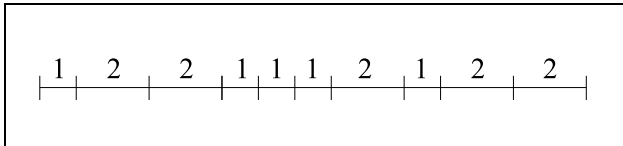


Figure 7: A complete solution for the example problem Π_{SM12s} .

4 Conclusions

In the paper, the DNA Simplified Partial Digest Problem was formulated and analyzed as a combinatorial search problem. The general error-free case was proved to be strongly NP-hard. On the other hand, a special form of the multiset resulting from the long digestion and composed of only 1s and 2s results in a simple, polynomial-time algorithm. The question remains open, whether or not, a similar approach can be used for the multisets composed of other restricted values (e.g. 1 and 3). An interesting problem is a question of approximability, i.e. an existence of polynomial-time algorithm constructing solutions being not far from the optimum one with respect to a suitably chosen optimality criterion.

Appendix

In the following, the original formulation of the Simplified Partial Digest Problem without errors in instances, from Blazewicz *et al.* (2001), is presented. We need some additional definitions.

$\mathcal{C} = \langle c_1, c_2, \dots, c_{2n} \rangle$ — multiset A sorted in non-decreasing order,

$L = c_i + c_{2n-i+1}$, for any i — the length of the analyzed strand,

$\mathcal{P}_i = \langle c_i, c_{2n-i+1} \rangle$, $i = 1..2n$ — an ordered pair of complementary fragments (i.e. summing up to L), where c_i is called the predecessor,

$\mathcal{Q} = \{q_1, q_2, \dots, q_n\}$ — a set of ordered pairs, where $q_i = \mathcal{P}_i$ or $q_i = \mathcal{P}_{2n-i+1}$, $i = 1..n$,

$\mathcal{O} = \langle o_1, o_2, \dots, o_n \rangle$ — the list of predecessors from every q_i , $i = 1..n$, sorted in non-decreasing order,

$\mathcal{R} = \{r_1, r_2, \dots, r_{n+1}\}$ — a multiset derived from \mathcal{O} according to the following rule: $r_1 = o_1$, $r_i = o_i - o_{i-1}$, $i = 2..n$, $r_{n+1} = L - o_n$.

Now, the problem is:

Problem 1' *Simplified Partial Digest Problem — search version.*

Instance: Multiset $A = \{a_1, a_2, \dots, a_{2n}\}$ of lengths of restriction fragments coming from short digestion and multiset $B = \{b_1, b_2, \dots, b_{n+1}\}$ of lengths of restriction fragments coming from long digestion, A and B containing no errors.

Answer: The set \mathcal{Q} such that the corresponding multiset \mathcal{R} is equal to B .

References

- Benzer, S. (1959), 'On the topology of the genetic fine structure', *Proceedings of the National Academy of Sciences of the USA* **45**, 1607–1620.
- Blazewicz, J., Formanowicz P., Kasprzak M., Jaroszewski M. & Markiewicz W.T. (2001), 'Construction of DNA restriction maps based on a simplified experiment', *Bioinformatics* **17**, 398–404.
- Blazewicz, J. & Jaroszewski M. (2003), 'New algorithm for the Simplified Partial Digest Problem', *Lecture Notes in Bioinformatics* **2812**, 95–110.
- Blazewicz, J. & Kasprzak M. (2003), 'Complexity of DNA sequencing by hybridization', *Theoretical Computer Science* **290**, 1459–1473.
- Cieliebak, M. & Eidenbenz S. (2004), 'Measurement errors make the partial digest problem NP-hard', *Lecture Notes in Computer Science* **2976**, 379–390.
- Cieliebak, M., Eidenbenz S. & Penna P. (2003), 'Noisy data make the partial digest problem NP-hard', *Lecture Notes in Bioinformatics* **2812**, 111–123.
- Fogel, G.B. & Corne D.W., eds. (2003), *Evolutionary Computations in Bioinformatics*, Morgan Kaufman: Boston.
- Garey, M.R. & Johnson D.S. (1979), *Computers and Intractability. A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company: San Francisco.
- Goldstein, L. & Waterman M.S. (1987), 'Mapping DNA by stochastic relaxation', *Advanced Applied Mathematics* **8**, 194–207.
- Johnson, D.S. (1985), 'The NP-completeness column: an ongoing guide', *Journal of Algorithms* **6**, 291–305.
- Karp, R.M. & Shamir R. (1998), Algorithms for optical mapping, in 'Proceedings of the Second International Conference on Computational Biology (RECOMB)', New York, 117–124.
- Newberg, L. & Naor D. (1993), 'A lower bound on the number of solutions of the probed partial digest problem', *Advanced Applied Mathematics* **14**, 172–183.
- Pandurangan, G. & Ramesh H. (2002), 'The restriction mapping problem revisited', *Journal of Computer and System Sciences* **65**, 526–544.
- Pevzner, P.A. (2000), *Computational Molecular Biology: An Algorithmic Approach*, MIT Press: Cambridge.
- Schmitt, W. & Waterman M.S. (1991), 'Multiple solutions of DNA restriction mapping problem', *Advanced Applied Mathematics* **12**, 412–427.

- Schwartz, D.C., Li X., Hernandez L.I., Ramnarain S.P., Huff E.J. & Wang Y.K. (1993), 'Ordered restriction maps of *Saccharomyces cerevisiae* chromosomes constructed by optical mapping', *Science* **262**, 110–114.
- Setubal, J. & Meidanis J. (1997), *Introduction to Computational Biology*, PWS Publishing Company: Boston.
- Skiena, S.S. (1997), Geometric reconstruction problems, in J.E. Goodman & J. O'Rourke, eds, 'Handbook of Discrete and Computational Geometry', CRC Press: Boca Raton, 481–490.
- Skiena, S.S., Smith W.D. & Lemke P. (1990), Reconstructing sets from interpoint distances, in 'Proceedings of Sixth ACM Symposium on Computational Geometry', 332–339.
- Skiena, S.S. & Sundaram G. (1994), 'A partial digest approach to restriction site mapping', *Bulletin of Mathematical Biology* **56**, 275–294.
- Waterman, M.S. (1995), *Introduction to Computational Biology. Maps, Sequences and Genomes*, Chapman and Hall: London.

On the Approximability of Maximum and Minimum Edge Clique Partition Problems

Anders Dessmark[†] Jesper Jansson[†] Andrzej Lingas[†] Eva-Marta Lundell[†]
Mia Persson^{*}

[†]Department of Computer Science, Lund University, 22100 Lund, Sweden.
Email: andrzej@cs.lth.se

^{*}School of Technology and Society, Malmö University College, 20506 Malmö, Sweden.
Email: mia@cs.lth.se

Abstract

We consider the following clustering problems: given a general undirected graph, partition its vertices into disjoint clusters such that each cluster forms a clique and the number of edges within the clusters is maximized (*Max-ECP*), or the number of edges between clusters is minimized (*Min-ECP*). These problems arise naturally in the DNA clone classification. We investigate the hardness of finding such partitions and provide approximation algorithms. Further, we show that greedy strategies yield constant factor approximations for graph classes for which maximum cliques can be found efficiently.

Keywords: Approximation algorithms, clique partition

1 Introduction

The *correlation clustering* problem has gained a lot of attention recently (Ailon, Charikar & Newman 2005, Bansal, Blum & Chawla 2004, Charikar, Guruswami & Wirth 2003, Demaine & Immorlica 2003, Emanuel & Fiat 2003, Swamy 2004); given a complete graph with edges labeled “+” (similar) or “-” (dissimilar), find a partition of the vertices into clusters that agrees as much as possible with the edge labels, i.e., that maximizes the *agreements* (the number of “+” edges inside clusters plus the number of “-” edges between cluster) or that minimizes the *disagreements* (the number of “-” edges inside clusters plus the number of “+” edges between clusters).

In this paper, we consider a special variant of the correlation clustering problem in which there are no negative edge labels. Instead, we omit an edge between two vertices of a dissimilar pair. Furthermore, we require an edge between each pair of vertices in a cluster, i.e., every cluster must form a clique. We consider the following two combinatorial optimization problems. The *maximum edge clique partition problem* (*Max-ECP* for short) aims to find a partition of the vertices into cliques such that the total number of edges within all cliques is maximized. The related minimization version of this problem, the *minimum edge clique partition problem* (*Min-ECP* for short), is defined analogously with the exception that the total number of edges between the cliques is minimized.

The *Max-ECP* and *Min-ECP* problems first have been considered in the setting of DNA clone classi-

fication (Figuerola, Goldstein, Jiang, Kurowski, Lingas & Persson 2005). In order to characterize cDNA and ribosomal DNA (rDNA) gene libraries, the powerful DNA array based method *oligonucleotide fingerprinting* is commonly used (see, e.g., (Drmanac, Stavropoulos, Labat, Vonau, Hauser, Soares & Drmanac 1996, Herwig, Poustka, Müller, Bull, Lehrach & O’Brien 1999, Valinsky, Della Vedova, Jiang & Borneman 2002, Valinsky, Della Vedova, Scupham, Alvey, Figuerola, Yin, Hartin, Chrobak, Crowley, Jiang & Borneman 2002)). A key step in this method is the cluster analysis, which aims to cluster together similar data, i.e., the *fingerprints*. The problem of clustering binarized fingerprint data such that the number of clusters is minimized was first studied and motivated in (Figuerola, Borneman & Jiang 2004). In (Figuerola, Goldstein, Jiang, Kurowski, Lingas & Persson 2005), Figuerola *et al.* propose new approaches of partitioning binarized fingerprints into disjoint clusters in order to maximize the number of pairs of similar fingerprints lying inside the clusters (equivalently, minimize the number of pairs of similar fingerprints lying in different clusters). These problems can hence be viewed as the *Max-ECP* and *Min-ECP* problems where the vertices are the binarized fingerprints and the edges between them indicate their similarity.

Related results

The well studied correlation clustering problem was first introduced for complete graphs by Bansal *et al.* (Bansal, Blum & Chawla 2004). It has applications in many areas (see, e.g., (Bansal, Blum & Chawla 2004, Demaine & Immorlica 2003)). As noted in (Bansal, Blum & Chawla 2004), the problem of maximizing agreements and minimizing disagreements are equivalent at optimality but differ from the point of view of approximation. In (Bansal, Blum & Chawla 2004), it was established that these problems are NP-hard for complete graphs, and a PTAS was given in the case of maximizing agreements, whereas a constant factor approximation is given in the case of minimizing disagreements. This constant factor approximation was later improved by Charikar *et al.* (Charikar, Guruswami & Wirth 2003) where a factor 4 approximation algorithm is given for complete graphs based on linear programming relaxation. The latter problem was also proved to be APX-hard.

The problems of maximizing agreements and minimizing disagreements were later generalized to include non-necessarily complete graphs with edge weights in (Charikar, Guruswami & Wirth 2003). A factor 0.7664 approximation algorithm based on the rounding of a semidefinite programming relaxation for the problem of maximizing agreements for general

weighted graphs was given in (Charikar, Guruswami & Wirth 2003), but this factor was later improved to 0.7666 by Swamy (Swamy 2004). As for the problem of minimizing disagreements, a factor $O(\log n)$ approximation algorithm for general weighted graphs was proposed (independently) in (Charikar, Guruswami & Wirth 2003), (Demaine & Immorlica 2003), and (Emanuel & Fiat 2003). Recently, Ailon *et al.* (Ailon, Charikar & Newman 2005) have provided a randomized expected 3-approximation algorithm for minimizing disagreements. In the case of weighted complete graphs, which satisfies probability constraints ($w_{ij}^+ + w_{ij}^- = 1$ for edge (i, j)) and triangle inequality constraints ($w_{ik}^- \leq w_{ij}^- + w_{jk}^-$) on the edges, they have provided a factor 2 approximation algorithm.

The APX-hardness of the unweighted version of *Min-ECP* has been established by Shamir *et al.* (Shamir, Sharan & Tsur 2002). They have also presented results in the case when a solution must contain exactly p clusters; *Min-ECP* is solvable in polynomial time for $p = 2$ but NP-complete for $p > 2$.

Our results

In this paper, we investigate the approximability of *Max-ECP* and *Min-ECP*. Specifically, we prove that *Max-ECP* on general, undirected graphs is hard to approximate within a factor of $n^{1-o(1)}$, unless $\text{NP} \subseteq \text{ZPTIME}(2^{(\log n)^{O(1)}})$. On the other hand, we give an n -approximation algorithm running in polynomial time for this problem. In the case of *Min-ECP* we provide a polynomial-time $O(\log n)$ -approximation algorithm for this problem on general, undirected graphs with non-negative weights. We also prove that this problem is NP-hard to approximate within $1 + \frac{1}{880} - \epsilon$, for any $\epsilon > 0$. We further consider the greedy heuristic and show that it yields a 2-approximation for both *Max-ECP* and *Min-ECP*, under the assumption that the largest clique can be determined in polynomial time. Thus, the greedy method could be applied in practice only to graph classes for which maximum cliques can be found efficiently, for instance chordal graphs, line graphs and circular-arc graphs (cf. (Figuerola, Borneman & Jiang 2004)). We also note that these bounds are actually tight. Figure 1 summarizes our contributions.

Problem	Lower Bound	Upper Bound
Max-ECP	$n^{1-o(1)}$	n
weightedMin-ECP	$1 + \frac{1}{880} - \epsilon$	$O(\log n)$
GreedyMax-ECP	2	2
GreedyMin-ECP	2	2

Figure 1: Summary of results.

Our paper is structured as follows. We give more formal definitions of *Max-ECP* and *Min-ECP* in Section 2. In Section 3, we provide a factor n approximation algorithm for *Max-ECP*. In Section 4, we give a lower bound on approximability of *Max-ECP*. In Section 5, we provide a polynomial-time $O(\log n)$ -approximation algorithm for the weighted version of *Min-ECP* and in section 6, we derive a lower bound on approximability of *Min-ECP*. Finally, in Section 7, we consider the greedy algorithm for *Max-ECP* and *Min-ECP* and prove that it yields a 2-approximation.

2 Preliminaries

The formal definition of *Max-ECP* and *Min-ECP* is as follows.

Definition 1 Let $G = (V, E)$ be an undirected graph and let $n = |V|$. The problem of *maximum edge clique partition* (*Max-ECP* for short) is to find a partition of V into disjoint subsets V_1, \dots, V_k such that for each $1 \leq i \leq k$, any two vertices in V_i share an edge and the total number of edges within the subsets V_1, \dots, V_k is maximized.

The problem of *minimum edge clique partition* (*Min-ECP* for short) is defined analogously to *Max-ECP* with the exception that the total number of edges between the subsets V_1, \dots, V_k is minimized.

Note that an exact solution to *Max-ECP* is an exact solution to *Min-ECP* and *vice versa*. The example shown in Figure 2 demonstrates two feasible solutions to *Max-ECP* and *Min-ECP*. As depicted in Figure 2(a), the total number of edges inside the clusters is 18, hence the solution to *Max-ECP* has a total cost of 18. On the contrary, the total number of edges outside the clusters in Figure 2(a) is 12, hence the solution to *Min-ECP* has a total cost of 12. The optimal clustering is depicted in Figure 2(b), with the total cost of 24 for *Max-ECP* and the total cost of 6 for *Min-ECP*.

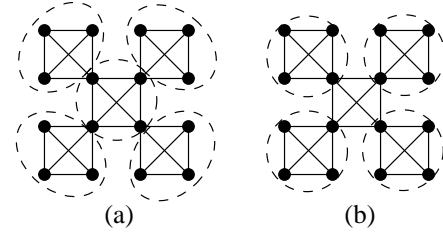


Figure 2: A feasible solution and the optimal solution to *Max-ECP* and *Min-ECP*.

3 A polynomial-time n -approximation algorithm for *Max-ECP*

Max-ECP is NP-hard and even hard to approximate within a factor $n^{1-O(1/(\log n)^\gamma)}$, for some constant γ , as proved in the next section. On the positive side, we prove in this section that *Max-ECP* admits a polynomial-time, factor k approximation algorithm, where k is the number of vertices in the largest clique. The approximation algorithm works as follows: Find a maximum matching in G and output it and the singletons containing the vertices not covered by the matching as a clique partition.

Theorem 1 Let k be the number of vertices in the largest clique in G . *Max-ECP* can be approximated within a factor of k in polynomial time.

Proof: Denote by $\text{OPT}(G)$ and $\text{APPR}(G)$ the total number of edges within cliques in an optimal solution for *Max-ECP* on G and in the solution returned by the approximation algorithm described above, respectively. Let (V_1, V_2, \dots, V_m) be an optimal solution for *Max-ECP* on G . There is a matching in G which, for $i = 1, \dots, m$, includes at least $\frac{|V_i|-1}{2}$ edges from the clique induced by V_i . Since for $i = 1, \dots, m$, $k \geq |V_i|$, such a matching includes at least the $\frac{1}{k}$ fraction of edges from each of the m cliques induced by V_1, V_2, \dots, V_m . Hence, $\text{APPR}(G) \geq \text{OPT}(G) / k$ holds.

4 A lower bound on the approximability of *Max-ECP*

The maximum clique problem is known to not admit an approximation within $n^{1-O(1/(\log n)^\gamma)}$ for some constant γ unless $\mathcal{NP} \subseteq \text{ZPTIME}(2^{(\log n)^{O(1)}})$ (Khot 2001). It follows that aforementioned lower bound on approximability holds for graphs on n vertices having a clique of size m not less than n^{1-x} , where $x = O(1/(\log n)^\gamma)$. Consider such a graph G . An optimal solution to *Max-ECP* for G has at least $\binom{m}{2}$ edges. Hence, an n^{1-3x} approximation to *Max-ECP* for G has at least $m(m-1)/(2n^{1-3x})$ edges. The size of maximum clique in the approximate solution to *Max-ECP* is minimized if all cliques have equal size h . In this case the total number of edges in the approximate solution is $\binom{h}{2}n/h$ which is less than $nh/2$. Hence, we obtain the inequality $m(m-1)/(2n^{1-3x}) \leq nh/2$ which by our assumptions on G and m yields $h = \Omega(n^x)$. This implies n^{1-x} approximation of the maximum clique problem in G which contradicts (Khot 2001). Thus, we obtain the following theorem.

Theorem 2 Unless $\mathcal{NP} \subseteq \text{ZPTIME}(2^{(\log n)^{O(1)}})$, the *Max-ECP* problem does not admit an $n^{1-O(1/(\log n)^\gamma)}$ approximation, for some constant γ .

5 A polynomial-time $O(\log n)$ -approximation algorithm for weighted *Min-ECP*

Min-ECP can be approximated within a factor of $O(\log n)$ in polynomial time, even for edge-weighted graphs with arbitrary non-negative weights, as follows.

Let $G = (V, E)$ be a given instance of *Min-ECP* in which each edge e has a non-negative weight $w(e)$. Define $W = \max_{e \in E} w(e)$. Construct an edge-weighted, edge-labeled, complete graph $G' = (V, E')$, where each $e \in E'$ is labeled by '+' and assigned weight $w(e)$ if $e \in E$, or labeled by '-' and assigned weight $W \cdot n^2 \log^2 n$ if $e \notin E$. Run any one of the polynomial-time $O(\log n)$ -approximation algorithms for Minimum Disagreement Correlation Clustering for weighted graphs (Charikar, Guruswami & Wirth 2003, Demaine & Immorlica 2003, Emanuel & Fiat 2003) on G' to obtain a clustering \mathcal{C}' for V , and return the set \mathcal{S} of subgraphs of G induced by \mathcal{C}' .

Lemma 1 For any two vertices $u, v \in V$ which are not joined by an edge in G , u and v do not belong to the same cluster in \mathcal{C}' .

Proof: Suppose u and v belong to the same cluster in \mathcal{C}' . Then the clustering obtained from \mathcal{C}' by placing u in a singleton cluster would have a disagreement score lower than that of \mathcal{C}' by a factor of $\omega(\log n)$, which is a contradiction.

By Lemma 1, the vertices from each cluster in \mathcal{C}' form a clique in G . Since the clusters in \mathcal{C}' are disjoint, \mathcal{S} is a partition of G into cliques, which proves the correctness of the method.

Next, we consider the approximation ratio. For any partition M of G into cliques, denote by $ECP(M)$ the ECP score for M , i.e., the sum of all weights of edges whose two endpoints belong to different cliques in M . Similarly, for any clustering M' of G' , let $Disagree(M')$ be the disagreement correlation clustering score for M' . Finally, $MinECP(G)$ and $MinDisagree(G')$ denote the minimum possible scores of ECP for G and $Disagree$ for G' , respectively.

Lemma 2 $ECP(\mathcal{S})$ is at most $O(\log n)$ times $MinECP(G)$.

Proof: Let M be a partition of G into cliques which minimizes ECP , and let M' be the clustering of G' induced by the cliques in M . Then, since only edges labeled by '+' contribute to $Disagree(M')$, we obtain $MinECP(G) = ECP(M) = Disagree(M') \geq MinDisagree(G')$.

Next, observe that $ECP(\mathcal{S})$ is equal to $Disagree(\mathcal{C}')$ because only edges labeled by '+' contribute to $Disagree(\mathcal{C}')$ by Lemma 1. Moreover, $Disagree(\mathcal{C}')$ is at most $O(\log n)$ times $MinDisagree(G')$. It follows that $ECP(\mathcal{S})$ is at most $O(\log n)$ times $MinECP(G)$.

To summarize:

Theorem 3 Weighted *Min-ECP* can be approximated within a factor of $O(\log n)$ in polynomial time.

6 A lower bound for *Min-ECP*

Shamir *et al.* have established the APX-hardness of unweighted *Min-ECP* by a reduction from a special variant of set cover in (Shamir, Sharan & Tsur 2002). It follows by (Shamir, Sharan & Tsur 2002) that the *Min-ECP* problem cannot have a polynomial-time approximation scheme unless $P=NP$. However, no explicit lower bound on the approximation factor for *Min-ECP* achievable in polynomial time is known in the literature.

In this section, we present a new reduction from the so called three way cut problem to the weighted *Min-ECP* problem which yields an explicit lower bound on the approximation factor.

The problem of three way cut (3WC) is to find a minimum number of edges whose removal disconnects three distinguished vertices.

Let A and B be two optimization problems (maximization or minimization). A linearly reduces to B if there are two polynomial time algorithms h and g , and constants $\alpha, \beta > 0$ such that

1. For an instance a of A , algorithm h produces an instance $b = h(a)$ of B such that the cost of an optimal solution for b , $opt(b)$, is at most $\alpha \cdot opt(a)$, and
2. For a , $b = h(a)$, and any solution y of b , algorithm g produces a solution x of a such that $|cost(x) - opt(a)| \leq \beta |cost(y) - opt(b)|$.

By (Dahlhaus, Johnson, Papadimitriou, Seymour & Yannakakis 1994), if A linearly reduces to B and B has a polynomial-time $1 + \epsilon$ approximation algorithm then A has a polynomial-time $(1 + \alpha\beta\epsilon)$ approximation algorithm.

Max-Cut is the problem of finding, for an undirected graph with vertex set V , a partition V_1, V_2 of V such that the number of edges $\{u, v\}$ where $\{u, v\} \cap V_1$ and $\{u, v\} \cap V_2$ are both nonempty is maximized.

In (Dahlhaus, Johnson, Papadimitriou, Seymour & Yannakakis 1994), Dahlhaus *et al.* presented a linear reduction of the Max-Cut problem to 3WC in order to prove that 3WC is APX-hard. Since Max-Cut is APX-hard (Håstad 2001), the APX-hardness of 3WC follows. In the aforementioned reduction $\alpha = 56$ and $\beta = 1$ (Dahlhaus, Johnson, Papadimitriou, Seymour & Yannakakis 1994). In fact, α can be decreased to 55 by the proof of Theorem 5 in (Dahlhaus, Johnson, Papadimitriou, Seymour & Yannakakis 1994)¹.

¹In the proof of Theorem 5 in (Dahlhaus, Johnson, Papadimitriou, Seymour & Yannakakis 1994), observe that $OPT_{3WC}(f(G)) = 56 \cdot \frac{|E|}{2} - K \leq 56 \cdot OPT_{Max-Cut}(G) - OPT_{Max-Cut}(G) = 55 \cdot OPT_{Max-Cut}(G)$

On the other hand, Håstad has shown that for any $\epsilon > 0$, it is NP-hard to approximate Max-Cut within $1 + \frac{1}{16} - \epsilon$ (Håstad 2001). Hence, we obtain the following lemma.

Lemma 3 *For any $\epsilon > 0$, it is NP-hard to approximate 3WC within $1 + \frac{1}{880} - \epsilon$.*

To reduce 3WC to weighted *Min-ECP*, fix an arbitrary $\delta > 0$, and transform any given instance of 3WC on n vertices to an instance of *Min-ECP* as follows:

1. Assign the weight 1 to each edge in the instance.
2. For each non-adjacent pair u, v of vertices in the instance insert an edge of weight δ/n^2 .
3. For each distinguished vertex s_i , $i = 1, 2, 3$, add an auxiliary vertex u_i and make it adjacent with each vertex of the instance. Assign the weight n^2 to each of the three edges (s_i, u_i) and the weight δ/n^2 to the remaining edges incident to the vertices u_i , $i = 1, 2, 3$.

Figure 3 demonstrates how the transformation from an instance of 3WC to an instance of *Min-ECP* works.

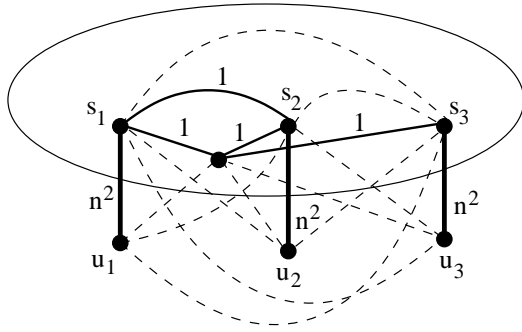


Figure 3: Transformation from 3WC to *Min-ECP*.

In this figure, note that a dashed line between a pair of vertices indicates an edge with weight δ/n^2 .

Note that in an optimal *Min-ECP* solution to the transformed instance each of the pairs s_i, u_i , $i = 1, 2, 3$ belongs to a separate clique and the total weight of the edges outside all the cliques in the optimal solution is between *cut* and *cut* + δ where *cut* stands for the value of an optimal solution to the instance of 3WC.

Suppose that for some $\epsilon > 0$, weighted *Min-ECP* could be approximated in polynomial time within a factor of f where $f \leq 1 + \frac{1}{880} - \epsilon$. Then using the set of edges between the three cliques in an approximate solution for weighted *Min-ECP* as an approximate solution for 3WC would yield a three-way cut for the original graph of cardinality at most $f \cdot (\text{cut} + \delta) \leq (f + f \cdot \delta) \cdot \text{cut}$. By setting $\delta = \frac{\epsilon}{2 \cdot (1 + \frac{1}{880} - \epsilon)}$, we could approximate 3WC in polynomial time within $1 + \frac{1}{880} - \epsilon/2$. We obtain a contradiction with Lemma 3. Hence, we obtain the following theorem.

Theorem 4 *For any $\epsilon > 0$, it is NP-hard to approximate weighted *Min-ECP* within $1 + \frac{1}{880} - \epsilon$.*

7 Greedy method for *Max-ECP* and *Min-ECP*

The greedy strategy applies naturally to the *Max-ECP* and *Min-ECP* problems: iteratively pick the

largest clique until all elements have been partitioned into disjoint clusters. However, the problem of finding a maximum clique is itself known to be extremely hard to approximate (Khot 2001). Thus, the greedy method could be applied in practice only to graph classes for which maximum cliques can be found efficiently (cf. (Figuerola, Borneman & Jiang 2004)).

Theorem 5 *The greedy method yields a 2-approximation for *Max-ECP* and *Min-ECP*.*

Proof: Consider an optimal solution to the *Max-ECP* problem (or, the *Min-ECP* problem, respectively) and let us assume that it consists of m cliques. Let C be the largest clique, say on k vertices, picked by the greedy method. Suppose first that the intersection of C with any clique in the optimal partition is a singleton or empty. Thus, in a way, the at most $k(k-1)$ former clique edges are replaced with the $k(k-1)/2$ edges in C (or, the $k(k-1)/2$ edges in C previously outside the cliques with at most $k(k-1)$ new edges outside the cliques, respectively). In the remaining case, if the intersection of C with any of the cliques in the optimal partition contains more than one vertex, less than $k(k-1)$ former clique edges are replaced by the $k(k-1)/2$ edges in C (or, the $k(k-1)/2$ edges in C previously outside the cliques are replaced by less than $k(k-1)$ new edges outside the cliques, respectively). By iterating the argument, we obtain the theorem.

The example shown in Figure 4 demonstrates that our upper bound on the approximation factor of the greedy method for *Max-ECP* is tight. Simply, the greedy method may produce n 2-cliques and $2n$ 1-cliques (singletons) yielding n edges whereas the optimal clique partition consists of $2n$ 2-cliques yielding $2n$ edges.

Figure 4 is also a tight example for greedy *Min-ECP*. Note that the number of edges between cliques will be $2n$ in the approximate solution, whereas the optimum contains n edges between the $2n$ 2-cliques.

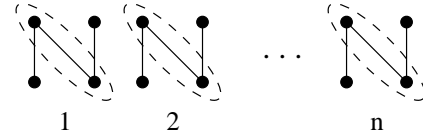


Figure 4: An example illustrating the worst-case performance of the greedy strategy for *Max-ECP* and *Min-ECP*.

8 Final remarks

By using rather maximum weight matching than maximum cardinality matching we can easily generalize our n -approximation method for *Max-ECP* to include edge weights.

It is an interesting open problem whether or not the gap between the upper and lower bounds on approximability of *Min-ECP* could be tightened.

A careful reader might observe that our approximation hardness result for *Max-ECP* does not hold for the graph classes for which our greedy method could be applied practically. The complexity and approximation status of *Max-ECP* and *Min-ECP* for the aforementioned graph classes are interesting open problems.

References

Ailon, N., Charikar, M. & Newman, A. (2005), Aggregating inconsistent information: Ranking and

- Clustering, in 'Proc. 37th Annual ACM Symposium on Theory of Computing (STOC 2005)', pp. 684–693.
- Bansal, N., Blum, A. & Chawla, S. (2004), 'Correlation Clustering', *Machine Learning* **56**(1–3), 89–113.
- Charikar, M., Guruswami, V. & Wirth, A. (2003), Clustering with Qualitative Information, in 'Proc. 44th Annual Symposium on Foundations of Computer Science (FOCS 2003)', pp. 524–533.
- Dahlhaus, E., Johnson, D.S., Papadimitriou, C.H., Seymour, P.D. & Yannakakis, M. (1994), 'The Complexity of Multiterminal Cuts', *SIAM J. Comput.* **23**, 864–894.
- Demaine, E. & Immorlica, N. (2003)Correlation Clustering with Partial Information, in 'Proc. 6th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX 2003)', pp. 1–13.
- Drmanac, S., Stavropoulos, N.A., Labat, I., Vonau, J., Hauser, B., Soares, M.B. & Drmanac, R. (1996), 'Gene-representing cDNA clusters defined by hybridization of 57,419 clones from infant brain libraries with short oligonucleotide probes', *Genomics* **37**, 29–40.
- Emanuel, D. & Fiat, A. (2003)Correlation Clustering – Minimizing Disagreements on Arbitrary Weighted Graphs, in 'Proc. 11th Annual European Symposium on Algorithms (ESA 2003)', pp. 208–220.
- Figueroa, A., Borneman, J. & Jiang, T. (2004), 'Clustering binary fingerprint vectors with missing values for DNA array data analysis', *Journal of Computational Biology* **11**(5), 887–901.
- Figueroa, A., Goldstein, A., Jiang, T., Kurowski, M., Lingas, A. & Persson, M. (2005)Approximate Clustering of Fingerprint Vectors with Missing Values, in 'Proc. 11th Computing: The Australasian Theory Symposium (CATS 2005)', pp. 57–60.
- Herwig, R., Poustka, A.J., Müller, C., Bull, C., Lehrach, H. & O'Brien, J. (1999), 'Large-scale clustering of cDNA-fingerprinting data', *Genome research* **9**, 1093–1105.
- Håstad, J. (2001), 'Some optimal inapproximability results', *Journal of the ACM* **48**(4), 798–859.
- Khot, S. (2001), Improved inapproximability results for MaxClique, chromatic number, and approximate graph coloring, in 'Proc. 42th Annual Symposium on Foundations of Computer Science (FOCS 2001)', pp. 600–609.
- Shamir, R., Sharan, R. & Tsur, D. (2002)Cluster Graph Modification Problems, in 'Proc. 28th International Workshop on Graph Theoretic Concepts in Computer Science (WG 2002)', pp. 379–390.
- Swamy, C. (2004)Correlation Clustering: maximizing agreements via semidefinite programming, in 'Proc. 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2004)', pp. 526–527.
- Valinsky, L., Della Vedova, G., Jiang, T. & Borneman, J. (2002), 'Oligonucleotide fingerprinting of ribosomal RNA genes for analysis of fungal community composition', *Applied and Environmental Microbiology* **68**, 5999–6004.
- Valinsky, L., Della Vedova, G., Scupham, A., Alvey, S., Figueroa, A., Yin, B., Hartin, R., Chrobak, M., Crowley, D., Jiang, T. & Borneman, J. (2002), 'Analysis of bacterial community composition by oligonucleotide fingerprinting of rRNA genes', *Applied and Environmental Microbiology* **68**, 3243–3250.

Faster Algorithms for Finding Missing Patterns

Shuai Cheng Li

School of Computer Science
University of Waterloo
Waterloo ON N2L 3G1 Canada
Email: scl@cs.uwaterloo.ca

Abstract

The missing pattern pair problem, introduced in (Inenaga, Kivioja & Mäkinen 2004), was motivated by the need for optimization in Polymerase Chain Reaction, a technique commonly used in bioinformatics. The problem is to find a pair of patterns of the shortest total length within a string of length n , where the two patterns do not occur within a distance α anywhere in the string. Inenaga *et al.* (Inenaga et al. 2004) gave an algorithm with time complexity $O(\min\{\alpha n \log n, n^2\})$ to solve this problem. In this paper we propose an algorithm of time complexity $O(\min\{\alpha n \log n, n^{3/2}\})$, improving on the quadratic bound part of the earlier algorithm. We also design a simple algorithm of time complexity $O(\frac{n^2}{\alpha} \log^2 n)$, which is $O(n \log^2 n)$ if $\alpha = \Theta(n)$.

Keywords: pattern discovery, complexity, algorithm.

1 Introduction

Pattern discovery problems are among the most intensively studied problems in bioinformatics (Wang, Shapiro & Shasha 1999). An example of such problems is that of finding a pattern which does not appear in a given string — this is known as the *missing pattern problem*. This problem can be solved in time $O(n)$ where n is the length of the given string.

Inenaga *et al.* (Inenaga et al. 2004) introduced the problem called *missing pattern pair* (MPP) problem, where we are to find a pair of patterns of the shortest total length which do not appear in a given string S within a predefined distance α . An algorithm of time complexity $O(\min\{\alpha n \log n, n^2\})$ was given in (Inenaga et al. 2004). The problem has practical use in optimizing the sensitivity of Polymerase Chain Reaction methods — a standard technique for producing many copies of a region of DNA. In this paper we give an algorithm of $O(\min\{\alpha n \log n, n^{3/2}\})$ runtime, which should improve performance for the cases where α is large.

In the following subsection, a brief review of the biological motivation of this problem is presented, for a detailed version, please refer to (Inenaga et al. 2004).

1.1 Biological Motivations

Polymerase Chain Reaction (PCR) is used routinely to producing multiple copies of a sub sequence of

DNA. Primers in PCR refer a pair of short sequence. The two primers hybridize to their binding side of a target sequence, and this flanking the target sequence and makes the duplication of the flanked area possible. One of the problem is that the primers can bind to other sites rather than the targeting sites, and result in incorrect flanking. In order to overcome this problem, *Adapter primers* is designed. For specific primers, adapter primes bind short sequence to them. It is argued in (Inenaga et al. 2004) that the PCR process will be facilitated by identifying a *shortest missing pair* as the adapter primers.

This paper is organized as follows: in Section 2 we give the definitions and problem reviews. In Section 3 we propose a new method for solving the missing pattern problem in time $O(n)$, then we present how the method can be extended to solve the missing pattern pair problem in time $O(\min\{\alpha n \log n, n^{3/2}\})$ in Section 4. In Section 5, an approach with running time $O(\frac{n^2}{\alpha} \log^2 n)$ is proposed, which has a complexity of $O(n \log^2 n)$ when $\alpha = \Theta(n)$. Section 6 concludes this paper.

2 Preliminaries

We try to follow as much as possible notations from other literature. The symbol N denotes the set of natural numbers. The symbol N^+ denotes the set of positive natural numbers.

Let Σ be a finite alphabet of size σ (σ is assumed to be a constant in (Inenaga et al. 2004) and in this paper¹). A *word* or a *pattern* is a string of symbols over Σ , where the latter is more typically used to refer to a substring of some (longer) word. The length of a word S is denoted as $|S|$. The character at position i of a string S is written $S[i]$ (the starting position being 0); while the substring of S from position i to position j ($i \leq j$) is written as $S[i : j]$.

A pattern P of length k is said to *occur* at position j of a string S if and only if $j + k - 1 < n$ and $P = S[j : j + k - 1]$. The set of all the positions that a pattern P occurs in a string S is denoted $Occ(P, S)$. For example, $Occ(AA, "AAATGCTAA") = \{0, 1, 7\}$. A pattern P is a *single missing pattern* (SMP) w.r.t a string S over Σ if it does not occur at any position of S , that is $Occ(P, S) = \emptyset$.

2.1 Sequence of k -mers

A k -mer is a word of length k . For any $k \in N^+$, we let Σ^k denote the set of all the words over Σ of length k , that is, all the k -mers.

Given a string S and $k \in N^+$, we can construct a sequence of $|S| - k + 1$ k -mers by the positions which they occur, namely, $S[0 : k - 1], S[1 :$

Copyright copyright 2006, Australian Computer Society, Inc. This paper appeared at Computing: The Australasian Theory Symposium (CATS2006), Hobart, Australia. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 51. Barry Jay and Joachim Gudmundsson, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.

¹ Σ is assumed to be $\{A, C, G, T\}$ in the original application of missing pattern.

$k], \dots, S[n - k, n - 1]$ (k -mers in the sequence may not be distinct). For any string S and $k \in N^+$, we let \mathcal{SEQ}_S^k denote this corresponding sequence of k -mers. The i -th k -mer in the sequence \mathcal{SEQ}_S^k is written $\mathcal{SEQ}_S^k[i]$ (the first k -mer being $\mathcal{SEQ}_S^k[0]$). The subsequence of \mathcal{SEQ}_S^k from position i to j is written as $\mathcal{SEQ}_S^k[i : j]$.

We let each character in Σ be represented by a number from 0 to $\sigma - 1$. Each k -mer, say s , can then be written as a string of numbers $i_0 i_1 \dots i_{k-1}$ where each number i_j in the string has a value from 0 to $\sigma - 1$ which represents the character at $s[j]$. For example, the 8-mer “AAATATGG” may be written as “00020211” (or simply 20211), where 0 represents A, 1 represents G, and 2 represents T. In this way, for any fixed k , there is a 1-1 mapping from Σ^k to the natural numbers from 0 to $\sigma^k - 1$. This representation of k -mers is useful as indices to arrays or lookup tables. This coding is simplified version of the hashing idea from (Karp & Rabin 1987), here we use a 1-1 mapping, as σ is assumed to be a constant.

Unless stated otherwise, for any $k \in N^+$ and string S , we assume members in Σ^k and \mathcal{SEQ}_S^k to be of this natural number form. Assuming that k is small ($k \leq \log_\sigma n + 1$, the same assumption is made in (Inenaga et al. 2004) and a suffix tree is used, in which each integer has $O(\log n)$ bits), each k -mer $v \in \Sigma^k$ uses only $O(1)$ space. We shall now show that given any S and k , the computation of \mathcal{SEQ}_S^k in such a representation can be performed very efficiently.

Given any S and k , we first compute $\mathcal{SEQ}_S^k[0]$, then for any $i > 0$, $\mathcal{SEQ}_S^k[i]$ can be computed with the value of the k -mer at position $i - 1$ by the formula:

$$\begin{aligned} \mathcal{SEQ}_S^k[i] &= \sigma(\mathcal{SEQ}_S^k[i - 1] - \lfloor \frac{\mathcal{SEQ}_S^k[i - 1]}{\sigma^{k-1}} \rfloor \sigma^{k-1}) \\ &\quad + S[i + k - 1] \end{aligned}$$

Thus we have the following result.

Lemma 1 For any string S and $k \in N^+$, \mathcal{SEQ}_S^k can be computed in $O(|S|)$ time.

Similarly, \mathcal{SEQ}_S^{k-1} can be efficiently computed from \mathcal{SEQ}_S^k , by the following.

$$\mathcal{SEQ}_S^{k-1}[i - 1] = \lfloor \mathcal{SEQ}_S^k[i - 1] / \sigma \rfloor \quad (1)$$

We let $\text{content}(\mathcal{SEQ}_S^k[i : j])$ denote the set of all k -mers which appears in the subsequence of k -mers $\mathcal{SEQ}_S^k[i : j]$. For simplicity, $\text{content}(\mathcal{SEQ}_S^k[0 : |\mathcal{SEQ}_S^k| - 1])$ is written as $\text{content}(\mathcal{SEQ}_S^k)$. Let $\mathcal{SEQ}_S^k(i, d) = \mathcal{SEQ}_S^k[A : B]$ where $A = \max\{0, i - d + 1\}$ and $B = \min\{|S| - 1, i + d - 1\}$. Intuitively, $\mathcal{SEQ}_S^k(i, d)$ are the k -mers of up to a distance d from the position i . $\text{content}(\mathcal{SEQ}_S^k(i, d))$ are the distinct k -mers of up to a distance d from the position i .

Lemma 2 $v \in \text{content}(\mathcal{SEQ}_S^{k-1})$ if and only if there exists $v' \in \text{content}(\mathcal{SEQ}_S^k)$ such that $v = \lfloor v' / \sigma \rfloor$, or $v = \mathcal{SEQ}_S^{k-1}[|S| - k + 1]$.

Lemma 2 and Equation 1 together show that $\text{content}(\mathcal{SEQ}_S^{k-1})$ can be computed completely from $\text{content}(\mathcal{SEQ}_S^k)$ in time $O(|\text{content}(\mathcal{SEQ}_S^k)|)$.

3 Shortest Missing Pattern Problem

The shortest missing pattern problem, proposed in (Inenaga et al. 2004), is to find the shortest single missing pattern (SMP) w.r.t. a given string S . (Inenaga et al. 2004) has proposed a solution based on suffix trees which runs in time $O(|S|)$ and space $O(|S|)$. It is well known that the suffix tree has a large overhead and is difficult to implement. We propose an alternative here which uses the same order of time and space, but finds all the shortest SMPs w.r.t. to a string S and easy to implement. Below we reproduce a Lemma from (Inenaga et al. 2004) which we need to show our result.

Lemma 3 For any string S , there exists an SMP of length $\lceil \log_\sigma(|S| + 1) \rceil$ w.r.t. S .

Proof. For any k , there are a total of σ^k possible k -mers (that is, $|\Sigma^k| = \sigma^k$). A string S , on the other hand, has at most distinct $|S| - k + 1$ k -mers. If $\sigma^k > |S| - k + 1$, then there is a k -mer which does not occur in S . Hence for any $k \geq \lceil \log_\sigma(|S| + 1) \rceil$, there exists a k -mer which does not occur in S . ■

Below we list the algorithm which finds the shortest SMPs. The algorithm conducts an exhaustive search of all the k -mers, with decreasing values of k , beginning from $\lceil \log_\sigma(|S| + 1) \rceil$. The output is a number k of the shortest length of the missing patterns, and an array of bits \mathcal{B} where for each $v \in \Sigma^k$, $\mathcal{B}[v] = 1$ if and only if v is a SMP.

Algorithm 1: Find all shortest SMPs

1. Let $l = \lceil \log_\sigma(|S| + 1) \rceil$.
2. Compute $\text{content}(\mathcal{SEQ}_S^l)$.
3. For $k = l$ to 1,
4. Allocate an array \mathcal{B} of σ^k bits, initializing each bit to 0.
5. For each $v \in \text{content}(\mathcal{SEQ}_S^k)$, set $\mathcal{B}[v]$ to 1.
6. (Note that elements in $\text{content}(\mathcal{SEQ}_S^k)$ range from 0 to $\sigma^k - 1$.)
7. If all the bits in \mathcal{B} are set to 1 (in which case there is no missing pattern of length k or below), output $k + 1$ and \mathcal{B}' (i.e. SMPs found in the previous iteration). (Note: by Lemma 3 there is at least one SMP at iteration $k = l$.)
9. Let $\mathcal{B}' = \mathcal{B}$, and compute $\text{content}(\mathcal{SEQ}_S^{k-1})$ from $\text{content}(\mathcal{SEQ}_S^k)$ (using Equation 1 and Lemma 2).
10. End

Since $\text{content}(\mathcal{SEQ}_S^{k-1})$, \mathcal{B} and \mathcal{B}' are of size σ^k for the k -th iteration, the space requirement is $O(\sigma^l) = O(|S|)$. By Lemma 1, line-2 can be computed in time $O(|S|)$. There are a total of at most l iterations for the loop at line-3, where each iteration takes time $O(\sigma^k)$. Thus the total time is $O(\sum_{k=1}^l \sigma^k)$. Since $\sum_{k=1}^l \sigma^k = \sigma^l + \sum_{k=1}^{l-1} \sigma^k = \sigma^l + \frac{\sigma^l - 1}{\sigma - 1} \leq 2\sigma^l$, the time complexity is $O(\sigma^l) = O(|S|)$.

Theorem 4 Given any string S , Algorithm 1 finds all the shortest SMP w.r.t S in $O(|S|)$ time, using $O(|S|)$ space.

4 Missing Pattern Pair Problem

A missing pattern pair (MPP) P_1 and P_2 with threshold α (written $\langle P_1, P_2 \rangle_\alpha$) w.r.t. a string S is a pair of patterns where:

1. either P_1 or P_2 is an SMP w.r.t S ; or
2. both P_1 and P_2 occur in S , and $\forall p_1 \in \text{Occ}(P_1, S)$, $\forall p_2 \in \text{Occ}(P_2, S)$, $|p_1 - p_2| > \alpha$. That is, no occurrences of P_1 occur within a distance of α from P_2 (and vice versa).

Our aim is to find a missing pattern pair with the shortest total length.

MISSING PATTERN PAIR (MPP) PROBLEM
(INENAGA ET AL. 2004)

Input: String S and $\alpha \in N^+$.
Output: An MPP $\langle P_1, P_2 \rangle_\alpha$ w.r.t. S with minimal $|P_1| + |P_2|$.

An algorithm for the MMP Problem with time complexity $O(\min\{\alpha n \log n, n^2\})$ was given in (Inenaga et al. 2004).

4.1 Preliminaries

We first introduce the *Dynamic Perfect Hashing* data structure which will be used in our algorithm.

The *Dynamic Perfect Hashing* (Dietzfelbinger, Karlin, Mehlhorn & Der 1994) is a data structure which manages a dictionary (a set of key-data pair) with $O(1)$ amortized runtime cost in the following operations: *insert*(k), *delete*(k), and *getdata*(k), where k is the key used in the operation. Its space requirement is linearly proportional to the number of elements managed. We write $k \in \mathcal{H}$ just in case k is the key for a key-data pair in \mathcal{H} ; $\mathcal{H}[k]$ denotes the data part of the key-data pair in \mathcal{H} with key k .

4.2 Finding Missing Pattern Pair

This is our strategy for solving the MMP problem. We first run Algorithm 1 on the input string S . If it returns an SMP of length ℓ (note that by Lemma 3 Algorithm 1 must return some $\ell \in N^+$), we know:

1. any shortest MPP $\langle P_1, P_2 \rangle_\alpha$ must have $|P_1| + |P_2| \leq \ell$.
2. for any shortest MPP $\langle P_1, P_2 \rangle_\alpha$, if $|P_1| + |P_2| < \ell$, then both P_1 and P_2 occurs in S .

Based on the output of Algorithm 1, we then exhaustively search for all MPPs $\langle P_1, P_2 \rangle_\alpha$ of total length ℓ , $\ell - 1, \dots, 1$.

We first give a subroutine (Algorithm 2) that looks for MPPs $\langle P_1, P_2 \rangle$ where the lengths of P_1 and P_2 are fixed to, say, l_1 and l_2 respectively. Without loss of generality we let $l_1 \geq l_2$. By the argument above we assume that P_1 and P_2 both appear in the given string.

The subroutine first computes the sequences of k -mers $\mathcal{SEQ}_S^{l_1}$ and $\mathcal{SEQ}_S^{l_2}$. It then computes, as output, a $\sigma^{l_1} \times \sigma^{l_2}$ matrix \mathcal{BB}_{l_1, l_2} where for all $u \in \Sigma^{=l_1}$ and $v \in \Sigma^{=l_2}$, $\mathcal{BB}_{l_1, l_2}[u][v] = 1$ iff there exists some $i, j \in N$ such that $\mathcal{SEQ}_S^{l_1}[i] = u$ and $\mathcal{SEQ}_S^{l_2}[j] = v$ and $|i - j| \leq \alpha$. That is, $\langle u, v \rangle_\alpha$ is an MPP just in case $\mathcal{BB}_{l_1, l_2}[u][v] = 1$. For notation simplicity $\mathcal{BB}_{l_1, l_2}[u]$ refers to the array entries $\mathcal{BB}_{l_1, l_2}[u][v]$ with $0 \leq v \leq \sigma^{l_2} - 1$. When it is clear from the context, \mathcal{BB}_{l_1, l_2} is written as \mathcal{BB} .

Algorithm 2: Find all MPPs $\langle P_1, P_2 \rangle$

where $|P_1| = l_1$ and $|P_2| = l_2$

1. Compute $\mathcal{SEQ}_S^{l_1}$, $\mathcal{SEQ}_S^{l_2}$, $\text{content}(\mathcal{SEQ}_S^{l_2}(0, \alpha))$.
2. Allocate a $\sigma^{l_1} \times \sigma^{l_2}$ 1-bit matrix \mathcal{BB} , initialize each bit in \mathcal{BB} to 0.

3. Prepare dictionary \mathcal{H} .
4. For each $u \in \text{content}(\mathcal{SEQ}_S^{l_2}(0, \alpha))$
5. Let $\mathcal{H}[u]$ be the number of occurrences of u in $\mathcal{SEQ}_S^{l_2}(0, \alpha)$.
6. For $i = 0, 1, \dots, |\mathcal{SEQ}_S^{l_1}| - 1$,
7. Let $u = \mathcal{SEQ}_S^{l_1}[i]$,
8. For each key $v \in \mathcal{H}$
(Note: $v \in \mathcal{H} \Rightarrow v \in \text{content}(\mathcal{SEQ}_S^{l_2}(i, \alpha))$).
9. Let $\mathcal{BB}[u][v] = 1$.
10. The following updates \mathcal{H} so that $v \in \mathcal{H} \Rightarrow v \in \text{content}(\mathcal{SEQ}_S^{l_2}(i + 1, \alpha))$.
11. If $i - \alpha \geq 0$
12. Let $v = \mathcal{SEQ}_S^{l_2}[i - \alpha]$.
13. Decrease $\mathcal{H}[v]$ by 1.
If $\mathcal{H}[v] = 0$ remove v from \mathcal{H} .
14. If $i + \alpha + 1 < |\mathcal{SEQ}_S^{l_1}|$
15. Let $v = \mathcal{SEQ}_S^{l_2}[i + \alpha + 1]$.
16. If $v \notin \mathcal{H}$, let $\mathcal{H}[v] = 1$,
else increment $\mathcal{H}[v]$ by 1.

In Algorithm 2, line 1 runs in time $O(|S|)$; while line 2–3 takes $O(|\mathcal{BB}|)$ time, that is, $O(\sigma^{l_1} \cdot \sigma^{l_2}) = O(\sigma^\ell)$. Line 4–5 runs in $O(\alpha)$ time. There a total of $|S|$ iterations for the loop at line 6. At each iteration i , \mathcal{H} contains at most $\min\{2\alpha + 1, \sigma^{l_2}\}$ distinct entries. Thus line 8–9 runs in $O(\min\{2\alpha + 1, \sigma^{l_2}\})$ per iteration. Line 10–16 runs in constant time for each iteration. Algorithm 2 hence runs in time $O(|S| \min\{2\alpha + 1, \sigma^{l_2}\})$.

Lemma 5 Given string S , $l_1, l_2 \in N^+$ where $l_1 \geq l_2$, Algorithm 2 takes time $O(\min\{|S|\sigma^{l_2}, |S|\alpha\})$ to find all the MPPs $\langle P_1, P_2 \rangle_\alpha$ w.r.t. S where $|P_1| = l_1$ and $|P_2| = l_2$.

4.3 Identify Missing Pattern Pair with Minimum Length

Firstly the following lemma can be deduced:

Lemma 6 Given \mathcal{BB}_{l_1, l_2} , $\mathcal{BB}_{l_1, l_2-1}$ can be computed with time $O(\sigma^{l_1+l_2} + \alpha)$.

Proof: There are two cases that $\mathcal{BB}_{l_1, l_2-1}[u][v] = 1$:

1. There exists v' such that $\mathcal{BB}_{l_1, l_2}[u][v'] = 1$ and with $v = \lfloor v' / \sigma \rfloor$
2. $u \in \text{content}(\mathcal{SEQ}_S^{l_1}(n - l_2 + 1, \alpha))$, and $v = \mathcal{SEQ}_S^{l_2-1}[n - l_2 + 1]$.

Totally there are $\sigma^{l_1+l_2}$ entries in \mathcal{BB}_{l_1, l_2} and we just need to scan through all the entries of \mathcal{BB}_{l_1, l_2} (which takes time cost σ^l) and to obtain $\text{content}(\mathcal{SEQ}_S^{l_1}, n - l' + 1, \alpha)$ (which takes time $O(\alpha)$) to compute $\mathcal{BB}_{l_1, l_2-1}$. ■

With these, we are now ready to wrap everything to obtain a better algorithm. Firstly we use the algorithm for Section 3.1 to identify ℓ . Then we iterate through all the possible combinations of (l_1, l_2) ($l_1 + l_2 \leq \ell$) to obtain all the shortest MPPs.

As the two cases for (l_1, l_2) and (l_2, l_1) are symmetric, we just need to search the cases where $l_1 \geq l_2$. Denote $\delta(l) = \min\{l_1, \ell - l_1\}$. The pseudo code is presented in Algorithm 3. For each l_1 , the algorithm will search through all the possible values l_2 (while avoiding the symmetric cases) by using the result from Lemma 6 to avoid recomputation. Algorithm 2 will be employed for the combination $(l_1, \delta(l_1))$ (given l_1 , the largest possible value for l_2 is $\delta(l_1)$).

Algorithm 3: Find the Shortest MPPs

1. Identify the shortest missing pattern length ℓ with Algorithm 1.
2. Compute $\text{content}(\mathcal{SEQ}_S^\ell)$.
3. For $l_1 = \ell - 1$ to 1
4. Compute $\mathcal{BB}_{l_1, \delta(l_1)}$ with Algorithm 2.
5. For $l_2 = \delta(l_1) - 1$ to 1.
6. Compute \mathcal{BB}_{l_1, l_2} with Lemma 6
7. Record if there is a length $l_1 + l_2$ missing pattern is found.

The time cost is dominated by line 3-7 for Algorithm 3. For each value of l_1 , line 4 takes time $O(|S| \min\{\sigma^{\delta(l_1)}, \alpha\})$, and line 5-6 takes time $O(\sum_{l_2=1}^{\delta(l_1)-1} (\sigma^{l_2} + \alpha))$.

$$\begin{aligned} \sum_{l_2=1}^{\delta(l_1)-1} (\sigma^{l_2} + \alpha) &\leq \sigma^{\delta(l_2)} \frac{1}{\sigma - 1} + (\delta(l_1) - 1)\alpha \\ &= O(\sigma^{\delta(l_1)} + (\delta(l_1) - 1)\alpha) \end{aligned}$$

As $l_1 < \ell = O(\log n)$, which means $\sigma^{\delta(l_1)} = O(|S|)$, also we know that $\delta(l_1) = O(\sigma^{\delta(l_1)})$ and $\alpha \leq |S|$. Combine all these in, we have:

$$\begin{aligned} |S| \min\{\sigma^{\delta(l_1)}, \alpha\} + \sigma^{\delta(l_1)} + (\delta(l_1) - 1)\alpha \\ &= O(|S| \min\{\sigma^{\delta(l_1)}, \alpha\} + |S| + \delta(l_1)\alpha) \\ &= O(\min\{|S| \sigma^{\delta(l_1)} + \delta(l_1)\alpha, |S|\alpha + \delta(l_1)\alpha\}) \\ &= O(|S| \min\{\sigma^{\delta(l_1)}, \alpha\}) \end{aligned}$$

Thus for each iteration of the outer loop, line 4-6 takes time $O(|S| \min\{\sigma^{\delta(l_1)}, \alpha\})$. Lastly, sum up the terms over the possible l_1 values, we have:

$$\begin{aligned} \sum_{l_1 \leq \ell} |S| \min\{\sigma^{\delta(l_1)}, \alpha\} &\leq |S| \min\left\{\sum_{l_1 \leq \ell} \sigma^{\delta(l_1)}, \sum_{l_1 \leq \ell} \alpha\right\} \\ &\leq |S| \min\left\{\sum_{l_1 \leq \ell} \sigma^{\min\{l_1, \ell - l_1\}}, \ell\alpha\right\} \\ &\leq |S| \min\left\{\sum_{l_1 \leq \lceil \frac{\ell}{2} \rceil} 2\sigma^{l_1}, \ell\alpha\right\} \\ &= O(|S| \min\{\sigma^{\lceil \frac{\ell}{2} \rceil}, \ell\alpha\}) \\ &= O(|S| \min\{\sqrt{|S|}, \alpha \log n\}) \end{aligned}$$

For the space complexity, it is clear that it is $O(|S| + \Delta)$, where Δ represents the output size. If we just want to identify one shortest MPP, the space complexity is $O(|S|)$. Formally, the time and space complexity are concluded in Theorem 7.

Theorem 7 *The missing pattern pair problem with a given string S of length n and a threshold α can be solved with time complexity $O(\min\{n^{3/2}, n\alpha \log n\})$ and space complexity $O(n)$ with Algorithm 3.*

5 Faster Algorithm with Large α

It may be noticed if we can replace Algorithm 2 in Algorithm 3 with a faster subroutine, Algorithm 3 will result in less running time. In this section, a faster procedure for large α is presented. A simple data structure named *Range Union Tree* is defined to serve the purpose of this algorithm.

5.1 Range Union Tree

For k -mer sequence \mathcal{SEQ}_S^k , a $\text{report}(i, j)$ query reports the set of distinct elements of the subsequence $\mathcal{SEQ}_S^k[i : j]$, that is $\text{content}(\mathcal{SEQ}_S^k[i : j])$. At the first glance, an array representation of \mathcal{SEQ}_S^k will be good enough to handle the queries. However this approach will not be efficient when the report queries are numerous and there are high duplications for the elements of \mathcal{SEQ}_S^k . To serve the usage of this paper, a balanced binary tree is adopted to represent \mathcal{SEQ}_S^k . An example is illustrated in Figure 1. Each element of \mathcal{SEQ}_S^k is assigned to a leaf node. For each internal node v , we store an ordered list of the distinct integers contains in the leaf node of the subtree which is rooted at v . The smallest and largest indices in \mathcal{SEQ}_S^k under each subtree is also maintained at each internal node respectively. The space usage for this tree is $O(n \log n)$ since each level of the tree requires space $O(n)$. To construct the tree, time cost $O(n \log n)$ is enough by a bottom-up manner (which is similar as the merge sort). For a report query, it is easy to see that we just need to union $O(\log n)$ ordered lists, which can be accomplished with time complexity $O(\sigma^k \log n)$, as σ^k is the upper bound of result list size. This tree is referred as the *Range Union Tree* (RUT) in this paper.

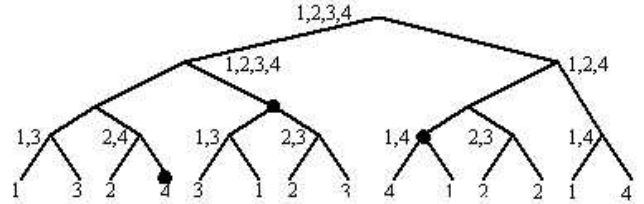


Figure 1: A RUT for sequence $\mathcal{SEQ}_S^k = \{1, 3, 2, 4, 3, 1, 2, 3, 4, 1, 3, 2, 1, 4\}$. To answer the query $\text{report}(3, 9)$ the element between $\mathcal{SEQ}_S^k[3]$ to $\mathcal{SEQ}_S^k[9]$, we just need to compute the union of these sets associated with the shaded nodes.

5.2 An Algorithm Based on RUT

Denote $\mathcal{SEQ}_S^l(Q, \alpha) = \bigcup_{i \in Q} \mathcal{SEQ}_S^l(i, \alpha)$. Let $Q_u^l = \text{Occ}(u, \mathcal{SEQ}_S^l)$. The task to compute $\mathcal{BB}[u]$ is essentially equivalent to compute $\text{content}(\mathcal{SEQ}_S^l(Q_u^l, \alpha))$. $\mathcal{SEQ}_S^l(Q_u^l, \alpha)$ consists a set of disjoint intervals (an interval here means a consecutive subsequence of indices eg. 1, 2, 3, 4, 7, 8, 9 is considered as two maximal disjointed intervals [1, 4], [7, 9]). The set of maximal disjointed intervals can be stored as an ordered list for each u and can be computed for all the u values simultaneously with time cost of $O(n)$ by scanning \mathcal{SEQ}_S^l once. An array \mathcal{A} with size σ^{l_1} with each entry indexed with u , $0 \leq u \leq \sigma^{l_1} - 1$ may be employed. The array entry at position u records the last interval's end index for u for the current scanning. While scanning the l_1 -mer with $u = \mathcal{SEQ}_S^{l_1}[i]$, by comparing $\max\{0, i - \alpha\}$ with $\mathcal{A}[i]$, we will know whether a new interval should be open, or the last interval should be just extended for u .

To compute $\text{content}(\mathcal{SEQ}_S^l(Q_u^l, \alpha))$ (we extend the content notation to a set of intervals), the set of intervals contained in $\mathcal{SEQ}_S^l(Q_u^l, \alpha)$ can be computed first, then with the RUT, the content of each interval can be obtained. Lastly the union of the content of the intervals can be identified. The number of

disjointed intervals for each u is bounded by $|S|/\alpha + c$ (from some constant c). For each u , we need query the RUT at most $|S|/\alpha + c$ times. Each query will cost time $O(\sigma^{l_2} \log n)$ and the time cost for each u , $0 \leq u \leq \sigma^{l_1}$ is $O(|S|/\alpha \sigma^{l_2} \log n)$. Thus this approach for computing \mathcal{BB} will result in an method with time complexity $O(\sigma^{l_1} |S|/\alpha \sigma^{l_2} \log n) = O(\frac{n^2}{\alpha} \log n)$.

Substitute it into Algorithm 3, we have:

Theorem 8 *Based on RUT, the missing pattern pair problem can be solved with time complexity of $O(\frac{n^2}{\alpha} \log^2 n)$*

6 Conclusion

In this paper, we proposed two deterministic algorithms for the missing pattern problem and have improved the bound for the MPP problem from $O(n^2)$ to $O(n^{3/2})$. Also we have demonstrated with a faster subroutine for a MPP for given lengths, a faster algorithm can be obtained under our framework.

References

- Dietzfelbinger, M., Karlin, A., Mehlhorn, K. & Der, F. M. (1994), 'Dynamic perfect hashing: Upper and lower bounds', *SIAM J. Comput.* **23**(4), 738–761.
- Inenaga, S., Kivioja, T. & Mäkinen, V. (2004), Finding missing patterns, in 'WABI', pp. 463–474.
- Karp, R. M. & Rabin, M. O. (1987), 'Efficient randomized pattern-matching algorithms', *IBM J. Res. Dev.* **31**(2), 249–260.
- Wang, J. T.-L., Shapiro, B. A. & Shasha, D., eds (1999), *Pattern Discovery in Biomolecular Data: Tools, Techniques and Applications*, Oxford University Press.

On the logical Implication of Multivalued Dependencies with Null Values

Sebastian Link[†]

Department of Information Systems, Information Science Research Centre
Massey University, Palmerston North, New Zealand
E-mail: S.Link@massey.ac.nz

Abstract

The implication of multivalued dependencies (MVDs) in relational databases has originally been defined in the context of some fixed finite universe (Fagin 1977, Zaniolo 1976). While axiomatisability, implication problem and many design problems have been intensely studied with respect to this notion, almost no research has been devoted towards the alternative notion of implication in which the underlying universe of attributes is left undetermined (Biskup 1980).

A milestone in the advancement of database systems was the permission of null values in databases. In particular, many achievements on MVDs have been extended to encompass incomplete information. Multivalued dependencies with null values (NMVDs) were defined and axiomatised in (Lien 1982). The definition of NMVDs is again based on a fixed underlying universe of attributes, and any complete set of inference rules requires therefore some version of the complementation rule.

In this paper we show that the axiomatisation in (Lien 1982) does not reflect the fact that the complementation rule is merely a means to achieve database normalisation. Moreover, we provide an alternative axiomatisation for NMVDs that does reflect this property. We also suggest an alternative notion for the implication of NMVDs in which the underlying universe is left undetermined, and propose several sound and complete sets of inference rules for this notion. Moreover, a correspondence between (minimal) axiomatisations in fixed universes that do reflect the property of complementation and (minimal) axiomatisations in undetermined universes is shown.

Keywords: Database Theory, Multivalued Dependency, Null Values, Implication, Axiomatisation

1 Introduction

Relational databases still form the core of most database management systems, even after more than three decades following their introduction in (Codd 1970). The relational model organises data into a collection of relations. These structures permit the storage of inconsistent data, inconsistent in the semantic

sense. Since this is not acceptable additional assertions, called dependencies, are formulated that every database is compelled to obey. There are many different classes of dependencies which can be utilised for improving the representation of the target database. Excellent surveys on relational dependencies can be found in (Fagin & Vardi 1986, Thalheim 1991).

Multivalued dependencies (MVDs) (Delobel 1978, Fagin 1977, Zaniolo 1976) are an important class of dependencies. A relation exhibits an MVD precisely when it is decomposable into two of its projections without loss of information (Fagin 1977). This property is fundamental to relational database design, in particular 4NF (Fagin 1977), and a lot of research has therefore been devoted to studying the behaviour of these dependencies. Recently, extensions of multivalued dependencies have been found very useful for various design problems in advanced data models such as the nested relational data model (Fischer, Saxton, Thomas & Van Gucht 1985), the Entity-Relationship model (Thalheim 2003), data models that support nested lists (Hartmann & Link 2004) and XML (Vincent & Liu 2003, Vincent, Liu & Liu 2003).

It is very rare in practice that the information in a database is complete. This observation has led to many extensions of the relational data model (Codd 1979, Lien 1982, Atzeni & Morfuni 1986, Levene & Loizou 1993, Levene & Loizou 1998, Johnson & Rosebrugh 2003) that can handle incomplete information. In particular, multivalued dependencies in the presence of null values (NMVDs) have been studied in (Lien 1982). The notion of an NMVD from (Lien 1982), as well as the original notion of an MVD (Fagin 1977), is dependent on the underlying set R of attributes. This dependence is reflected syntactically by the R -complementation rule which is part of the axiomatisation of NMVDs, see (Lien 1982). The complementation rule is special in the sense that it is the only inference rule which is dependent on R . In the absence of null values, this observation has led to further research (Mendelzon 1979, Biskup 1978, Biskup 1980, Hartmann & Link 2006, Link 2006) on the complementation rule. In particular, Biskup introduced an alternative notion of semantic implication in which the underlying universe is left undetermined (Biskup 1980). In the same paper it was shown that this notion can be captured syntactically by a sound and complete set of inference rules, denoted by \mathfrak{S}_0 . If $R\mathfrak{S}_0$ results from \mathfrak{S}_0 by adding the R -complementation rule, then $R\mathfrak{S}_0$ is R -sound and R -complete for the R -implication of MVDs. Moreover, every inference of an MVD by $R\mathfrak{S}_0$ can be turned into an inference of the same MVD in which the R -complementation rule is applied at most once, and if it is applied, then in the last step of the inference ($R\mathfrak{S}_0$ is said to be complementary). This indicates that the R -complementation rule simply re-

Copyright ©2006, Australian Computer Society, Inc. This paper appeared at Twelfth Computing: The Australasian Theory Symposium (CATS), Hobart, Tasmania. Conferences in Research and Practice in Information Technology, Vol. 51. Barry Jay and Joachim Gudmundsson, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

[†] Sebastian Link is supported by Marsden Funding, Royal Society of New Zealand

flects a part of the decomposition process, and is not necessarily essential for deriving valid consequences. Interestingly, this research has not been extended to encompass incomplete information, i.e., to NMVDs. Since research on (N)MVDs seems to experience a recent revival in the context of other data models (Fischer et al. 1985, Thalheim 2003, Hartmann & Link 2004, Vincent & Liu 2003, Vincent et al. 2003) it seems desirable to further extend the knowledge on (N)MVDs. An advancement of such knowledge may simplify the quest of finding suitable and comprehensible extensions of MVDs to currently popular data models.

In this paper we will extend the alternative notion of implication from MVDs (Biskup 1980) to the presence of null values. First, it is demonstrated that the sound and complete set $R\mathfrak{R}$ of inference rules for the R -implication of NMVDs from (Lien 1982) is not complementary. Moreover, we propose a sound and complete set $R\mathfrak{L}$ that is indeed complementary for the R -implication of NMVDs. Subsequently, we will identify a sound and complete set \mathfrak{L} of inference rules for the implication of NMVDs in undetermined universes. Thus, \mathfrak{L} does not permit the possibly semantically meaningless inference of complementation. Finally, the set \mathfrak{L} is extended to obtain a sound and complete set of inference rules for the implication of functional and multivalued dependencies in the presence of null values in undetermined universes. The problems studied in this paper are not just of theoretical interest. In practice one does not necessarily want to generate all consequences of a given set of NMVDs but only some of them. Such a task can be accomplished by using incomplete sets of inference rules. However, it is then essential to explore the power of such incomplete sets.

The paper is structured as follows. Section 2 repeats fundamental notions from the relational model of data as well as incomplete information. In particular, the notions for implication of multivalued dependencies in the presence of null values are highlighted. After the axiomatisation of NMVDs from (Lien 1982) has been reviewed Section 3 identifies R -sound and R -complete sets of inference rules for the R -implication of NMVDs that are also R -complementary. Furthermore, the alternative notion of implication for NMVDs in which universes are left undetermined is introduced, and a sound and complete set \mathfrak{L} of inference rules proposed. The result is extended to capture both functional and multivalued dependencies in the presence of nulls in undetermined universes. Some general results are proposed in Section 4 which show a correspondence between (minimal) complete sets of inference rules in undetermined universes and (minimal) complete and complementary sets of inference rules in fixed universes. Section 5 suggests some alternative axiomatisations for NMVDs in undetermined universes that only require weak versions of some of the inference rules in \mathfrak{L} . The paper concludes in Section 6.

2 MVDs in Relational Databases

We use this section to introduce some notation, and repeat notions and results for dependencies in the presence of null values.

2.1 Partial Relations

Let $\mathfrak{A} = \{A_1, A_2, \dots\}$ be a (countably) infinite set of attributes. A *relation schema* is a finite set $R = \{A_1, \dots, A_n\}$ of distinct symbols, called *attributes*, which represent column names of a relation. Each

attribute A_i of a relation schema is associated an infinite domain $dom(A_i)$ which represents the set of possible values that can occur in the column named A_i . In particular, it is assumed that every attribute may have a null value, denoted by $\nu \in dom(A_i)$. It may be noted that many kinds of null values have been proposed; for example, “missing” or “value unknown at present” (Codd 1975, Grant 1977, Grahne 1984), “non-existence” (Mikinouchi 1977), “inapplicable” (Grant 1977), “no information” (Zaniolo 1984) and “open” (Gottlob & Zicari 1988). The intention of the null value ν is to mean “undefined”, “inapplicable”, or “nonexistent”. For instance, the *maiden name* of a male *employee* may have a null value to mean inapplicable, or the *middle name* of an *employee* may have a null value to mean nonexistent.

If X and Y are sets of attributes, then we may write XY for $X \cup Y$. If $X = \{A_1, \dots, A_m\}$, then we may write $A_1 \dots A_m$ for X . In particular, we may write simply A to represent the singleton $\{A\}$. A *tuple* over $R = \{A_1, \dots, A_n\}$ (R -tuple or simply tuple, if R is understood) is a function $t : R \rightarrow \bigcup_{i=1}^n dom(A_i)$

with $t(A_i) \in dom(A_i)$ for $i = 1, \dots, n$. For $X \subseteq R$ let $t[X]$ denote the restriction of the tuple t over R on X , and $dom(X) = \prod_{A \in X} dom(A)$ the Cartesian product of the domains of attributes in X . A *relation* r over R is a finite set of tuples over R . The relation schema R is also called the domain $Dom(r)$ of the relation r over R . Suppose that t_1, t_2 are two tuples in the relation r over R . It is said that t_1 *subsumes* t_2 if for every attribute $A \in R$, either $t_1[A] = t_2[A]$ or $t_2[A] = \nu$ holds. For the remainder of this article, the following restriction will be imposed on the relations in a database: No relation in the database shall contain two tuples t_1 and t_2 such that t_1 subsumes t_2 . When no null value is present, this restriction amounts to saying that no two tuples are identical, an explicit requirement for database relations.

In order to contrast relations with and without null values, several terms are introduced. A relation r over R is said to be a *total relation* if it contains no null values. That is, if for any tuple $t \in r$ and any attribute $A \in R$, $t[A] \neq \nu$. If r is not a total relation, it is a *partial relation* or simply *relation*. For a tuple $t \in R$ and a set $X \subseteq R$, t is said to be X -total if for any $A \in X$, $t[A] \neq \nu$.

There are several operations on partial relations that are natural generalisations of their counterparts from total relations. These include projection and natural join. Let r be some relation over R . Let X be some attribute set of R . The *projection* of r on X , denoted by $r[X]$, is a set of tuples t for which (i) there is some $t_1 \in r$ such that $t = t_1[X]$ and (ii) there is no $t_2 \in r$ such that $t_2[X]$ subsumes t and $t_2[X] \neq t$. Let Y be some attribute set of R with $Y \subseteq X$. The *Y -total projection* of r on X , denoted by $r_Y[X]$, is the set $r_Y[X] = \{t \in r[X] \mid t \text{ is } Y\text{-total}\}$. Given an X -total relation r over R and an X -total relation s over S such that $X = R \cap S$ the *natural join* of r and s , denoted by $r \bowtie s$, is the relation over $R \cup S$ which contains exactly those tuples t such that there is some $t_1 \in r$ and some $t_2 \in s$ with $t_1 = t[R]$ and $t_2 = t[S]$.

2.2 Dependencies in the Presence of Nulls

Functional dependencies (FDs) between sets of attributes have always played a central role in the study of relational databases (Codd 1970, Codd 1972, Beeri & Bernstein 1979, Bernstein 1976, Bernstein & Goodman 1980), and seem to be central for the study of database design in other data models as well (Arenas & Libkin 2004, Hara & Davidson 1999, Hart-

mann & Link 2004, Levene & Loizou 1998, Tari, Stokes & Spaccapietra 1997, Weddell 1992, Wijzen 1999). The notion of a functional dependency is well-understood and the semantic interaction between these dependencies has been syntactically captured by Armstrong's well-known axioms (Armstrong 1974, Armstrong, Nakamura & Rudnicki 2002).

Let R be a relation schema. A *functional dependency with nulls* on R , abbreviated NFD, is a statement $X \rightarrow Y$ where $X, Y \subseteq R$. The NFD $X \rightarrow Y$ on R is satisfied by a partial relation r over R , denoted by $\models_r X \rightarrow Y$, if and only if for all $t_1, t_2 \in r$ the following holds: if t_1 and t_2 are X -total and $t_1[X] = t_2[X]$, then $t_1[Y] = t_2[Y]$. Therefore, whenever two tuples agree on a nonnull X -value, they agree on the Y -value, which may be partial.

FDs are incapable of modelling many important properties that database users have in mind. Multivalued dependencies (MVDs, (Delobel 1978, Fagin 1977, Zaniolo 1976)) provide a more general notion and offer a response to the shortcomings of FDs. MVDs in the presence of null values have been introduced in (Lien 1982).

A *multivalued dependency with nulls* on the relation schema R , abbreviated NMVD, is an expression $X \twoheadrightarrow Y$ where $X, Y \subseteq R$. A partial relation r over R satisfies the NMVD $X \twoheadrightarrow Y$ on R , denoted by $\models_r X \twoheadrightarrow Y$, if and only if for all $t_1, t_2 \in r$ the following holds: if t_1 and t_2 are X -total and $t_1[X] = t_2[X]$, then there is some $t \in r$ such that $t[XY] = t_1[XY]$ and $t[X(R - Y)] = t_2[X(R - Y)]$. Informally, the partial relation r satisfies $X \twoheadrightarrow Y$ when the total X -values determine the set of values on Y independently from the set of values on $R - Y$. This actually suggests that the relation schema R is overloaded in the sense that it carries two independent facts XY and $X(R - Y)$. More precisely, it is shown in (Lien 1982) that NMVDs provide a necessary and sufficient condition for a X -total relation to be decomposable into two of its projections without loss of information (in the sense that the original X -total relation is guaranteed to be the natural join of the two projections). This means that $\models_r X \twoheadrightarrow Y$ if and only if $r_X[R] = r_X[XY] \bowtie r_X[X(R - Y)]$. This characteristic of NMVDs is fundamental to database design and a lot of research has therefore been devoted to studying the behaviour of these dependencies. Recently, extensions of multivalued dependencies have been found very useful for various design problems in advanced data models such as the nested relational data model (Fischer et al. 1985), the Entity-Relationship model (Thalheim 2003), data models that support nested lists (Hartmann & Link 2004) and XML (Vincent & Liu 2003, Vincent et al. 2003).

For the design of a relational database schema dependencies are normally specified as semantic constraints on the relations which are intended to be instances of the schema. During the design process one usually needs to determine further dependencies which are logically implied by the given ones. In order to emphasise the dependence of implication from the underlying relation schema R we refer to R -implication.

Definition 2.1. Let R be a relation schema, and let $\Sigma = \{X_1 \twoheadrightarrow Y_1, \dots, X_k \twoheadrightarrow Y_k\}$ and $X \twoheadrightarrow Y$ be NMVDs on R , i.e., $X \cup Y \cup \bigcup_{i=1}^k (X_i \cup Y_i) \subseteq R$. Then Σ R -implies $X \twoheadrightarrow Y$ if and only if each partial relation r over R that satisfies all NMVDs in Σ also satisfies $X \twoheadrightarrow Y$. \square

In order to determine all logical consequences of a finite set of NMVDs one can use the following set of inference rules which was proposed in (Lien 1982).

Note that we use the natural complementation rule (Biskup 1978) instead of the complementation rule that was originally proposed (Lien 1982).

$$\begin{array}{ll}
 \frac{}{X \twoheadrightarrow Y} Y \subseteq X & \frac{X \twoheadrightarrow Y}{XU \twoheadrightarrow YV} V \subseteq U \\
 (\text{reflexivity, } \mathcal{R}) & (\text{augmentation, } \mathcal{A}) \\
 \\
 \frac{X \twoheadrightarrow Y}{X \twoheadrightarrow R - Y} & \frac{X \twoheadrightarrow Y, X \twoheadrightarrow Z}{X \twoheadrightarrow YZ} \\
 (R\text{-complementation, } \mathcal{C}_R) & (\text{union, } \mathcal{U}) \\
 \\
 \frac{X \twoheadrightarrow Y, X \twoheadrightarrow Z}{X \twoheadrightarrow Z - Y} & \frac{X \twoheadrightarrow Y, X \twoheadrightarrow Z}{X \twoheadrightarrow Y \cap Z} \\
 (\text{difference, } \mathcal{D}) & (\text{intersection, } \mathcal{I})
 \end{array}$$

In (Lien 1982) the set $R\mathfrak{K} = \{\mathcal{R}, \mathcal{A}, \mathcal{U}, \mathcal{C}_R\}$ of inference rules is proven to be both R -sound and R -complete for the R -implication of MVDs, on all relation schemata R . Let $\Sigma \cup \{\sigma\}$ be a set of NMVDs on the relation schema R . Let $\Sigma \vdash_{\mathfrak{K}} \sigma$ denote the inference of σ from a set Σ of NMVDs with respect to the set \mathfrak{S} of inference rules. Let $\Sigma_{\mathfrak{S}}^+ = \{\sigma \mid \Sigma \vdash_{\mathfrak{S}} \sigma\}$ denote the *syntactic hull* of Σ under inference using only rules from \mathfrak{S} . The set $R\mathfrak{S}$ is called *R -sound* for the R -implication of NMVDs iff for every set Σ of NMVDs on R we have $\Sigma_{R\mathfrak{S}}^+ \subseteq \Sigma_R^* = \{\sigma \mid \Sigma \text{ } R\text{-implies } \sigma\}$. The set $R\mathfrak{S}$ is called *R -complete* for the implication of NMVDs if and only if for every set Σ of NMVDs on R we have $\Sigma_R^* \subseteq \Sigma_{R\mathfrak{S}}^+$. The set $R\mathfrak{S}$ is called *sound* (complete) for the R -implication of NMVDs iff it is R -sound (R -complete) for the R -implication of NMVDs for all relation schemata R .

An interesting question is now whether all the rules of a certain set are really necessary to capture the R -implication of NMVDs for every R . More precisely, an inference rule \mathfrak{R} is said to be *R -independent* from the set $R\mathfrak{S}$ if and only if there is some set $\Sigma \cup \{\sigma\}$ of NMVDs on the relation schema R such that $\sigma \notin \Sigma_{R\mathfrak{S}}^+$, but $\sigma \in \Sigma_{R\mathfrak{S} \cup \{\mathfrak{R}\}}^+$. Moreover, the inference rule \mathfrak{R} is said to be *independent* from $R\mathfrak{S}$ if and only if there is some relation schema R such that \mathfrak{R} is R -independent from $R\mathfrak{S}$. Finally, a complete set $R\mathfrak{S}$ is said to be *minimal* for the R -implication of NMVDs if and only if every inference rule $\mathfrak{R} \in R\mathfrak{S}$ is independent from $R\mathfrak{S} - \{\mathfrak{R}\}$. This means that no proper subset of $R\mathfrak{S}$ is still complete.

Apart from $R\mathfrak{K}$ the following sets are also complete for the R -implication of NMVDs. This fact was not noticed in (Lien 1982), but is not difficult to see.

Theorem 2.1. The sets $R\mathfrak{K}_1 = \{\mathcal{R}, \mathcal{A}, \mathcal{I}, \mathcal{C}_R\}$ and $R\mathfrak{K}_2 = \{\mathcal{R}, \mathcal{A}, \mathcal{D}, \mathcal{C}_R\}$ are sound and complete for the R -implication of NMVDs.

Proof. The union rule \mathcal{U} is derivable from $\{\mathcal{I}, \mathcal{C}_R\}$

$$\frac{\frac{X \twoheadrightarrow Y}{X \twoheadrightarrow R - Y} \quad \frac{X \twoheadrightarrow Z}{X \twoheadrightarrow R - Z}}{X \twoheadrightarrow (R - Y) \cap (R - Z)} \\
 \frac{X \twoheadrightarrow (R - Y) \cap (R - Z)}{X \twoheadrightarrow R - ((R - Y) \cap (R - Z))} \\
 = Y \cup Z$$

and the intersection rule \mathcal{I} is derivable from $\{\mathcal{D}\}$

$$\frac{\frac{X \twoheadrightarrow Y \quad X \twoheadrightarrow Z}{X \twoheadrightarrow Z - Y} \quad X \twoheadrightarrow Z}{X \twoheadrightarrow Z - (Z - Y)} \\
 = Y \cap Z$$

\square

Note that $\frac{X \twoheadrightarrow Y, Y \twoheadrightarrow Z}{X \twoheadrightarrow Z - Y}$, the *pseudo-transitivity* rule \mathcal{T} , which is essential for MVDs (Beeri, Fagin & Howard 1977), is not R -sound in the presence of null values (Lien 1982).

2.3 NMVDs in Undetermined Universes

Consider a slight modification of the classical example (Fagin 1977) in which the NMVD $Employee \twoheadrightarrow Child$ is specified, i.e., the set of children is completely determined by an employee, independently from the rest of the information in any schema. If the relation schema R consists of the attributes $Employee$, $Child$ and $Salary$, then we may infer the NMVD $Employee \twoheadrightarrow Salary$ by means of the R -complementation rule. However, if the underlying relation schema R consists of the four attributes $Employee$, $Child$, $Salary$ and $Year$, then the NMVD $Employee \twoheadrightarrow Salary$ is no longer R -implied. Note the fundamental difference between the NMVDs

$Employee \twoheadrightarrow Child$ and $Employee \twoheadrightarrow Salary$.

The first NMVD has been specified to establish the relationship of employees and their children as a fact due to a set-valued correspondence. The second NMVD does not necessarily correspond to any semantic information, but simply results from the context in which $Employee$ and $Child$ are considered. If the context changes, the NMVD disappears. We can therefore observe the following:

- If consequences of NMVDs are inferred by a set $R\mathfrak{S}$ of inference rules with respect to a fixed universe R , applications of the R -complementation rule during any inference by $R\mathfrak{S}$ should be either completely avoided or limited to the very last step of the inference.
- It may be argued that consequences which are dependent on the underlying universe are in fact no consequences at all. This implies, however, that the notion of R -implication is not suitable.

We follow the first observation first, and come back to the second observation later. One may ask whether the R -sound and R -complete set $R\mathfrak{K} = \{\mathcal{R}, \mathcal{A}, \mathcal{U}, \mathcal{C}_R\}$ of inference rules reflects the property of R -complementation. More precisely, a complete set $R\mathfrak{S}$ of inference rules for the R -implication of (N)MVDs is said to be *complementary* iff it is R -complementary for all relation schemata R , i.e., for each $X \twoheadrightarrow Y \in \Sigma_{R\mathfrak{S}}^+$ there is an inference of $X \twoheadrightarrow Y$ from Σ by $R\mathfrak{S}$ in which the R -complementation rule \mathcal{C}_R is applied at most once and (if at all) as the last rule. The following example shows that the set $R\mathfrak{K}$ is not complementary.

Example 2.1. Let $\Sigma = \{A \twoheadrightarrow BC, A \twoheadrightarrow B\}$. The following table represents the syntactic hull $\Sigma_{\{\mathcal{R}, \mathcal{A}, \mathcal{U}\}}^+$ of Σ under inferences using $\{\mathcal{R}, \mathcal{A}, \mathcal{U}\}$. The NMVD $U \twoheadrightarrow V$ belongs to $\Sigma_{\{\mathcal{R}, \mathcal{A}, \mathcal{U}\}}^+$ iff the entry at row labelled U and column labelled V is a cross \times .

	\emptyset	A	B	C	AB	AC	BC	ABC
\emptyset	\times							
A	\times	\times	\times		\times		\times	\times
B	\times		\times					
C	\times			\times				
AB	\times	\times	\times		\times		\times	\times
AC	\times	\times	\times	\times	\times	\times	\times	\times
BC	\times		\times	\times			\times	
ABC	\times	\times	\times	\times	\times	\times	\times	\times

It shows in particular that $A \twoheadrightarrow C \notin \Sigma_{\{\mathcal{R}, \mathcal{A}, \mathcal{U}\}}^+$. Moreover, Lemma 3.1 shows that $A \twoheadrightarrow Y \notin \Sigma_{\{\mathcal{R}, \mathcal{A}, \mathcal{U}\}}^+$ for all Y such that $Y - \{A, B, C\} \neq \emptyset$. However, for $R = \{A, B, C, D\}$ we have $A \twoheadrightarrow C \in \Sigma_{R\mathfrak{K}}^+$, say by

$$\frac{\frac{A \twoheadrightarrow BC}{A \twoheadrightarrow AD} \quad A \twoheadrightarrow B}{A \twoheadrightarrow ABD} \quad .$$

Hence, in any such inference the rule \mathcal{C}_R must be used at least once, but since $R - \{C\} = \{A, B, D\}$ the R -complementation rule \mathcal{C}_R is not only used as the last rule. \square

Biskup introduced the following sound inference rules for the R -implication of MVDs

$$\frac{}{\emptyset \twoheadrightarrow \emptyset} \quad \frac{X \twoheadrightarrow Y, Y \twoheadrightarrow Z}{X \twoheadrightarrow YZ} \quad (\text{empty-set-axiom}, \mathcal{R}_\emptyset) \quad (\text{additive transitivity}, \mathcal{T}^*)$$

$$\frac{X \twoheadrightarrow Y, W \twoheadrightarrow Z}{X \twoheadrightarrow Y \cap Z} \quad Y \cap W = \emptyset \quad (\text{subset}, \mathcal{S})$$

and showed that the complete set $R\mathfrak{B} = \{\mathcal{R}_\emptyset, \mathcal{A}, \mathcal{T}, \mathcal{T}^*, \mathcal{S}, \mathcal{C}_R\}$ is also complementary. That is,

$$\begin{aligned} X \twoheadrightarrow Y \in \Sigma_{R\mathfrak{B}}^+ \\ \text{if and only if} \\ X \twoheadrightarrow Y \in \Sigma_{\mathfrak{B}}^+ \text{ or } X \twoheadrightarrow (R - Y) \in \Sigma_{\mathfrak{B}}^+ \end{aligned} \quad (2.1)$$

where $\Sigma = \{X_1 \twoheadrightarrow Y_1, \dots, X_k \twoheadrightarrow Y_k\}$ and $X \cup Y \cup \bigcup_{i=1}^k (X_i \cup Y_i) \subseteq R$.

Moreover, Biskup introduced an alternative notion of implication for MVDs (Biskup 1980), which leaves the underlying relation schema undetermined. This brings us back to our second observation from above. We will now generalise the notion of MVDs in undetermined universes to the presence of null values. An NMVD is a syntactic expression $X \twoheadrightarrow Y$ with $X, Y \subseteq \mathfrak{A}$. The NMVD $X \twoheadrightarrow Y$ is satisfied by some partial relation r if and only if $X \cup Y \subseteq \text{Dom}(r)$ and $r_X[\text{Dom}(r)] = r_X[XY] \bowtie r_X[X \cup (\text{Dom}(r) - Y)]$.

Definition 2.2. The set $\Sigma = \{X_1 \twoheadrightarrow Y_1, \dots, X_k \twoheadrightarrow Y_k\}$ of NMVDs implies the single NMVD $X \twoheadrightarrow Y$ if and only if for each partial relation r with $X \cup Y \cup \bigcup_{i=1}^k (X_i \cup Y_i) \subseteq \text{Dom}(r)$ the NMVD $X \twoheadrightarrow Y$ is satisfied by r whenever r already satisfies all NMVDs in Σ . \square

In this definition, the underlying relation schema is left undetermined. The only requirement is that the NMVDs must apply to the partial relations. The following fact is immediate and generalises a result from (Biskup 1980).

Theorem 2.2. Let $\Sigma = \{X_1 \twoheadrightarrow Y_1, \dots, X_k \twoheadrightarrow Y_k\}$ be a set of NMVDs, and $X \cup Y \cup \bigcup_{i=1}^k (X_i \cup Y_i) \subseteq R$. If Σ implies $X \twoheadrightarrow Y$, then Σ R -implies $X \twoheadrightarrow Y$. \square

The following example shows that the converse of Theorem 2.2 is false.

Example 2.2. For $R = \{Employee, Child, Salary\}$ and $\Sigma = \{Employee \twoheadrightarrow Child\}$ we have that Σ R -implies $Employee \twoheadrightarrow Salary$. However, Σ does not imply $Employee \twoheadrightarrow Salary$. Consider for instance the following partial relation r with domain $\{Employee, Child, Salary, Year\}$.

Employee	Child	Salary	Year
Don Juan	ν	4000	2004
Don Juan	ν	5000	2005

The two relations $r_{Employee}[Employee, Child]$

Employee	Child
Don Juan	ν

and $r_{Employee}[Employee, Salary, Year]$

Employee	Salary	Year
Don Juan	4000	2004
Don Juan	5000	2005

show that r satisfies the NMVD $Employee \twoheadrightarrow Child$. However, the two relations $r_{Employee}[Employee, Salary]$

Employee	Salary
Don Juan	4000
Don Juan	5000

and $r_{Employee}[Employee, Child, Year]$

Employee	Child	Year
Don Juan	ν	2004
Don Juan	ν	2005

indicate that r does not satisfy $Employee \twoheadrightarrow Salary$. Consequently, Σ does not imply $Employee \twoheadrightarrow Salary$. \square

A set \mathfrak{S} of inference rules is called *sound* for the implication of (N)MVDs if and only if for every finite set Σ of (N)MVDs we have $\Sigma_{\mathfrak{S}}^+ \subseteq \Sigma^* = \{\sigma \mid \Sigma \text{ implies } \sigma\}$. The set \mathfrak{S} is called *complete* for the implication of (N)MVDs if and only if for every finite set Σ of (N)MVDs we have $\Sigma^* \subseteq \Sigma_{\mathfrak{S}}^+$. An inference rule \mathfrak{R} is said to be *independent* from the set \mathfrak{S} if and only if there is some finite set $\Sigma \cup \{\sigma\}$ of (N)MVDs such that $\sigma \notin \Sigma_{\mathfrak{S}}^+$, but $\sigma \in \Sigma_{\mathfrak{S} \cup \{\mathfrak{R}\}}^+$. A complete set \mathfrak{S} of inference rules is said to be *minimal* for the implication of (N)MVDs if and only if every inference rule \mathfrak{R} in \mathfrak{S} is independent from $\mathfrak{S} - \{\mathfrak{R}\}$. This means that no proper subset of \mathfrak{S} is still complete for the implication of (N)MVDs.

It should be noted that the singletons $\mathcal{R}, \mathcal{A}, \mathcal{U}, \mathcal{D}, \mathcal{I}$ are all sound, and the R -complementation rule is R -sound, but not sound as just demonstrated by the previous example.

The second major result in (Biskup 1980) shows that the set $\mathfrak{B} = \{\mathcal{R}_\emptyset, \mathcal{A}, \mathcal{T}, \mathcal{T}^*, \mathcal{S}\}$ is sound and complete for the implication of MVDs in undetermined universes.

3 NMVDs in fixed and undetermined Universes

The last section has identified two major objectives:

1. Find a set $R\mathfrak{L}$ of inference rules which is sound, complete and complementary for the R -implication of NMVDs.
2. Identify a set \mathfrak{L} of inference rules which is sound and complete for the implication of NMVDs.

Biskup has successfully provided solutions to these two problems for MVDs, i.e., in the absence of null values. One may hope that the inclusion of the additive transitivity rule \mathcal{T}^* and/or subset rule \mathcal{S} into $R\mathfrak{R}$ result in a complete set of inference rules that is also complementary. Both rules, however, are not sound for the implication of NMVDs as the following example demonstrates.

Example 3.1. Consider the following partial relation r :

A	B	C	D
a	b_1	c_1	ν
a	b_2	c_2	ν

For $X = A$, $Y = BC$, $W = D$ and $Z = B$ we see that $\models_r X \twoheadrightarrow Y$ and $\models_r W \twoheadrightarrow Z$ with $Y \cap W = \emptyset$. However, $\not\models_r X \twoheadrightarrow Y \cap Z$. Note that r satisfies $W \twoheadrightarrow Z$ since the two tuples are not total on W . This shows the incorrectness of the subset rule for NMVDs.

The incorrectness of the additive transitivity rule follows from the following example. Consider the partial relation r :

A	B	C	D
a	ν	c_1	d_1
a	ν	c_2	d_2

For $X = A$, $Y = B$ and $Z = C$ we see that $\models_r X \twoheadrightarrow Y$ and $\models_r Y \twoheadrightarrow Z$. However, $\not\models_r X \twoheadrightarrow YZ$. Note that r satisfies $Y \twoheadrightarrow Z$ since the two tuples are not total on Y . \square

3.1 NMVDs in fixed Universes

Our first theorem shows that there are indeed complete sets which are complementary. In order to be precise, we give the following definition.

Let Σ be a finite set of NMVDs, and let \mathfrak{S} be a set of inference rules. A finite sequence of NMVDs $\gamma = [\sigma_1, \dots, \sigma_k]$ is called an *inference from Σ by \mathfrak{S}* if and only if each σ_i is either an element of Σ or is obtained by applying one of the rules of \mathfrak{S} to appropriate elements of $\{\sigma_1, \dots, \sigma_{i-1}\}$. We say that the inference γ infers σ_k (the last element of the sequence γ). The syntactic hull $\Sigma_{\mathfrak{S}}^+$ is the set of all NMVDs which can be inferred by some inference from Σ by \mathfrak{S} .

Theorem 3.1. Let Σ be a set of NMVDs on the relation schema R . For each inference γ from Σ by the set $R\mathfrak{R} = \{\mathcal{R}, \mathcal{A}, \mathcal{U}, \mathcal{C}_R\}$ there is an inference ξ from Σ by the set $R\mathfrak{L} = \{\mathcal{R}, \mathcal{A}, \mathcal{U}, \mathcal{D}, \mathcal{C}_R\}$ with the following properties:

1. γ and ξ infer the same NMVD.
2. In ξ the R -complementation rule \mathcal{C}_R is applied at most once.
3. If \mathcal{C}_R is applied in ξ , then \mathcal{C}_R is applied as the last rule.

Proof. The proof is by induction on the length l of the inference $\gamma = [\sigma_1, \dots, \sigma_l]$. If $l = 1$, then $\xi := \gamma$ has the desired properties. Let $l > 1$, and $\gamma = [\sigma_1, \dots, \sigma_l]$ be an inference from Σ by \mathfrak{L} which has length l . We consider four cases according to how σ_l was obtained from $[\sigma_1, \dots, \sigma_{l-1}]$.

Case 1. σ_l is either an element of Σ or was obtained using the reflexivity axiom \mathcal{R} . Then $\xi = [\sigma_l]$ has the desired properties.

Case 2. We obtain σ_l by applying the augmentation rule \mathcal{A} to the premise σ_i with $i < l$. Let ξ_i be obtained by using the induction hypothesis for $\gamma_i := [\sigma_1, \dots, \sigma_i]$. Consider the inference $\xi := [\xi_i, \sigma_l]$. If in ξ_i the rule \mathcal{C}_R is not applied, then ξ has the desired properties. If in ξ_i the rule \mathcal{C}_R is applied (as last rule), then the last two steps of ξ are of the form:

$$\frac{\frac{X \twoheadrightarrow Y}{X \twoheadrightarrow R - Y}}{XU \twoheadrightarrow (R - Y)V} V \subseteq U.$$

However, these two steps can be replaced as follows:

$$\frac{\frac{X \rightarrow Y}{XU \rightarrow Y} \quad \frac{XU \rightarrow V}{XU \rightarrow V}^{V \subseteq U \subseteq XU}}{\frac{XU \rightarrow Y - V}{XU \rightarrow R - (Y - V)}} = (R - Y)V$$

The result of this replacement is an inference with the desired properties.

Case 3. We obtain σ_l by applying the union rule \mathcal{U} to the premises σ_i and σ_j with $i, j < l$. Let ξ_i and ξ_j be obtained by using the induction hypothesis for $\gamma_i = [\sigma_1, \dots, \sigma_i]$ and $\gamma_j = [\sigma_1, \dots, \sigma_j]$, respectively. Consider the inference $\xi := [\xi_i, \xi_j, \sigma_l]$. We distinguish four cases according to the occurrences of the R -complementation rule \mathcal{C}_R in ξ_i and ξ_j .

Case 3.1. If \mathcal{C}_R is applied neither in ξ_i nor in ξ_j , then ξ has the desired properties.

Case 3.2. If \mathcal{C}_R is applied in ξ_i (as last rule), but not in ξ_j , then the last step of ξ_i and the last step of ξ are of the following form:

$$\frac{\frac{X \rightarrow Y}{X \rightarrow R - Y} \quad \frac{X \rightarrow Z}{X \rightarrow Z}}{X \rightarrow (R - Y)Z}$$

However, these steps can be replaced as follows:

$$\frac{\frac{X \rightarrow Z}{X \rightarrow Y - Z} \quad \frac{X \rightarrow Y}{X \rightarrow Y - Z}}{X \rightarrow R - (Y - Z)} = (R - Y)Z$$

The result of this replacement is an inference with the desired properties.

Case 3.3. If \mathcal{C}_R is applied in ξ_j (as last rule), but not in ξ_i , then the last step of ξ_j and the last step of ξ are of the following form:

$$\frac{\frac{X \rightarrow Y}{X \rightarrow Y} \quad \frac{X \rightarrow Z}{X \rightarrow R - Z}}{X \rightarrow Y(R - Z)}$$

However, these steps can be replaced as follows:

$$\frac{\frac{X \rightarrow Y}{X \rightarrow Z - Y} \quad \frac{X \rightarrow Z}{X \rightarrow Z - Y}}{X \rightarrow R - (Z - Y)} = Y(R - Z)$$

The result of this replacement is an inference with the desired properties.

Case 3.4. If \mathcal{C}_R is applied both in ξ_i and ξ_j (as last rule), then the last steps of ξ_i and ξ_j and the last step of ξ are of the following form:

$$\frac{\frac{X \rightarrow Y}{X \rightarrow R - Y} \quad \frac{X \rightarrow Z}{X \rightarrow R - Z}}{X \rightarrow (R - Y) \cup (R - Z)}$$

However, these steps can be replaced as follows:

$$\frac{\frac{\frac{X \rightarrow Y}{X \rightarrow Z - Y} \quad \frac{X \rightarrow Z}{X \rightarrow Z}}{X \rightarrow Z - (Z - Y)} = Y \cap Z}{X \rightarrow R - (Y \cap Z)} = (R - Y) \cup (R - Z)$$

The result of this replacement is an inference with the desired properties.

Case 4. We obtain σ_l by applying the R -complementation rule \mathcal{C}_R to the premise σ_i with $i < l$. Let ξ be obtained by using the induction hypothesis for $\gamma_i := [\sigma_1, \dots, \sigma_i]$. Consider the inference $\xi := [\xi_i, \sigma_l]$. If in ξ_i the rule \mathcal{C}_R is not applied, then ξ has the desired properties. If in ξ_i the rule \mathcal{C}_R is applied (as last rule), then the last two steps of ξ are of the following form:

$$\frac{\frac{X \rightarrow Y}{X \rightarrow R - Y}}{X \rightarrow R - (R - Y)} = Y$$

Hence, the inference obtained by removing these two steps from ξ has the desired properties. \square

The set $R\mathfrak{L}$ is complete for the R -implication of NMVDs since $R\mathfrak{L}$ is an extension of the complete set $R\mathfrak{K}$ (Lien 1982). While $R\mathfrak{K}$ is minimal the set $R\mathfrak{L}$ is not (the pseudo-difference rule \mathcal{D} can be omitted). However, $R\mathfrak{L}$ is complementary while $R\mathfrak{K}$ is not. A reasonable question is whether there is any minimal set $R\mathfrak{S}$ which is also complementary. This might be a reasonable task for future research.

3.2 NMVDs in Undetermined Universes

We now explore the power of the common part of the sets $R\mathfrak{L}$, namely $\mathfrak{L} = \{\mathcal{R}, \mathcal{A}, \mathcal{U}, \mathcal{D}\}$, which can be obtained from any of the sets $R\mathfrak{L}$ by removing the R -complementation rule \mathcal{C}_R . Hence, \mathfrak{L} does not permit the possibly semantically meaningless inference of complementation.

Theorem 3.1 states that for all relation schemata R the set $\mathfrak{L} = \{\mathcal{R}, \mathcal{A}, \mathcal{U}, \mathcal{D}\}$ is nearly R -complete. More precisely, we can formulate the following corollary.

Corollary 3.1. *Let $R \subseteq \mathfrak{A}$ be a finite set of attributes. Then for all finite sets $\Sigma = \{X_1 \rightarrow Y_1, \dots, X_k \rightarrow Y_k\}$ of NMVDs, for all NMVDs $X \rightarrow Y$ such that $X \cup Y \cup \bigcup_{i=1}^k (X_i \cup Y_i) \subseteq R$ we have that*

$$X \rightarrow Y \in \Sigma_{R\mathfrak{L}}^+ \text{ if and only if } X \rightarrow Y \in \Sigma_{\mathfrak{L}}^+ \text{ or } X \rightarrow (R - Y) \in \Sigma_{\mathfrak{L}}^+.$$

\square

Corollary 3.1 indicates that by the set \mathfrak{L} we can infer those consequences of a given set of NMVDs which are independent of the underlying relation schema R .

We shall prove now that the set \mathfrak{L} is actually sound and complete for the implication of NMVDs, in the sense of Definition 2.2, that is by \mathfrak{L} we can generate exactly all implications in an undetermined universe.

Lemma 3.1. *Let $\Sigma = \{X_1 \rightarrow Y_1, \dots, X_k \rightarrow Y_k\}$ be a finite set of NMVDs. If $X \rightarrow Y \in \Sigma_{\mathfrak{L}}^+$, then $Y \subseteq X \cup \bigcup_{i=1}^k Y_i$.*

Proof. We show that if $\gamma = [\sigma_1, \dots, \sigma_l]$ is an inference from Σ by \mathfrak{L} such that γ infers the NMVD $\sigma_l = X \rightarrow$

Y , then $Y \subseteq X \cup \bigcup_{i=1}^k Y_i$. The proof is by induction on the length l of γ . If $l = 1$, then $X \rightarrow Y$ was obtained either by application of the reflexivity axiom, i.e. $Y \subseteq X$, or it is an element of Σ . Thus we have $Y \subseteq X \cup \bigcup_{i=1}^k Y_i$ in any case.

Let $l > 1$. We consider four cases according to how σ_l was obtained from $[\sigma_1, \dots, \sigma_{l-1}]$.

Case 1. σ_l was obtained by application of the reflexivity axiom or it is an element of Σ . This is the same situation as for $l = 1$.

Case 2. σ_l was obtained by application of the augmentation rule \mathcal{A} to the premise σ_i with $i < l$. Then the last step of γ has the form

$$\frac{R \rightarrow S}{RU \rightarrow SV} V \subseteq U$$

where $\sigma_i = R \rightarrow S$ and $S \subseteq R \cup \bigcup_{i=1}^k Y_i$ by induction hypothesis, and $\sigma_l = RU \rightarrow SV$. Consequently, we have

$$SV \subseteq RU \cup \bigcup_{i=1}^k Y_i.$$

Case 3. σ_l was obtained by application of the union rule \mathcal{U} to the premises σ_i and σ_j with $i, j < l$. Then the last step of γ has the form

$$\frac{R \rightarrow S, R \rightarrow T}{R \rightarrow ST}$$

where $\sigma_i = R \rightarrow S$ and $S \subseteq R \cup \bigcup_{i=1}^k Y_i$ by induction hypothesis, $\sigma_j = R \rightarrow T$ and $T \subseteq R \cup \bigcup_{i=1}^k Y_i$ by induction hypothesis, and $\sigma_l = R \rightarrow ST$. Consequently, we have

$$ST \subseteq R \cup \bigcup_{i=1}^k Y_i.$$

Case 4. σ_l was obtained by application of the difference rule \mathcal{D} to the premises σ_i and σ_j with $i, j < l$. Then the last step of γ has the form

$$\frac{R \rightarrow S, R \rightarrow T}{R \rightarrow T - S}$$

where $\sigma_i = R \rightarrow S$ and $S \subseteq R \cup \bigcup_{i=1}^k Y_i$ by induction hypothesis, $\sigma_j = R \rightarrow T$ and $T \subseteq R \cup \bigcup_{i=1}^k Y_i$ by induction hypothesis, and $\sigma_l = R \rightarrow T - S$. Consequently, we have

$$T - S \subseteq R \cup \bigcup_{i=1}^k Y_i.$$

This concludes the proof. \square

Lemma 3.2. Let $\Sigma = \{X_1 \rightarrow Y_1, \dots, X_k \rightarrow Y_k\}$ be a finite set of NMVDs. Let $W := \bigcup_{i=1}^k (X_i \cup Y_i)$. If $X \rightarrow Y \in \Sigma_\Sigma^+$, then there is an inference $\gamma = [\sigma_1, \dots, \sigma_l]$ of $X \rightarrow Y$ from Σ by \mathcal{L} such that any attribute occurring in $\sigma_1, \dots, \sigma_{l-1}$ is an element of W .

Proof. Let $\bar{\xi} = [R_1 \rightarrow S_1, \dots, R_{l-1} \rightarrow S_{l-1}]$ be any inference of $X \rightarrow Y$ from Σ by \mathcal{L} . Consider the sequence

$$\xi := [R_1 \cap W \rightarrow S_1 \cap W, \dots, R_{l-1} \cap W \rightarrow S_{l-1} \cap W].$$

We claim that ξ is an inference of $X \cap W \rightarrow Y \cap W$ from Σ by \mathcal{L} . For if $R_i \rightarrow S_i$ is an element of Σ or was obtained by application of the reflexivity axiom

\mathcal{R} , then $R_i \cap W \rightarrow S_i \cap W = R_i \rightarrow S_i$. Moreover, one can verify that if $R_i \rightarrow S_i$ is the result of applying one of the rules $\mathcal{A}, \mathcal{U}, \mathcal{D}$ in $\bar{\xi}$, then $R_i \cap W \rightarrow S_i \cap W$ is the result of the same rule applied to the corresponding premises in ξ .

Now by Lemma 3.1 we know that $Y \subseteq X \cup \bigcup_{i=1}^k Y_i \subseteq X \cup W$, hence $Y - W \subseteq X$. However, this implies that we can infer $X \rightarrow Y$ from $X \cap W \rightarrow Y \cap W$ by the augmentation rule \mathcal{A} :

$$\frac{X \cap W \rightarrow Y \cap W}{\underbrace{(X \cap W) \cup X}_{=X} \rightarrow \underbrace{(Y \cap W) \cup (Y - W)}_{=Y}}.$$

Hence the inference $[\xi, X \rightarrow Y]$ has the desired properties. \square

Theorem 3.2. The set $\mathcal{L} = \{\mathcal{R}, \mathcal{A}, \mathcal{U}, \mathcal{D}\}$ is sound and complete for the implication of multivalued dependencies with null values.

Proof. Let $\Sigma = \{X_1 \rightarrow Y_1, \dots, X_k \rightarrow Y_k\}$ be a finite set of NMVDs, and let $X \rightarrow Y$ be an NMVD. We have to prove that

$$\Sigma \text{ implies } X \rightarrow Y \quad \text{iff} \quad X \rightarrow Y \in \Sigma_\Sigma^+. \quad (3.2)$$

Let $T := X \cup Y \cup \bigcup_{i=1}^k (X_i \cup Y_i)$. In order to prove the soundness of \mathcal{L} (if-part of (3.2)) we assume that $X \rightarrow Y \in \Sigma_\Sigma^+$ holds. Let r be any partial relation such that $T \subseteq \text{Dom}(r)$ and such that r satisfies $X_i \rightarrow Y_i \in \Sigma$ for all $i = 1, \dots, k$. We must show that r also satisfies $X \rightarrow Y$. According to Lemma 3.2 there is an inference γ of $X \rightarrow Y$ from Σ by \mathcal{L} such that $R \cup S \subseteq T \subseteq \text{Dom}(r)$ holds for each NMVD $R \rightarrow S$ occurring in γ . Since each rule of \mathcal{L} is sound we can conclude (by induction) that each NMVD occurring in γ is satisfied by r . Hence, r satisfies $X \rightarrow Y$ in particular.

In order to prove the completeness of \mathcal{L} (only if-part of (3.2)) we assume $X \rightarrow Y \notin \Sigma_\Sigma^+$. Let $R \subseteq \mathcal{A}$ be a finite set of attributes such that T is a proper subset of R , that is $T \subset R$. Consequently, $R - Y$ is not a subset of T . Hence, by Lemma 3.1, $X \rightarrow (R - Y) \notin \Sigma_\Sigma^+$. Now from $X \rightarrow Y \notin \Sigma_\Sigma^+$ and $X \rightarrow (R - Y) \notin \Sigma_\Sigma^+$ we conclude that $X \rightarrow Y \notin R\mathcal{L}$ by Corollary 3.1. Since $R\mathcal{L}$ is R -complete for the R -implication of NMVDs it follows that Σ does not R -imply $X \rightarrow Y$. Hence, Σ does not imply $X \rightarrow Y$ by Theorem 2.2. \square

3.3 All minimal Sets of Inference Rules

One may ask whether there are any other subsets of $\{\mathcal{R}, \mathcal{A}, \mathcal{U}, \mathcal{D}\}$ which are also complete for the implication of NMVDs. The proof of the following result consists of independence proofs. These independence proofs have been computationally verified using GNU pascal (providing set arithmetic) programs.

Theorem 3.3. \mathcal{L} is the only minimal complete subset of $\{\mathcal{R}, \mathcal{A}, \mathcal{U}, \mathcal{D}\}$ which is sound and complete for the implication of NMVDs.

Proof. We show that all four inference rules of \mathcal{L} are essential for gaining completeness, i.e., each inference rule is independent from the rest of the inference rules.

- The reflexivity axiom \mathcal{R} is independent from $\mathfrak{S} = \{\mathcal{A}, \mathcal{I}, \mathcal{D}, \mathcal{U}\}$. Let $\Sigma = \emptyset$, and $\sigma = \emptyset \twoheadrightarrow \emptyset$. Since $\sigma \notin \Sigma_{\mathfrak{S}}^+$, but $\sigma \in \Sigma_{\mathfrak{S} \cup \{\mathcal{R}\}}^+$ we have found witnesses Σ and σ for the independence of \mathcal{R} from \mathfrak{S} .
- The augmentation rule \mathcal{A} is independent from $\mathfrak{S} = \{\mathcal{R}, \mathcal{I}, \mathcal{D}, \mathcal{U}\}$. Let $\Sigma = \{A \twoheadrightarrow B\}$, and $\sigma = AC \twoheadrightarrow B$. The following table represents the closure $\Sigma_{\mathfrak{S}}^+$ of Σ under \mathfrak{S} neglecting all remaining trivial NMVDs $X \twoheadrightarrow Y$ with $Y \subseteq X$ and $X, Y \subseteq \mathcal{A}$.

	\emptyset	A	B	C	AB	AC	BC	ABC
\emptyset	x							
A	x	x	x		x			
B	x		x					
C	x			x				
AB	x	x	x	x	x			
AC	x	x		x		x		
BC	x		x	x			x	
ABC	x	x	x	x	x	x	x	x

Since $\sigma \notin \Sigma_{\mathfrak{S}}^+$, but $\sigma \in \Sigma_{\mathfrak{S} \cup \{\mathcal{A}\}}^+$ we have found witnesses Σ and σ for the independence of \mathcal{A} from \mathfrak{S} .

- The union rule \mathcal{U} is independent from $\mathfrak{S} = \{\mathcal{R}, \mathcal{A}, \mathcal{I}, \mathcal{D}\}$. Let $\Sigma = \{A \twoheadrightarrow B, A \twoheadrightarrow C\}$, and $\sigma = A \twoheadrightarrow BC$. The following table represents the closure $\Sigma_{\mathfrak{S}}^+$ of Σ under \mathfrak{S} neglecting all remaining trivial NMVDs $X \twoheadrightarrow Y$ with $Y \subseteq X$ and $X, Y \subseteq \mathcal{A}$.

	\emptyset	A	B	C	AB	AC	BC	ABC
\emptyset	x							
A	x	x	x	x	x	x		
B	x		x					
C	x			x				
AB	x	x	x	x	x	x	x	x
AC	x	x	x	x	x	x	x	x
BC	x		x	x			x	
ABC	x	x	x	x	x	x	x	x

Since $\sigma \notin \Sigma_{\mathfrak{S}}^+$, but $\sigma \in \Sigma_{\mathfrak{S} \cup \{\mathcal{U}\}}^+$ we have found witnesses Σ and σ for the independence of \mathcal{U} from \mathfrak{S} .

- The difference rule \mathcal{D} is independent from $\mathfrak{S} = \{\mathcal{R}, \mathcal{A}, \mathcal{I}, \mathcal{U}\}$. Let $\Sigma = \{A \twoheadrightarrow BC, A \twoheadrightarrow B\}$, and $\sigma = A \twoheadrightarrow C$. The following table represents the closure $\Sigma_{\mathfrak{S}}^+$ of Σ under \mathfrak{S} neglecting all remaining trivial NMVDs $X \twoheadrightarrow Y$ with $Y \subseteq X$ and $X, Y \subseteq \mathcal{A}$.

	\emptyset	A	B	C	AB	AC	BC	ABC
\emptyset	x							
A	x	x	x		x		x	x
B	x		x					
C	x			x				
AB	x	x	x		x		x	x
AC	x	x	x	x	x	x	x	x
BC	x		x	x			x	
ABC	x	x	x	x	x	x	x	x

Since $\sigma \notin \Sigma_{\mathfrak{S}}^+$, but $\sigma \in \Sigma_{\mathfrak{S} \cup \{\mathcal{D}\}}^+$ we have found witnesses Σ and σ for the independence of \mathcal{D} from \mathfrak{S} . \square

3.4 NFDs and NMVDs in undetermined Universes

Finally, we use Theorem 3.2 and the results from (Lien 1982) to obtain an axiomatisation for NFDs and NMVDs in undetermined universes.

Theorem 3.4. *The following set of inference rules*

$$\begin{array}{c}
\frac{}{X \twoheadrightarrow Y} \text{Y} \subseteq X \quad \frac{X \twoheadrightarrow Y}{XU \twoheadrightarrow YV} \text{V} \subseteq U \quad \frac{X \twoheadrightarrow Y, X \twoheadrightarrow Z}{X \twoheadrightarrow YZ} \\
\frac{X \twoheadrightarrow Y}{X \twoheadrightarrow Z} \text{Z} \subseteq Y \quad \frac{X \twoheadrightarrow Y}{X \twoheadrightarrow Y} \quad \frac{}{X \twoheadrightarrow Y} \text{Y} \subseteq X \\
\frac{X \twoheadrightarrow Y}{XU \twoheadrightarrow YV} \text{V} \subseteq U \quad \frac{X \twoheadrightarrow Y, X \twoheadrightarrow Z}{X \twoheadrightarrow YZ} \quad \frac{X \twoheadrightarrow Y, X \twoheadrightarrow Z}{X \twoheadrightarrow Z - Y}
\end{array}$$

is sound and complete for the implication of functional and multivalued dependencies with null values. \square

The reflexivity rule for NMVDs is certainly redundant in this set of inference rules. It has been included to emphasize the fact that NFDs and NMVDs can be dealt with separately (even when they are specified together). This is entirely different from traditional relational databases without null values where FDs and MVDs have been shown to interact non-trivially (Beeri et al. 1977).

4 General Results

We will use this section to show an equivalence between complete sets of inference rules in undetermined universes and complete and complementary sets in fixed universes. Due to space limitations we omit the proofs in this section. For a set \mathfrak{S} of inference rules that is sound for the implication of NMVDs let $R\mathfrak{S}$ denote the set $\mathfrak{S} \cup \{\mathcal{C}_R\}$.

Theorem 4.1. *Let \mathfrak{S} be a sound set of inference rules for the implication of NMVDs. The set \mathfrak{S} is complete for the implication of NMVDs if and only if the set $R\mathfrak{S}$ is complete and complementary for the R-implication of NMVDs.* \square

This equivalence from Theorem 4.1 can even be extended to minimal complete sets of inference rules.

Corollary 4.1. *Let \mathfrak{S} be a sound set of inference rules for the implication of NMVDs. The set \mathfrak{S} is minimal and complete for the implication of NMVDs if and only if the set $R\mathfrak{S}$ is complete and complementary for the R-implication of NMVDs, and there is no inference rule $\mathfrak{R} \in \mathfrak{S}$ such that the set $R(\mathfrak{S} - \{\mathfrak{R}\})$ is still both complete and complementary for the R-implication of NMVDs.* \square

The last corollary helps us finding all subsets of $\{\mathcal{R}, \mathcal{A}, \mathcal{U}, \mathcal{I}, \mathcal{D}\}$ that are complete and complementary for the R-implication of NMVDs.

Lemma 4.1. *The R-complementation rule \mathcal{C}_R is independent from $\mathfrak{S} = \{\mathcal{R}, \mathcal{A}, \mathcal{U}, \mathcal{I}, \mathcal{D}\}$.*

Proof. Let $R = A$, $\Sigma = \emptyset$ and $\sigma = \emptyset \twoheadrightarrow A$. Since $\sigma \notin \Sigma_{\mathfrak{S}}^+$, but $\sigma \in \Sigma_{\mathfrak{S} \cup \{\mathcal{C}_R\}}^+$ we have found witnesses R , Σ and σ for the independence of \mathcal{C}_R from \mathfrak{S} . \square

The next corollary is a consequence of Theorem 3.3, Corollary 4.1 and Lemma 4.1.

Corollary 4.2. *There are no proper subsets of $R\mathfrak{L}$ which are both complete and complementary for the R-implication of NMVDs.* \square

5 Minimising Minimality

Recall that a complete set \mathfrak{S} of inference rules is said to be minimal iff none of the rules in \mathfrak{S} can be omitted from \mathfrak{S} without losing completeness. In this sense the set $\mathfrak{L} = \{\mathcal{R}, \mathcal{A}, \mathcal{U}, \mathcal{D}\}$ is minimal for the implication of NMVDs. A stricter version of minimality would include that the side conditions of all inference rules cannot be weakened. For instance, since both the

reflexivity axiom $\frac{}{X \rightarrow Y}^{Y \subseteq X}$ and the augmentation rule $\frac{X \rightarrow Y}{XU \rightarrow YV}^{V \subseteq U}$ are present in \mathfrak{L} one may replace the reflexivity axiom \mathcal{R} by the empty-set-axiom \mathcal{R}_\emptyset :

$\frac{}{\emptyset \rightarrow \emptyset}$ and still maintain completeness. In fact, the empty-set-axiom \mathcal{R}_\emptyset is a very weak form of the reflexivity axiom \mathcal{R} representing just the single instance of \mathcal{R} where $X = Y = \emptyset$. However, \mathcal{R} is derivable from $\{\mathcal{R}_\emptyset, \mathcal{A}\}$:

$$\frac{\frac{}{\emptyset \rightarrow \emptyset}}{X \rightarrow Y}^{Y \subseteq X} .$$

Theorem 5.1. *The set $\{\mathcal{R}_\emptyset, \mathcal{A}, \mathcal{U}, \mathcal{D}\}$ is sound and complete for the implication of multivalued dependencies with null values.* \square

Instead of weakening the reflexivity axiom, one may replace the augmentation rule \mathcal{A} by the weak augmentation rule \mathcal{W} : $\frac{X \rightarrow Y}{XA \rightarrow Y}$ which is a very restricted form of augmentation in which $V = \emptyset$ and $U = A$ is a singleton. However, \mathcal{A} can be derived from $\{\mathcal{R}, \mathcal{W}, \mathcal{U}\}$ as follows (suppose $U = \{A_1, \dots, A_k\}$):

$$\frac{\frac{\frac{X \rightarrow Y}{XA_1 \rightarrow Y} \quad \vdots \quad \frac{X \rightarrow Y}{XA_k \rightarrow Y}}{XA_1 \dots A_k \rightarrow Y} \quad \frac{XU \rightarrow V}{XU \rightarrow YV}}{XU \rightarrow YV}^{V \subseteq U \subseteq XU} .$$

The reflexivity axiom \mathcal{R} may also be replaced by the empty-set-axiom \mathcal{R}_\emptyset and the attribute axiom

$\mathcal{A}t$: $\frac{A \rightarrow A}{A \rightarrow A}$. In fact, \mathcal{R} can be derived from $\{\mathcal{R}_\emptyset, \mathcal{A}t, \mathcal{W}, \mathcal{U}\}$. If $Y = \emptyset$ and X consists of k attributes, then we apply the empty-set-axiom \mathcal{R}_\emptyset first to derive $\emptyset \rightarrow \emptyset$. Subsequently, the weak augmentation rule \mathcal{W} is applied k times to derive $X \rightarrow \emptyset$. In case that $Y = \{B_1, \dots, B_l\}$ and X has k attributes, $k \geq l$, we derive $B_1 \rightarrow B_1, \dots, B_l \rightarrow B_l$ by l applications of the attribute axiom $\mathcal{A}t$. Subsequently, we apply the weak augmentation rule \mathcal{W} to each of these NMVDs k times to derive $X \rightarrow B_1, \dots, X \rightarrow B_l$. Finally, the union rule \mathcal{U} is applied $l - 1$ times to derive $X \rightarrow Y$.

Theorem 5.2. *The set $\{\mathcal{R}_\emptyset, \mathcal{A}t, \mathcal{W}, \mathcal{U}, \mathcal{D}\}$ is sound and complete for the implication of multivalued dependencies with null values.* \square

6 Conclusion

We have explored multivalued dependencies in the presence of null values (NMVDs) with meaning “undefined”, “inapplicable”, or “non-existent”. It was shown that Lien’s original axiomatisation of NMVDs (Lien 1982) is not complementary. That is, there are inferences of NMVDs in which the application of the complementation rule can neither be avoided nor deferred until the last step of the inference. The fact that the complementation rule simply reflects a part

of the normalisation process is therefore not reflected by Lien’s axiomatisation. In this paper sound and complete sets of inference rules for the R -implication of NMVDs have been proposed that are indeed complementary. Moreover, Biskup’s alternative notion of implication for MVDs, in which the underlying universe is left undetermined, was extended to the presence of null values. Several sound and complete sets of inference rules for the implication of NMVDs have been proposed, which can be extended to cover both functional and multivalued dependencies in the presence of null values. The results clarify the role of the R -complementation rule for NMVDs, and may simplify the quest of finding suitable and comprehensible notions of multivalued dependencies in the context of advanced database models. Moreover, the results clarify the power of several R -incomplete subsets.

Some interesting problems warrant future research. While the R -implication problem of MVDs has received a considerable amount of interest with the best current time bound proposed in (Galil 1982), no research has been devoted to the corresponding R -implication problem of NMVDs. In the spirit of our article it seems also interesting to investigate the implication problem of (N)MVDs in undetermined universes, and maybe derive further correspondences between implication and R -implication.

An interesting open problem is to generalise the approach in (Levene & Loizou 1998) from functional to multivalued dependencies. The approach uses a possible world semantics exploring all extensions of an incomplete database to a complete database. Weak MVDs must be satisfied by some possible world while strong MVDs are satisfied by all possible worlds.

References

- Arenas, M. & Libkin, L. (2004), ‘A normal form for XML documents’, *Transactions on Database Systems (ToDS)* **29**(1), 195–232.
- Armstrong, W. W. (1974), ‘Dependency structures of database relationships’, *Information Processing* **74**, 580–583.
- Armstrong, W. W., Nakamura, Y. & Rudnicki, P. (2002), ‘Armstrong’s axioms’, *Journal of formalized Mathematics* **14**.
- Atzeni, P. & Morfuni, N. (1986), ‘Functional dependencies and constraints on null values in database relations’, *Information and Control* **70**(1), 1–31.
- Beeri, C. & Bernstein, P. A. (1979), ‘Computational problems related to the design of normal form relational schemata’, *ACM Trans. Database Syst.* **4**(1), 30–59.
- Beeri, C., Fagin, R. & Howard, J. H. (1977), A complete axiomatization for functional and multivalued dependencies in database relations, in ‘Proceedings of the SIGMOD International Conference on Management of Data’, ACM, pp. 47–61.
- Bernstein, P. (1976), ‘Synthesizing third normal form relations from functional dependencies’, *ACM Trans. Database Syst.* **1**(4), 277–298.
- Bernstein, P. A. & Goodman, N. (1980), What does Boyce-Codd normal form do?, in ‘Proceedings of the 6th International Conference on Very Large Data Bases’, IEEE Computer Society, pp. 245–259.

- Biskup, J. (1978), 'On the complementation rule for multivalued dependencies in database relations', *Acta Informatica* **10**(3), 297–305.
- Biskup, J. (1980), 'Inferences of multivalued dependencies in fixed and undetermined universes', *Theor. Comput. Sci.* **10**(1), 93–106.
- Codd, E. F. (1970), 'A relational model of data for large shared data banks', *Commun. ACM* **13**(6), 377–387.
- Codd, E. F. (1972), Further normalization of the database relational model, in 'Courant Computer Science Symposia 6: Data Base Systems', Prentice-Hall, pp. 33–64.
- Codd, E. F. (1975), 'Understanding relations', *ACM SIGFIDET FDT Bulletin* **7**(3-4), 23–28.
- Codd, E. F. (1979), 'Extending the database relational model to capture more meaning', *Transactions on Database Systems (ToDS)* **4**(4), 397–434.
- Delobel, C. (1978), 'Normalisation and hierarchical dependencies in the relational data model', *ACM Trans. Database Syst.* **3**(3), 201–222.
- Fagin, R. (1977), 'Multivalued dependencies and a new normal form for relational databases', *ACM Trans. Database Syst.* **2**(3), 262–278.
- Fagin, R. & Vardi, M. Y. (1986), The theory of data dependencies: a survey, in 'Mathematics of Information Processing: Proceedings of Symposia in Applied Mathematics', American Mathematical Society, pp. 19–71.
- Fischer, P. C., Saxton, L. V., Thomas, S. J. & Van Gucht, D. (1985), 'Interactions between dependencies and nested relational structures', *J. Comput. Syst. Sci.* **31**(3), 343–354.
- Galil, Z. (1982), 'An almost linear-time algorithm for computing a dependency basis in a relational database', *J. ACM* **29**(1), 96–102.
- Gottlob, G. & Zicari, R. (1988), Closed world databases opened through null values, in 'Proceedings of the 14th International Conference on Very Large Data Bases (VLDB)', pp. 50–61.
- Grahne, G. (1984), Dependency satisfaction in databases with incomplete information, in 'Proceedings of the 10th International Conference on Very Large Data Bases (VLDB)', pp. 37–45.
- Grant, J. (1977), 'Null values in a relational database', *Information Processing Letters* **6**(5), 156–157.
- Hara, C. & Davidson, S. (1999), Reasoning about nested functional dependencies, in 'Proceedings of the 18th SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems', ACM, pp. 91–100.
- Hartmann, S. & Link, S. (2004), Multi-valued dependencies in the presence of lists, in 'Proceedings of the 23rd SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems', ACM, pp. 330–341.
- Hartmann, S. & Link, S. (2006), 'On a problem of Fagin concerning multivalued dependencies in relational databases', accepted for Theoretical Computer Science (TCS).
- Johnson, M. & Rosebrugh, R. (2003), 'Three approaches to partiality in the sketch data model', *Electronic Notes in Theoretical Computer Science* **78**, 1–18.
- Levene, M. & Loizou, G. (1993), 'Semantics for null extended nested relations', *Transactions on Database Systems (ToDS)* **18**(3), 414–459.
- Levene, M. & Loizou, G. (1998), 'Axiomatisation of functional dependencies in incomplete relations', *Theoretical Computer Science (TCS)* **206**(1-2), 283–300.
- Lien, Y. E. (1982), 'On the equivalence of data models', *Journal of the ACM* **29**(2), 333–363.
- Link, S. (2006), 'On multivalued dependencies in fixed and undetermined universes', accepted for the 4th International Symposium on Foundations of Information and Knowledge Systems (FoIKS), Springer, LNCS.
- Mendelzon, A. (1979), 'On axiomatising multivalued dependencies in relational databases', *J. ACM* **26**(1), 37–44.
- Mikinouchi, A. (1977), A consideration on normal form of not-necessarily-normalised relation in the relational data model, in 'Proceedings of the 3rd International Conference on Very Large Data Bases (VLDB)', IEEE Computer Society, pp. 447–453.
- Tari, Z., Stokes, J. & Spaccapietra, S. (1997), 'Object normal forms and dependency constraints for object-oriented schemata', *ACM Trans. Database Syst.* **22**, 513–569.
- Thalheim, B. (1991), *Dependencies in Relational Databases*, Teubner-Verlag.
- Thalheim, B. (2003), Conceptual treatment of multivalued dependencies, in 'Conceptual Modeling - ER 2003, 22nd International Conference on Conceptual Modeling, Chicago, IL, USA, October 13-16, 2003, Proceedings', number 2813 in 'Lecture Notes in Computer Science', Springer, pp. 363–375.
- Vincent, M. & Liu, J. (2003), Multivalued dependencies in XML, in 'Proceedings of the 20th British National Conference on Databases', number 2712 in 'Lecture Notes in Computer Science', Springer, pp. 4–18.
- Vincent, M., Liu, J. & Liu, C. (2003), A redundancy free 4NF for XML, in 'Proceedings of the 1st International XML Database Symposium', number 2824 in 'Lecture Notes in Computer Science', Springer, pp. 254–266.
- Weddell, G. (1992), 'Reasoning about functional dependencies generalized for semantic data models', *ACM Trans. Database Syst.* **17**(1), 32–64.
- Wijzen, J. (1999), 'Temporal FDs on complex objects', *ACM Trans. Database Syst.* **24**(1), 127–176.
- Zaniolo, C. (1976), Analysis and Design of Relational Schemata for Database Systems, PhD thesis, UCLA, Tech. Rep. UCLA-ENG-7769.
- Zaniolo, C. (1984), 'Database relations with null values', *Journal of Computer and System Sciences* **28**(1), 142–166.

Boolean equation solving as graph traversal

Brian Herlihy

Peter Schachte*

Harald S ndergaard

Department of Computer Science and Software Engineering
The University of Melbourne, Vic. 3010, Australia
Email: {bther1,schachte,harald}@csse.unimelb.edu.au

Abstract

We present a new method for finding closed forms of recursive Boolean function definitions. Traditionally, these closed forms are found by iteratively approximating until a fixed point is reached. Conceptually, our new method replaces each k -ary function by 2^k Boolean variables defined by mutual recursion. The introduction of an exponential number of variables is mitigated by the simplicity of their definitions and by the use of a novel variant of ROBDDs to avoid repeated computation. Experimental evaluation suggests that this approach is significantly faster than Kleene iteration for examples that would require many Kleene iteration steps.

Keywords: Boolean functions, fixed points, decision diagrams

1 Introduction

The need to obtain closed forms of recursively defined functions arises in many fields of mathematics and computer science, including formal language theory, database theory, and formal semantics. The theoretical foundations for solving recursive definitions were laid a long time ago, in the fields of order and fixed point theory. Improved algorithms, however, are still being developed for special applications.

A typical case is *abstract interpretation*, which is concerned with automated inference of program properties, as needed, for example, by optimising compilers and other program analysis and transformation tools (Cousot & Cousot 1977). Problems in abstract interpretation boil down to finding extreme fixed points (usually least fixed points) in structures that can be fairly complex. The meaning of a program is assumed to be given as a fixed point characterisation: the least fixed point of some functional F defined over a domain of program states, based on the semantic domains for the programming language. Program analysis is then obtained by faithfully abstracting F to work on some *abstract domain* instead, that is, on a domain of approximate program states. Usually the process can be divided into two stages: firstly constructing a set of (mutually) recursively defined functions as a conservative approximation of the aspect of the behaviour of the components of the

program one is interested in, and secondly finding a closed-form “solution” of the recursive equations.

The usual approach to finding fixed points for recursively defined equations is *Kleene iteration*. The essential idea is to transform the set of mutually recursively defined functions into a single non-recursive functional that takes a tuple of closed form tentative solutions to these equations and uses the original definitions of the functions, modified to use the tentative solutions in place of recursive calls, to compute a better approximation. Kleene iteration repeatedly applies this functional to the result of the previous iteration until a fixed point is reached. If the functional is monotone on a lattice with the ascending-chain property, and iteration begins with the least element, the process is guaranteed to terminate with the least fixed point, *i.e.*, the strongest closed form solution to the initial set of equations.

Consider, for example, the following equation:

$$f(x, y, z) = (x \wedge (y \leftrightarrow z)) \vee \exists u, v. ((x \leftrightarrow (u \wedge v)) \wedge f(v, u, z))$$

This function arises in the groundness analysis of a Prolog program to find the last element of a list using an accumulating parameter. To apply Kleene iteration to this, one defines a functional

$$F(f) = \lambda x, y, z. ((x \wedge (y \leftrightarrow z)) \vee \exists u, v. ((x \leftrightarrow (u \wedge v)) \wedge f(v, u, z)))$$

Note that the call to f is no longer recursive; it now invokes the parameter to F . Kleene iteration would begin by applying F to the least possible function f , which is $\lambda x, y, z. 0$. For brevity, henceforth we shall agree that the arguments of this function are x, y , and z , and omit the λ . Thus the result of $F(0)$ is $(x \wedge (y \leftrightarrow z)) \vee \exists u, v. ((x \leftrightarrow (u \wedge v)) \wedge 0) = x \wedge (y \leftrightarrow z)$. Next we apply F to this and get $(x \wedge (y \leftrightarrow z)) \vee \exists u, v. ((x \leftrightarrow (u \wedge v)) \wedge v \wedge (u \leftrightarrow z)) = (x \wedge (y \leftrightarrow z)) \vee (x \leftrightarrow z)$. Applying F to this, we get $(x \wedge y) \rightarrow z$. Finally, applying F to this, we get back $(x \wedge y) \rightarrow z$, indicating a fixed point has been reached.

The first contribution of this paper is to propose an alternative view of this problem. Consider again the original function f . We build a table in which each row shows a possible input to f and the value of f for that input. In each row of the table, we substitute the input for that row, and all possible combinations of values for the existentially quantified variables. Thus the table will contain no variables whatsoever; all that remains will be Boolean constants and recursive calls. Furthermore, since each recursive call has all arguments statically known, it refers to a fixed row of the table. The following table illustrates the same example f shown above using the new approach (we use subscripts for function arguments).

*Peter Schachte’s work on this project has been supported in part by NICTA Victoria Laboratories.

Copyright   2006, Australian Computer Society, Inc. This paper appeared at Computing: The Australasian Theory Symposium (CATS2006), Hobart, Australia. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 51. Barry Jay and Joachim Gudmundsson, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.

$$\begin{aligned}
f_{000} &= 0 \vee \exists u, v. (\neg(u \wedge v) \wedge f_{vu0}) \\
f_{001} &= 0 \vee \exists u, v. (\neg(u \wedge v) \wedge f_{vu1}) \\
f_{010} &= 0 \vee \exists u, v. (\neg(u \wedge v) \wedge f_{vu0}) \\
f_{011} &= 0 \vee \exists u, v. (\neg(u \wedge v) \wedge f_{vu1}) \\
f_{100} &= 1 \vee \exists u, v. (u \wedge v \wedge f_{vu0}) \\
f_{101} &= 0 \vee \exists u, v. (u \wedge v \wedge f_{vu1}) \\
f_{110} &= 0 \vee \exists u, v. (u \wedge v \wedge f_{vu0}) \\
f_{111} &= 1 \vee \exists u, v. (u \wedge v \wedge f_{vu1})
\end{aligned}$$

Now we view this as a set of eight mutually-*recursively* defined variables. Of these, f_{100} and f_{111} are unambiguously determined to be 1. Also, f_{101} is defined to be f_{111} , which is 1, and f_{000} , f_{001} , f_{010} , and f_{011} are all defined to be disjunctions including either f_{100} or f_{101} , both of which are now known to be 1, so all are 1. This leaves only f_{110} , which is defined to be f_{110} . This indicates both truth values will be fixed points for this row. Assigning it 0 yields the least fixed point $(x \wedge y) \rightarrow z$, as given by Kleene iteration. Plugging in 1 yields $f(x, y, z) = 1$ which can also be verified to be a fixed point.

In this example, only two fixed points exist. In general, there will be 2^n fixed points, where n is the number of strongly-connected components (SCCs) in the dependency graph among rows in the truth table.

It is possible, for the domain of Boolean functions of arity n , to give a recursive definition for which the ascending Kleene sequence has maximal possible length: $2^n + 1$. In fact this can be achieved in several different ways. The following definition exemplifies this for a function of arity 4. What is defined in this cumbersome manner is the constant function 1, but it takes 16 Kleene iteration steps to determine this.

$$\begin{aligned}
p(v_1, v_2, v_3, v_4) &= \\
&\exists v_5. (v_4 \wedge p(v_1, v_2, v_3, v_5)) \\
&\vee (v_3 \wedge p(v_1, v_2, v_4, 1)) \\
&\vee (v_2 \wedge (v_3 \leftrightarrow v_4) \wedge p(v_1, v_3, 1, 1)) \\
&\vee (v_1 \wedge (v_2 \leftrightarrow v_3) \wedge (v_3 \leftrightarrow v_4) \wedge p(v_2, 1, 1, 1)) \\
&\vee ((v_1 \leftrightarrow v_2) \wedge (v_2 \leftrightarrow v_3) \wedge (v_3 \leftrightarrow v_4))
\end{aligned}$$

Again, this is a definition that arises in the groundness analysis of a certain Prolog program, although the program would not appear in the typical Prolog programmer's collection. It comes from one of several families suggested by Codish (1999) and Genaim, Howe & Codish (2001) as particularly challenging for analysis, because of their heavily iterative nature. In fact, the challenge that these programs have posed to our existing analysis tools (which make use of straight-forward Kleene iteration) has been the primary motivation for the work reported here.

The remainder of this paper will proceed as follows. In Section 2 we recall some basic concepts from fixed point theory and decision diagrams. Section 3 introduces a data structure, a variant of ROBDDs, and an algorithm to efficiently solve recursive definitions of Boolean functions. Section 4 reports on the experimental evaluation of the algorithm, Section 5 discusses related work, and Section 6 concludes.

2 Preliminaries

2.1 Lattices and fixed points

A *partial ordering* is a binary relation that is reflexive, transitive, and antisymmetric. A set equipped with a partial ordering is a *poset*. Let (X, \leq) be a poset. A (possibly empty) subset Y of X is a *chain* iff for all $y, y' \in Y$, $y \leq y' \vee y' \leq y$.

Let (X, \leq) be a poset. An element $x \in X$ is an *upper bound* for $Y \subseteq X$ iff $y \leq x$ for all $y \in Y$. Dually

we may define a *lower bound* for Y . An upper bound x for Y is the *least upper bound* for Y iff, for every upper bound x' for Y , $x \leq x'$, and when it exists, we denote it by $\bigsqcup Y$. Dually we may define the *greatest lower bound* $\bigsqcap Y$ for Y .

A poset X for which every subset possesses a least upper bound and a greatest lower bound is a *complete lattice*. We denote the least element $\bigsqcap X$ of X by \perp_X (or usually just \perp). X is *ascending chain finite* iff every ascending chain in X is finite.

Let (X, \leq) and (Z, \preceq) be posets. $F : X \rightarrow Z$ is *monotone* iff $x \leq x' \rightarrow F(x) \preceq F(x')$ for all $x, x' \in X$. A *fixed point* for $F : X \rightarrow X$ is an element $x \in X$ such that $x = F(x)$. If X is a complete lattice, then the set of fixed points for a monotone $F : X \rightarrow X$ is itself a complete lattice. The least element of this lattice is the *least fixed point* for F , and we denote it by $\text{lfp}(F)$.

2.2 ROBDDs

Let $\mathbb{B} = \{1, 0\}$. The set of Boolean functions is $\text{Bool} = \bigcup_{n \in \mathbb{N}} \mathbb{B}^n \rightarrow \mathbb{B}$. Let the set \mathcal{V} of propositional variables be equipped with a total ordering \prec .

Binary decision diagrams (BDDs) are defined inductively as follows:

- 0 is a BDD.
- 1 is a BDD.
- If $x \in \mathcal{V}$ and R_1 and R_2 are BDDs then $\text{ite}(x, R_1, R_2)$ is a BDD.

The meaning of a BDD is given as follows:

$$\begin{aligned}
\llbracket 0 \rrbracket &= 0 \\
\llbracket 1 \rrbracket &= 1 \\
\llbracket \text{ite}(x, R_1, R_2) \rrbracket &= (x \wedge \llbracket R_1 \rrbracket) \vee (\neg x \wedge \llbracket R_2 \rrbracket)
\end{aligned}$$

Let $R = \text{ite}(x, R_1, R_2)$. A BDD R' *appears in* R iff $R' = R$ or R' appears in R_1 or R_2 . We define $\text{vars}(R) = \{v \mid \text{ite}(v, _, _) \text{ appears in } R\}$.

A BDD is an *OBDD* iff it is 0 or 1 or if it is $\text{ite}(x, R_1, R_2)$ where R_1 and R_2 are OBDDs, and $\forall x' \in \text{vars}(R_1) \cup \text{vars}(R_2) : x \prec x'$.

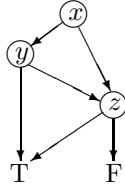
An OBDD R is an *ROBDD* (Reduced Ordered Binary Decision Diagram (Bryant 1992)) iff for all BDDs R_1 and R_2 appearing in R , $R_1 = R_2$ when $\llbracket R_1 \rrbracket = \llbracket R_2 \rrbracket$. Practical implementations (Brace, Rudell & Bryant 1990) use a function $\text{mknd}(x, R_1, R_2)$ to create all ROBDD nodes as follows:

1. If $R_1 = R_2$, return R_1 instead of a new node, as $\llbracket \text{ite}(x, R_1, R_2) \rrbracket = \llbracket R_1 \rrbracket$.
2. If an identical ROBDD was previously built, return that one instead of a new one; this is accomplished by keeping a hash table, called the *unique table*, of all previously created nodes.
3. Otherwise, return $\text{ite}(x, R_1, R_2)$.

This ensures that ROBDDs are strongly canonical: a shallow equality test is sufficient to determine whether or not two ROBDDs represent the same Boolean function.

Figure 1 gives a visual presentation of a simple ROBDD. For all diagrams in this paper, the leftmost of the two outgoing edges from any non-terminal node is the 1-edge and the rightmost is the 0-edge.

There are 2^{2^n} different Boolean functions of arity n , and it has been shown (Liaw & Lin 1992) that an ROBDD for an n -input Boolean function requires

Figure 1: An ROBDD for the function $(x \wedge y) \vee z$.

at most $(2^n/n)$ nodes, so the worst-case complexity is exponential. However, the size of ROBDDs can vary enormously, not only with the ordering of variables, but also with the type of functions that are represented. For a great variety of Boolean functions, the ROBDD representation has polynomial complexity, which makes ROBDDs very attractive for a wide variety of applications.

Many important properties of Boolean functions can be easily tested from the ROBDD form of a function. Testing for unsatisfiable or tautological functions can be done in constant time: an unsatisfiable function is represented by the 0-terminal and a tautological function is represented by the 1-terminal. If the satisfiability of a function requires a particular variable to have a value v , then all nodes labelled with that variable will have their $(\neg v)$ -edge point directly to the 0-terminal. Finally, the ROBDD representation of a function that is independent of a particular variable will have no nodes labelled by that variable.

3 Solving as graph manipulation

Before we embark on describing the new algorithm, we describe an algorithm that served as inspiration.

3.1 A depth-first approach

The algorithm we describe first is due to Le Charlier & Van Hentenryck (1992). It is rather more general than ours, indeed it is presented as a “universal top-down fixed point algorithm”. It is well described (and appraised) by Fecht & Seidl (1999). Figure 2 is essentially Fecht and Seidl’s presentation of algorithm **TD**, as they name it.¹

The idea behind the algorithm is to maintain a “partial table” σ that maps variables to their current (approximate) values. In Figure 2, σ is a globally accessible table.

Suppose we have initiated solving for x and in the process find that we need the value of y . The approach is to eagerly turn to solving for y , that is, to find the next approximation to y ’s final value. However, there are two situations in which eagerness is abandoned and the current approximation $\sigma(y)$ to y ’s value is used instead. One is where we are already in an iteration initiated for y . A set *Called* is thus maintained to keep track of variables for which an iteration has been started but not completed. The other situation is where solving for y is deemed useless because no variable that influences y has changed its value. A set *Stable* is thus maintained to keep track of stable variables. A procedure call *destabilize*(x) makes sure that the implications of changing the value of x are tracked, so that *Stable* remains reliable.

¹Fecht and Seidl’s version is incorrect, probably as a result of a typographic error. The line marked with (*) in Figure 2 is missing in Fecht and Seidl’s presentation. As a result, the closed form of, say, $f(x, y) = x \vee f(y, x)$ comes out incorrectly as $f(x, y) = x$.

```

procedure main
   $\sigma := \emptyset$ ; Stable :=  $\emptyset$ ; Called :=  $\emptyset$ ; infl :=  $\emptyset$ ;
  foreach  $x \in V$  do solve( $x$ ) od
end

procedure solve( $x : V$ )
  if  $x \notin \textit{Stable}$  and  $x \notin \textit{Called}$  then
    if  $x \notin \textit{dom}(\sigma)$  then
       $\sigma(x) := \perp$ ; infl( $x$ ) :=  $\emptyset$ 
    fi;
    Called := Called  $\cup \{x\}$ ;
    do
      Stable := Stable  $\cup \{x\}$ ;
      old :=  $\sigma(x)$ ;
       $\sigma(x) := \textit{evalrhs}(x, \lambda y. \textit{eval}(x, y))$ ;
      if  $\sigma(x) \neq \textit{old}$  then
        Stable := Stable  $\setminus \{x\}$ ; (*)
        destabilize( $x$ )
      fi;
    until  $x \in \textit{Stable}$ ;
    Called := Called  $\setminus \{x\}$ 
  fi
end

function eval( $x, y : V$ ) :  $D$ 
  solve( $y$ );
  infl( $y$ ) := infl( $y$ )  $\cup \{x\}$ ;
  return  $\sigma(y)$ 
end

procedure destabilize( $x : V$ )
  temp := infl( $x$ ); infl( $x$ ) :=  $\emptyset$ ;
  foreach  $y \in \textit{temp}$  do
    Stable := Stable  $\setminus \{y\}$ ;
    destabilize( $y$ )
  od
end

```

Figure 2: Algorithm **TD** (after Fecht and Seidl)

The role of *evalrhs*(x, \cdot) is to evaluate the right-hand side of x ’s definition. This evaluation has access to (the global) σ . However, to evaluate eagerly, and to (dynamically) keep track of variable dependencies, $\lambda y. \textit{eval}(x, y)$ is used instead of σ . The table *infl* is used for the bookkeeping—*infl*(y) is the set of variables that may depend on y . Essentially, *eval*(x, y) provides for the solving for y in a context of solving for x , updating *infl* to track the dependency.

3.2 A data structure for equation systems

The algorithm just presented is general. We now turn to the special case of finding closed forms for (mutually) recursively defined Boolean functions and the second contribution of this paper: a new data structure and algorithm for this special case.

To be more formal, assume we are given a set \mathbb{F} of Boolean function names. As a convenience, we also fix a set \mathcal{V} of variables, with its total ordering $<$. We let the smallest n variables serve as the sequence of formal parameters for all Boolean functions, with all larger variables serving as local variables in definition bodies. Then we can define the set of Boolean function bodies and definitions as:

$$\begin{aligned}\text{Bod} &= \mathbb{B} \cup \mathcal{V} \cup (\mathbb{F} \times \mathcal{V}^n) \cup \{x \wedge y \mid x, y \in \text{Bod}\} \cup \\ &\quad \{x \vee y \mid x, y \in \text{Bod}\} \cup \{\neg x \mid x \in \mathcal{V}\} \\ \text{Def} &= \mathbb{F} \rightarrow \text{Bod}\end{aligned}$$

Here the logical connectives have their usual interpretation, and local variables are implicitly existentially quantified over the definition body.

We define the set of closed form Boolean function definitions to omit function applications:

$$\text{Bod}_c = \mathbb{B} \cup \mathcal{V} \cup \{x \wedge y \mid x, y \in \text{Bod}_c\} \cup \{x \vee y \mid x, y \in \text{Bod}_c\} \cup \{\neg x \mid x \in \mathcal{V}\}$$

$$\text{Def}_c = \mathbb{F} \rightarrow \text{Bod}_c$$

Note that $\text{Def}_c \subseteq \text{Def}$. Furthermore, because $\{\wedge, \vee, \neg\}$ are functionally complete for **Bool**, and by de Morgan's laws, Bod_c is equivalent to $\mathbb{B}^n \rightarrow \mathbb{B}$, and thus is a subset of **Bool**. Let Bod_c be ordered by entailment.

Now we define a functional $F : \text{Def} \rightarrow \text{Def}_c \rightarrow \text{Def}_c$ in terms of $E : \text{Def}_c \rightarrow \text{Bod}_c \rightarrow \text{Bod}_c$ as follows:

$$F D C = (E C) \circ D$$

$$\begin{aligned}E C t &= t & \text{where } t &\in \mathbb{B} \\ E C v &= v & \text{where } v &\in \mathcal{V}\end{aligned}$$

$$E C (f v_1 \dots v_n) = C f v_1 \dots v_n$$

$$E C (c_1 \wedge c_2) = (E C c_1) \wedge (E C c_2)$$

$$E C (c_1 \vee c_2) = (E C c_1) \vee (E C c_2)$$

$$E C (\neg v) = \neg v$$

We say $C \in \text{Def}_c$ is a *closed form* for equation system $D \in \text{Def}$ iff C is a fixed point of $F D$, that is, if $C = F D C$. Note that due to the way negation is included in our definition of **Bod**, F is monotone, ensuring that $F D$ has fixed points. In what follows we shall take the closed form of $D \in \text{Def}$ to be $\text{lfp}(F D)$, although any fixed point will do, and our algorithm can be modified to find all fixed points.

For convenience we shall restrict the syntax of **Bod** to a specialised disjunctive normal form that separates closed parts of function definitions as follows:

$$\text{Bod} = \{f \vee d \mid f \in \text{Bod}_c \wedge d \in \text{Dis}\}$$

$$\text{Dis} = \text{Con} \cup \{d_1 \vee d_2 \mid d_1, d_2 \in \text{Dis}\}$$

$$\text{Con} = \{f \wedge c \mid f \in \text{Bod}_c \wedge c \in \text{Ap}\}$$

$$\text{Ap} = (\mathbb{F} \times \mathcal{V}^n) \cup \{c_1 \wedge c_2 \mid c_1, c_2 \in \text{Ap}\}$$

By distributivity, commutativity, and associativity of conjunction and disjunction, this new definition has equivalent expressiveness to the previous one. However, it makes our algorithms simpler.

3.3 A tabular view of equation solving

Consider the Boolean function definitions:

$$f(x, y) = (x \leftrightarrow y)$$

$$g(x, y) = (x \leftrightarrow y) \vee g(y, x)$$

$$h(x) = g(0, x)$$

Although these definitions do not fit the specified syntax, it is easy to rewrite them as equivalent definitions that do:

$$f : (v_1 \wedge v_2) \vee (\neg v_1 \wedge \neg v_2)$$

$$g : (v_1 \wedge v_2) \vee (\neg v_1 \wedge \neg v_2) \vee g(v_2, v_1)$$

$$h : \neg v_{n+1} \wedge g(v_{n+1}, v_1)$$

v_1	v_2	$g(v_1, \dots, v_n)$
0	0	1
0	1	$g(1, 0)$
1	0	$g(0, 1)$
1	1	1

v_1	$h(v_1, \dots, v_n)$
0	$g(0, 0)$
1	$g(0, 1)$

Table 1: Tabular view of example

v_1	v_2	$g(v_1, v_2)$	iteration		
			0	1	2
0	0	1	0	1	1
0	1	$g(1, 0)$	0	0	0
1	0	$g(0, 1)$	0	0	0
1	1	1	0	1	1

Table 2: Tabular Kleene iteration

Note that v_1 and v_2 are the first two formal parameters, and v_{n+1} is the smallest local variable, implicitly existentially quantified over the function body.

Table 1 presents the values of these functions for each combination of inputs. We have omitted f for brevity, as it is defined without reference to any functions, so its table is an ordinary truth table. Since g only involves parameter variables v_1 and v_2 , we need only consider 4 cases, and h only requires 2 cases. To handle the unconstrained variable v_{n+1} in the definition of h , we use the disjunction of the body with v_{n+1} given value 1, and with it given value 0.

These tables do not show a final solution to these equations, as they are not in closed form. However, we can use Kleene iteration to find a closed form, as shown in Table 2. Assuming we wish to find the least fixed point, we begin in iteration 0 by assigning the value 0 to all rows. In iteration 1, we compute the value for each row, using the values assigned in iteration 0 wherever the definition of a row refers to another row (or the same row). Iteration 2 repeats the process, using the values from row 1. In this example, iteration 1 is a fixed point, as confirmed in iteration 2.

3.4 Towards ROBDDs

The tabular approach taken above will work well for functions of low arity. However, for functions with dozens of arguments, it quickly becomes unworkable. In this subsection we shall reformulate the tabular approach to work on an ROBDD-like structure.

Our revision of the ROBDD structure is similar to our relaxation of the truth table: we allow Boolean function invocations, as well as 0 and 1, as sinks of the structure. Just as we did for the tabular approach, we take advantage of the fact that, at the sinks, the values of all relevant formal parameter variables are known. Thus the “formulae” we allow for sinks are in fact just references to other sinks in the structure. Figure 3 shows the example of Section 3.3 in this view.

The structure is an ordered binary decision tree down to its leaf nodes; however, leaf nodes may refer to one another, even cyclically. It is also ordered in the same sense as an OBDD. It is not reduced, as the destinations of links from a leaf node depend upon that node's position in the tree. We refer to this structure as a tree because of its underlying structure, although it is, strictly speaking, a directed graph.

In the style of an ROBDD we would like to be able to evaluate the function for a given input by following the arcs until we reach a terminal node. If we

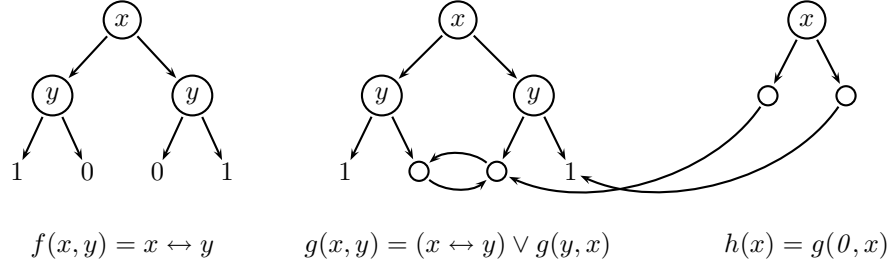


Figure 3: ROBDD-like views of recursive definitions

wish to evaluate the structure in Figure 3 for $g(0, 0)$ or $g(1, 1)$ then we have no problems. But when we attempt evaluation of $g(0, 1)$ we are referred to the value of $g(1, 0)$, and vice versa. Solving this problem is the focus of the remainder of this section. Before we can tackle that, however, we must generalise our data structure.

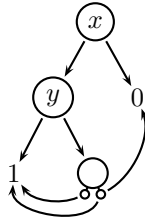
The example of Figure 3 does not show the full generality of function bodies permitted by **Bod**. In particular, it does not illustrate disjunctions of conjunctions of function invocations, nor does it make clear how to handle existentially quantified variables in function calls.

To handle disjunctions of conjunctions of function invocations, we generalise our definition of the DDS structure to specify that leaf nodes in the structure may be either 0, 1, or a disjunction of conjunctions of Boolean function invocations. We shall depict the new sink node graphically as an unlabelled node with a number of “pimples,” each of which refers to a number of other sinks in the structure. The intended meaning is that the pimpiled node should be interpreted as true iff at least one of the pimples are true. A pimple is considered to be true if all the nodes it refers to are true.

Consider the recursive definition:

$$f(x, y) = \begin{aligned} & x \wedge y \\ & \vee x \wedge f(y, x) \wedge f(x, x) \\ & \vee x \wedge f(x, x) \end{aligned}$$

This definition can be depicted as follows:



The pimpiled node in this example represents the expression $(1 \wedge 0) \vee 1 = 1$.

Existentially quantified variables are slightly harder to handle. A naive approach would be to treat an existentially quantified variable the same as any other, and simply extend the tree with new variables. While this approach would indeed work, it leaves the task of eliminating the existentially quantified variables.

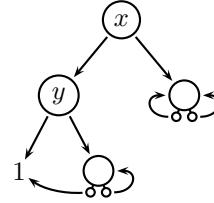
Since existentially quantified variables are all greater (in the variable ordering) than all formal parameter variables, they are always placed on the tree's fringe. Thus eliminating these variables is a matter of disjoining all the sinks below the greatest parameter variable. Since our representation of sinks explicitly allows disjunctions, handling existentially quantified

variables is simply a matter of coalescing the disjunctions that would appear anywhere under the greatest parameter variable into a single sink node.

Consider, for example, the function:

$$f(x, y) = (x \wedge y) \vee (\exists u. f(x, u))$$

We treat the $\exists u. f(x, u)$ expression as if it were (the equivalent) $f(x, 0) \vee f(x, 1)$, leading to this:



3.5 Dendritic decision structures

We can improve on the **TD** algorithm by exploiting properties specific to our Boolean domain. Two observations are crucial here. First, as the recursively defined objects are simply truth values, living in a domain of height one, once a variable's value changes, it never needs to be re-visited. Second, as a fringe node in the structure effectively is an expression in DNF, a simple bookkeeping technique allows us to determine an appropriate time to re-evaluate a variable. Ideally we would like to re-evaluate a variable only when that evaluation would trigger a change of value. The idea is that, when the number of conjuncts in each disjunct is high, we often find that re-evaluation of one conjunct does not change the value of the conjunction. If a variable's right hand side has d disjuncts with c conjuncts in each, then it is possible for all but one conjunct in each disjunct to become true without making the whole equation true. This means it may take $(c - 1)d + 1$ evaluations before the variable becomes true.

It would naturally be more efficient if we could cache the work we did during the first evaluation and re-use it, instead of repeating it. The third contribution of this paper is to propose a very simple data structure and corresponding algorithm that does this very efficiently.

We consider three possible states for a variable: Undetermined (\perp), definitely true (1), and definitely false (0). Evaluation follows the rules of Kleene's logic:

\neg	
\perp	\perp
0	1
1	0

\wedge	\perp	0	1
\perp	\perp	0	\perp
0	0	0	0
1	\perp	0	1

\vee	\perp	0	1
\perp	\perp	\perp	1
0	\perp	0	1
1	1	1	1

To implement these rules we introduce an efficient structure to represent a partially evaluated disjunction. A “pimple” off a node v becomes a *receptor*

which keeps count of the number of nodes that must be true before v becomes true. Dependency arrows become reversed, turning into *axons* pointing from nodes to receptors. We call the resulting structures *dendritic decision structures*, or simply DDSs. Their purpose is to speed up processing of cyclic dependencies discovered during evaluation.

Take the situation where a depends on b and b depends on a . If we start evaluation at a we discover a cycle when we request a 's value during evaluation of b . In the standard recursive algorithm we would record the information that b depends on a , and use that to remember that b must be re-evaluated if a 's value ever changes. In our new approach we create an axon at a and a receptor at b . The receptor has a value of 1, indicating that there is a conjunction in b with one and only one axon targeting it. Note that we only create a receptor for a node if the value of that variable depends upon as yet undetermined variables.

If the value of a changes to 1 then we follow the axon to the receptor, which is decremented. If and when a receptor reaches 0, all values in the conjunction in b represented by the receptor have become 1. In that case, b has been sufficiently stimulated to change and is given value 1.

3.6 The algorithm

Figure 4 gives the algorithm that uses DDSs.

The method uses a principle of aggressive propagation of truth. For each node x , a set of axons point to the nodes that depend on the value of x . More specifically, the axons point to receptors that represent conjunctions containing x . We use the notation $u \mapsto (v, c)$ for an axon that emanates from node u and points to the receptor c associated with node v . The procedure *fire* shows what happens when a node u has been determined to be 1: Every receptor (v, c) pointed to is decremented, and if and when it reaches 0, v is deemed to have value 1 as well. Solving is then complete as far as v is concerned, and the immediate consequences of v taking value 1 are pursued: v activates its axons.

As in the previous algorithm, a set *Called* keeps track of nodes for which an iteration has been started but not completed. Another set, *Evaluated*, keeps track of nodes that no longer need be considered. This, however, does not just mean nodes that have a final value 0 or 1. Once a node has been examined, it is not "solved for" again; it is considered "evaluated". However, it may still change its value from \perp , through appropriate activation of axons that point to the node.

3.7 An example

Figure 5 gives an example to illustrate the benefits of axons and receptors. If we begin evaluation at a and evaluate right-hand sides from left to right then we immediately find we must evaluate b , which sends us to c and then to d , which refers back to a . At this point a receptor is associated with d , starting with the value 1, and an axon is created at a pointing to that receptor. If a ever changes value later, we can immediately determine that d depends on it and update its value directly.

Continuing with evaluating d we reach the reference to b , also in the same conjunction. b is also under evaluation, so we create an axon on b pointing back to the same receptor and increase the value of the receptor to 2. If both a and b take the value 1 then the receptor will reach 0 and d will be set to 1.

```

procedure main
   $\sigma := \emptyset$ ; Called :=  $\emptyset$ ; Evaluated :=  $\emptyset$ ;
  foreach  $x \in V$  do solve( $x$ ) od
end

function solve( $x : V$ ) :  $D$ 
  /* Solve for variable  $x$  */
  if  $x \in \textit{Evaluated}$  then
    /*  $x$  has final value or is  $\perp$  and has receptors */
    return  $\sigma(x)$ 
  elseif  $x \in \textit{Called}$  then
    /*  $x$  is currently under evaluation */
    return  $\perp$ 
  else
    /*  $x$  has not been examined before */
    Called := Called  $\cup \{x\}$ ;
     $\sigma(x) := \textit{evalrhs}(x)$ ;
    Called := Called  $\setminus \{x\}$ ;
    Evaluated := Evaluated  $\cup \{x\}$ ;
    if  $\sigma(x) = 1$  then fire( $x$ ) fi;
    return  $\sigma(x)$ 
  fi
end

function evalrhs( $x : V$ ) :  $D$ 
  /* Evaluate right hand side of  $x$ 's definition */
  foreach disjunct  $d$  in rhs( $x$ ) do
    Uncertain :=  $\emptyset$ ;
    foreach conjunct  $c$  in  $d$  do
      if solve( $c$ ) =  $\perp$  then
        Uncertain := Uncertain  $\cup \{c\}$ 
      fi;
    od;
    if Uncertain  $\neq \emptyset$  then
      Create receptor ( $x, c$ ) with value  $| \textit{Uncertain} |$ ;
      foreach  $u \in \textit{Uncertain}$  do
        Create an axon  $u \mapsto (x, c)$ 
      od
    fi
  od;
  if all conjuncts of any disjunct have value 1 then
    return 1
  elseif  $x$  has receptors then
    return  $\perp$ 
  fi;
  return 0
end

procedure fire( $u : V$ )
  /* Activate all axons from node  $u$  */
  foreach axon  $u \mapsto (v, c)$  do
    Decrement  $c$ ;
    if  $c = 0$  then
       $\sigma(v) := 1$ ;
      Evaluated := Evaluated  $\cup \{v\}$ ;
      fire( $v$ )
    fi
  od
end

```

Figure 4: The algorithm based on DDSs

We do the same with the second conjunction in d 's right-hand side, creating a second axon at a and an axon at c , both pointing to another receptor with value 2. This concludes the evaluation of d . We will never evaluate d again, although we may visit it later as the result of an axon firing. The configuration of

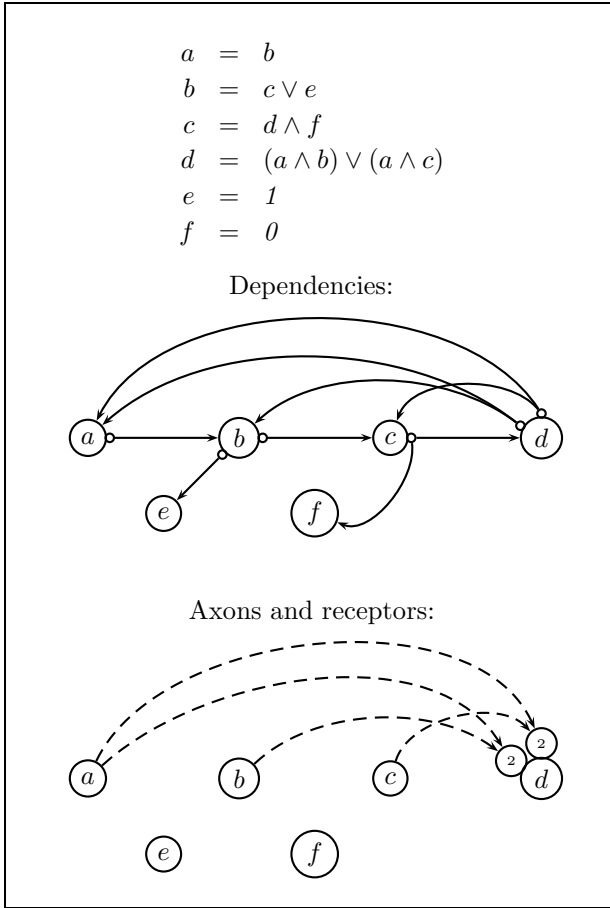
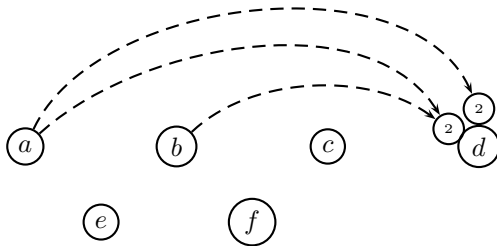


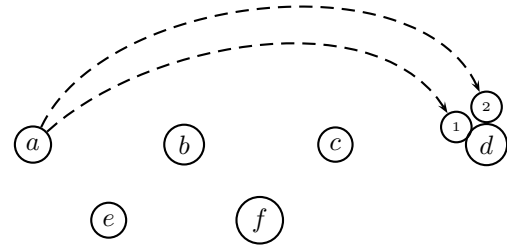
Figure 5: Use of axons and receptors

receptors and axons after processing of d is shown in figure 5. If we had started from a different definition, the configuration could be different. Note that in this figure and the following diagrams we omit the tree structure above the leaf nodes, as they do not contribute to the discussion.

Evaluation now returns to c . The value of d remains \perp for now. c requires f , which immediately evaluates to 0 , so c has value 0 also. Receptors and axons are now:



Evaluation of c has finished, so we return to evaluating b . This requires e which immediately evaluates to 1 . As the two variables that b depends on are disjoint, the value of b is 1 . Now the axon emanating from b is fired, reducing the value of its target receptor to 1 . No action is taken at d , as all receptors are still non-zero:

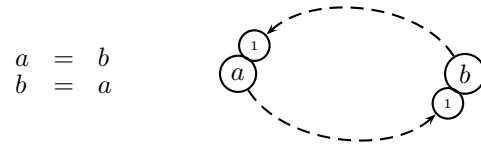


Finally, we return to evaluation of a . It depends only on b which now has value 1 , so a is 1 . This causes two axons to fire. The order of firing is not important, so let us consider the axons in the order in which they were created. The first axon's target receptor is decremented to 1 . The second axon's target receptor is decremented to 0 . This receptor represents a conjunction in which all conjuncts are 1 , so we can set the value of d to 1 . At this point we should activate d 's axons as well, but there are no axons for this node. Evaluation of a has finished, and the system is solved completely. The solution is:

$$\begin{aligned}
 a &= b = d = e = 1 \\
 c &= f = 0
 \end{aligned}$$

3.8 Unresolved axons

It is possible to be left with a cycle of axons and receptors when we finish evaluation of the initial variable. The simplest example of this involves just two variables.

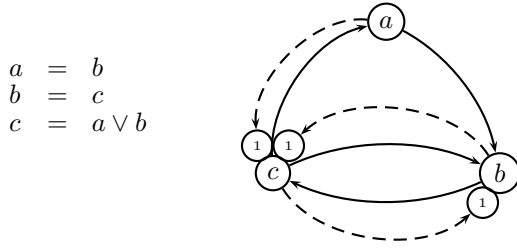


Clearly, this system has two solutions. In general, for n sets of variables which are linked by unresolved axons there are 2^n solutions to the system of equations. The sets of variables are SCCs, considering axons as edges. We obtain the least solution by setting all variables in all SCCs to 0 , but *any* solution can be generated easily at this point.

3.9 Fine-tuning the algorithm

We use shortcut Boolean evaluation within *evalrhs* for a considerable gain in efficiency. Once we have determined that a conjunct is 0 , there is no need to continue evaluation of other conjuncts within that disjunct. Likewise, once any disjunct is determined to be 1 , we can finish evaluation of the variable altogether.

If we have decided in advance that we want to find the *least* solution then we can avoid creating some axons and receptors. By incorporating Tarjan's SCC detection algorithm (Tarjan 1972) into our algorithm we can determine when we are at the "head" of an SCC—the first variable of the SCC we discovered (also the variable we finish evaluation of last). If a variable evaluating to \perp is known to be the head of an SCC then we may safely set it to 0 , as we will be doing that in the final phase in any case. It is also unnecessary to place any receptors on the head of an SCC as it always takes a final value. If we are lucky this will resolve all variables in the SCC to a final value. If we are unlucky we may get a case such as the following:



The diagram shows the state when a finishes evaluation. As a is the head of an SCC and its value is \perp , we set it to θ . This disables the receptor it points to, but cannot resolve the value of c . Likewise, b does not receive a final value.

To avoid the final phase of finding unresolved axons and setting them to θ then we can keep a record of all variables which were given receptors within each SCC. When we finish with the head of the SCC we can go through this list and set any variables which still have the value \perp to θ . This helps to reduce the amount of space needed by the algorithm, as fewer axons and receptors are in existence at any one time.

4 Experimental evaluation

We have evaluated the algorithm discussed above, including the optimisations discussed in Section 3.9, for groundness analysis over the domain *Pos* for Prolog programs (Armstrong, Marriott, Schachte & Søndergaard 1998). The analysis derives the possible groundness states on success of each predicate in a program, possibly consisting of many source files. All source files are read into memory simultaneously, but predicates are only analysed simultaneously when required. We have implemented the fixed point algorithm in C and the abstracter in Prolog, using an established ROBDD manipulation library (Schachte 1996), written in C, in both.

The computing/test environment used has the following characteristics: desktop system with X server running, a timer with 1ms resolution (not the standard 10ms), Debian GNU/Linux version 3.1, Linux kernel version 2.6.8, Intel Pentium 4 3.0GHz CPU with 1 MB cache, 1 GB memory.

Here are the characteristics of the benchmarking undertaken:

- Timings include only fixed point processing, not the translation to Boolean form.
- Each test was performed sufficiently often to consume 10 minutes, up to a maximum of 1000 repetitions, and the smallest time was taken.
- The majority of programs benchmarked had results under 2 ms for all algorithms. These programs are not included in the table as the results were considered too inaccurate.
- NT stands for a single repetition which took more than 60 seconds.

4.1 Challenging examples

Table 3 looks at the performance of the new algorithm on the “challenge” examples for Kleene iteration. These examples are difficult because they require the maximum number of iteration steps to find a fixed point — a number which grows exponentially in the number of arguments.

Table 3: Challenge benchmark results (ms)

Benchmark	Kleene	DDs
chain8	7.20	1.50
chain10	48.87	6.42
chain12	508.65	28.62
chain14	8382.47	130.52
chain16	NT	588.78
def8	8.56	0.87
def10	59.02	3.39
def12	569.65	15.10
def14	11362.96	66.41
def16	NT	295.14
challenge6	2.15	8.09
challenge8	18.84	150.27
challenge10	292.38	2767.88
challenge12	10099.08	48657.50

Table 4: Standard benchmark results (ms)

Benchmark	Kleene	DDs
reducer	2.11	2.82
press	2.13	2.26
ann	2.42	7.49
bryant	2.46	20.83
ili	2.61	2.78
qplan	2.71	3.29
parser_dcg	3.09	4.42
neural	3.54	3.68
scc1	3.73	4.00
ga	3.98	4.81
simple_analyzer	5.29	21.10
sim_v5-2	5.73	6.15
peval	7.51	9.17
chat_parser	13.08	240.96
chat	13.36	247.43
chat_80	39.37	345.94

Each program is parametrised over an integer. The minimum value for each parameter is that for which analysis took at least 2ms for at least one algorithm. For **chain** and **def**, the value of the parameter is also the number of argument variables in each predicate. For **challenge**, the number of argument variables is twice the value of the parameter.

The **chain** class of programs are due to Codish (1999). The **def** and **challenge** classes of programs are due to Genaim et al. (2001). We already met an example recursive definition arising from the **def** family in Section 1: the function p is the one that arises from **def4**.

Examining traces of execution with Valgrind (Nethercote & Seward 2003) reveals much about the behaviour of both algorithms. Of course, the cost of Kleene iteration grows quickly as the number of arguments is increased because the number of iteration steps grows exponentially. But to make matters worse, the size of the data structures grows with the number of arguments, so the cost of each iteration step also grows. The new algorithm does create and traverse dependency chains of exponential length (this appears to be unavoidable in some cases), however it only traverses each chain once.

The relative performance of the **challenge** class of programs is interesting to observe. For all sizes of **challenge** problems considered in our testing, Kleene iteration outperforms our algorithm. However, note

that for the higher arity cases, the time cost of Kleene iteration grows faster than that of our algorithm. For slightly higher arities, our algorithm will overtake Kleene iteration. In fact, for all three problem classes in Table 3, the cost of Kleene iteration grows substantially faster than the cost of our algorithm.

4.2 Standard benchmarks

Table 4 shows results for a number of small standard test cases. For these, Kleene iteration performs better or much better than our new algorithm. The new algorithm can be quite efficient, but it only takes one predicate with high arity or a large number of existentially quantified variables to make it impractical.

The examples for which our algorithm perform significantly slower merit closer examination. Analysis of the chat parser (`chat`, `chat_parser`, `chat.80`) was particularly expensive. Most of the analysis time was spent on an SCC which contains a predicate `possessive/14`. This predicate contains 8 local variables which are abstracted as existentially quantified variables. For each of the 2^{14} combinations of argument variables, there are 2^8 combinations of values for the existentially quantified variables, any of which could make the object variable take value T . Our algorithm checks each of these combinations, leading to poor performance in these cases. Clearly, for our approach to be practical for groundness analysis, it will be necessary to find ways to avoid complete exploration of all possible inputs and all possible valuations of existentially quantified variables.

5 Related work

One broad class of fixed point algorithms use a worklist as a basic structure. These algorithms usually include some method of detecting dependencies between variables. The general idea is to add variables into the worklist whenever a value they depend on has been changed. Variables are then selected from the worklist to be re-evaluated until the worklist is empty.

The simplest worklist algorithm adds *all* variables to the worklist when *any* variable changes value. This requires no knowledge of dependencies between variables and is trivially correct. However, if variables from the worklist are chosen in a poor order, the method may end up taking the maximum number of iterations. We can improve on this by making a preliminary pass over all the right hand sides, and finding which variables depend (statically) on other variables. For each variable x we want to know the set $infl(x)$, that is, the variables that depend on x . The algorithm of Kildall (1973) uses this idea. Many variants of this idea are possible, depending on policy for removal of worklist items, and the order in which elements are added. A good approach appears to be *eager evaluation* (Wunderwald 1995).

More efficient worklist algorithms track “dynamic” dependencies. The dependencies that can be read off a recursive definition over-approximate the actual dependencies, which may well change during evaluation. For example, raising the value of x from $x = F$ to $x = T$ in the expression $x \vee y$ removes its dependence on the value of y . “Dynamic” algorithms have such independence detection built in. Variants include the methods of Jørgensen (1994) and Vergauwen, Wauwman & Lewi (1994), as well the **W** algorithm of Fecht & Seidl (1999).

A clever alternative to worklist algorithms with dynamic dependency detection is offered by Le Charlier & Van Hentenryck (1992). Their work is based on the observation that when a variable has many dependencies, it is highly likely that the value we get will change during the computation. It would be better to evaluate those variables that do not depend on others first, followed by evaluating those which depend on variables which already have their final value, and so on. The method of Le Charlier & Van Hentenryck aims at maintaining the precision provided by dynamic dependency detection while at the same time processing variables in an optimal order. For a variable which has not been evaluated yet, the worklist solver uses the initial approximation \perp . Le Charlier & Van Hentenryck, on the other hand, suspend evaluation of the current variable and eagerly begin evaluation of the needed variable. This leads to the recursive algorithm shown in Figure 2. A further refinement is the **WRT** algorithm of Fecht & Seidl (1999).

Our algorithm is based on Le Charlier & Van Hentenryck (1992), but is otherwise incomparable to the methods discussed here, as it exploits properties of the Boolean domain to make shortcuts not available to a general algorithm.

Fecht & Seidl (1999) examine the choice of which equation to re-evaluate first in a system of recursively defined equations. They demonstrate that a worklist based solver which uses timestamps as a method of dynamic SCC detection is generally more efficient than the approach of Le Charlier and Van Hentenryck. As a measure of efficiency they use the number of evaluations of right hand sides. The complexity of each right hand side is not taken into account. The improved efficiency consists of differences in handling strongly connected components. These differences do not always result in an improvement however.

The issues discussed by Fecht and Seidl are not relevant to us. Once we have found an SCC using depth first search, we treat it as a single entity and solve it at once. At the valuation level, we examine each valuation only once, revisiting it only when required by an axon. This makes a comparison based on number of evaluations of right hand sides inappropriate.

Englebert, Le Charlier, Roland & Van Hentenryck (1993) introduce two optimisations which can be applied to many fixpointing algorithms. One is clause prefix dependency, and the other is caching of operations. The first improvement avoids re-evaluating a clause prefix when no abstract value on which it depends has been updated. The second improvement consists of caching all operations on substitutions and reusing the results whenever possible. They note that the second optimisation largely subsumes the first, as the caching eliminates most of the cost of evaluating clause prefixes. Analysis time was reduced by around 28% in a C implementation by these methods. Our approach of using DDSs effectively implements this caching optimisation.

Fecht & Seidl (1998) look at limiting the calculations needed to raise a particular value in the right hand side from its previous approximation to the new approximation. Instead of recalculating the right hand side from the new value, they calculate the right hand side for the difference between the new and old values, defined as $diff(d_{old}, d_{new}) = d_{diff}$ such that $d_{old} \sqcup d_{diff} = d_{new}$. The result is then joined with the result of the old calculation. In our setting the domain has only two values, so this optimisation is not useful.

6 Conclusion

We have presented a novel approach to the problem of finding closed forms of recursively defined Boolean functions. Our approach centers on the idea of solving these equations one valuation at a time. The advantage of this is that the solution for a single valuation is ultimately a Boolean value. This means that, since we are climbing an ascending chain of height one, once the solution for a valuation changes from an initial value of false, it will not change again.

To capitalise on this advantage, we have introduced a new data structure, the dendritic decision structure. This data structure captures the dependency relation among valuations, leading to an algorithm for finding closed forms that very efficiently handles problems requiring many iterations when solved by Kleene iteration.

This new algorithm, like Kleene iteration, suffers from very bad worst-case performance. Unfortunately, this worst-case performance occurs in our testing domain of groundness analysis of common programs for our algorithm, but only on rare or artificial cases for Kleene iteration. Therefore we do not yet consider this algorithm suitable for practical use in groundness analysis. Further work is needed to make it competitive in general. In particular, it will be necessary to find a way to avoid building and exploring the entire DDS structure for a function whenever possible. It will also be important to avoid exploring all valuations for existentially quantified variables whenever possible.

On the positive side, it is clear that a hybrid approach would give the best of both worlds. We can perform a few steps of Kleene iteration, which is enough to solve the majority of inputs. If this does not produce a solution, then we can use the last approximation from Kleene iteration as a starting point for our algorithm. This is easily accomplished by replacing the fixed part of the definitions of all functions in the SCC being solved with the corresponding current Kleene approximation. Based on the benchmark results, such an approach would be efficient for all cases except **challenge**, which is difficult for all algorithms.

Acknowledgements

We would like to thank Martin Harris and Adrian Teo for their contributions to early versions of the algorithms and data structures presented in this paper.

References

- Armstrong, T., Marriott, K., Schachte, P. & Søndergaard, H. (1998), 'Two classes of Boolean functions for dependency analysis', *Science of Computer Programming* **31**(1), 3–45.
- Brace, K., Rudell, R. & Bryant, R. (1990), Efficient implementation of a BDD package, in 'Proc. Twenty-seventh ACM/IEEE Design Automation Conf.', pp. 40–45.
- Bryant, R. (1992), 'Symbolic Boolean manipulation with ordered binary-decision diagrams', *ACM Computing Surveys* **24**(3), 293–318.
- Codish, M. (1999), 'Worst-case groundness analysis using positive Boolean functions', *Journal of Logic Programming* **41**(1), 125–128.
- Cousot, P. & Cousot, R. (1977), Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, in 'Proc. Fourth ACM Symp. Principles of Programming Languages', ACM Press, pp. 238–252.
- Englebert, V., Le Charlier, B., Roland, D. & Van Hentenryck, P. (1993), 'Generic abstract interpretation algorithms for Prolog: Two optimization techniques and their experimental evaluation', *Software Practice and Experience* **23**(4), 419–459.
- Fecht, C. & Seidl, H. (1998), Propagating differences: An efficient new fixpoint algorithm for distributed constraint systems, in C. Hankin, ed., 'Proc. ESOP'98', Vol. 1381 of *Lecture Notes in Computer Science*, Springer, pp. 90–104.
- Fecht, C. & Seidl, H. (1999), 'A faster solver for general systems of equations', *Science of Computer Programming* **35**(2–3), 137–161.
- Genaim, S., Howe, J. M. & Codish, M. (2001), 'Worst-case groundness analysis using definite Boolean functions', *Theory and Practice of Logic Programming* **1**, 611–615.
- Jørgensen, N. (1994), Finding fixpoints in finite function spaces using neededness analysis and chaotic iteration, in B. Le Charlier, ed., 'Static Analysis', Vol. 864 of *Lecture Notes in Computer Science*, Springer, pp. 329–345.
- Kildall, G. A. (1973), A unified approach to global program optimization, in 'Proc. First Annual ACM SIGACT-SIGPLAN Symp. Principles of Programming Languages', ACM Press, pp. 194–206.
- Le Charlier, B. & Van Hentenryck, P. (1992), A universal top-down fixpoint algorithm, Technical Report CS-92-25, Brown University, Department of Computer Science.
- Liaw, H.-T. & Lin, C.-S. (1992), 'On the OBDD-representation of general Boolean functions', *IEEE Transactions on Computers* **C-41**(6), 661–664.
- Nethercote, N. & Seward, J. (2003), 'Valgrind: A program supervision framework', *Electronic Notes in Theoretical Computer Science* **89**(2).
- Schachte, P. (1996), Efficient ROBDD operations for program analysis, in K. Ramamohanarao, ed., 'ACSC'96: Proceedings of the 19th Australasian Computer Science Conference', Australian Computer Science Communications, pp. 347–356.
- Tarjan, R. E. (1972), 'Depth-first search and linear graph algorithms', *SIAM Journal of Computing* **1**(2), 146–170.
- Vergauwen, B., Wauman, J. & Lewi, J. (1994), Efficient fixpoint computation, in B. Le Charlier, ed., 'Static Analysis', Vol. 864 of *Lecture Notes in Computer Science*, Springer, pp. 314–328.
- Wunderwald, J. E. (1995), Memoing evaluation by source-to-source transformation, in M. Proietti, ed., 'Logic Program Synthesis and Transformation', Vol. 1048 of *Lecture Notes in Computer Science*, Springer, pp. 17–32.

Learnability of Term Rewrite Systems from Positive Examples

M.R.K. Krishna Rao

Information and Computer Science Department
King Fahd University of Petroleum and Minerals,
Dhahran 31261, Saudi Arabia.
Email: krishna@ccse.kfupm.edu.sa

Abstract

Learning from examples is an important characteristic feature of intelligence in both natural and artificial intelligent agents. In this paper, we study learnability of term rewriting systems from positive examples alone. We define a class of linear-bounded term rewriting systems that are inferable from positive examples. In linear-bounded term rewriting systems, nesting of defined symbols is allowed in right-hand sides, unlike the class of flat systems considered in Krishna Rao [8]. The class of linear-bounded TRSs is rich enough to include many divide-and-conquer programs like addition, logarithm, tree-count, list-count, split, append, reverse etc.

1 Introduction

Starting from the influential works of Gold [5] and Blum and Blum [3], a lot of effort has gone into developing a rich theory about inductive inference and the classes of concepts which can be learned from both positive (examples) and negative data (counterexamples) and the classes of concepts which can be learned from positive data alone. The study of inferability from positive data alone is important because negative information is hard to obtain in practice – positive examples are much easier to generate by conducting experiments than the negative examples in general. In his seminal paper [5] on inductive inference, Gold proved that even simple classes of concepts like the class of regular languages cannot be inferred from positive examples alone. This strong negative result disappointed the scientists in the field until Angluin [1] has given a characterization of the classes of concepts that can be inferred from positive data alone and exhibited a few nontrivial classes of concepts inferable from positive data. This influential paper inspired further research on the inductive inference from positive data. Since then many positive results are published about inductive inference of logic programs and pattern languages from positive data (see a.o., [9, 2, 10, 7, 8]). To the best of our knowledge, inductive inference of term rewriting systems from positive data has not received much attention – [8] is the only publication on this topic so far.

In the last few decades, term rewriting systems have played a fundamental role in the analysis and implementation of abstract data type specifications, decidability of word problems, computability theory,

design of functional programming languages (e.g. Miranda), integration of functional and logic programming paradigms, and artificial intelligence – theorem proving and automated reasoning.

In this paper, we propose a class of linear-bounded term rewriting systems that are inferable from positive examples. Linear-bounded TRSs have a nice property that the size of redexes in an innermost derivation starting from a flat term t is bounded by the size of the initial term t . This property ensures that we only need to consider rewrite rules whose sides are bounded by the size of the examples in learning linear-bounded TRSs from positive data.

The class of linear-bounded TRSs is rich enough to include many divide-and-conquer programs like addition, logarithm, tree-count, list-count, split, append, reverse etc. The relation between the class of linear-bounded TRSs and the class of simple flat TRSs recently introduced in [8] is discussed in a later section. In particular, flat TRSs can define functions (like doubling), whose output is bigger in size than the input, which is not possible with linear-bounded TRSs. On the other hand, flat TRSs do not allow nesting of defined symbols in the rewrite rules, which means that we cannot define functions like reverse and quick-sort that can be defined by a linear-bounded TRS. Due to space restrictions, many proofs are omitted.

The rest of the paper is organized as follows. The next section gives preliminary definitions and results needed about inductive inference. In section 3, we define the class of linear-bounded TRSs and establish a few properties of them in section 4. The inferability of linear-bounded TRSs from positive data is established in section 5. The final section concludes with a discussion on open problems.

2 Preliminaries

We assume that the reader is familiar with the basic terminology of term rewriting and inductive inference and use the standard terminology from [6, 4] and [5, 9]. The alphabet of a first order language L is a tuple $\langle \Sigma, \mathcal{X} \rangle$ of mutually disjoint sets such that Σ is a finite set of function symbols and \mathcal{X} is a set of variables. In the following, $\mathcal{T}(\Sigma, \mathcal{X})$ denotes the set of terms constructed from the function symbols in Σ and the variables in \mathcal{X} . The size of a term t , denoted by $|t|$, is defined as the number of occurrences of symbols (except the punctuation symbols) occurring in it.

Definition 1 A *term rewriting system* (TRS, for short) \mathcal{R} is a pair (Σ, R) consisting of a set Σ of function symbols and a set R of rewrite rules of the form $l \rightarrow r$ satisfying:

- (a) l and r are first order terms in $\mathcal{T}(\Sigma, \mathcal{X})$,
- (b) left-hand-side l is not a variable and
- (c) each variable occurring in r also occurs in l .

Example 1 The following TRS defines multiplication over natural numbers.

$$\begin{aligned} a(0, y) &\rightarrow y \\ a(s(x), y) &\rightarrow s(a(x, y)) \end{aligned}$$

$$\begin{aligned} m(0, y) &\rightarrow 0 \\ m(s(x), y) &\rightarrow a(y, m(x, y)) \end{aligned}$$

Here, a stands for addition and m stands for multiplication. \diamond

Definition 2 A context $C[\dots]$ is a term in $\mathcal{T}(\Sigma \cup \{\diamond\}, \mathcal{X})$. If $C[\dots]$ is a context containing n occurrences of \diamond and t_1, \dots, t_n are terms then $C[t_1, \dots, t_n]$ is the result of replacing the occurrences of \diamond from left to right by t_1, \dots, t_n . A context containing precisely 1 occurrence of \diamond is denoted $C[\]$.

Definition 3 The rewrite relation $\Rightarrow_{\mathcal{R}}$ induced by a TRS \mathcal{R} is defined as follows: $s \Rightarrow_{\mathcal{R}} t$ if there is a rewrite rule $l \rightarrow r$ in \mathcal{R} , a substitution σ and a context $C[\]$ such that $s \equiv C[l\sigma]$ and $t \equiv C[r\sigma]$. We say that s reduces to t in one rewrite (or reduction) step if $s \Rightarrow_{\mathcal{R}} t$ and say s reduces to t (or t is reachable from s) if $s \Rightarrow_{\mathcal{R}}^* t$, where $\Rightarrow_{\mathcal{R}}^*$ is the transitive-reflexive closure of $\Rightarrow_{\mathcal{R}}$. The subterm $l\sigma$ in s is called a *redex*. A redex is an *innermost redex* if no proper subterm of it is a redex. A derivation $s \Rightarrow_{\mathcal{R}}^* t$ is an *innermost derivation* if each reduction step in it reduces an innermost redex.

Example 2 The following innermost derivation shows a computation of the value of the term $m(s(s(0)), s(s(s(0))))$ by the above TRS.

$$\begin{aligned} &m(s(s(0)), s(s(s(0)))) \\ &\Rightarrow a(s(s(s(0))), m(s(s(0)), s(s(s(0)))) \\ &\Rightarrow a(s(s(s(0))), a(s(s(s(0))), m(0, s(s(s(0))))) \\ &\Rightarrow a(s(s(s(0))), a(s(s(s(0))), 0)) \\ &\Rightarrow a(s(s(s(0))), s(a(s(s(0))), 0)) \\ &\Rightarrow a(s(s(s(0))), s(s(a(s(0)), 0))) \\ &\Rightarrow a(s(s(s(0))), s(s(s(a(0), 0)))) \\ &\Rightarrow a(s(s(s(0))), s(s(s(0)))) \\ &\Rightarrow s(a(s(s(0))), s(s(s(0)))) \\ &\Rightarrow s(s(a(s(0)), s(s(s(0))))) \\ &\Rightarrow s(s(s(a(0), s(s(s(0))))) \\ &\Rightarrow s(s(s(s(s(s(0)))))) \end{aligned}$$

This is one of the many possible derivations from $m(s(s(0)), s(s(s(0))))$. Since the system is both terminating and confluent, every derivation (innermost or not) from this term ends in the same final value $s(s(s(s(s(s(0))))))$. \diamond

Remark 1 The conditions (b) left-hand-side l is not a variable and (c) each variable occurring in r also occurs in l of Definition 1 avoid trivial nonterminating computations. If a rewrite rule $x \rightarrow r$ with a variable left-hand-side is present in a TRS, every term can be rewritten by this rule and hence no normal form exist resulting in nonterminating computations. If the right-hand-side r contains a variable y not present in the left-hand-side l of a rule $l \rightarrow r$ such that $r \equiv C[y]$, then the term l can be rewritten to $C[l]$ (substitution σ replacing the extra-variable by l) resulting in ever growing terms and obvious nontermination.

Definition 4 Let U and E be two recursively enumerable sets, whose elements are called *objects* and *expressions* respectively.

- A *concept* is a subset $\Gamma \subseteq U$.
- An *example* is a tuple $\langle A, a \rangle$ where $A \in U$ and $a = \text{true}$ or false . Example $\langle A, a \rangle$ is *positive* if $a = \text{true}$ and *negative* otherwise.

- A concept Γ is *consistent* with a sequence of examples $\langle A_1, a_1 \rangle, \dots, \langle A_m, a_m \rangle$ when $A_i \in \Gamma$ if and only if $a_i = \text{true}$, for each $i \in [1, m]$.
- A *formal system* is a finite subset $R \subseteq E$.
- A *semantic mapping* is a mapping Φ from formal systems to concepts.
- We say that a formal system R *defines* a concept Γ if $\Phi(R) = \Gamma$.

Definition 5 A *concept defining framework* is a triple $\langle U, E, \Phi \rangle$ of a universe U of objects, a set E of expressions and a semantic mapping Φ .

Definition 6 A class of concepts $C = \{\Gamma_1, \Gamma_2, \dots\}$ is an *indexed family of recursive concepts* if there exists an algorithm that decides whether $w \in \Gamma_i$ for any object w and natural number i .

Here onwards, we fix a concept defining framework $\langle U, E, \Phi \rangle$ arbitrarily and only consider indexed families of recursive concepts.

Definition 7 A *positive presentation* of a nonempty concept $\Gamma \subseteq U$ is an infinite sequence w_1, w_2, \dots of objects (positive examples) such that $\{w_i \mid i \geq 1\} = \Gamma$.

An *inference machine* is an effective procedure that requests an object as an example from time to time and produces a concept (or a formal system defining a concept) as a conjecture from time to time. Given a positive presentation $\sigma = w_1, w_2, \dots$, an inference machine IM generates a sequence of conjectures g_1, g_2, \dots . We say that IM *converges to* g on input σ if the sequence of conjectures g_1, g_2, \dots is finite and ends in g or there exists a positive integer k_0 such that $g_k = g$ for all $k \geq k_0$.

Definition 8 A class C of concepts is *inferable from positive data* if there exists an inference machine IM such that for any $\Gamma \in C$ and any positive presentation σ of Γ , IM converges to a formal system g such that $\Phi(g) = \Gamma$.

We need the following result of Shinohara [9] in proving our result.

Definition 9 A semantic mapping Φ is *monotonic* if $R \subseteq R'$ implies $\Phi(R) \subseteq \Phi(R')$. A formal system R is *reduced w.r.t.* $S \subseteq U$ if $S \subseteq \Phi(R)$ and $S \not\subseteq \Phi(R')$ for any proper subset $R' \subset R$.

Definition 10 A concept defining framework $\mathcal{C} = \langle U, E, \Phi \rangle$ has *bounded finite thickness* if

1. Φ is monotonic and
2. for any finite set $S \subseteq U$ and any $m \geq 0$, the set $\{\Phi(R) \mid R \text{ is reduced w.r.t. } S \text{ and } |R| \leq m\}$ is finite.

Theorem 1 (Shinohara [9])

If a concept defining framework $\mathcal{C} = \langle U, E, \Phi \rangle$ has bounded finite thickness, then the class

$$C^m = \{\Phi(R) \mid R \subseteq E, |R| \leq m\}$$

of concepts is inferable from positive data for every $m \geq 1$.

3 Linear-bounded Term Rewriting Systems

In the following, we partition Σ into set D of defined symbols that may occur as the outermost symbol of left-hand-side of rules and set C of constructor symbols that do not occur as the outermost symbol of left-hand-side of rules.

Definition 11 The set $D_{\mathcal{R}}$ of *defined* symbols of a term rewriting system $\mathcal{R}(\mathcal{F}, R)$ is defined as $\{\text{root}(l) \mid l \rightarrow r \in R\}$ and the set $C_{\mathcal{R}}$ of *constructor* symbols of $\mathcal{R}(\mathcal{F}, R)$ is defined as $\mathcal{F} - D_{\mathcal{R}}$.

To show the defined and constructor symbols explicitly, we may write the above rewrite system as $\mathcal{R}(D_{\mathcal{R}}, C_{\mathcal{R}}, R)$ and omit the subscript when such omission does not cause any confusion. The terms containing no defined symbols are called constructor terms, and we refer to the terms of the form $f(t_1, \dots, t_n)$ such that f is a defined symbol and t_1, \dots, t_n are constructor terms as level 1 terms. In this paper, we only consider constructor systems – left-hand sides are level 1 terms.

We need the following definition in the sequel.

Definition 12 An argument filter is a mapping π that assigns to every defined symbol of arity n , a list of argument positions $[i_1, \dots, i_k]$ such that $1 \leq i_1 < i_2 < \dots < i_k \leq n$.

Unlike the usual practice in termination (and context sensitive rewriting) literature, we use argument filters only for defined symbols and do not distinguish the case of $\pi(f)$ being a single argument.

The following notion of parametric size over constructor terms and level 1 terms is central to our results.

Definition 13 For a constructor term t , the *parametric size* $[t]$ of t is defined recursively as follows:

- if t is a variable x then $[t]$ is a linear expression x ,
- if t is a constant then $[t]$ is zero,
- if $t = f(t_1, \dots, t_n)$ then $[t]$ is a linear expression $1 + [t_1] + \dots + [t_n]$.

For a level 1 term $t \equiv f(t_1, \dots, t_n)$, the *parametric size* $[t]$ of t is defined as $[t_{i_1}] + \dots + [t_{i_k}]$ when $\pi(f) = [i_1, \dots, i_k]$.

Example 3 The parametric sizes of constructor terms \mathbf{a} , $\mathbf{h}(\mathbf{a}, \mathbf{x}, \mathbf{b})$, $\mathbf{h}(\mathbf{g}(\mathbf{a}), \mathbf{g}(\mathbf{g}(\mathbf{x})), \mathbf{g}(\mathbf{y}))$ are 0 , $x + 1$, $5 + x + y$ respectively. The parametric size of level 1 term $\mathbf{f}(\mathbf{g}(\mathbf{a}), \mathbf{g}(\mathbf{g}(\mathbf{x})), \mathbf{g}(\mathbf{y}))$ with argument filter $\pi(f) = [1, 3]$ is $[g(a)] + [g(y)] = 1 + 1 + y = 2 + y$. \diamond

The following function **LIgen** generates a set of equations and two sets of linear inequalities from a given rewrite rule $l \rightarrow r$ in a constructor system and an argument filter π (note that π is used by this function implicitly through Def. 13). It uses fresh variables $\text{Var}_1, \text{Var}_2, \dots$ which do not occur in the rewrite system under consideration.

function LIgen($l \rightarrow r, \pi$);

begin

$EQ := \phi$; $LI_1 := \phi$; $LI_2 := \phi$;

$i := 0$; /* counter for fresh variables. */

while r contains defined symbols **do**

begin

Let $r \equiv C[u_1, \dots, u_m]$, showing all the level 1 subterms of r ;

$r := C[\text{Var}_{i+1}, \dots, \text{Var}_{i+m}]$;

$EQ := EQ \cup \{\text{Var}_{i+1} = u_1, \dots, \text{Var}_{i+m} = u_m\}$;

for $j := 1$ **to** m **do**

begin

$\text{ineq1}_{i+j} := [u_j] \geq \text{Var}_{i+j}$;

$\text{ineq2}_{i+j} := [l] \geq [u_j]$

end

$LI_1 := LI_1 \cup \{\text{ineq1}_{i+1}, \dots, \text{ineq1}_{i+m}\}$;

$LI_2 := LI_2 \cup \{\text{ineq2}_{i+1}, \dots, \text{ineq2}_{i+m}\}$;

$i := i + m$

end;

$LI_2 := LI_2 \cup \{\text{ineq2}_0 : [l] \geq [r]\}$;

end;

The above function **LIgen** introduces one fresh variable (and one equation in EQ and one inequality each in LI_1 and LI_2) corresponding to each defined symbol in the right-hand side term r of the rule $l \rightarrow r$. If a defined symbol f occurs above another defined symbol g in r and variables Var_i and Var_j correspond to f and g respectively, then $i > j$. The set EQ of equations and the numbering of inequalities are only needed in the proofs in the sequel.

Now, we are in a position to define the class of linear-bounded TRSS.

Definition 14 Let \mathcal{R} be a constructor system and π be an argument filter. Then, \mathcal{R} is a linear-bounded system w.r.t. π if each rule in it is linear-bounded w.r.t. π . A rewrite $l \rightarrow r$ is linear-bounded w.r.t. π if the inequalities in LI_1 imply each inequality in LI_2 , where LI_1 and LI_2 are the sets of inequalities generated by the function **LIgen**($l \rightarrow r, \pi$).

A constructor system is linear-bounded if it is linear-bounded w.r.t. some argument filter π .

Remark 2 The validity of (linear) inequalities is traditionally defined as the follows: *the inequality expression1 \geq expression2 is valid if and only if it is valid for all possible assignments of values to variables in it.* In the sequel, we only talk of sizes which are obviously non-negative and hence *the inequality expression1 \geq expression2 is valid if and only if it is valid for all possible assignments of non-negative values to variables in it.* According to this, $X + 1 > X$ is valid but $X + Y > X$ is not valid because Y can take a zero value and $X + 0$ is not greater than X . Similarly, $2X > X$ is not valid because X can take a zero value. However, both $X + Y \geq X$ and $2X \geq X$ are valid.

The following examples illustrate the concept of linear-bounded systems. We use short notations $LI_1(l \rightarrow r)$ and $LI_2(l \rightarrow r)$ to denote the inequalities generated by **LIgen**($l \rightarrow r, \pi$), when π is clear from the context (and write LI_1 and LI_2 when the rule is also clear).

Example 4 Consider the following constructor system reversing a list.

```
app(nil, y) → y
app(cons(x, z), y) → cons(x, app(z, y))
rev(nil) → nil
rev(cons(x, z)) → app(rev(z), cons(x, nil))
```

We show this system to be linear-bounded w.r.t. argument filter π such that $\pi(\text{app}) = [1, 2]$ and $\pi(\text{rev}) = [1]$. For the first rule, $LI_1 = \phi$ and $LI_2 = \{y \geq y\}$. Since $y \geq y$ is a valid inequality, LI_1 obviously implies LI_2 and hence this rule is linear-bounded. Similarly, the third rule can be easily shown to be linear-bounded (with $LI_1 = \phi$ and $LI_2 = \{0 \geq 0\}$).

For the second rule, $LI_1 = \{z + y \geq \text{Var}_1\}$ and $LI_2 = \{x + y + z + 1 \geq z + y, x + y + z + 1 \geq x + \text{Var}_1 + 1\}$. The first inequality $x + y + z + 1 \geq z + y$

in LI_2 is a valid inequality and the second inequality $x + y + z + 1 \geq x + \text{Var}_1 + 1$ in LI_2 is implied by the inequality $z + y \geq \text{Var}_1$ in LI_1 . Therefore, this rule is linear-bounded.

For the fourth rule, $LI_1 = \{z \geq \text{Var}_1, \text{Var}_1 + x + 1 \geq \text{Var}_2\}$ and $LI_2 = \{x + z + 1 \geq z, x + z + 1 \geq \text{Var}_1 + x + 1, x + z + 1 \geq \text{Var}_2\}$. The first inequality $x + z + 1 \geq z$ in LI_2 is a valid inequality, the second inequality $x + z + 1 \geq \text{Var}_1 + x + 1$ in LI_2 is implied by the inequality $z \geq \text{Var}_1$ in LI_1 , and the third inequality $x + z + 1 \geq \text{Var}_2$ in LI_2 is implied by the two inequalities $z \geq \text{Var}_1$ and $\text{Var}_1 + x + 1 \geq \text{Var}_2$ in LI_1 . Therefore, this rule is linear-bounded too. \diamond

4 Some Properties of Linear-bounded Systems

In this section, we prove some properties of linear-bounded systems. A nice property of the class of linear-bounded systems is that it is decidable whether a given TRS is linear-bounded or not, as this problem can be reduced to the satisfiability problem of linear inequalities.

Theorem 2 *It is decidable whether a TRS \mathcal{R} is linear-bounded or not.*

The following theorem captures the basic idea of linear-bounded systems – the size of output is bounded by the size of input.

Theorem 3 Let \mathcal{R} be a linear-bounded TRS and t be a level 1 term with root in D . If $t \Rightarrow^* v$ is an innermost derivation and v is a constructor term (i.e., a normal form), then the parametric sizes of t and v satisfy the property $[t] \geq [v]$.

Further, the size of any innermost redex in the above derivation is bounded by the size of the initial term.

Theorem 4 Let \mathcal{R} be a linear-bounded TRS and t be a level 1 term with root in D . If $t \Rightarrow^* u$ is an innermost derivation such that w is an innermost redex in u , then the parametric sizes of t and w satisfy the property $[t] \geq [w]$.

The above characteristic properties of linear-bounded TRSs ensure that it is decidable whether a flat term t reduces to a constructor term u by a linear-bounded system or not.

Theorem 5 *If t is a level 1 term with root in D , u is a constructor term and R is a linear-bounded TRS, it is decidable whether $t \Rightarrow_R^* u$ or not.*

5 Inferability of linear-bounded TRSs from Positive Data

In this section, we establish inductive inferability of linear-bounded TRSs from positive data.

Definition 15 Let LB_k be the set of all linear-bounded rules of the form $l \rightarrow r$ such that $|l| + |r| \leq k$, FC be the cartesian product of (a) the set of all level 1 terms with root in D and (b) the set of all constructor terms, and Φ be a semantic mapping such that $\Phi(R)$ is the relation $\{(s, t) \mid s \Rightarrow_R^* t, s \text{ is a level 1 term with root in } D \text{ and } t \text{ is a constructor term}\}$. The concept defining framework $\langle LB_k, FC, \Phi \rangle$ is denoted by \mathbf{LBF}_k .

The following Lemma follows from Theorems 5 and 2.

Lemma 1 The class of rewrite relations defined by linear-bounded TRSs is an indexed family of recursive concepts.

The following theorem plays the predominant role in proving our main result.

Theorem 6 *The concept defining framework $\mathbf{LBF}_k = \langle LB_k, FC, \Phi \rangle$ has bounded finite thickness.*

Proof: Since Φ is the rewrite relation, it is obviously monotonic, i.e., $\Phi(R_1) \subseteq \Phi(R_2)$ whenever $R_1 \subseteq R_2$.

Consider a finite relation $S \subseteq FC$ and a TRS $R \subseteq LB_k$ containing at most $m \geq 1$ rules such that R is reduced w.r.t. S . Let n be an integer such that $n \geq [t]$ for every $(t, u) \in S$. Let S' be the set of innermost redexes in innermost derivations $t \Rightarrow^* u$ such that $(t, u) \in S$. By Theorem 4, $n \geq [w]$ for every term $w \in S'$. Since R is reduced w.r.t. S , every rule in R is used in derivations of S . Hence, $n \geq [l]$ for every rule $l \rightarrow r \in R$.

Since Σ is finite, there are only finitely many linear-bounded TRSs containing at most m rules of the form $l \rightarrow r$ such that $|l| \geq [l]$ and $|l| + |r| \leq k$ (except for the renaming of variables)¹. Therefore, the set $\{\Phi(R) \mid R \text{ is reduced w.r.t. } S \text{ and contains at most } m \text{ rules}\}$ is finite. Hence, the concept defining framework $\langle LB_k, FC, \Phi \rangle$ has bounded finite thickness. \diamond

From this Theorem, Lemma 1 and Theorem 1, we obtain our main result.

Theorem 7 *For every $m, k \geq 1$, the class of linear-bounded TRSs with at most m rules of size at most k is inferable from positive data.*

6 Discussion

In this paper, we study inductive inference of term rewriting systems from positive data. A class of linear-bounded TRSs *inferable from positive data* is defined. This class of TRSs is rich enough to include divide-and-conquer programs like addition, logarithm, tree-count, list-count, split, append, reverse etc. To the best of our knowledge, only known results about inductive inference of term rewriting systems from positive data are from [8], where the class of simple flat² TRSs is shown to be inferable from positive data. The classes of simple flat TRSs and linear-bounded TRSs are incomparable for the following reasons.

1. Linear-bounded TRSs only capture functions whose output is bounded by the size of the inputs. Functions like addition, list-count, split, append, reverse have such a property. But functions like multiplication and doubling are beyond linear-bounded TRSs as the size of their output is bigger than that of the input. The following simple flat TRS computes the double of a given list (output contains each element of the input twice as often).

```
double(nil) → nil
double(cons(H, T)) → cons(H, cons(H, double(T)))
```

¹Since argument filters –and hence the parametric linear– ignore some arguments, the additional condition $|l| + |r| \leq k$ is needed. In [8], no argument filters are used and hence this additional condition is not needed.

²A TRS is simple flat if defined symbols are not nested in any rule and the sum of the sizes of arguments of defined symbols in the right-hand side of a rule is bounded by the sum of the sizes of arguments of defined symbols in the left-hand side [8].

This shows that there are functions that can be computed by simple flat TRSs but not by linear-bounded TRSs.

2. The rewrite system for computing reverse of a list given in Example 4 is linear-bounded, but it is beyond simple flat TRSs as it involves nesting of defined symbols. This shows that there are functions that can be computed by linear-bounded but not by simple flat TRSs.

Open problem: In view of this incomparability of the simple flat TRSs and linear-bounded TRSs, it will be very useful to work towards extending the frontiers of inferable classes of rewrite systems and characterize some classes of TRSs having the expressive power of both simple flat TRSs and linear-bounded TRSs, and yet inferable from positive data.

Acknowledgements: The author would like to thank the King Fahd University of Petroleum and Minerals for the generous support provided by it in conducting this research.

References

- [1] D. Angluin (1980), *Inductive inference of formal languages from positive data*, Information and Control **45**, pp. 117-135.
- [2] H. Arimura and T. Shinohara (1994), *Inductive inference of Prolog programs with linear data dependency from positive data*, Proc. Information Modelling and Knowledge Bases V, pp. 365-375, IOS press.
- [3] L. Blum and M. Blum (1975), *Towards a mathematical theory of inductive inference*, Information and Control **28**, pp. 125-155.
- [4] N. Dershowitz and J.-P. Jouannaud (1990), *Rewrite Systems*, In J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science*, Vol. B, pp. 243-320, North-Holland.
- [5] E.M. Gold (1967), *Language identification in the limit*, Information and Control **10**, pp. 447-474.
- [6] J.W. Klop (1992), *Term Rewriting Systems*, in S. Abramsky, D. Gabby and T. Maibaum (ed.), *Handbook of Logic in Computer Science*, Vol. 1, Oxford Press, 1992.
- [7] M. R. K. Krishna Rao (2000), *Some classes of prolog programs inferable from positive data*, Theor. Comput. Sci. **241**, pp. 211-234.
- [8] M. R. K. Krishna Rao (2004), *Inductive Inference of Term Rewriting Systems from Positive Data*, in Proc. of Algorithmic Learning Theory, ALT'04, Lecture Notes in Artificial Intelligence **3244**, pp. 69-82, Springer-Verlag.
- [9] T. Shinohara (1991), *Inductive inference of monotonic formal systems from positive data*, New Generation Computing **8**, pp. 371-384.
- [10] Takeshi Shinohara, Hiroki Arimura (2000), *Inductive inference of unbounded unions of pattern languages from positive data*, Theor. Comput. Sci. **241**, pp. 191-209.
- [11] Y. Toyama (1987), *Counterexamples to termination for the direct sum of term rewriting systems*, Inf. Process. Lett. **25**, pp. 141-143.

On-demand Bounded Broadcast Scheduling with Tight Deadlines*

Chung Keung Poon¹

Feifeng Zheng²

Yinfeng Xu²

¹ Department of Computer Science, City University of Hong Kong, Hong Kong

² School of Management, Xi'an Jiaotong University, Xi'an, Shaanxi, China

Email: ckpoon@cs.cityu.edu.hk,

zhengff@mailst.xjtu.edu.cn, yfxu@mail.xjtu.edu.cn

Abstract

We investigate a scheduling problem motivated by pull-based data delivering systems where there is a server keeping a number of pages; and clients requesting the same page can be satisfied simultaneously by one broadcast.

The HEU algorithm of Woeginger (1994) is proven to be optimal in maximizing the number of satisfied requests when the pages have equal length and the requests have tight deadlines. However, we show that when there are maximum bounds on the number and weight of requests at any time in the system, the HEU algorithm is not optimal. We then propose a modified algorithm, VAR, which is optimal for this case.

Keywords: Competitive analysis; Broadcast scheduling; Tight deadline.

1 Introduction

In this paper, we study an on-line scheduling problem motivated by pull-based broadcasting systems. In such a system, there is a server with a number of pages to be broadcasted to clients upon requests. Each request r arrives at an arbitrary time $a(r)$ and requests for some page $p(r)$. The request is satisfied if page $p(r)$ is broadcasted completely before the deadline $d(r)$. Since multiple requests to the same page can be satisfied simultaneously in a single broadcast, the challenge here is to determine the schedule, i.e., which page to broadcast at what time, so as to satisfy as many simultaneous requests as possible. More precisely, our objective is to maximize the number of satisfied requests. Furthermore, we assume that the requests are given in an on-line fashion: the attributes of a request r are only known at the time of arrival $a(r)$. Before that, nothing about r (not even its existence) is known. Thus we need to design an on-line algorithm which makes decisions on the fly as the requests come. An on-line algorithm A is said to be c -competitive if on every input, the number of satisfied requests in the schedule produced by an optimal offline algorithm, denoted by OPT, is at most c times that of A .

It is easy to see that without pre-emption, an on-line algorithm A cannot have a bounded competitive ratio. Imagine A is broadcasting a page for a set J of

requests when another set R of $\alpha|J|$ new requests for another page with tight deadlines arrive. If A does not abort J , OPT will broadcast the page requested by R and no other requests come later. Thus, the number of requests satisfied by OPT is α times that satisfied by A . Since α can be arbitrarily large, A does not have a bounded competitive ratio. In this paper, we consider the *preemption-restart model* in which the server may pre-empt a broadcast before its completion and the pre-empted request can only be satisfied by broadcasting the requested page from the beginning again.

Previously, Jiang and Vaidya (1998) studied the problem assuming knowledge of the probability distribution of the requests. Kim and Chwa (2003) were the first to design algorithms with provable worst case performance bounds without assuming such knowledge. When pages are of equal length, they presented an online algorithm (called AC) and proved it to be 5.828-competitive. Using a tighter analysis, Chan et al (2004) showed that their GREEDY algorithm is 5-competitive for the general case and 4-competitive for the special case where all requests have tight deadlines. (The GREEDY algorithm is actually the AC algorithm with a modified parameter which we called the *abortion ratio*.)

This special case of tight deadlines is especially interesting. Here, a request r must be served immediately upon its arrival or else it can never be satisfied. This case is actually related to the *weighted interval scheduling problem* in which the input is a set of weighted intervals with fixed starting and ending points and the goal is to schedule the intervals so that no two scheduled intervals overlap and the weighted sum of scheduled intervals is maximized. By viewing the number of requests to the same page in our broadcast scheduling problem as the weight of an interval in the weighted interval scheduling problem, it is easy to see that algorithms for on-line interval scheduling can be applied to our problem. For this problem, Woeginger (1994) proposed the HEU algorithm (which is the same as GREEDY) and proved it to be 4-competitive. Furthermore, he showed that it is optimal by giving a matching lower bound for any deterministic algorithm.

To overcome the lower bound of 4, Miyazawa and Erlebach (2004) considered randomization. In particular, they designed a randomized algorithm called VirtualWeight that has an expected competitive ratio of 3 when the weights are monotonic non-decreasing and the intervals have agreeable endpoints, i.e., intervals that have earlier start point will have earlier end point. This leads to the following question: Are there other special cases of the problem in which we can obtain an on-line algorithm with competitive ratio less than 4?

In many situations, various kinds of information to be broadcasted may be of different importance, or the

*This research was fully supported by a grant from the Research Grants Council of the Hong Kong Special Administration Region, China [Project No. City U 1164/04E] and by NSF of China under grants 10371094 and 70471035.

Copyright ©2006, Australian Computer Society, Inc. This paper appeared at Computing: The Australasian Theory Symposium (CATS2006), Hobart, Australia. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 51. Barry Jay and Joachim Gudmundsson, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.

usefulness of the same information varies for different requesters or at distinct time periods. To model this situation, we assume that the requests are weighted to represent their different importance. Moreover, there may be an upper bound on the number of requests that can be satisfied by one broadcast in some situations. For example, it can be due to the limitation of the physical system, or simply because the number of clients in a system is bounded. Based on the above two factors, if the weight of an arbitrary request has a bounded range, the maximum total weight gained in a broadcast (i.e., the sum of the weight of every request satisfied in that broadcast) is also bounded. We denote by M such an upper bound.

1.1 Our Results

In this paper, we show that when each page has equal length and there is an upper bound M on the satisfied weight in one broadcast, the deterministic algorithm, HEU, has competitive ratio less than 4. Note that the lower bound of Woeginger applies to the unit length job case as well. Thus, the upper bound on the total weight of satisfied requests in one broadcast is essential for achieving a competitive ratio of less than 4. Furthermore, we show that HEU is actually not an optimal algorithm in this case. We show that there exists an on-line algorithm with competitive ratio arbitrarily close to c^* where c^* is a value depending on M . The main novelty in our algorithm is that it adaptively changes its abortion ratio according to the recent history. We also show a matching lower bound for any deterministic algorithm. Thus our algorithm is optimal for this case. To implement the algorithm, we need to compute the value of c^* . Unfortunately, we are not able to derive a closed-form formula for c^* in terms of M . For different values of M , we compute numeric values on c^* and compare them with those of HEU.

1.2 Related Work

Broadcast scheduling is such an important problem that various different models have been studied in the literature. The one closest to ours seems to be that of Kalyanasundaran and Velauthapillai (2003) which assumes that the server has a number of files partitioned into pages of equal length and a client can receive a file as long as the pages are sent in a cyclic order. In another model, the server is allowed to break down a page into different parts and send them over the network simultaneously. Yet, some may assume that the receivers have certain amount of buffer so that part of a page previously broadcasted can be cached. All these are not allowed in our model.

For our model, most of the previous works concentrate on minimizing the flow time where the flow time of a request is the time elapsed between its arrival and its completion. For example, Acharya and Muthukrishnan (1998) and Edmonds and Pruhs (2002) studied the problem of minimizing the total flow time while Bartal and Muthukrishnan (2000) studied the minimization of the maximum flow time. Aksoy and Franklin (1998) presented a practical parameterized algorithm and evaluated it with extensive experiments.

The rest of the paper is organized as follows. Section 2 gives some basic notations and preliminaries on our problem. In section 3 we analyze the competitiveness of HEU, and in section 4 we present and analyse our heuristic algorithm VAR. We prove a lower bound of competitive ratio in section 5 and present some numeric computations on the competitive ratios of HEU and VAR in section 6. Finally, section 7 concludes this paper.

2 Notations and Preliminaries

Since all the pages have equal length, we assume without loss of generality that each page has unit length. Given an input I (a set of requests) and an on-line algorithm A , we denote by $S_A(I)$ and $S^*(I)$ the schedules produced by A and by an optimal off-line algorithm on I respectively. When A and I are understood from the context, we will simply denote the schedules by S and S^* respectively.

A schedule S is a sequence of broadcasts (J_1, J_2, \dots) where each broadcast J_i is a set of requests to the same page started being served at time $s(J_i)$. The broadcasts are indexed such that $s(J_i) < s(J_{i'})$ for $i < i'$. Denote by $|J_i|$ the total weight of requests in J_i . For convenience, assume that each request has weight at least 1 and hence $|J_i| \geq 1$. If $s(J_i) + 1 > s(J_{i+1})$, then the broadcast J_i is *aborted* by J_{i+1} ; otherwise J_i is said to be *completed*. We denote by $|S|$ the total weight of satisfied requests in the schedule S , i.e., we only count those completed requests.

To analyze a schedule produced by an arbitrary algorithm A , we often find it convenient to partition the schedule into *basic subschedules*. A basic subschedule is a maximal sequence of broadcasts $\delta = (J_0, J_1, \dots, J_m)$ such that for all $0 \leq i \leq m-1$, J_i is aborted by J_{i+1} and only J_m is completed. Since the sequence is maximal, there is either an idle interval or a completed broadcast immediately before J_0 . Note that m may equal 0 if there is no aborted broadcasts in δ .

3 The HEU Algorithm

In this section we will analyse the performance of HEU. For simplicity, we assume $M = 2^k$ for some integer $k \geq 1$.

Suppose HEU is broadcasting a page for a set J of requests when new requests arrive and let R be the request set with the largest weight. (If there are more than one pages having requests with the same weight, ties are broken arbitrarily.) Then HEU aborts J if and only if R is at least twice the size of J . (We say that HEU has abortion ratio 2.) Note that there is no pending requests since the requests have tight deadlines. Also, aborted requests will never be satisfied.

Theorem 3.1 *Suppose each broadcast can satisfy a total weight of at most $M = 2^k$ for some positive integer k . Then HEU is $4(1 - 1/M)$ -competitive.*

Proof. The proof technique is similar to that of Chan et al. (2004). Consider a basic subschedule $\delta = (J_0, J_1, \dots, J_m)$ produced by HEU. Recall that J_i ($0 \leq i \leq m-1$) is aborted while J_m is completed by HEU. Thus, HEU gains $|J_m|$ in this basic subschedule.

For OPT, denote by O_i the broadcast OPT starts when HEU is serving J_i ($0 \leq i \leq m$), i.e., O_i is the broadcast started by OPT in the interval $[s(J_i), s(J_{i+1}))$ for $i < m$ and in $[s(J_i), s(J_i) + 1)$ for $i = m$. Let $|O_i|$ be the total weight of requests in O_i . Since OPT has complete knowledge of all the requests, we assume that all its broadcasts will run to completion. Thus, OPT can gain $\sum_{i=0}^m |O_i|$ within this basic subschedule. On the other hand, requests in J_i can never be satisfied later by OPT due to their tight deadlines. We will show that $\sum_{i=0}^m |O_i|$ is at most $4(1 - 1/M)$ of $|J_m|$. Clearly, this implies the same overall competitive ratio.

By the construction of HEU, $|J_i| \leq \frac{1}{2}|J_{i+1}|$ ($0 \leq i \leq m-1$) and hence $|J_m| \geq 2^{m-i}|J_i| \geq 2^m|J_0|$.

On the other hand, we have $|O_i| < 2|J_i| \leq |J_{i+1}|$, or else HEU will abort J_i to start O_i , contradicting that J_i is aborted by J_{i+1} . Thus, $|O_i| < (\frac{1}{2})^{m-i-1}|J_m|$ and hence

$$\begin{aligned} \sum_{i=0}^m |O_i| &< \sum_{i=0}^{m-1} (\frac{1}{2})^{m-i-1} |J_m| + |O_m| \\ &= 2[1 - (\frac{1}{2})^m] |J_m| + |O_m|. \end{aligned}$$

Since $|J_0| \geq 1$, we have $|J_m| \geq 2^m |J_0| \geq 2^m$. On the other hand, $|J_m|$ is no more than the upper bound $M = 2^k$. Hence, $m \leq k$. We will discuss the relationship of m and k in two cases.

(Case 1: $m = k$). In this case J_m is J_k . Since $2^k \leq |J_k| \leq 2^k$, we have $|J_k| = 2^k = M$ and then $|O_k| \leq M = |J_k|$. Thus, $\sum_{i=0}^k |O_i| < \{2[1 - (\frac{1}{2})^k] + 1\} |J_k| \leq 4[1 - (\frac{1}{2})^k] |J_k|$.

(Case 2: $m \leq k - 1$). Since O_m will not abort J_m , $|O_m| \leq 2|J_m|$ holds. Then, $\sum_{i=0}^m |O_i| < \{2[1 - (\frac{1}{2})^m] + 2\} |J_m| \leq \{2[1 - (\frac{1}{2})^{k-1}] + 2\} |J_m| = 4[1 - (\frac{1}{2})^k] |J_m|$.

In both cases, $\sum_{i=0}^m |O_i|/|J_m| < 4[1 - (\frac{1}{2})^k]$, that is, HEU is $4(1 - 1/M)$ -competitive. ■

4 The VAR Algorithm

In this section we will present and analyze our algorithm, called VAR, which is a heuristic depending on the upper bound M .

Suppose VAR is broadcasting a page for the set J of requests when new requests arrive. Let R be the largest set of requests for the same page, i.e., largest total weight, ties broken arbitrarily. Further, let J' be the empty set if the machine is idle before J was started. Otherwise, let J' be the set of broadcasts aborted by J . Then VAR will abort J to serve R if and only if $|R| \geq c(|J| - |J'|)$ where c is a constant in the range $(2, 4)$ to be determined according to M . Thus, we can view VAR as aborting J if $|R|/|J|$ is larger than a certain abortion ratio which is changing according the previous aborted job.

Consider a basic subschedule $\delta = (J_0, J_1, \dots, J_m)$ produced by VAR. Define a sequence of values $\{r_i\}_{0 \leq i < m}$ such that $r_i = |J_{i+1}|/|J_i|$. Also, define another sequence $\{\tilde{r}_i\}_{0 \leq i < m}$ such that $\tilde{r}_0 = c$, and for $i \geq 1$, $\tilde{r}_i = c(1 - \frac{1}{\tilde{r}_{i-1}})$. Then the following two lemmas hold.

Lemma 1 *For all $i \geq 0$ such that $\tilde{r}_i > 1$, we have $\tilde{r}_{i+1} < \tilde{r}_i$.*

Proof. $\tilde{r}_1 = c - c/\tilde{r}_0 < \tilde{r}_0 = c$. Suppose that $\tilde{r}_i < \tilde{r}_{i-1}$ hold for some $i > 0$. Then we have that $\tilde{r}_{i+1} = (c - \frac{c}{\tilde{r}_i}) < (c - \frac{c}{\tilde{r}_{i-1}}) = \tilde{r}_i$ and the lemma follows. ■

Lemma 2 *For all $i \geq 0$ such that $\tilde{r}_i > 1$, we have $r_i \geq \tilde{r}_i$.*

Proof. By construction of VAR, we have that $r_0 \geq c = \tilde{r}_0$. Suppose that $r_i \geq \tilde{r}_i$ holds for some $i \geq 0$. Then by construction of VAR, $r_{i+1} = |J_{i+2}|/|J_{i+1}| \geq c(1 - \frac{|J_i|}{|J_{i+1}|}) = c(1 - \frac{1}{r_i}) \geq c(1 - \frac{1}{\tilde{r}_i}) = \tilde{r}_{i+1}$. Hence, the lemma follows. ■

Lemma 3 *Assume $2 < c < 4$. Then for any $M \geq 2$, there exists an integer N such that $\tilde{r}_N > 1$ but $\tilde{r}_{N+1} \leq 1$.*

Proof. We will assume $\tilde{r}_i > 1$ for all $i \geq 0$ and derive a contradiction. Expressing \tilde{r}_i in terms of \tilde{r}_{i+1} , we have

$$\tilde{r}_i = \frac{c}{c - \tilde{r}_{i+1}}.$$

Consider the difference between consecutive elements in the sequence $\{\tilde{r}_i\}$, we have

$$\begin{aligned} \tilde{r}_i - \tilde{r}_{i+1} &= \frac{c - c\tilde{r}_{i+1} + \tilde{r}_{i+1}^2}{c - \tilde{r}_{i+1}} \\ &\geq \frac{c(1 - c/4)}{c - \tilde{r}_{i+1}} \\ &> \frac{c(1 - c/4)}{c - 1} \end{aligned}$$

where for the first inequality, we use the fact that $(\tilde{r}_{i+1} - \frac{c}{2})^2 \geq 0$, and for the second inequality, we use the assumption that $\tilde{r}_{i+1} > 1$. Define $\Delta = \frac{c(1-c/4)}{c-1}$. So $\Delta > 0$ (for $2 < c < 4$) and is independent of the subscript i . That means $\tilde{r}_0 > \tilde{r}_N + N\Delta > 1 + N\Delta$ which is greater than c for $N \geq 3/\Delta$, a contradiction. Hence the lemma follows. ■

Note that N is monotonic increasing with c . We claim that the product $\tilde{r}_0 \tilde{r}_1 \dots \tilde{r}_N$ is also monotonic increasing with c . First, $\tilde{r}_0 = c$ and $\tilde{r}_1 = c - 1$ are obviously monotonic increasing with c . Suppose that $\tilde{r}_i = c(1 - \frac{1}{\tilde{r}_{i-1}})$ is monotonic increasing with c for some $1 < i < N$, then $\tilde{r}_{i+1} = c(1 - \frac{1}{\tilde{r}_i})$ is monotonic increasing with c for $(1 - \frac{1}{\tilde{r}_i})$ is monotonic increasing with \tilde{r}_i . Given an $M \geq 2$, we define c^* as the (unique) value of $c \in (2, 4)$ such that $\tilde{r}_0 \tilde{r}_1 \dots \tilde{r}_N = M$. Then we have the following theorem.

Theorem 4.1 *For any $M \geq 2$, VAR is c -competitive if $c > c^*$ and $2 < c < 4$.*

Proof. Similar to the analysis in Theorem 3.1, we consider a basic subschedule $\delta = (J_0, J_1, \dots, J_m)$ produced by VAR. Denote by $|\delta|$ what VAR gains in δ , and by $|\delta^*|$ what OPT gains in δ respectively. Let O_i and $|O_i|$ be as defined before.

By construction of VAR and definition of r_i , $|J_m| \geq r_{m-1}|J_{m-1}| \geq \dots \geq r_{m-1} \dots r_0 |J_0| \geq r_{m-1} \dots r_0$.

We claim that $m - 1 < N$. Otherwise, by Lemma 2, definition of c^* , and the assumption that $c > c^*$, we have that $|J_m| \geq r_N \dots r_1 r_0 \geq \tilde{r}_N \dots \tilde{r}_1 \tilde{r}_0 > M$, contradicting the assumption that each broadcast can satisfy a total weight of at most M . In other words, an arbitrary basic subschedule has finite length.

By the construction of δ , VAR gains $|J_m|$ in δ , i.e., $|\delta| = |J_m|$. Now we bound $|\delta^*|$. For OPT, $|O_i| < c(|J_i| - |J_{i-1}|)$ and $|O_0| < c|J_0|$. Otherwise O_i would have aborted J_i , contradicting that J_i is aborted by J_{i+1} . Thus, $|\delta^*| = \sum_{i=0}^m |O_i| < c|J_0| + \sum_{i=1}^m c(|J_i| - |J_{i-1}|) = c|J_m|$.

Since all requests have tight deadlines, OPT cannot start any broadcast in δ after VAR does. Hence, $|\delta^*|/|\delta| \leq c$ and VAR is c -competitive. ■

5 A Lower Bound

In this section we will prove that VAR is an optimal algorithm for this case, that is, c^* is a lower bound of the competitive ratio for any deterministic algorithm for this problem.

Theorem 5.1 *No deterministic algorithm can be better than c^* -competitive when all requests have tight deadlines.*

Proof. The main idea of the proof is similar to that of Woeginger (1994). We start our proof by a lemma and a definition of request sets. With the definition of \tilde{r}_i , we have the following lemma.

Lemma 4 *Let $\{r_i^*\}_{i \geq 0}$ be the sequence $\{\tilde{r}_i\}_{i \geq 0}$ when $c = c^*$. Then $r_0^* = c^*$, $r_1^* = c^* - 1$, and for $i > 1$, $r_i^* = c^* - 1 - \sum_{p=1}^{i-1} \prod_{q=1}^p \frac{1}{r_{i-q}^*}$.*

Proof. By the definition of r_i^* , we have that $r_0^* = c^*$, $r_1^* = c^* - 1$, and for $i > 1$, $r_i^* = (c^* - \frac{c^*}{r_{i-1}^*})$. Then $r_2^* = (c^* - \frac{c^*}{r_1^*}) = c^* - 1 - \frac{c^* - r_1^*}{r_1^*} = c^* - 1 - \frac{1}{r_1^*}$. Suppose $r_i^* = c^* - 1 - \sum_{p=1}^{i-1} \prod_{q=1}^p \frac{1}{r_{i-q}^*}$ holds. Then $c^* - r_i^* = 1 + \sum_{p=1}^{i-1} \prod_{q=1}^p \frac{1}{r_{i-q}^*}$. We have that

$$\begin{aligned} r_{i+1}^* &= c^* - \frac{c^*}{r_i^*} \\ &= c^* - 1 - \frac{c^* - r_i^*}{r_i^*} \\ &= c^* - 1 - \frac{1 + \sum_{p=1}^{i-1} \prod_{q=1}^p \frac{1}{r_{i-q}^*}}{r_i^*} \\ &= c^* - 1 - \sum_{p=1}^i \prod_{q=1}^p \frac{1}{r_{i+1-q}^*}. \end{aligned}$$

Hence, the lemma follows. ■

We define request sets $SET(J_i, R_i, \sigma_i) = \{K_{i,1}, \dots, K_{i,q}\}$ where $K_{i,j}$ is a set of requests for the same page, $J_i = K_{i,1}$, $R_i = K_{i,q}$, and all requests in SET are of tight deadlines. Thus, each request needs to be served on its arrival or else it cannot be satisfied. Here, $\sigma_i = \sigma/2^i$ such that $\sum_{i=0}^{\infty} \sigma_i < 2\sigma$ where $\sigma > 0$ is an arbitrarily small real number and its value will be determined later. Denote by $|K_{i,j}|$ the total weight of requests in $K_{i,j}$. SET has the following three properties:

1. $|K_{i,j}| < |K_{i,j+1}| \leq |K_{i,j}| + \sigma_i$ for $1 \leq j \leq q-1$.
2. $|R_i| = r_i^* |J_i|$ where by Lemma 4, $r_0^* = c^*$ and for $i \geq 1$, $r_i^* = c^* - 1 - \sum_{p=1}^{i-1} \prod_{q=1}^p \frac{1}{r_{i-q}^*}$.
3. Denote by $a(K_{i,j})$ the arrival time of request set $K_{i,j}$. Then $a(K_{i,j})$ satisfies $0 < a(K_{i,1}) < \dots < a(K_{i,q}) < a(K_{i,1}) + 1 < \dots < a(K_{i,q}) + 1$.

Since $a(K_{i,q}) < a(K_{i,1}) + 1$, all request sets $K_{i,j}$ pairwise collide with each other.

Let $0 < \theta < 1$ be an arbitrarily small real. We will introduce an input list of request sets $SET(*, *, *)$ and an adversary that forces an arbitrary on-line algorithm A to act poorly on the list and to have the competitive ratio at least $c^* - \theta$. The adversary proceeds in steps and in each step, A is fed by a $SET(*, *, *)$, whose exact structure depends on the behavior of A during the preceding steps, and A is forced to abort the current broadcast. After finite steps, what A gains will be a factor of approximately $c^* - \theta$ away from that of OPT.

Step 1. The adversary releases $SET(J_0, R_0, \sigma_0)$ where $|J_0| = 1$. A can process at most one of the request sets due to the construction of SET . If A selects the smallest request set J_0 , the adversary will

release no more requests. OPT will serve R_0 and what OPT gains is c^* times of what A does, making A lose. Thus, A has to select some other request set J_1 . We define the request set arriving immediately before J_1 as J'_1 . OPT will serve J'_1 . By construction of SET , we have $|J'_1| \geq |J_1| - \sigma_0$.

Step 2. The adversary presents $SET(J_1, R_1, \sigma_1)$ to A so that it comes after OPT completes J'_1 but before A completes its current service. If A does not abort the current broadcast, it cannot serve any request in $SET(J_1, R_1, \sigma_1)$ and the adversary releases no more requests later. OPT will then select to serve R_1 after finishing J'_1 . In this case A completes $|J_1|$ requests and OPT gains $|R_1| + |J'_1| \geq r_1^* |J_1| + (|J_1| - \sigma_0) = c^* |J_1| - \sigma$, and then A loses for σ can be arbitrarily small. Therefore, A is forced to abort the current broadcast when $SET(J_1, R_1, \sigma_1)$ arrives. Moreover, A will not select the smallest set of requests in $SET(J_1, R_1, \sigma_1)$ or else it loses for the weight of the smallest request set is the same as that of the aborted one. Denote by J_2 the new set of requests selected by A , and we define the set of requests arriving immediately before J_2 as J'_2 . OPT will then select to serve J'_2 after finishing J'_1 . Similarly, $|J'_2| \geq |J_2| - \sigma_1$.

This is repeated over and over again. In step i , $SET(J_{i-1}, R_{i-1}, \sigma_{i-1})$ releases after OPT completes J'_{i-1} but before A completes its current service. A is forced again to abort the current broadcast. Otherwise if A continues the current broadcast with $|J_{i-1}|$ requests, the adversary will release no more requests. OPT will then select to serve R_{i-1} after finishing J'_{i-1} . In this case, A gains $|J_{i-1}|$ and OPT gains

$$|R_{i-1}| + \sum_{j=1}^{i-1} |J'_j| \geq r_{i-1}^* |J_{i-1}| + \sum_{j=1}^{i-1} |J_j| - \sum_{j=0}^{i-2} \sigma_j. \quad (1)$$

Since J_{i-1} is a request set in $SET(J_{i-2}, R_{i-2}, \sigma_{i-2})$, $|J_{i-1}| \leq |R_{i-2}| = r_{i-2}^* |J_{i-2}| \leq \dots \leq (\prod_{p=j}^{i-2} r_p^*) |J_j|$ holds for $1 \leq j \leq i-2$, and hence $\sum_{j=1}^{i-2} |J_j| \geq \left(\sum_{j=1}^{i-2} \prod_{p=1}^{j-1} \frac{1}{r_{i-1-p}^*} \right) |J_{i-1}|$. Moreover, $\sum_{j=0}^{i-2} \sigma_j < 2\sigma$. Together with Lemma 4, inequality (1) turns out to be

$$\begin{aligned} &|R_{i-1}| + \sum_{j=1}^{i-1} |J'_j| \\ &> \left(r_{i-1}^* + 1 + \sum_{j=1}^{i-2} \prod_{p=1}^{j-1} \frac{1}{r_{i-1-p}^*} \right) |J_{i-1}| - 2\sigma \\ &= c^* |J_{i-1}| - 2\sigma. \end{aligned} \quad (2)$$

Then A loses for σ can be arbitrarily small. So, A has to make an abortion in step i .

However, by Lemmas 1 and 3, we know that r_i^* is decreasing as i increases and eventually $r_{N+1}^* \leq 1$. So, the list of $SET(*, *, *)$ has finite length.

Specifically, assume A is broadcasting a set of request J_{N+1} . Then at step $N+2$, we give $SET(J_{N+1}, R_{N+1}, \sigma_{N+1})$ (which contains only two request sets both of weight $|J_{N+1}|$) after OPT completes J'_{N+1} but before A completes J_{N+1} in $SET(J_N, R_N, \sigma_N)$. Whether A continues its current broadcast or starts a new set of requests in $SET(J_{N+1}, R_{N+1}, \sigma_{N+1})$, it gains $|J_{N+1}|$. On the other hand, OPT starts R_{N+1} after finishing J'_{N+1}

in $SET(J_N, R_N, \sigma_N)$ and then gains totally

$$\begin{aligned} |R_{N+1}| + \sum_{j=1}^{N+1} |J'_j| \\ \geq r_{N+1}^* |J_{N+1}| + \sum_{j=1}^{N+1} |J'_j| \\ > c^* |J_{N+1}| - 2\sigma \end{aligned}$$

using the same analysis as inequality (2). Now set $\sigma \leq \theta |J_{N+1}|/2$. Then, A has competitive ratio at least $c^* - \theta$. Since θ can be arbitrarily small, the theorem is proved. ■

6 Numeric Values

According to the definition of c^* , it depends on $M = 2^k$. In this section, we will discuss the variation of c^* and compare it with the competitive ratio of HEU for different value of M . In the computer program, we set $(\tilde{r}_0 \tilde{r}_1 \cdots \tilde{r}_N) - M \leq 0.001$, the criteria of ending computation. Figure 1 shows the relationship between k and the competitive ratio of VAR and HEU respectively where c^{HEU} denotes the competitive ratio of HEU.

n	c^*	k	\tilde{r}_N	\tilde{r}_{N+1}
1	2.9312	3	1.414	0.859
2	3.7486	10	1.177	0.564
3	3.9206	20	1.103	0.366
4	3.9962	100	1.024	0.094

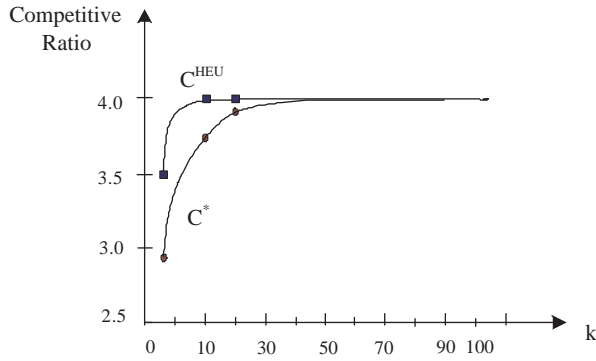


Figure 1: Relationship between c^* and k .

For VAR, when $k = 3$, its competitive ratio $c^* = 2.9312$, and as k increases, c^* increases with an obviously decreasing speed and asymptotically approaches 4 from below. Hence, we conjecture that if $M \rightarrow \infty$, VAR has the competitive ratio 4 in the case of tight deadline.

By figure 1, $c^* < c^{HEU}$ holds for each k . That is, VAR is better than HEU. For instance, when $k = (3, 10, 20, 100)$, $c^* = (2.9312, 3.7486, 3.9206, 3.9962)$ and by Theorem 3.1, $c^{HEU} = (3.50, 3.9961, 4.00, 4.00)$.

7 Conclusion

In this paper, we discuss the broadcasting problem with the upper bound on the number and weight of requests in the system. We proposed the VAR algorithm and prove its competitiveness. By experiment, we show that VAR is better than HEU introduced by Woeginger (1994) in the case of tight deadline.

We also prove that VAR matches the lower bound of competitive ratio for deterministic algorithms.

Since both VAR and HEU do not consider the deadlines of requests, their competitive ratios when not all requests have tight deadlines are simply those in the tight deadline case plus one. It is conceivable that the competitive ratio can be further improved in the case of arbitrary deadline if the deadlines are taken into consideration. In fact, Zheng et al. (2004) have constructed a 4.56-competitive algorithm in Kim's model, in which they considered the deadlines of requests. An obvious open problem is: when there is an upper bound on request weight and number, does there exist a better algorithm that considers request deadline?

References

- Aacharya, S. & Muthukrishnan, S. (1998), Scheduling on-demand broadcasts: new metrics and algorithms, in 'ACM/IEEE International Conference on Mobile Computing and Networking', pp. 43–54.
- Aksoy, D. & Franklin, M. (1998), Scheduling for large-scale on-demand data broadcasting, in 'IEEE Conference on Computer Communications (INFOCOM'98)', pp. 652–659.
- Bartal, Y. & Muthukrishnan, S. (2000), Minimizing maximum response time in scheduling broadcasts, in 'Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms', pp. 558–559.
- Chan, W., Lam, T., Ting, H. & Wong, P. (2004), New results on on-demand broadcasting with deadline via job scheduling with cancellation, in '10th International Computing and Combinatorics Conference (COCOON'04)', Vol. 3106 of *Lecture Notes in Computer Science*, pp. 210–218.
- Edmonds, J. & Pruhs, K. (2002), Broadcast scheduling when fairness is fine, in 'Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms', pp. 421–430.
- Jiang, S. & Vaidya, N. (1998), Scheduling algorithms for a data broadcast system: minimizing variance of the response time, Technical Report TR98-005, Texas A&M University.
- Kalyanasundaram, B. & Velauthapillai, M. (2003), On-demand broadcasting under deadline, in '11th European Symposium on Algorithms', Vol. 2832 of *Lecture Notes in Computer Science*, pp. 313–324.
- Kim, J. & Chwa, K. (2003), Scheduling broadcasts with deadlines, in '9th International Computing and Combinatorics Conference (COCOON'03)', Vol. 2697 of *Lecture Notes in Computer Science*, pp. 415–424.
- Miyazawa, H. & Erlebach, T. (2004), 'An improved randomized on-line algorithm for a weighted interval selection problem', *Journal of Scheduling* **7**, 293–311.
- Woeginger, G. (1994), 'On-line scheduling of jobs with fixed start and end times', *Theoretical Computer Science* **130**, 5–16.
- Zheng, F., Fung, S., Chin, F., Poon, C. & Xu, Y. (2004), Improved on-line broadcast scheduling with deadlines. manuscript.

Greedy algorithms for on-line set-covering and related problems

Giorgio Ausiello¹

Aristotelis Giannakos²

Vangelis Th. Paschos²

¹Dipartimento di Informatica e Sistemistica
Università degli Studi di Roma "La Sapienza"
Via salaria 113, 00198, Roma, Italy
Email: ausiello@dis.uniroma1.it

²LAMSADE
Université Paris-Dauphine and CNRS UMR 7024
Place du Maréchal De Lattre de Tassigny, 75775 Paris Cedex 16, France
Emails: {giannako,paschos}@lamsade.dauphine.fr

Abstract

We study the following on-line model for set-covering: elements of a ground set of size n arrive one-by-one and with any such element c_i , arrives also the name of some set S_{i_0} containing c_i and covering the most of the uncovered ground set-elements (obviously, these elements have not been yet revealed). For this model we analyze a simple greedy algorithm consisting of taking S_{i_0} into the cover, only if c_i is not already covered. We prove that the competitive ratio of this algorithm is \sqrt{n} and that it is asymptotically optimal for the model dealt, since no on-line algorithm can do better than $\sqrt{n/2}$. We next show that this model can also be used for solving minimum dominating set with competitive ratio bounded above by the square root of the size of the input graph. We finally deal with the maximum budget saving problem. Here, an initial budget is allotted that is destined to cover the cost of an algorithm for solving set-covering. The objective is to maximize the savings on the initial budget. We show that when this budget is at least equal to \sqrt{n} times the size of the optimal (off-line) solution of the instance under consideration, then the natural greedy off-line algorithm is asymptotically optimal.

Keywords: Set-covering, On-line algorithm, Competitive ratio, Dominating set, Budget saving

1 Introduction

Let C be a ground set of n elements and \mathcal{S} a family of m subsets of C such that $\cup_{S \in \mathcal{S}} S = C$. The set covering problem consists of finding a family $\mathcal{S}' \subseteq \mathcal{S}$, of minimum cardinality, such that $\cup_{S \in \mathcal{S}'} S = C$. In what follows, for an element $c_i \in C$, we set $F_i = \{S_j \in \mathcal{S} : c_i \in S_j\}$ and $f_i = |F_i|$; also, we set $f = \max\{f_i : i = 1, \dots, n\}$.

The set covering problem has been extensively studied over the past decades. It has been shown to be **NP**-hard in Karp (1972) and $O(\log n)$ -approximable for both weighted and unweighted cases (see Chvátal (1979), for the former, and Johnson (1974), Lovász (1975) and Slavík (1996), for the latter; see also Paschos (1997) for a comprehensive survey on the subject). As it is shown by Feige (1998), this approximation ratio is the best achievable, unless $\mathbf{NP} \subseteq \mathbf{DTIME}(n^{O(\log \log n)})$, i.e., unless problems

in **NP** could be proved solvable by slightly super-polynomial algorithms.

In *on-line* computation, one can assume that the instance is not known in advance but it is revealed step-by-step. Upon arrival of new data, one has to decide irrevocably which of these data are to be taken in the solution under construction. The fact that the instance is not known in advance, gives rise to several on-line models specified by the ways in which the final instance is revealed, or by the amount of information that is achieved by the on-line algorithm at each step. In any of these models, one has to devise algorithms, called on-line algorithms, constructing feasible solutions whose values are as close as possible to optimal off-line values, i.e., to values of optimal solutions assuming that the final instance is completely known in advance. The closeness of an on-line solution to an optimal off-line one is measured by the so-called competitive ratio $m(x, y)/\text{opt}(x)$, where x is an instance of the problem dealt, y the solution computed by the on-line algorithm dealt, $m(x, y)$ its value and $\text{opt}(x)$ the value of an optimal off-line solution. This measure for on-line computation has been introduced by Sleator *et al.* (1985).

Informally, the basic on-line set-covering model adopted here is the following: elements of a ground set of size n arrive one-by-one and with any such element c_i , arrives also the name of some set S_{i_0} containing c_i and covering the most of the ground set-elements that have not been yet covered. Clearly, any uncovered element is yet unrevealed. For this model we analyze a simple greedy algorithm consisting of taking S_{i_0} into the cover, only if c_i is not already covered. We prove that the competitive ratio of this algorithm is \sqrt{n} and that it is asymptotically optimal for the model dealt, since no on-line algorithm can do better than $\sqrt{n/2}$. This model generalizes the one proposed in Alon *et al.* (2003) and, furthermore, it uses a very simple, fast and intuitive algorithm that could be seen as the on-line counterpart of the natural greedy (off-line) set-covering algorithm.

In Alon *et al.* (2003), the following on-line set covering model has been studied. We suppose that we are given an instance (\mathcal{S}, C) that it is known in advance, but it is possible that only a part of it, i.e., a sub-instance (\mathcal{S}_p, C_p) of (\mathcal{S}, C) will finally arrive; this sub-instance is not known in advance. A picturesque way to apprehend the model is to think of the elements of C as lights initially switched off. Elements switch on (get activated) one-by-one. Any time an element c gets activated, the algorithm has to decide which among the sets of \mathcal{S} containing c has to be included in the solution under construction (since we assume that (\mathcal{S}, C) is known in advance, all these sets are also known). In other words, the algorithm has to keep an online cover for the activated elements. The

algorithm proposed for this model achieves competitive ratio $O(\log n \log m)$ (even if less than n elements of C will be finally switched on and less than m subsets of \mathcal{S} include these elements).

The on-line model dealt here and studied in Section 2, is inspired, yet quite different, from the one of Alon *et al.* (2003). Given C , \mathcal{S} (not known in advance as Alon *et al.* (2003) assumes) and an arrival sequence $\Sigma = (\sigma_1, \dots, \sigma_n)$ of the elements of C (i.e., elements of C are switched on following the order $\sigma_1, \dots, \sigma_n$), the objective is to find, for any $i \in \{1, \dots, n\}$, a family $\mathcal{S}'_i \subseteq \mathcal{S}$ such that $\{\sigma_1, \dots, \sigma_i\} \subseteq \cup_{S \in \mathcal{S}'_i} S$. For any σ_i , $i = 1, \dots$, we denote by S_i^j , $j = 1, \dots, f_i$, the sets of \mathcal{S} containing σ_i , by \bar{S}_i^j the subset of the elements of S_i^j still remaining uncovered and by δ_i^j the cardinality of \bar{S}_i^j . By f_i , we denote the frequency of σ_i , i.e., the number of sets in \mathcal{S} containing σ_i . When σ_i switches on, the only information revealed is the name of some set $S_i^{j_0} \in \operatorname{argmax}\{\delta_i^j, j = 1, \dots, f_i\}$. So, no a priori knowledge of the topology of the instance (\mathcal{S}, C) is assumed by the model. In particular, we do not have to know which are the yet uncovered elements of $S_i^{j_0}$ but only the fact that their number is maximum with respect to any other S_i^j .

The algorithm that we study for this model, called **LGREEDY** in the sequel, informally works as follows: once an element $\sigma_i \in C$ switches on, if σ_i is not already covered, then set $S_i^{j_0} \in \operatorname{argmax}\{\delta_i^j, j = 1, \dots, f_i\}$ is added in the cover under construction. Clearly, by the way **LGREEDY** works, the content of $\bar{S}_i^{j_0}$ is still unrevealed. This algorithm follows the same principle as the natural greedy algorithm for (off-line) minimum set covering, called **FGREEDY** in the sequel, modulo the fact that this principle applies not to the whole instance (\mathcal{S}, C) that is to be finally revealed, but to the part of (\mathcal{S}, C) induced by the elements of C that, at a given moment, are switched off (even if the topology of this part is not known). We prove that the competitive ratio of this algorithm is \sqrt{n} and also that there exist arbitrarily large instances for which this ratio is at least $\sqrt{n/2}$. We then show that the set-covering model dealt can be used to solve also minimum dominating set, within competitive ratio \sqrt{n} where n is the order of the input graph. Minimum dominating set is defined as follows: given a graph $G(V, E)$, we wish to determine the least subset $V' \subseteq V$ that dominates the rest of the vertices, i.e., a subset $V' \subseteq V$ such that for all $u \in V \setminus V'$ there exists $v \in V'$ for which $(u, v) \in E$.

In Section 3, we provide a lower bound, equal to $\sqrt{n/2}$ for the competitiveness of a whole class of on-line algorithms running on our model. These algorithms are the ones that construct a cover by taking, at any activation step, at least one set containing some not yet covered recently activated ground element. Based upon this result, one can conclude that **LGREEDY** is asymptotically optimal for this class and for the model adopted.

There exist several reasons motivating, to our opinion, the study of the model dealt in this paper. The first one is that the algorithm used is as it has already been mentioned, a kind of on-line alternative of the famous greedy algorithm for set-covering. Hence, analysis of its competitiveness is interesting by itself. The second reason is that a basic and very interesting feature of the model dealt is its very small memory requirement, since the only information needed is the binary encoding of the name of $S_i^{j_0}$. This is a major difference between our approach and the one of Alon *et al.* (2003). There, anytime an element gets

activated, the algorithm needs to compute the value of a potential function using an updated weight parameter for each element and then chooses covering sets in a suitable way so that this potential be non-increasing; the greedy online algorithm in our model needs only a constant number of memory places, making it more appropriate for handling very large instances with very few hardware resources.

In many real-life problems, it is meaningful to relax the main specification of the online setting, that is, to keep a solution for any partially revealed instance, in order to achieve a better solution quality. In this sense, a possible relaxation is to consider that several algorithms collaborate in order to return the final solution. The costs of using these algorithms can be different the ones from the others, depending upon the sizes of the solutions computed, the time overheads they take in order to produce them, etc. Moreover, we can assume that an initial common budget is allotted to all these algorithms and that this budget is large enough to allow use of at least one of the algorithms at hand to solve the problem without exceeding it. A nice objective could be in this case, to use these algorithms in such a way that a maximum of the initial budget is saved. For the case of set-covering, the following budget-model, giving rise to what we call *maximum budget saving problem* is considered in Section 4. We assume that two algorithms collaborate to solve it: the **LGREEDY** and the **FGREEDY**. The application cost of the former is just the cardinality of the solution it finally computes, while, for the latter, its application cost is the cardinality of its solutions augmented by an overhead due, for example, to the fact that it is allowed to wait before making its decisions. For an instance x of set-covering, the initial budget considered is $B(x) = \sqrt{n} \operatorname{opt}(x)$ (this is in order that at least **LGREEDY** is able to compute a solution of x without exceeding the budget for any x). Denote by $c(x, y)$ the cost of using **A** in order to compute a cover y for x . The objective is to maximize the quantity $B(x) - c(x, y)$ and, obviously, the maximum possible economy on x is $B(x) - \operatorname{opt}(x)$. We show in Section 4 that there exists a natural algorithm-cost model such that **FGREEDY** is asymptotically optimal for maximum budget saving.

Before closing this section, let us quote another very interesting approach that could be considered to be at midway between semi-on-line approaches and solutions-stability ones, developed in Gambosi *et al.* (1997). There, the problem tackled is the maintenance of approximation ratio achieved by an algorithm while the set covering instance undergoes limited changes. More precisely, assume a set covering instance (\mathcal{S}, C) and a solution \mathcal{S}' for it. How many insertions of some of the ground elements in subsets that did not previously contain these elements produce an instance for which the solution \mathcal{S}' of the initial instance guarantees the same approximation ratio in both of them? It is shown in Gambosi *et al.* (1997) that if solution \mathcal{S}' has been produced by application of the natural greedy algorithm achieving approximation ratio $O(\log n)$ (see Chvátal (1979)), then after $O(\log n)$ such insertions initial solution \mathcal{S}' still guarantees the same approximation ratio.

2 A greedy on-line algorithm

As already mentioned, the model studied in this section assumes an arrival sequence $\Sigma = (\sigma_1, \dots, \sigma_n)$ of the elements of C , and the objective is to find, for any $i \in \{1, \dots, n\}$, a family $\mathcal{S}'_i \subseteq \mathcal{S}$ such that $\{\sigma_1, \dots, \sigma_i\} \subseteq \cup_{S \in \mathcal{S}'_i} S$. Once an element σ_i , $i = 1, \dots$, switches on, the encoding for $S_i^{j_0} \in$

$\text{argmax}\{\delta_i^j, j = 1, \dots, f_i\}$ is also revealed.

For this model, we propose the following algorithm, **LGREEDY**, where, although it is not necessary, we suppose for reasons of simplicity of algorithm's specification that n is known to it:

- set $S'_0 = \emptyset$;
- for $i = 1$ to n do (σ_i switches on): if σ_i is not already covered by S'_{i-1} ,
 - then set $S'_i = S'_{i-1} \cup \{\text{argmax}\{\delta_i^j : j = 1, \dots, f_i\}\}$;
 - else set $S'_i = S'_{i-1}$;
- output $S' = S'_n$.

Theorem 1. Consider an instance (S, C) of minimum set covering with $|C| = n$. Consider also the on-line model introduced above, and denote by $S^* = \{S_1^*, \dots, S_{k^*}^*\}$ an optimal off-line solution on (S, C) . Then, the competitive ratio of **LGREEDY** is bounded above by $\min\{\sqrt{2n/k^*}, \sqrt{n}\}$. Furthermore, there exist large enough instances for which this ratio is at least $\sqrt{n/2}$.

Proof. Fix an arrival sequence $\Sigma = (\sigma_1, \dots, \sigma_n)$ and denote by c_1, \dots, c_k , its *critical elements*, i.e., the elements having entailed introduction of a set in S' . In other words, critical elements of Σ are all elements c_i such that c_i was not yet covered by the cover under construction upon its arrival. Assume also that the final cover S' consists of k sets, namely, S_1, \dots, S_k , where S_1 has been introduced in S' due to c_1 , S_2 due to c_2 , and so on.

Let $\delta(S_i)$ be the increase of the number of covered elements just after having taken S_i in the greedy cover (recall that if S_i has been added in S' for critical element $c_i = \sigma_j$, $\delta(S_i) = \max\{\delta_j^1, \dots, \delta_j^{f_j}\}$). We have:

$$\delta(S_1) = |S_1| \quad (1)$$

and, for $2 \leq i \leq k$,

$$\delta(S_i) = \left| \bigcup_{\ell=1}^i S_\ell \right| - \left| \bigcup_{\ell=1}^{i-1} S_\ell \right| \quad (2)$$

Fix now an optimal off-line solution S^* of cardinality k^* . Any of the critical elements c_1, \dots, c_k can be associated to the set of smallest index in S^* containing it. For any $S_i^* \in S^*$, we denote by \hat{S}_i^* , the set of the critical elements associated with S_i^* (obviously, $\hat{S}_i^* \subseteq S_i^*$). The *critical content* $h(S_i^*)$ of any $S_i^* \in S^*$ is defined as the number of critical elements associated to it as described before, i.e., $h(S_i^*) = |\hat{S}_i^*|$.

Let S_1^*, \dots, S_r^* be the sets in S^* of positive critical contents $h(S_1^*), \dots, h(S_r^*)$, respectively. Clearly,

$$\sum_{i=1}^r h(S_i^*) = k \quad (3)$$

$$r \leq k^* \quad (4)$$

For any S_i^* , let $c_i^1, \dots, c_i^{h(S_i^*)}$ be the elements of its critical content ordered according to their position in the arrival sequence Σ ; in other words, following our assumptions, $\hat{S}_i^* = \{c_i^1, \dots, c_i^{h(S_i^*)}\}$ (recall that $\hat{S}_i^* \subseteq S_i^*$).

Suppose, without loss of generality, that, for $\ell = 1, \dots, h(S_i^*)$, the set $S_{j_\ell} \in S$ has been introduced in S' when the critical element c_i^ℓ has been activated. At

the moment of the arrival of c_i^1 , the set S_i^* is also a candidate set for S' . The fact that S_{j_1} has been chosen instead of S_i^* means that $\delta(S_{j_1}) \geq \delta(S_i^*)$; hence, since as noticed just above, $\hat{S}_i^* \subseteq S_i^*$, the following holds immediately: $\delta(S_{j_1}) \geq \delta(S_i^*) \geq |\hat{S}_i^*| = h(S_i^*)$. When c_i^2 gets activated, the set S_i^* has lost some of its elements that have been covered by some sets already chosen by the algorithm. In any case, it has lost c_i^1 (covered by S_{j_1}). So, following the arguments developed just above for S_{j_1} , $\delta(S_{j_2}) \geq h(S_i^*) - 1$, and so on (quantities $\delta(\cdot)$ are defined either by (1), or by (2)). So, dealing with c_i^ℓ , the following holds:

$$h(S_i^*) - \ell + 1 \leq \delta(S_{j_\ell}) \quad (5)$$

For example, consider the illustration of Figure 1. Let S^* be a set of the fixed optimal cover S^* and denote by \hat{S} the set of its critical elements, c^1 , c^2 and c^3 (ranged in the order they have been activated). Let S be the set chosen by **LGREEDY** to cover c^2 . The shadowed parts of S^* , \hat{S} and S correspond to elements already covered by **LGREEDY** at the moment of arrival of c^2 . At this moment, S must contain at least as many uncovered elements as S^* does and a fortiori at least one uncovered element for any yet uncovered critical element of S^* (two uncovered elements for S appear below the dashed line for c^3 and c^4).

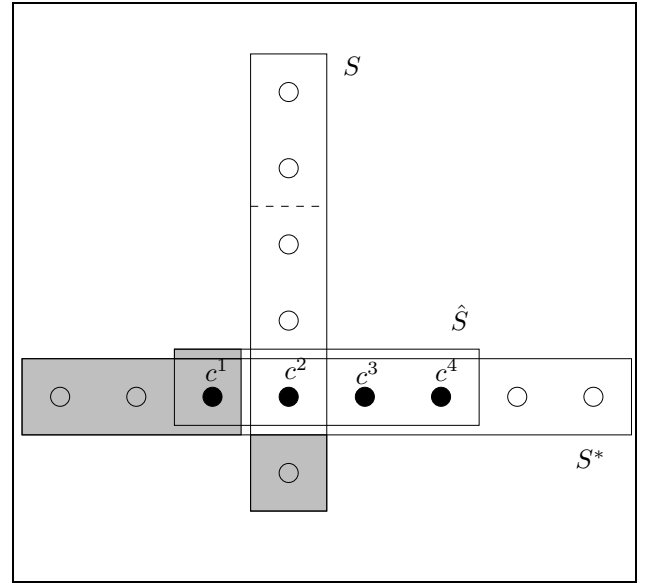


Figure 1: An example for (5)

Summing up inequalities (5), for $\ell = 1, \dots, h(S_i^*)$, and setting $\sum_{\ell=1}^{h(S_i^*)} \delta(S_{j_\ell}) = n_i$, we finally get for S_i^* :

$$\begin{aligned} \frac{h(S_i^*) (h(S_i^*) + 1)}{2} &\leq \sum_{\ell=1}^{h(S_i^*)} \delta(S_{j_\ell}) = n_i \\ \implies h(S_i^*) &\leq \sqrt{2n_i} \end{aligned} \quad (6)$$

Set, for $1 \leq i \leq r$, $n_i = \alpha_i n$, for some $\alpha_i \in [0, 1]$. Then, $\sum_{i=1}^r \alpha_i = 1$ and

$$\sum_{i=1}^r \sqrt{\alpha_i} \leq \sqrt{r} \quad (7)$$

Using (3), (4), (6) and (7), we get:

$$k = \sum_{i=1}^r h(S_i^*) \leq \sqrt{2n} \sum_{i=1}^r \sqrt{\alpha_i} \leq \sqrt{r} \sqrt{2n} \leq \sqrt{k^*} \sqrt{2n} \quad (8)$$

Dividing the first and the last members of (8) by k^* , we get:

$$\frac{k}{k^*} \leq \sqrt{\frac{2n}{k^*}} \quad (9)$$

On the other hand, remark that, if $k^* = 1$, i.e., if there exists $S^* \in \mathcal{S}$ such that $\mathcal{S}^* = \{S^*\}$, then LGREEDY would have chosen it from the beginning of its running in order to cover σ_1 ; next, no additional set would have entered the \mathcal{S}' . Consequently, we can assume that $k^* \geq 2$ and, using (9),

$$\frac{k}{k^*} \leq \sqrt{n} \quad (10)$$

Combination of (9) and (10) concludes the competitive ratio claimed.

Fix an integer N and consider the following instance (\mathcal{S}, C) of minimum set covering:

$$\begin{aligned} C &= \left\{ 1, \dots, \frac{N(N+1)}{2} \right\} \\ S_1 &= \{1, \dots, N\} \\ S_2 &= \{N+1, \dots, 2N-1\} \\ &\vdots \\ S_N &= \left\{ \frac{N(N+1)}{2} \right\} \\ S_{N+1} &= \left\{ (i-1)N - \frac{i(i-3)}{2} : i = 1, \dots, N \right\} \\ S_{N+2} &= C \setminus S_{N+1} \end{aligned}$$

Consider the arrival sequence $(1, \dots, N(N+1)/2)$. LGREEDY might compute the cover $\mathcal{S}' = \{S_i, 1 \leq i \leq N\}$, while the optimal one is $\mathcal{S}^* = \{S_{N+1}, S_{N+2}\}$. Hence, the competitive ratio in this case would be $N/2$, with $N = (-1 + \sqrt{1+8n})/2$ which is asymptotically equal to $\sqrt{n/2}$ as claimed.

For example, consider Figure 2. For Σ starting with 1, 6, 10, 13, 15, LGREEDY may have chosen sets:

$$\{1, 2, 3, 4, 5\}, \{6, 7, 8, 9\}, \{10, 11, 12\}, \{13, 14\}, \{15\}$$

respectively, while the optimal cover would consist of the two sets:

$$\{1, 6, 10, 13, 15\} \quad (11)$$

$$\{2, 3, 4, 5, 7, 8, 9, 11, 12, 14\} \quad (12)$$

The proof of the theorem is now complete. ■

Revisit (9), set $\Delta = \max_{S_i \in \mathcal{S}} \{|S_i|\}$ and take into account the obvious inequality: $k^* \geq n/\Delta$. Then, the following result is immediately derived from Theorem 1.

Corollary 1. *The competitive ratio of LGREEDY is bounded above by $\sqrt{2\Delta}$.*

The set-covering model dealt here is very economic and thus suitable to solve very large instances. Indeed, its memory requirements are extremely reduced since the only information LGREEDY needs at any step i is the encoding of the name of a set $S_i^{j_0} \in \arg\max\{\delta_i^j, j = 1, \dots, f_i\}$. This is not the case for

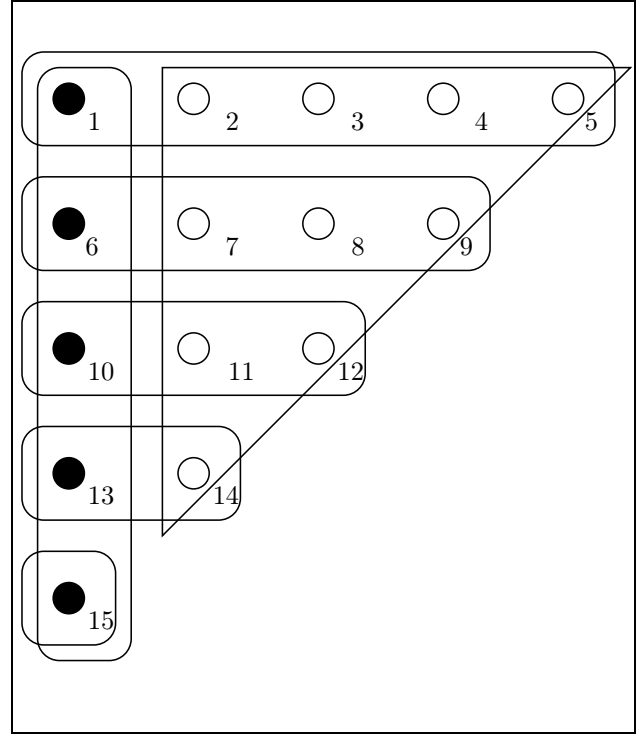


Figure 2: The ratio $\sqrt{n/2}$ for LGREEDY is asymptotically attained

the intensive computations implied by the model of Alon *et al.* (2003).

Let us note that the model dealt above can be used to solve a natural on-line version of the minimum dominating set problem. Given a graph $G(V, E)$ with $|V| = n$, assume that its vertices switch on one-by-one. Any time a vertex σ_i does so, the name of its neighbor with the most neighbors still switched off is announced. Denote by v_{i_0} such neighbor of σ_i . If σ_i is not yet dominated by the partial dominating set V' already constructed, then v_{i_0} enters V' .

Consider the following classical reduction from minimum dominating set to set covering: the vertex-set V of the input-graph G becomes both the family of subsets and the ground set of the set covering instance (hence, both items have size n) and for any vertex $v_i \in V$, the corresponding set contains v_i itself together with its neighbors in G . It is easy to see that any set cover of size k in the so-constructed set covering instance corresponds to a dominating set of the same size in G and vice-versa. Remark also that the dominating set model just assumed on G is exactly, with respect to the transformation just sketched, the set-covering model dealt before. Consequently, the following result follows immediately.

Proposition 1. *The on-line set-covering algorithm of Theorem 1 is \sqrt{n} -competitive for minimum dominating set in graphs of order n .*

Note also that the counter-example instance given in the proof of Theorem 1 can be slightly modified to fit the case where, at each step, whenever a yet uncovered element arrives, the algorithm is allowed to take in the cover a constant number of sets containing it and such that the number of elements yet switched off that belong to these sets is maximized. For some $\rho > 1$ and for some integer N , consider the following instance:

$$\mathcal{S} = \left\{ X, Y, S_i^j : 1 \leq i \leq N, 1 \leq j \leq \rho \right\}$$

$$\begin{aligned}
C &= \bigcup_{i=1}^N \bigcup_{j=1}^{\rho} S_i^j \\
(|C| &= \rho \frac{N(N-1)}{2} + N = n) \\
X &= \{x_1, \dots, x_N\} \\
|S_i^j| &= N - i + 1 \text{ for } i = 1, \dots, N \\
S_i^j \cap S_l^k &= \emptyset, \text{ if } i \neq l \\
S_i^j \cap S_i^k &= \{x_i\}, \text{ if } j \neq k \\
Y &= C \setminus X
\end{aligned}$$

Consider the arrival sequence where x_1, \dots, x_N are firstly revealed. LGREEDY might take in the cover all the S_i^j 's, while the optimal cover is $\{X, Y\}$. In this case, the competitive ratio is $\rho N/2$, with

$$N = \frac{\rho - 2}{2\rho} + \sqrt{\left(\frac{\rho - 2}{2\rho}\right)^2 + 2\frac{n}{\rho}}$$

i.e., the value of the ratio is asymptotically $\sqrt{\rho n/2}$.

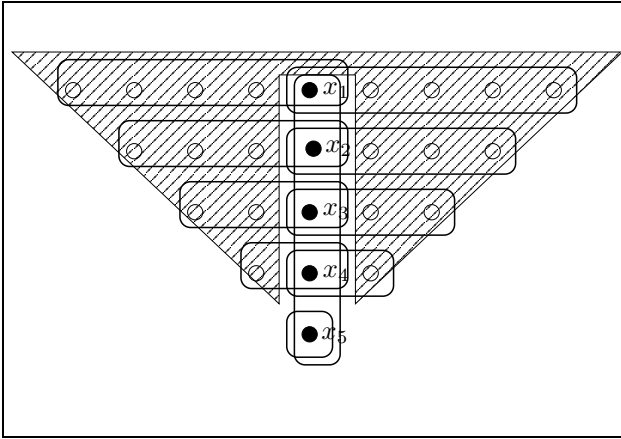


Figure 3: A counter-example for the case where the algorithm is allowed to take a constant number of sets containing a recently arrived element

For example, set $\rho = 2$ and $N = 5$ and consider the instance of Figure 3. For Σ starting with x_1, x_2, x_3, x_4, x_5 , the algorithm may insert to the cover the sets depicted as “rows”, while the optimal cover would consist of the “column”-set $\{x_1, x_2, x_3, x_4, x_5\}$ together with the “big” set containing the rest of the elements (drawn striped in Figure 3).

In the weighted version of set-covering, any set S of \mathcal{S} is assigned with a non-negative weight $w(S)$, and a cover S' of the least possible total weight $W = \sum_{S \in S'} w(S)$ has to be computed. A natural modification of LGREEDY in order to deal with weighted set-covering is to put in the cover, whenever a still uncovered element arrives, a set S_i containing it that minimizes the quantity $w(S_i)/\delta(S_i)$. Unfortunately, this modification cannot perform satisfactorily. Consider, for example, an instance of weighted set-covering consisting of a ground set $C = \{x_1, \dots, x_n\}$, and three sets, $S = C$ with $w(S) = n$, $X = \{x_1\}$ with $w(X) = 1$ and $Y = C \setminus \{x_1\}$ with $w(Y) = 0$. If x_1 arrives first, the algorithm could have chosen S to cover it, yielding a cover for the overall instance of total weight n , while the optimal cover would be $\{X, Y\}$ of total weight 1.

3 A lower bound for a whole class of on-line algorithms

We now prove that, in the general model, no on-line algorithm can achieve, for the model dealt, competitive ratio better than $\sqrt{n/2}$, even if it is allowed to choose at any step more than one set to be introduced in the solution.

Proposition 2. *Let A be an on-line algorithm for set-covering such that, at any step, it takes in the cover at least one set containing some not yet covered arriving element. Let k_A be the size of the cover computed by A and k^* be the size of the optimal cover. Then, $k_A/k^* \geq \sqrt{n/2}$.*

Proof. Consider the following set-covering instance built, for any integer N , upon a ground set $S = \{x_{ij} : 1 \leq j \leq i \leq N\}$; obviously, $|C| = n = N(N+1)/2$. A *path-set of order i* is defined as a set containing $N-i+1$ elements $\{x_{ij_i}, \dots, x_{Nj_N}\}$. The set-system \mathcal{S} of the instance contains all possible path-sets of each order i , $1 \leq i \leq N$. Clearly, there exist $N!/0!$ path-sets of order 1, $N!/1!$ path-sets of order 2, and so on and, finally, $N!/(N-1)!$ path-sets of order N , i.e., in all $N!(1 + \dots + 1/(N-1)!) \approx eN!$ path-sets. Finally, the set-system \mathcal{S} is completed with an additional set Y containing all elements of C but those of some path-set of order 1, that will be specified later (hence, $|Y| = n - N$).

As long as there exist uncovered elements, the adversary may choose to have an uncovered element x_{ij} of the lowest possible i arriving, which will be contained only in all path-sets of order less than or equal to i . Notice that as long as algorithm A has $r < N$ sets inserted in the cover, there will be at least one element x_{r+1j} for some j , $1 \leq j \leq k+1$, not yet covered. Suppose that after the arrival of σ_t , the size of the cover computed by A gets equal to, or greater than, N . Clearly, $1 \leq t \leq N$. At time $t+1$, a new element arrives, contained in some path-sets and in Y , which can be now specified as consisting of all elements in C except of the elements of some path-set S^* of order 1 containing $\sigma_1, \dots, \sigma_t$; the rest of the arrival sequence is indifferent.

Clearly the optimum cover in this case would have been path-set S^* together with set Y ; hence, $k_A/k^* \geq N/2$, with N tending to $\sqrt{2n}$ as n increases.

For example consider the instance of Figure 4, with $N = 5$ (the elements of C are depicted as cycles labelled by (i, j) for $1 \leq j \leq i \leq 3$). The S_i sets can be thought of as paths terminating to a sink on the directed graph of Figure 4(a). Assume that $(1, 1)$ arrives, and algorithm A chooses sets $\{(1, 1), (2, 1), (3, 1)\}, \{(1, 1), (2, 2), (3, 2)\}$ for covering it; the uncovered element $(3, 3)$ arrives next, so A has to cover it by, say, the set $\{(2, 1), (3, 3)\}$ (Figure 4(b)). The optimal cover might consist of set $\{(1, 1), (2, 2), (3, 3)\}$ together with a big set consisting of the rest of the elements, that could not have been revealed to A upon arrival of $(1, 1)$, or of $(3, 3)$ (Figure 4(c)).

It is easy to see that the above construction can be directly generalized so that the same result holds also in the case that the on-line algorithm is allowed to take more than one sets at a time in the cover: if $\sigma_1 = x_{i\ell}$, then as long as the size of the online cover is less than N , there exists always some $i_{\ell-1} < i_\ell \leq N$ and some j_{i_ℓ} for which $x_{i_\ell j_{i_\ell}}$ is yet uncovered. Hence, if σ_ℓ is this element, then the algorithm will have to put some sets in the cover. Finally, the algorithm will have put N sets in the cover, while the optimum will always be of size 2. ■

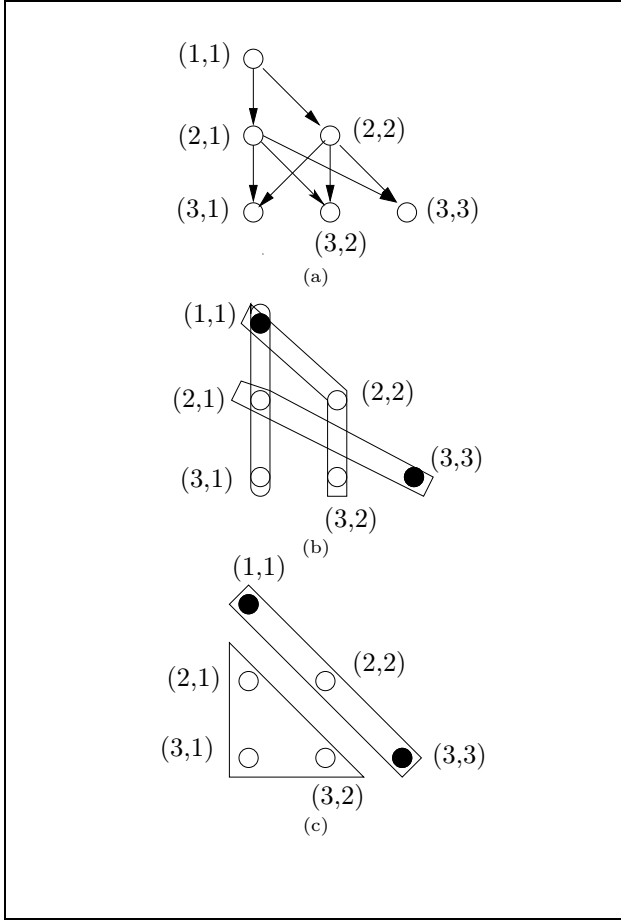


Figure 4: The counter-example of Proposition 2 for $N = 5$

4 The maximum budget saving problem

In this section, we study a kind of dual version of the minimum set-covering, the maximum budget saving problem. Here, we are allotted an initial budget $B(\mathcal{S}, C)$ destined to cover the cost of an algorithm that solves minimum set-covering on (\mathcal{S}, C) . Any such algorithm has its own cost that is a function of the size of the solution produced, of the time overheads it takes in order to compute it, etc. Our objective is to maximize our savings, i.e., the difference between the initial budget and the cost of the algorithm. For simplicity, we assume that the maximum saving ever possible to be performed is $B(\mathcal{S}, C) - k^*$, where, as previously, k^* is the size of an optimum set-cover of (\mathcal{S}, C) .

We consider here that the set-covering instance arrives on-line. If a purely on-line algorithm is used to solve it, then its cost equals the size of the solution computed; otherwise, if the algorithm allows itself to wait in order to solve the instance (partly or totally) off-line then, its cost is the sum of the size of the solution computed plus a fine that is equal to some root, of order strictly smaller than 1, of the solution that would be computed by a purely on-line algorithm. We suppose that the budget allotted is equal to $k^*\sqrt{n}$, where $n = |C|$. This assumption on $B(\mathcal{S}, C)$ is quite natural. It corresponds to a kind of feasible cost for an algorithm; this is algorithm LGREEDY presented in Section 2.

The interpretation of this model is the following. We are allotted a budget corresponding to the cost of an algorithm always solving set-covering. In this way, we are sure that we can always construct a feasible solution for it. Furthermore, by the second part of The-

orem 1, it is very risky to be allotted less than $k^*\sqrt{n}$ since there exist instances where the bound \sqrt{n} is attained. On the other hand, we can have at our disposal a bunch of on-line or off-line set-covering algorithms, any one having its proper cost as described just above, from which we have to choose the one whose use will allow us to perform the maximum possible economy with respect to our initial budget. The fact that the measure of the optimum solution for maximum budget saving is $B(\mathcal{S}, C) - k^*$, has also a natural interpretation: we can assume that there exist an arrival sequence Σ for C such that, for any $\sigma_i \in \Sigma$, an oracle can always choose to cover σ_i with the same set with which σ_i is covered in an optimum off-line solution for instance (\mathcal{S}, C) . Under this assumption for the measure of the optimum budget saving solution, this problem is clearly NP-hard since it implies computation of an optimum solution for minimum set-covering. Finally, denoting by $c_A(\mathcal{S}, C)$ the cost of algorithm A when solving minimum set-covering on (\mathcal{S}, C) , the approximation ratio of maximum set saving is equal to:

$$\frac{B(\mathcal{S}, C) - c_A(\mathcal{S}, C)}{B(\mathcal{S}, C) - k^*} \quad (13)$$

Obviously this ratio is smaller than 1 and, furthermore, the closer the ratio to 1, the better the algorithm achieving it.

Theorem 2. *Under the model adopted, FGREEDY is asymptotically optimum for maximum budget saving.*

Proof. Consider an instance (\mathcal{S}, C) of minimum set-covering and denote by k_F and k_L , the sizes of the solutions computed by algorithms FGREEDY and LGREEDY, respectively. By what has been assumed just above, denoting by c_F the cost of using FGREEDY, there exist some $\epsilon > 0$ such that:

$$c_F(\mathcal{S}, C) = k_F + k_L^{1-\epsilon} \quad (14)$$

Moreover, the following inequalities hold, the first one from Slavík (1996) and the second one from Theorem 1:

$$k_F \leq k^* \log n \quad (15)$$

$$k_L \leq k^* \sqrt{n} \quad (16)$$

Using (14), (15) and (16), we get the following inequality for $c_F(\mathcal{S}, C)$:

$$c_F(\mathcal{S}, C) \leq k^{*1-\epsilon} n^{\frac{1-\epsilon}{2}} + k^* \log n \leq \left(n^{\frac{1-\epsilon}{2}} + \log n \right) k^* \quad (17)$$

On the other hand, as assumed above:

$$B(\mathcal{S}, C) = k^* \sqrt{n} \quad (18)$$

Using (13), (17) and (18), we obtain:

$$\begin{aligned} \frac{B(\mathcal{S}, C) - c_F(\mathcal{S}, C)}{B(\mathcal{S}, C) - k^*} &\geq \frac{k^* \sqrt{n} - \left(n^{\frac{1-\epsilon}{2}} + \log n \right) k^*}{k^* \sqrt{n} - k^*} \\ &= \frac{\sqrt{n} - \left(n^{\frac{1-\epsilon}{2}} + \log n \right)}{\sqrt{n} - 1} \quad (19) \end{aligned}$$

It is easy to see that, for n large enough, the last term of (19) tends to 1, and the statement claimed by the theorem is true. ■

Remark also that if we are allotted with a budget equal to $k^* \log n \log m$ (i.e., the cost of the on-line algorithm of Alon *et al.* (2003)) and we assume that

the fine paid by algorithm **FGREEDY** is also computed with respect to the algorithm of Alon *et al.* (2003), then a similar analysis as in the proof of Theorem 2 leads to the same result, i.e., that **FGREEDY** remains asymptotically optimum.

Also, if the budget allotted is $k^* \sqrt{n}$ and one calls the on-line algorithm of Alon *et al.* (2003), this latter algorithm is asymptotically optimum for maximum budget saving.

5 Conclusions

We have introduced an on-line model associated with a natural greedy on-line algorithm achieving non-trivial competitive ratio \sqrt{n} . Moreover, we have shown that this simple algorithm is strongly competitive since no on-line algorithm for this model, even if it introduces in the cover more than one sets at a time, can guarantee better than $\sqrt{n}/2$. One of the features of our model is that the algorithm can run with an extremely small amount of memory and disk requirements and hence it is suitable for solving very large instances.

Next, we have introduced and studied the maximum budget saving problem. Here, we have relaxed irrevocability in the solution construction by allowing the algorithm to delay its decisions modulo some fine to be paid. For such a model we have shown that the natural greedy off-line algorithm is asymptotically optimal.

A subject for further research is the extension of our models to deal with minimum-weight set-covering. For this version work is in progress.

References

- Alon, N., Awerbuch, B., Azar, Y., Buchvinder, N. & Naor, S. (2003), The online set cover problem, in 'Proc. STOC'03', pp. 100–105.
- Chvátal, V. (1979), 'A greedy-heuristic for the set covering problem', *Math. Oper. Res.* **4**, 233–235.
- Feige, U. (1998), 'A threshold of $\ln n$ for approximating set cover', *J. Assoc. Comput. Mach.* **45**, 634–652.
- Gambosi, G., Protasi, M. & Talamo, M. (1997), 'Preserving approximation in the min-weighted set cover problem', *Discrete Appl. Math.* **73**, 13–22.
- Johnson, D. S. (1974), 'Approximation algorithms for combinatorial problems', *J. Comput. System Sci.* **9**, 256–278.
- Karp, R. M. (1972), Reducibility among combinatorial problems, in R. E. Miller & J. W. Thatcher, eds, 'Complexity of computer computations', Plenum Press, New York, pp. 85–103.
- Lovász, L. (1975), 'On the ratio of optimal integral and fractional covers', *Discrete Math.* **13**, 383–390.
- Paschos, V. (1997), 'A survey about how optimal solutions to some covering and packing problems can be approximated', *ACM Comput. Surveys* **29**(2), 171–209.
- Slavík, P. (1996), A tight analysis of the greedy algorithm for set cover, in 'Proc. STOC'96', pp. 435–441.
- Sleator, D. & Tarjan, R. E. (1985), 'Amortized efficiency of list update and paging rules', *Commun. ACM* **28**(2), 202–208.

Author Index

Asahiro, Yuichi, 11
 Asano, Tetsuo, 3
 Ausiello, Giorgio, 145

Blazewicz, Jacek, 93

Dessmark, Anders, 101

Giannakos, Aristotelis, 145
 Gudmundsson, Joachim, iii

Harland, James, 79
 Hasunuma, Toro, 21
 Herlihy, Brian, 123

Jansson, Jesper, 101
 Jay, Barry, iii

Kasprzak, Marta, 93
 Kelarev, A.V., 87
 Klein, Gerwin, 53
 Kolanski, Rafal, 53

Li, Shuai Cheng, 107
 Lingas, Andrzej, 101
 Link, Sebastian, 113
 Lundell, Eva-Marta, 101

Miyano, Eiji, 11

Ono, Hirotaka, 11

Paschos, Vangelis Th., 145
 Persson, Mia, 101
 Poon, Chung Keung, 139

Rao, M.R.K. Krishna, 133

Saabas, Ando, 27
 Schachte, Peter, 123
 Smid, Michiel, 7
 Søndergaard, Harald, 123

Takaoka, Tadao, 69
 Tian, Ye Henry, 41

Uustalu, Tarmo, 27

Violich, Stephen, 69

Xu, Yinfeng, 139

Zenmyo, Kouhei, 11
 Zheng, Feifeng, 139

Recent Volumes in the CRPIT Series

ISSN 1445-1336

Listed below are some of the latest volumes published in the ACS Series *Conferences in Research and Practice in Information Technology*. The full text of most papers (in either PDF or Postscript format) is available at the series website <http://crpit.com>.

Volume 41 - Theory of Computing 2005

Edited by Mike Atkinson, *University of Otago, New Zealand* and Frank Dehne, *Griffith University, Australia*. January, 2005. 1-920-68223-6.

Contains the papers presented at the Eleventh Computing: The Australasian Theory Symposium (CATS2005), Newcastle, NSW, Australia, January/February 2005.

Volume 42 - Computing Education 2005

Edited by Alison L. Young, *UNITEC, New Zealand* and Denise Tolhurst, *University of New South Wales, Australia*. January, 2005. 1-920-68224-4.

Contains the papers presented at the Seventh Australasian Computing Education Conference (ACE2005), Newcastle, NSW, Australia, January/February 2005.

Volume 43 - Conceptual Modelling 2005

Edited by Sven Hartmann, *Massey University, New Zealand* and Markus Stumptner, *University of South Australia*. January, 2005. 1-920-68225-2.

Contains the papers presented at the Second Asia-Pacific Conference on Conceptual Modelling (APCCM2005), Newcastle, NSW, Australia, January/February 2005.

Volume 44 - ACSW Frontiers 2005

Edited by Rajkumar Buyya, *University of Melbourne*, Paul Coddington, *University of Adelaide*, Paul Montague, *Motorola Australia Software Centre*, Rei Safavi-Naini, *University of Wollongong*, Nicholas Sheppard, *University of Wollongong* and Andrew Wendelborn, *University of Adelaide*. January, 2005. 1-920-68226-0.

Contains the papers presented at the Australasian Workshop on Grid Computing and e-Research (AusGrid 2005) and the Third Australasian Information Security Workshop (AISW 2005), Newcastle, NSW, Australia, January/February 2005.

Volume 45 - Information Visualisation 2005

Edited by Seok-Hee Hong, *NICTA, Australia*. January, 2005. 1-920-68227-9.

Contains the papers presented at the Asia-Pacific Symposium on Information Visualisation, APVis.au, Sydney, Australia, January 2005.

Volume 46 - ICT in Education

Edited by Graham Low, *University of New South Wales, Australia*. October, 2005. 1-920-68228-7.

Contains selected refereed papers presented at the South East Asia Regional Computer Confederation (SEARCC) 2005: ICT Building Bridges Conference, Sydney, Australia, September 2005.

Volume 47 - Safety Critical Systems and Software 2004

Edited by Tony Cant, *University of Queensland*. March, 2005. 1-920-68229-5.

Contains all papers presented at the Ninth Australian Workshop on Safety-Related Programmable Systems, (SCS2004), Brisbane, Australia, October 2004.

Volume 48 - Computer Science 2006

Edited by Vladimir Estivill-Castro, *Griffith University* and Gillian Dobbie, *University of Auckland, New Zealand*. January, 2006. 1-920-68230-9.

Contains the papers presented at the Twenty-Ninth Australasian Computer Science Conference (ACSC2006), Hobart, Tasmania, Australia, January 2006.

Volume 49 - Database Technologies 2006

Edited by Gillian Dobbie, *University of Auckland, New Zealand* and James Bailey, *University of Melbourne*. January, 2006. 1-920-68231-7.

Contains the papers presented at the Seventeenth Australasian Database Conference (ADC2006), Hobart, Tasmania, Australia, January 2006.

Volume 50 - User Interfaces 2006

Edited by Wayne Piekarski, *University of South Australia*. January, 2006. 1-920-68232-5.

Contains the papers presented at the Seventh Australasian User Interface Conference (AUIC2006), Hobart, Tasmania, Australia, January 2006.

Volume 51 - Theory of Computing 2006

Edited by Barry Jay, *UTS, Australia* and Joachim Gudmundsson, *NICTA, Australia*. January, 2006. 1-920-68233-3.

Contains the papers presented at the Twelfth Computing: The Australasian Theory Symposium (CATS2006), Hobart, Tasmania, Australia, January 2006.

Volume 52 - Computing Education 2006

Edited by Denise Tolhurst, *University of New South Wales, Australia* and Samuel Mann, *Otago Polytechnic, Otago, New Zealand*. January, 2006. 1-920-68234-1.

Contains the papers presented at the Eighth Australasian Computing Education Conference (ACE2006), Hobart, Tasmania, Australia, January 2006.

Volume 53 - Conceptual Modelling 2006

Edited by Markus Stumptner, *University of South Australia*, Sven Hartmann, *Massey University, New Zealand* and Yasushi Kiyoki, *Keio University, Japan*. January, 2006. 1-920-68235-X.

Contains the papers presented at the Third Asia-Pacific Conference on Conceptual Modelling (APCCM2006), Hobart, Tasmania, Australia, January 2006.

Volume 54 - ACSW Frontiers 2006

Edited by Rajkumar Buyya, *University of Melbourne*, Tianchi Ma, *University of Melbourne*, Rei Safavi-Naini, *University of Wollongong*, Chris Steketee, *University of South Australia* and Willy Susilo, *University of Wollongong*. January, 2006. 1-920-68236-8.

Contains the papers presented at the Fourth Australasian Workshop on Grid Computing and e-Research (AusGrid 2006) and the Fourth Australasian Information Security Workshop (AISW 2006), Hobart, Tasmania, Australia, January 2006.

Volume 55 - Safety Critical Systems and Software 2005

Edited by Tony Cant, *University of Queensland*. Late 2005. 1-920-68237-6.

Contains all papers presented at the 10th Australian Workshop on Safety Related Programmable Systems, August 2005, Sydney, Australia.

Volume 56 - Visual Information Processing 2005

Edited by Hong Yan, *City University of Hong Kong*, Jesse Jin, *University of Newcastle, Australia*, Zhi-qiang Liu, *City University of Hong Kong* and Daniel Yeung, *Hong Kong Polytechnic University*. Late 2005. 1-920-68238-4.

Contains papers from the Asia-Pacific Workshop on Visual Information Processing (VIP2005), Hong Kong, December 2005.

Volume 57 - Multimodal User Interaction 2005

Edited by Fang Chen and Julien Epps, *National ICT Australia*. December, 2005. 1-920-68239-2.

Contains the proceedings of the Multimodal User Interaction Workshop 2005, NICTA-HCSNet, Sydney, Australia, 13-14 September 2005.

Volume 58 - Advances in Ontologies 2005

Edited by Thomas Meyer, *National ICT Australia, Sydney* and Mehmet Orgun, *Macquarie University*. December, 2005. 1-920-68240-6.

Contains the proceedings of the Australasian Ontology Workshop (AOW 2005), Sydney, Australia, 6 December 2005.