The book "Conferences in Research and Practice in Information Technology, Volume 141" contains the proceedings of the "Theory of Computing 2013" conference. This volume is published in "Australian Computer Science Communications, Volume 35, Number 8".
Table of Contents


Preface ........................................................................................................................................................................ vii
Programme Committee .................................................................................................................................................... viii
Organising Committee .................................................................................................................................................... ix
Welcome from the Organising Committee ................................................... x
CORE - Computing Research & Education ........................................................... xi
ACSW Conferences and the Australian Computer Science Communications ........................................................................... xii
ACSW and CATS 2013 Sponsors ................................................................................................................................. xiv

Contributed Papers

Range-Aggregate Queries for Geometric Extent Problems .......................................................... 3
Peter Brass, Christian Knauer, Chan-Su Shin, Michiel Smid, Ivo Vigan

Competitive Online Algorithms for Multiple-Machine Power Management and Weighted Flow Time . 11
Ho-Leung Chan, Sze-Hang Chan, Tak-Wah Lam, Lap-Kei Lee, Rongbin Li, Chi-Man Liu

Rotated Library Sort ......................................................................................................................... 21
Franky Lam, Raymond K. Wong

Improved Inapproximability Results for the Shortest Superstring and Related Problems .............. 27
Marek Karpinski, Richard Schmied

Complexity of Counting Output Patterns of Logic Circuits ........................................................... 37
Kei Uchizawa, Zhenghong Wang, Hiroki Morizumi, Xiao Zhou

Busy Beaver Machines and the Observant Otter Heuristic ........................................................ 43
James Harland

Author Index ............................................................................................................................................................... 53
Preface

The papers in this volume were presented at the 19th edition of Computing: Australasian Theory Symposium (CATS 2013), held on 30 January 2013 in Adelaide, South Australia. They comprise Volume 141 of the Conferences in Research and Practice in Information Technology (CRPIT).

The 21-member program committee accepted 7 of the 15 submissions. The committee was ably assisted by 13 external reviewers. We thank everyone for their commitment towards the quality of this Symposium.

Given the time constraints, we at least attempted to verify the main claims in all submissions, and we provided technical feedback to authors. We hope that most of the accepted papers will appear subsequently, possibly in greater detail, in peer-reviewed scientific journals.

I thank the ACSW team, especially Alex Potanin and Ivan Lee, for their continued generous assistance, Simeon Simoff from CRPIT for publication advice, Tony Zara for managing the CATS 2013 website, and Julián Mestre and Tasos Viglas for their invaluable experience as past CATS PC chairs.

Anthony Wirth
University of Melbourne
CATS 2013 Programme Committee Chair
January 2013
Programme Committee

Chair

Anthony Wirth, University of Melbourne, Australia

Members

Sang Won Bae, Kyonggi University, South Korea
Amit Chakrabarti, Dartmouth College, USA
Ning Chen, Nanyang Technological University, Singapore
Daniel Delling, Microsoft Research, USA
Michael Dinitz, The Weizmann Institute of Science Rehovot, Israel
Riccardo Dondi, Università degli studi di Bergamo, Italy
Vida Dujmovic, Carleton University, Canada
Fabrizio Frati, Universita’ Roma Tre, Italy / University of Sydney, Australia
Inge Gørtz, Technical University of Denmark, Denmark
Joachim Gudmundsson, University of Sydney, Australia
Martin Hoefer, RWTH Aachen University, Germany
Nicole Immorlica, Northwestern University, USA
Catherine McCartin, Massey University, New Zealand
Mirka Miller, University of Newcastle, Australia
Kerri Morgan, Monash University, Australia
Atri Rudra, University at Buffalo, USA
Harald Søndergaard, University of Melbourne, Australia
David Steurer, Microsoft Research, USA
Prudence Wong, University of Liverpool, U.K.
Dinghao Wu, Pennsylvania State University, USA

Additional Reviewers

Philip Bille
Gianluca Della Vedova
Ramiro Feria-Puron
Shinhaeng Jo
Matias Korman
Chris Martens
Heiner Marxen

Pascal Michel
Hebert Perez-Roses
Yuri Pirola
Paul Subotic
Jiajun Wu
Yitong Yin
Organising Committee

Chair
Dr. Ivan Lee

Finance Chair
Dr. Wolfgang Mayer

Publication Chair
Dr. Raymond Choo

Local Arrangement Chair
Dr. Grant Wigley

Registration Chair
Dr. Jinhai Cai
Welcome from the Organising Committee

On behalf of the Organising Committee, it is our pleasure to welcome you to Adelaide and to the 2013 Australasian Computer Science Week (ACSW 2013). Adelaide is the capital city of South Australia, and it is one of the most liveable cities in the world. ACSW 2013 will be hosted in the City West Campus of University of South Australia (UniSA), which is situated at the north-west corner of the Adelaide city centre.

ACSW is the premier event for Computer Science researchers in Australasia. ACSW2013 consists of conferences covering a wide range of topics in Computer Science and related area, including:

- Australasian Computer Science Conference (ACSC) (Chaired by Bruce Thomas)
- Australasian Database Conference (ADC) (Chaired by Hua Wang and Rui Zhang)
- Australasian Computing Education Conference (ACE) (Chaired by Angela Carbone and Jacqueline Whalley)
- Australasian Information Security Conference (AISC) (Chaired by Clark Thomborson and Udaya Parampalli)
- Australasian User Interface Conference (AUIC) (Chaired by Ross T. Smith and Burkhard C. Wünsche)
- Computing: Australasian Theory Symposium (CATS) (Chaired by Tony Wirth)
- Australasian Symposium on Parallel and Distributed Computing (AusPDC) (Chaired by Bahman Javadi and Saurabh Kumar Garg)
- Australasian Workshop on Health Informatics and Knowledge Management (HIKM) (Chaired by Kathleen Gray and Andy Koronios)
- Asia-Pacific Conference on Conceptual Modelling (APCCM) (Chaired by Flavio Ferrarotti and Georg Grossmann)
- Australasian Web Conference (AWC2013) (Chaired by Helen Ashman, Michael Sheng and Andrew Trotman)

In addition to the technical program, we also put together social activities for further interactions among our participants. A welcome reception will be held at Rockford Hotel’s Rooftop Pool area, to enjoy the fresh air and panoramic views of the cityscape during Adelaide’s dry summer season. The conference banquet will be held in Adelaide Convention Centre’s Panorama Suite, to experience an expansive view of Adelaide’s serene riverside parklands through the suite’s seamless floor to ceiling windows.

Organising a conference is an enormous amount of work even with many hands and a very smooth cooperation, and this year has been no exception. We would like to share with you our gratitude towards all members of the organising committee for their dedication to the success of ACSW2013. Working like one person for a common goal in the demanding task of ACSW organisation made us proud that we got involved in this effort. We also thank all conference co-chairs and reviewers, for putting together conference programs which is the heart of ACSW. Special thanks goes to Alex Potanin, who shared valuable experiences in organising ACSW and provided endless help as the steering committee chair. We’d also like to thank Elyse Perin from UniSA, for her true dedication and tireless work in conference registration and event organisation. Last, but not least, we would like to thank all speakers and attendees, and we look forward to several stimulating discussions.

We hope your stay here will be both rewarding and memorable.

Ivan Lee
School of Information Technology & Mathematical Sciences
ACSW2013 General Chair
January, 2013
CORE welcomes all delegates to ACSW2013 in Adelaide. CORE, the peak body representing academic computer science in Australia and New Zealand, is responsible for the annual ACSW series of meetings, which are a unique opportunity for our community to network and to discuss research and topics of mutual interest. The original component conferences - ACSC, ADC, and CATS, which formed the basis of ACSW in the mid 1990s - now share this week with eight other events - ACE, AIASC, AUIC, AusPDC, HIKM, ACDC, APCCM and AWC which build on the diversity of the Australasian computing community.

In 2013, we have again chosen to feature a small number of keynote speakers from across the discipline: Riccardo Bellazzi (HIKM), and Divyakant Agrawal (ADC), Maki Sugimoto (AUIC), and Wen Gao. I thank them for their contributions to ACSW2013. I also thank invited speakers in some of the individual conferences, and the CORE award winner Michael Sheng (CORE Chris Wallace Award). The efforts of the conference chairs and their program committees have led to strong programs in all the conferences, thanks very much for all your efforts. Thanks are particularly due to Ivan Lee and his colleagues for organising what promises to be a strong event.

The past year has been turbulent for our disciplines. ERA2012 included conferences as we had pushed for, but as a peer review discipline. This turned out to be good for our disciplines, with many more Universities being assessed and an overall improvement in the visibility of research in our disciplines. The next step must be to improve our relative success rates in ARC grant schemes, the most likely hypothesis for our low rates of success is how harshly we assess each others’ proposals, a phenomenon which demonstrably occurs in the US NFS. As a US Head of Dept explained to me, ”in CS we circle the wagons and shoot within”.

Beyond research issues, in 2013 CORE will also need to focus on education issues, including in Schools. The likelihood that the future will have less computers is small, yet where are the numbers of students we need? In the US there has been massive growth in undergraduate CS numbers of 25 to 40% in many places, which we should aim to replicate. ACSW will feature a joint CORE, ACDCIT, NICTA and ACS discussion on ICT Skills, which will inform our future directions.

CORE’s existence is due to the support of the member departments in Australia and New Zealand, and I thank them for their ongoing contributions, in commitment and in financial support. Finally, I am grateful to all those who gave their time to CORE in 2012; in particular, I thank Alex Potanin, Alan Fekete, Aditya Ghose, Justin Zobel, John Grundy, and those of you who contribute to the discussions on the CORE mailing lists. There are three main lists: csprofs, cshods and members. You are all eligible for the members list if your department is a member. Please do sign up via http://lists.core.edu.au/mailman/listinfo - we try to keep the volume low but relevance high in the mailing lists.

I am standing down as President at this ACSW. I have enjoyed the role, and am pleased to have had some positive impact on ERA2012 during my time. Thank you all for the opportunity to represent you for the last 3 years.

Tom Gedeon
President, CORE
January, 2013
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.

**2014. Volume 36. Host and Venue - AUT University, Auckland, New Zealand.**

**2013. Volume 35. Host and Venue - University of South Australia, Adelaide, SA.**

**2012. Volume 34. Host and Venue - RMIT University, Melbourne, VIC.**

**2011. Volume 33. Host and Venue - Curtin University of Technology, Perth, WA.**

**2010. Volume 32. Host and Venue - Queensland University of Technology, Brisbane, QLD.**

**2009. Volume 31. Host and Venue - Victoria University, Wellington, New Zealand.**

**2008. Volume 30. Host and Venue - University of Wollongong, NSW.**

**2007. Volume 29. Host and Venue - University of Ballarat, VIC. First running of HDKM.**

**2006. Volume 28. Host and Venue - University of Tasmania, TAS.**

**2005. Volume 27. Host - University of Newcastle, NSW. APBC held separately from 2005.**


**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.**


**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.**


**1995. Volume 17. Hosts - Flinders University, University of Adelaide and University of South Australia. Venue - Glenelg, SA.**


**1992. Volume 14. Host and Venue - University of Tasmania, TAS. (ADC held separately at La Trobe University).**


**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.**


**1987. Volume 9. Host and Venue - Deakin University, VIC.**

**1986. Volume 8. Host and Venue - Australian National University, Canberra, ACT.**


**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

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACDC</td>
<td>Australasian Computing Doctoral Consortium</td>
</tr>
<tr>
<td>ACE</td>
<td>Australasian Computer Education Conference</td>
</tr>
<tr>
<td>ACSC</td>
<td>Australasian Computer Science Conference</td>
</tr>
<tr>
<td>ACSW</td>
<td>Australasian Computer Science Week</td>
</tr>
<tr>
<td>ADC</td>
<td>Australasian Database Conference</td>
</tr>
<tr>
<td>AISC</td>
<td>Australasian Information Security Conference</td>
</tr>
<tr>
<td>APCCM</td>
<td>Asia-Pacific Conference on Conceptual Modelling</td>
</tr>
<tr>
<td>AUIC</td>
<td>Australasian User Interface Conference</td>
</tr>
<tr>
<td>AusPDC</td>
<td>Australasian Symposium on Parallel and Distributed Computing (replaces AusGrid)</td>
</tr>
<tr>
<td>AWC</td>
<td>Australasian Web Conference</td>
</tr>
<tr>
<td>CATS</td>
<td>Computing; Australasian Theory Symposium</td>
</tr>
<tr>
<td>HIKM</td>
<td>Australasian Workshop on Health Informatics and Knowledge Management</td>
</tr>
</tbody>
</table>

Note that various name changes have occurred, which have been indicated in the Conference Acronyms sections in respective CRPIT volumes.
ACSW and CATS 2013 Sponsors

We wish to thank the following sponsors for their contribution towards this conference.

CORE - Computing Research and Education,
www.core.edu.au

Australian Computer Society,
www.acs.org.au

University of South Australia,
www.unisa.edu.au/
Range-Aggregate Queries for Geometric Extent Problems

Peter Brass1 Christian Knauer2 Chan-Su Shin3 Michiel Smid4 Ivo Vigan5

1 Department of Computer Science
The City College of New York, New York, NY, 10031, USA.
Email: peter@cs.ccny.cuny.edu

2 Institute of Computer Science
Universität Bayreuth, 95440 Bayreuth, Germany.
Email: christian.knauer@uni-bayreuth.de

3 Department of Digital Information Engineering
Hankuk University of Foreign Studies, Yongin, 449-791, Korea.
Email: cssin@hufs.ac.kr

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

5 Department of Computer Science
City University of New York, The Graduate Center, New York, NY 10016, USA
Email: ivigan@gc.cuny.edu

Abstract

Let $S$ be a set of $n$ points in the plane. We present data structures that solve range-aggregate query problems on three geometric extent measure problems. Using these data structures, we can report, for any axis-parallel query rectangle $Q$, the area/perimeter of the convex hull, the width, and the radius of the smallest enclosing disk of the points in $S \cap Q$.

Keywords: Computational geometry, range-aggregate query, orthogonal range query, convex hull, width, smallest enclosing disk.

1 Introduction

In the range searching problem, we have to preprocess a given set $S$ of geometric objects (such as points) into a data structure such that, for any query range $Q$, the value $f(S \cap Q)$ can be computed efficiently. Examples of such functions $f$ include algebraic ones such as “sum”, “minimum”, and “maximum” over the weights pre-assigned to the objects of $S$, and geometric ones, such as “closest pair”, “diameter”, and “width” of $S \cap Q$.


In general, geometric aggregate functions are not decomposable, i.e., the answer $f(S \cap Q)$ cannot be derived efficiently from the answers of the subsets which form a partition of $S \cap Q$. Because of this, different techniques have to be developed to answer these kind of queries.

Let $S$ be a set of $n$ points in the plane. For the case when $f$ is the function “closest pair” and $Q$ is an axis-parallel rectangle, (Sharathkumar & Gupta 2007) and (Gupta et al. 2009) present data structures that solve the problem using $O(n \cdot \text{polylog}(n))$ space and $O(\text{polylog}(n))$ query time. These structures, however, have $\Omega(n^2)$ preprocessing time. (Abam et al. 2009) show that variants of these structures can be constructed in $O(n \cdot \text{polylog}(n))$ time, while preserving the bounds on the space and query time. (Abam et al. 2009) also show that closest pair queries in a halfplane can be answered in close to $O(\sqrt{n})$ time using $O(n \cdot \text{polylog}(n))$ space.

For the case when $f$ is the function “maximal points” and $Q$ is an axis-parallel rectangle, (Brodal & Tsakalidis 2011) and (Das et al. 2012) present data structures having size $O(n \cdot \text{polylog}(n))$ and query time $O(\text{polylog}(n) + k)$, where $k$ is the size of the output.

Assume that $f$ is the function “diameter” and $Q$ a data structure such that, for any given query range $Q$, the value $f(S \cap Q)$ can be computed efficiently.
is an axis-parallel rectangle. Let $k$ be a parameter with $1 \leq k \leq n$. Gupta et al. (2009) present a data structure of size $O((n+(n/k)^2)\log^5 n)$ having query time $O(k \log^5 n)$. If we aim for $O(polylog(n))$ query time, this structure uses close to quadratic space. On the other hand, using $O(n \cdot polylog(n))$ space, the query time will be $\Omega(\sqrt{n})$. In fact, (Davoodi et al. 2012) present evidence that this trade-off cannot be improved.

Let $f$ be a function that can be approximated using coresets; examples are “diameter”, “width”, and “smallest enclosing disk”. For the case when $Q$ is an axis-parallel rectangle, an approximation to $f(S \cap Q)$ can be computed in $O(polylog(n))$ time using $O(n \cdot polylog(n))$ space; see (Gupta et al. 2009, Nekrich & Smid 2010).

1.1 Our contributions

We present data structures for solving exactly range-aggregate query problems on $S$ of $n$ points in the plane and axis-parallel query regions $Q$, for three geometric extent measures: the area/perimeter of the convex hull of $S \cap Q$, the width of $S \cap Q$, and the radius of the smallest enclosing disk of $S \cap Q$. Our results are summarized in Table 1. These are the first non-trivial results for solving these queries exactly; previously, only non-trivial results were known for approximation versions of these query problems.

2 Preliminaries

Let $S$ be a set of $n$ points in the plane. We assume that the points in $S$ are in general position. Let $Q := [a_x, b_x] \times [a_y, b_y]$ be a query range, where $a_x \leq b_x$ and $a_y \leq b_y$.

A standard method to identify $S \cap Q$ is by storing the points of $S$ in a range tree; see (de Berg et al. 2008). Using this data structure, identifying $S \cap Q$ is done in two phases: (1) Find all points of $S$ lying in the vertical strip of $Q$ defined by the $x$-interval $[a_x, b_x]$. (2) Select the points in the $y$-interval $[a_y, b_y]$ among the points in $[a_x, b_x]$.

Let $T$ be a primary range tree, i.e., a balanced binary search tree in which the leaf nodes store the points in $S$ in non-decreasing order of their $x$-coordinates from left to right. For every node $u$, we denote by $S(u)$ the canonical set of points in $S$ that are stored at the leaf nodes of the subtree rooted at $u$. It is well-known that a subset of points in the $x$-interval $[a_x, b_x]$ can be represented as the disjoint union of $O(\log n)$ canonical subsets. If $S(u)$ contributes to $O(\log n)$ of these canonical subsets, then $u$ is called a canonical node for the interval. For a given $x$-interval, we can identify these canonical nodes in $O(\log n)$ time by traversing $T$ from the root.

We associate each node $u$ in $T$ with a secondary range tree $T_y(u)$, built on the $y$-coordinates of the points in $S(u)$. Then we can identify $O(\log n)$ canonical subsets (or nodes) in $T_y(u)$ whose points lie in $[a_y, b_y]$.

As a result, for a given query range $Q$, we can compute, in $O(\log^2 n)$ time, a sequence of $O(\log^2 n)$ canonical nodes $v_1, \ldots, v_m$ in the secondary range trees, such that

$$S \cap Q = \bigcup_{i=1}^m S(v_i).$$

In addition, we will associate each node in each $T_y(\cdot)$ with additional preprocessed information depending on the individual problem.

The width of a point set $V$ is defined to be the minimum distance between any two parallel lines such that $V$ is contained in the strip bounded by these lines. A smallest enclosing disk is a minimum-radius disk which encloses all points of $V$.

Let $ch(S(u))$, width($S(u)$), and sed($S(u)$) (in short, ch($u$), width($u$), and sed($u$)) denote the convex hull, the width, and the smallest enclosing disk for the points of $S(u)$, for any node $u$ in the range tree. We use $|S|$ and $|T|$ to denote the cardinality of the set $S$ and the number of nodes in the tree $T$.

3 Convex hull queries

We consider the problem of computing the area and perimeter of ch($S \cap Q$) for a given axis-parallel query rectangle $Q$.

Additional information. We maintain a two-dimensional range tree mentioned in the previous section. At each node $v$ of each secondary range tree $T_y(\cdot)$, we store three additional pieces of information:

1. ch($v$), the convex hull of $S(v)$,
2. area(ch($v$)), the area of ch($v$), and
3. $T_A(v)$, a balanced binary search tree whose leaf nodes store the points of ch($v$) in counterclockwise order. Let $z$ be an internal node of $T_A(v)$ with at least three points $p_i, p_{i+1}, \ldots, p_j$, $i < j$, at the leaf nodes of its subtree in $T_A(v)$. At each $z$, we store the area of ch($v$) to the right of $p_j p_i$, i.e., area(ch($\{p_i, \ldots, p_j\}$)). Once we know this area, we can easily get the area of ch($v$) to the left of $p_j p_i$ as area(ch($v$)) – area(ch($\{p_i, \ldots, p_j\}$)).

Let us check how much space this additional information requires. For any node $v$ in a fixed secondary range tree $T_y(u)$ for some $u \in T$, it takes $O(|S(v)| \log |S(v)|)$ time and $O(|S(v)|)$ space to store both ch($v$) and area(ch($v$)). Computing the area information stored in $T_A(v)$ is done in a bottom-up fashion, thus it takes a time of $O(|S(v)| \log |S(v)|)$ and a

<table>
<thead>
<tr>
<th>query type</th>
<th>query time</th>
<th>preprocessing time</th>
<th>preprocessing space</th>
<th>reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>convex hull</td>
<td>O(log^5 n)</td>
<td>O(n log^4 n)</td>
<td>O(n log^2 n)</td>
<td>Theorem 1</td>
</tr>
<tr>
<td>area/perimeter report</td>
<td>O(log^5 n + h)</td>
<td>O(n log^4 n)</td>
<td>O(n log^2 n)</td>
<td></td>
</tr>
<tr>
<td>width</td>
<td>O(k log^4 n + log^8 n)</td>
<td>O((n^2/k) log^7 n)</td>
<td>O((n^2/k) log^6 n)</td>
<td>Theorem 2</td>
</tr>
<tr>
<td>smallest enclosing disk</td>
<td>O(log^8 n)</td>
<td>O(n log^2 n)</td>
<td>O(n log^2 n)</td>
<td>Theorem 3</td>
</tr>
</tbody>
</table>

Table 1: Our contributions; $k$ is a parameter between 1 and $n$. $h$ is the output size of the convex hull.
space of $O(|S(v)|)$. For a fixed $T_y(u)$, we need a time of
$$\sum_{v \in T_y(u)} O(|S(v)| \log |S(v)|) = O(|T_y(u)| \log^2 |T_y(u)|),$$
and a space of
$$\sum_{v \in T_y(u)} O(|S(v)|) = O(|T_y(u)| \log |T_y(u)|).$$

Thus, for the range tree $T$, we need a time of
$$\sum_{u \in T} O(|T_y(u)| \log^2 |T_y(u)|) = O(|T| \log^3 |T|),$$
which is $O(n \log^3 n)$. By a similar analysis, we need $O(n \log^2 n)$ space. As a result, the data structure can be built in $O(n \log^3 n)$ time and uses $O(n \log^2 n)$ space.

**Query.** To compute $\text{ch}(S \cap Q)$ and its area for the query range $Q$, we first identify $O(\log^2 n)$ canonical nodes $v_1, \ldots, v_m$ in all secondary range trees such that $S \cap Q = \bigcup_{1 \leq i \leq m} S(v_i)$. Using $(v_i)$ stored at each canonical node $v_i$, we next compute $\text{ch}(S \cap Q) = \text{ch}(\text{ch}(v_1) \cup \ldots \cup \text{ch}(v_m))$ together with its area in $O(\log^3 n)$ time as follows.

To compute $\text{ch}(S \cap Q)$, we compute two outer tangents between convex hulls $\text{ch}(v_i)$ and $\text{ch}(v_j)$, i.e., tangent lines containing two hulls in their same sides, for all pairs $(v_i, v_j)$ with $i \neq j$. Since any two convex hulls are disjoint, we can apply the prune-and-search algorithm by Kirkpatrick & Snoeyink (1995) to compute them in $O(\log(|S(v_i)| + |S(v_j)|)) = O(\log n)$ time.

This takes $O(\log^5 n)$ total time. For each $\text{ch}(v_i)$, we now collect the tangents incident to each point $p_k \in \text{ch}(v_i)$ which has at least one tangent. Let $e$ and $e'$ be two edges incident to $p_k$ on $\text{ch}(v_i)$ where $e$ appears before $e'$ in counterclockwise order. Among the tangents from $p_k$, we choose the one with the smallest angle with respect to the line containing $e$ in counterclockwise direction. Denote the chosen tangent by $t_k$. We store such $t_k$ for all points $p_k$ having at least one tangent in a list $L_t$ in counterclockwise order. Clearly $L_t$ contains $O(\log^2 n)$ tangents, which can be sorted angularly in $O(\log^2 n \log \log n)$ time. As a result, the ordered lists $L_t$ for all $1 \leq i \leq m$ can be computed in $O(\log^5 n)$ time.

We now compute $\text{ch}(S \cap Q)$ by traversing the computed tangents. The method is similar to the gift-wrapping convex hull algorithm (O’Rourke 1998). We start with the southermmost $\text{ch}(v_i)$. Starting from $p_k$, we want to find the first point in counterclockwise direction along the edges of $\text{ch}(v_i)$ which has a tangent in $L_t$. This is equivalent to finding two consecutive points in $L_t$ such that $p_k$ lies between the two points along the boundary of $\text{ch}(v_i)$. This can be done by a binary search over the indices of the points in $L_t$, which takes $O(\log |L_t|) = O(\log \log n)$ time. Traverse the found tangent, say $t_l$, to reach a point $p_l$ in another convex hull $\text{ch}(v_j)$. Again we use the list $L_l$ to find the next point on $\text{ch}(v_j)$ which has a tangent, and traverse it. We continue in this way until we return to $\text{ch}(v_i)$, and thereby complete the construction of $\text{ch}(\text{ch}(v_1) \cup \ldots \cup \text{ch}(v_m))$ in $O(2^k \log \log n)$ time.

As a result, we can compute $\text{ch}(S \cap Q)$ in $O(2^k \log \log n)$ time, in the sense that if we have to report its edges explicitly, then we can do it in $O(\log^5 n + |\text{ch}(S \cap Q)|)$ time.

Finally we calculate area($\text{ch}(S \cap Q)$). Consider any $\text{ch}(v_i)$ whose boundary appears on the boundary of $\text{ch}(S \cap Q)$. Then, as shown in Figure 1, we assume that the boundary of $\text{ch}(v_i)$ appears on the boundary $\text{ch}(S \cap Q)$ from a point $p_a$ to a point $p_b$. Let $C_i$ be the convex polygon with points $p_a, p_{a+1}, \ldots, p_{b-1}, p_b$. We now compute area($C_i$) as follows. We traverse $\text{ch}(v_i)$ to search the leaf nodes storing $p_a$ and $p_b$, and obtain two paths to $p_a$ and $p_b$. We collect the positive areas stored at the canonical nodes “below” both paths in $\text{ch}(v_i)$. Since the positive area is defined for three or more consecutive points in $\text{ch}(v_i)$, deleting such points from $C_i$ results in a smaller convex polygon $C'_i \subset C_i$. Then $C'_i$ consists of $O(\log n)$ points because there are $O(\log n)$ canonical nodes in $\text{ch}(v_i)$ and among them only $O(1)$ canonical nodes have one or two points in their subtrees. Now area($C_i$) is the sum of the areas stored at the nodes with positive area plus area($C'_i$).

Since area($C'_i$) can be directly computed in $O(|C'_i|)$ = $O(\log n)$ time, we can compute area($C_i$) in $O(\log n)$ time. By the same method, we compute area($C_i$) for all $\text{ch}(v_i)$ which belong to the boundary of $\text{ch}(S \cap Q)$ in $O(\log^3 n)$ time.

Let $C := \bigcup_i C_i$ and $C' := \text{ch}(S \cap Q) \setminus C$; see Figure 2. Note that area($C$) = $\sum_i$ area($C_i$), so it can be computed in $O(\log^2 n)$ time. In the worst case, all the canonical nodes $\text{ch}(v_i)$ can contribute to the boundary of $\text{ch}(S \cap Q)$, so $C'$ can have $O(\log^2 n)$ points on its boundary and area($C'$) can be directly computed in the same time. Since area($\text{ch}(S \cap Q)$) =
The area of the convex hull of a set of points in a plane is given by the formula $\text{area}(C) + \text{area}(C')$, where $\text{area}(\text{conv}(S \cap Q))$ can be computed in $O(\log^2 n)$ time.

The most time-consuming step in the computation of $\text{area}(\text{conv}(S \cap Q))$ is to compute the tangents for all possible pairs of $O(n^2)$ canonical nodes in $S \cap Q$. Thus we can answer $\text{area}(\text{conv}(S \cap Q))$ in $O(\log^2 n)$ time. The perimeter of $\text{conv}(S \cap Q)$ can be obtained in a similar way.

**Theorem 1** Let $S$ be a set of $n$ points in the plane. In $O(n \log^2 n)$ time, we can construct a data structure of size $O(n \log^2 n)$, such that for any axis-parallel query rectangle $Q$, we can report $\text{conv}(S \cap Q)$ in $O(\log^2 n + K)$ time and compute the area or the perimeter of $\text{conv}(S \cap Q)$ in $O(\log^2 n)$ time. Here $K = |\text{conv}(S \cap Q)|$.

**4 Width queries**

For the diameter query problem, the diameter $diam(S \cap Q)$ is determined by exactly two points of $S \cap Q$. Each of them belongs to one of the $O(n^2)$ canonical subContainments $(S(v_1), \ldots, S(v_m))$. If $i$ and $j$ are the indices such that $S(v_i)$ and $S(v_j)$ contain these two points (with $i \neq j$ being possible), then, $diam(S \cap Q) = diam(S(v_i) \cup S(v_j))$. As a result, $\text{diam}(S \cap Q)$ is the maximum value of $\text{diam}(S(v_i) \cup S(v_j))$ for all pairs $1 \leq i, j \leq m$. If we compute in advance a table of $\text{diam}(S(v_i) \cup S(v_j))$ for all pairs $(v_i, v_j)$, then we can simply look up the entries in the table which correspond to the canonical subset pairs for $S \cap Q$.

This approach is not applicable to the width problem: The value of width($S \cap Q$) is determined by three points of $S$ and we can easily construct an example for which width($S \cap Q$) is not in the set of width($S(v_i) \cup S(v_j) \cup S(v_k)$) over all triples of canonical subContainments for $S \cap Q$. Furthermore, the width problem is essentially a non-convex optimization problem, unlike the smallest enclosing circle problem which can use properties associated with convex programming (Eppstein 1992). This is what makes the width problem difficult.

**Additional information.** To store an additional information, we use data structures that (Chan 2003) uses to maintain the width of a set of points in a dynamic way.

The width of $S$, $\text{width}(S)$, is determined by three points on $\text{conv}(S)$. We consider a dual transformation such that a point $(a, b)$ in the primal plane maps to the line $y = ax + b$ in the dual plane. Then, in the dual plane, the set of all lines above the convex hull $S$ becomes an unbounded convex polygon $A$ in the positive $y$-direction, and the set of all lines below the convex hull becomes an unbounded convex polygon $B$ in the negative $y$-direction; see Figure 3. The strip containing $S$ is mapped to a vertical segment in the dual plane which connects either a vertex of $\partial A$ and a point on $\partial B$ or a point on $\partial A$ and a vertex of $\partial B$.

So width$(S)$ is attained by the minimum vertical distance between $\partial A$ and $\partial B$. If we denote by $d(A, B)$ the minimum vertical distance between $\partial A$ and $\partial B$ in the dual plane, then width$(S) = d(A, B)$.

Chan (2003) built two data structures $Y(A)$ and $Z(A, B)$ for two convex hulls $A$ and $B$ defined above in the dual plane, which support the following queries:

1. $Y(A)$ can compute $d(A, e)$, for any query line segment $e$ below $A$, in $O(|\log^2 |A||)$ time. $Y(B)$ is defined in a symmetric way.
2. $Z(A, B)$ can compute $d(A, \gamma)$, for any chain $\gamma \subset \partial B$ with two arbitrary endpoints on $\partial B$, in $O(|\log |B||)$ time. $Z(B, A)$ is defined similarly. In fact, $Z(A, B)$ and $Z(B, A)$ are based on $Y(A)$ and $Y(B)$.

The data structure $Y(A)$ can be built in $O(|A| \log^2 |A|)$ time and space, and $Z(A, B)$ in $O(|A| \log^2 |A| + |B| \log^2 |B|)$ time and space (Chan 2003).

For a fixed parameter $1 \leq k \leq n$, a node $v$ in any secondary structure $T_y(v)$ (in short, $T_y$) is said to be big if $|S(v)| \geq k$, otherwise small. In a fixed $T_y$, there are $O(|T_y|/k)$ big nodes and the number of levels of $T_y$ containing big nodes is $O(\log(|T_y|/k))$.

At each “big” node $v$ in any $T_y$, we maintain the following three additional pieces of information:

1. $A(v)$ and $B(v)$ as the dual structure for $\text{conv}(S(v))$,
2. $Y(A(v))$ and $Y(B(v))$ by Chan (Chan 2003),
3. $Z(A(v), B(v))$ and $Z(B(v), A(w))$ for every big node $w$ in any $T_y$ by Chan (Chan 2003).

Let us check the amount of space needed for these structures. If $T_y$ has a big node, then $T_y$ has at least $k$ nodes, so there are only $O(|T_y|/k) = O(n/k)$ internal nodes in the primary tree $T$ having such $T_y$.

Since $T_y$ can have at most $|T_y|/k$ big nodes, the number of big nodes in all $T_y$ stored at one level in $T$ is $O(|T_y|/k) = O(n/k)$. In total, there are $O((n/k) \log(n/k))$ big nodes.

For the first additional information, we need $O(|T_y|)$ space for all big nodes at the same level in a fixed $T_y$. Since $T_y$ has $O(\log(n/k))$ levels, we need $O(|T_y| \log(n/k))$ space for $T_y$. For $T_y$ stored at the same level of $T$, the required space is $O(\sum |T_y| \log(n/k)) = O(n \log(n/k))$. Only $O(\log(n/k))$ levels in $T$ store $T_y$ which contains big nodes, so the space amount for the first one is $O(n \log^2(n/k))$.

For the second one, we can similarly check that it needs $O(n \log^2 n \log^2(n/k))$ space.

For the third one, we first check the space needed for $Z(A(v), \cdot)$ and $Z(B(v), \cdot)$ of a fixed $v$. The space associated with $w$ in some $T_y$ is $O(|T_y| \log^2 n \log(n/k))$. Summing up all $T_y$, it becomes $O(n \log^2 n \log^2(n/k))$. Since we have $O((n/k) \log(n/k))$ big nodes $v$ in $T$, the total space is $O(n^2/k \log^2 n \log^2(n/k))$.

As a result, we need $O((n^2/k) \log^2 n \log^2(n/k))$ space for the three additional pieces of information. By a similar analysis, we can show that it takes $O(n^2/k \log^2 n \log^2(n/k))$ time to prepare them.
Query. Let \( Q := [a_x, b_x] \times [a_y, b_y] \), where \( a_x < b_x \) and \( a_y < b_y \). We first identify the \( O(\log^2 n) \) canonical nodes for \( S \cap Q \). These canonical nodes (or canonical subsets) partition \( Q \), as in Figure 4(a), into disjoint rectangular regions, each of which is associated with a specific canonical node.

We merge the regions for the “small” canonical nodes (equivalently, collect the points stored in the small canonical nodes) such that the merged regions are still rectangular and are disjoint from each other as in Figure 4(b). By the definition of small nodes, it is easy to show that the number of resulting merged regions is \( O(\log n) \), and the number of points of \( S \cap Q \) lying in each merged region is \( O(k) \). We again call a small node the union of the small canonical nodes in the region. We denote the small nodes by \( u_1, \ldots, u_l \) and the big nodes by \( v_1, \ldots, v_m \), where \( l = O(\log n) \) and \( m = O(\log^2 n) \). Let \( C \) be the union of all small and big canonical nodes for \( S \cap Q \).

The width of \( S \cap Q \) is the width of the points stored at the canonical nodes of \( C \). For any node pair \( u, v \in C \), width(\( S(u) \cup S(v) \)) is the width of the convex hull of \( S(u) \cup S(v) \). In the dual plane, this is the minimum vertical distance of \( A(u) \cap A(v) \) and \( B(u) \cap B(v) \), i.e., width(\( S(u) \cup S(v) \)) = \( d(A(u) \cap A(v), B(u) \cap B(v)) \). Let \( A := \cap_{u \in C} A(u) \) and \( B := \cap_{u \in C} B(u) \). Then width(\( S \cap Q \)) = \( d(A, B) \).

Since \( S(u) \) and \( S(v) \), for any two distinct \( u, v \in C \), are separated either horizontally or vertically, \( \partial A(u) \) and \( \partial A(v) \) intersect at most once, and \( \partial B(u) \) and \( \partial B(v) \) also intersect at most once. Using this property, we can compute the intersection points by applying the dual version of the method that we already used in Section 3 to compute the convex hull of the convex hulls of canonical subsets in the primal plane. This is done in \( O(\log^2 n) \) time. Recall here that small nodes do not have the data structures for convex hulls, but we can construct them from scratch in \( O(k \log k) \) time for each small node, so it takes \( O(k \log k \log n) \) total time for \( O(\log n) \) small nodes. Thus, in \( O(k \log^2 n + \log^2 n) \) time, we can know which portion of which \( \partial A(v) \) (resp. \( \partial B(v) \)) contributes to \( \partial A \) (resp. \( \partial B \)).

As in Figure 5, draw the vertical lines at these intersection points on each of \( A \) and \( B \), and compute the intersections of the vertical lines with the opposite boundary. Since there are \( O(\log^2 n) \) vertical lines, we can find such intersections in \( O(\log^2 n) \) time by binary searches. As a result, these lines divide the plane into \( O(\log^2 n) \) vertical slabs \( \tau \), and in a slab \( \tau \) only one \( A(v) \) (resp., only one \( B(u) \)) contributes to \( A \cap \tau \) (resp., \( B \cap \tau \)). It is clear that width(\( S \cap Q \)) is the minimum of \( d(A(v), B(u) \cap \tau) \) over all slabs \( \tau \).

Fix \( \tau \) and assume that \( \partial A(v) \) and \( \partial B(u) \) have the chains whose edges in \( \tau \) coincide with \( \partial A \cap \tau \) and \( \partial B \cap \tau \), respectively. We have three cases to find the distance \( d(A(v), B(u) \cap \tau) \). If both \( u \) and \( v \) are big nodes, then we ask the distance to \( Z(A(v), B(u)) \) and \( Z(B(u), A(v)) \) in \( O(\log n) \) time. If one of them is small, say \( u \), then we compute \( d(A(v), e) \) for each edge \( e \in \partial B(u) \cap \tau \) by asking to \( Y(A(v)) \), and again compute \( d(B(v), e') \) for each \( e' \in \partial A(u) \cap \tau \), and also compute \( d(A(v), e') \) for each \( e' \in \partial A(u) \cap \tau \) by asking to \( Y(B(u)) \), which are both done in \( O(k \log^2 n) \) time. If they are both small, then we compute the distance directly by two linear scans; one between \( \partial A(v) \) and \( \partial B(u) \) for a point set \( S \) with \( \log n \) points, and transform the lifted points, using the duality transform, to the dual space. Then the distance of any axis-parallel query rectangle \( Q \), the width of \( S \cap Q \) can be computed in \( O(k \log^2 n + \log^2 n) \) time.

For example, setting \( k = n^2 \), we can answer a query in \( O(n^2 \log^4 n) \) time using \( O(n^{2-\epsilon} \log^5 n) \) space.

5 Smallest enclosing disk queries

A smallest enclosing disk \( \text{sed}(S) \) for a point set \( S \) is determined by two or three points on its boundary. To find it, we first lift the points in \( S \) onto a paraboloid in three dimensions, and transform the lifted points, using the duality transform, to the dual space. Then the radius of \( \text{sed}(S) \) becomes the minimum vertical distance between a convex polyhedron and a paraboloid.

We can compute the distance by adapting the three-dimensional linear programming algorithm developed for a semi-dynamic environment by (Epstein 1992), which guarantees polylogarithmic query time with a data structure of subquadratic size.
5.1 The dual problem

The lifting map and duality transform are well explained in the literature; refer, e.g., to Section 5.7 in the book by O’Rourke (O’Rourke 1998). For completeness, we explain these transformations.

The lifting map. We define the lifting map, which maps a point in $\mathbb{R}^2$ to a point on the paraboloid $P : z = x^2 + y^2$ in $\mathbb{R}^3$:

$$p = (x, y) \mapsto p_1 = (x, y, x^2 + y^2).$$

Let $C$ be a circle in $\mathbb{R}^2$ with center $(a, b)$ and radius $r$. Let $H_C$ be the non-vertical plane defined by

$$H_C : z = 2ax + 2by + r^2 - a^2 - b^2.$$

Let $p = (x, y)$ be a point in $\mathbb{R}^2$. Then $p$ is on, inside, or outside $C$ if and only if $p_1$ is on, below, or above $H_C$, respectively.

For a set $S$ of $n$ points, define

$$S_1 = \{p_1 \mid p \in S\}.$$

Let $C$ be a circle with center $(a, b)$ and radius $r$, and assume that $C$ encloses all points in $S$. Then all points of $S_1$ are on or below the plane $H_C$. Consider a plane $H_C'$ which is parallel to $H_C$ and tangent to the paraboloid $P$. Then

$$H_C' : z = 2ax + 2by - a^2 - b^2.$$

The vertical distance between $H_C$ and $H_C'$ is equal to $r^2$. Thus the following observation holds.

**Observation 1** The radius of the smallest enclosing disk of $S$ is the vertical distance between two parallel planes $H$ and $H'$ such that

1. all points of $S_1$ are on or below $H$,
2. $H$ contains either a face or an edge of the upper convex hull of $S_1$, and
3. $H'$ is tangent to the paraboloid $P$.

The duality transform. We now define the duality transform, which maps any non-vertical plane in $\mathbb{R}^3$ to a point in $\mathbb{R}^3$ as follows:

$$H : z = ax + by + c \mapsto H^* = (a/2, b/2, c).$$

Let $S$ be a set of points in $\mathbb{R}^3$ and define

$$S^* = \{H^* \mid H \text{ is a non-vertical plane on or above ch}(S)\}.$$

Then the following observation holds:

**Observation 2** Let $S$ be a set of $n$ points in $\mathbb{R}^3$. The set $S^*$ is convex and unbounded in the positive $z$-direction.

Using the paraboloid $P : z = x^2 + y^2$, we define the set

$$P^* = \{H^* \mid H \text{ is a non-vertical plane on or below } P\}.$$ 

Let $B^*$ denote the boundary of $P^*$. Then we also have a similar observation as we did for $S^*$.

**Observation 3** The set $P^*$ is convex and unbounded in the negative $z$-direction. Furthermore $P^*$ is the paraboloid $z = -x^2 - y^2$.

Dual problem. We are now ready to define the dual problem of the original problem of computing the radius of $\text{sed}(S)$ for a point set $S$.

We get $S_1$ by the lifting map, and $S^*_1$ by the dual transform. Then $S_1$ is the set of points in the primal space, and $S^*_1$ is the set of points in the dual space. Also we apply the duality transform to map the paraboloid $P$ with equation $z = x^2 + y^2$ to the paraboloid $P^*$ with equation $z = -x^2 - y^2$.

Let us define a set $B^*$ of points in the dual space as follows:

$$B^* = \{H^* \mid H \text{ is a non-vertical plane containing some face of the upper hull of } S_1\}.$$

Let $B^*$ be the set of all the points, in the dual space, “on” or “vertically above” the lower convex hull of $B^*$. Then $B^*$ is a convex polyhedron unbounded in the positive $z$-direction, which is fully contained in $S^*_1$. We also easily prove that $B^*$ and $P^*$ are disjoint.

Consider two parallel planes $H$ and $H'$ such that all points of $S_1$ are on or below $H$, and $H'$ is tangent to the paraboloid $P$. If the distance between $H$ and $H'$ gives the radius of $\text{sed}(S)$, then in the dual space, the point $H^*$ is on the boundary of $S^*_1$ and the point $(H')^*$ is on the paraboloid $P^*$. Furthermore, since $\text{sed}(S)$ contains either two or three points on its boundary, $H$ must contain either an edge or a face of the upper convex hull of $S_1$, which implies that $H^*$ is either an edge or a vertex of the lower convex hull of $B^*$, i.e., an edge or a vertex of $B^*$. Thus we have the following fact.

**Fact 1** Let $S$ be a set of $n$ points in the plane. The radius of the smallest enclosing disk $\text{sed}(S)$ of $S$ is equal to the minimum vertical distance between $B^*$ and $P^*$.

Let us go back to our query problem. Let $v_1, \ldots, v_m$ be the canonical nodes in $T_y$ for $S \cap Q$. Then $S \cap Q = \bigcup_{i=1}^m S(v_i)$. We want to compute $\text{sed}(S \cap Q)$. For each $v_i$, we define $S_1(v_i)$, $S^*_1(v_i)$, and $B^*(v_i)$ for the canonical subset $S(v_i)$ as above. To guarantee that a disk contains all points in $S \cap Q$, its associated plane $H$ in the lifting space must be on or above $S_1(v_i)$, i.e., on or above $\text{ch}(S(v_i))$. But this means that $H^*$ is a point on the boundary of $\bigcap_{i=1}^m S^*_1(v_i)$ in the dual space. More precisely, $H^*$ is a point on $\bigcap_{i=1}^m B^*(v_i)$. As a result, computing the smallest enclosing disk of $S \cap Q$ is equivalent to computing the minimum vertical distance between the paraboloid $P^*$ and the intersection of the convex polyhedra $B^*(v_1), \ldots, B^*(v_m)$. Actually, Eppstein (1992) already explained this dual transformation to maintain the smallest enclosing disk of points in two dimensions in a semi-dynamic setting; see Corollary 1 in (Eppstein 1992).

5.2 Data structure and algorithm

Let $A$ be a convex polyhedron of $n$ vertices. We represent $A$ by a hierarchical representation of $A$, as given by Dobkin & Kirkpatrick (1990). See also Section 7.10 in the book by O’Rourke (1998) for a detailed explanation.

A sequence $\text{hier}(A) = A_1, \ldots, A_k$ of convex polyhedra is said to be a hierarchical representation of $A$ if

1. $A_1 = A$ and $A_k$ is a tetrahedron,
Proceedings of the Nineteenth Computing: The Australasian Theory Symposium (CATS 2013), Adelaide, Australia

2. \( A_{i+1} \subseteq A_i \) for \( 1 \leq i < k \),
3. \( V(A_{i+1}) \cap V(A_i) \) for \( 1 \leq i < k \), where \( V(A_j) \) denotes the set of vertices of \( A_j \), and
4. the vertices of \( V(A_i) \backslash V(A_{i+1}) \) form an independent set in \( A_i \) for \( 1 \leq i < k \).

Dobkin & Kirkpatrick (1990) presented an algorithm to construct \( \text{hier}(A) \) in \( O(n) \) time such that (1) \( k = O(\log n) \), (2) \( \sum_{i} |V(A_i)| = O(n) \), and (3) the maximum degree of the vertices of \( V(A_i) \backslash V(A_{i+1}) \) in \( A_i \) is a constant, say at most 8. They also showed the following crucial lemmas.

**Lemma 1** (Dobkin & Kirkpatrick 1990) Given a hierarchical representation \( \text{hier}(A) = A_1, \ldots, A_k \) and any query plane \( H \) such that \( A_{i+1} \subseteq H^+ \) for some \( i \), then either \( A_i \subseteq H^+ \) or there exists a unique vertex \( v \in V(A_i) \) such that \( v \in H^- \), where \( H^+ \) and \( H^- \) denote the half-space above and below \( H \), respectively. Furthermore, such \( v \) can be found in constant time.

**Lemma 2** (Dobkin & Kirkpatrick 1990) Given a hierarchical representation \( \text{hier}(A) \) and any query plane \( H \) that does not intersect \( A \), the minimum vertical distance between \( A \) and \( H \) can be computed in \( O(\log |A|) \) time.

Let \( d(A, B) \) be the minimum vertical distance between disjoint convex sets \( A \) and \( B \). The algorithm given in Lemma 2 computes \( d(A, B) \) when \( B \) is a plane. We can simply adapt the algorithm in Lemma 2 to compute \( d(\tilde{A}, B) \) for the case when \( B \) is a paraboloid:

**Lemma 3** Let \( A \) be a convex polyhedron which is unbounded in the positive z-direction, let \( P^* \) be the paraboloid \( z = -x^2 - y^2 \), and assume that \( A \) is above \( P^* \). If \( \text{hier}(A) = \{A_1, \ldots, A_k\} \) is given, we can compute \( d(A_i, P^*) \) in \( O(\log |A|) \) time.

**Proof.** We first compute \( d(A_k, P^*) \) in constant time, which is possible because \( A_k \) is a tetrahedron. We now update a pair \((a_i, p_i)\) of points \( a_i \in \partial A_i \) and \( p_i \in P^* \) realizing \( d(A_i, P^*) \), as \( i \) is decremented from \( k \) to 1. Since \( d(A_k, P^*) \) is equal to \( (a_k, p_k) \), when we translate \( P^* \) in the positive z-direction by \( d(A_k, P^*) \), it first hits \( A_k \) at \( a_k \). Let \( H_{P^*} \) be the plane tangent to \( P^* \) at the point \( p_k \), and let \( H_A \) be the plane through \( a_k \) parallel to \( H_{P^*} \). Then it follows that \( A_i \) is above \( H_A \) and \( P^* \) is below \( H_{P^*} \), i.e., their interiors are separated by both of \( H_{P^*} \) and \( H_A \). We now compute \( d(A_{i-1}, P^*) \) by identifying \((a_{i-1}, p_{i-1})\). Since \( A_{i-1} = (A_{i-1} \cap H_A^+) \cup (A_{i-1} \cap H_A^-) \)
\[ d(A_{i-1}, P^*) = \min\{d(A_{i-1} \cap H_A^+, P^*), d(A_{i-1} \cap H_A^-, P^*)\}. \]

Clearly, \( d(A_{i-1} \cap H_A^+, P^*) \) is attained by \((a_{i-1}, p_{i-1})\). Thus it suffices to compute \( d(A_{i-1} \cap H_A^-, P^*) \). By Lemma 1, there can only be one vertex \( v \in A_{i-1} \cap H_A^- \) and it can be identified in constant time. Thus if such \( v \) exists, then \( A_{i-1} \cap H_A^- \) is of constant complexity because \( v \) has constant degree in \( A_{i-1} \) by definition of \( \text{hier}(A) \). This allows us to compute \( d(A_{i-1} \cap H_A^-, P^*) \) in \( O(1) \) time. Therefore, we can update \((a_{i-1}, p_{i-1})\) in \( O(1) \) time. Since \( k = O(\log |A|) \), the total time is \( O(\log |A|) \).

**Additional information.** At each node \( v \) in any secondary range tree \( T_p \), we maintain only one additional structure as follows:

1. \( \text{hier}(B^*(v)) \), a hierarchical representation for the convex polyhedron \( B^*(v) \).

Since \( \text{hier}(A) \) can be constructed in \( O(|A|) \) time (Dobkin & Kirkpatrick 1990), this information can be computed in \( O(n \log^2 n) \) time and space.

**Query.** Let \( v_1, \ldots, v_m \) be the canonical nodes for \( S \cap Q \). Recall that \( m = O(\log^2 n) \). As mentioned earlier, we need to compute
\[ d(\bigcap_{i=1}^m B^*(v_i), P^*). \]

Let us consider the elementary case that \( m = 1 \), i.e., computing \( d(B^*(v), P^*) \). This can be done in \( O(\log^2 n) \) time by Lemma 3. Once we can solve this elementary case in \( O(\log^2 n) \) time, we can employ the algorithm by Eppstein (1992) as follows:

**Lemma 4** (Eppstein 1992) Given \( m \) convex polyhedra represented by their hierarchical representations, we can optimize any given objective function over their common intersection in \( O(\gamma \cdot m^2 \log^2 n) \) time, provided that the elementary problem of optimizing the function over one convex polyhedron can be done in \( O(\gamma) \) time.

In our case, since \( \gamma = O(\log n) \) and \( m = O(\log^2 n) \), we can compute \( d(\bigcap_{i=1}^m B^*(v_i), P^*) \) in \( O(\log^5 n) \) time. Thus, the radius of \( \text{sed}(S \cap Q) \) can be computed within the same time bound.

**Theorem 3** Let \( S \) be a set of \( n \) points in the plane. In \( O(n \log^2 n) \) time, we can construct a data structure of size \( O(n \log^2 n) \), such that for any axis-parallel query rectangle \( Q \), the radius of the smallest enclosing disk of \( S \cap Q \) can be computed in \( O(\log^5 n) \) time.

6 Concluding remarks

An immediate open question is to construct more efficient data structures for the three extent measures. It might be quite possible to reduce a few logarithmic factors from the current time bounds.

Another interesting question is whether width queries can be answered in \( O(\text{polylog}(n)) \) time using a data structure of size \( O(n \cdot \text{polylog}(n)) \).

It would be interesting to consider other extent measures such as the largest empty disk within the convex hull of \( S \cap Q \), and the minimum annulus containing \( S \cap Q \). Finally, it would be interesting to consider different query ranges, such as circles or half-planes, or to extend to higher dimensions.

**References**


Competitive Online Algorithms for Multiple-Machine
Power Management and Weighted Flow Time

Ho-Leung Chan  Sze-Hang Chan  Tak-Wah Lam *  Lap-Kei Lee
Rongbin Li  Chi-Man Liu

Department of Computer Science,
University of Hong Kong
Email: {hchan, shchan, twlam, lklee, rbl, cmliu}@cs.hku.hk

Abstract

We consider online job scheduling together with power management on multiple machines. In this model, jobs with arbitrary sizes and weights arrive online, and each machine consumes different amounts of energy when it is processing a job, idling or sleeping. A scheduler has to maintain a good balance of the states of the machines to avoid energy wastage, while giving an efficient schedule of the jobs. We consider a recently well-studied objective of minimizing the total weighted flow time of the jobs plus the total energy usage. For the special case where all jobs have the same weight, competitive algorithms have been obtained (Lam et al. 2009, Chan et al. 2011). This paper gives a non-trivial potential analysis of a weighted generalization of the power management algorithm in (Chan et al. 2011), coupled with a classical scheduling algorithm HDF. This leads to the first competitive result for minimizing weighted flow time plus energy. The result can be extended to the dynamic speed scaling model where the scheduler can vary the speed of individual machines to process the jobs and the energy usage depends on the speed of the machines.

Keywords: sleep management, weighted flow time, energy efficiency, potential analysis.

1 Introduction

Server farms with hundreds to thousands of machines are common nowadays, and energy consumption has become an important concern. It has been reported that the energy cost of running a machine for three years indeed exceeds the hardware cost of the machine (Belady 2007). When a machine is turned on, its energy consumption consists of the static power, which is consumed even when the machine is idle, and the dynamic power, which is the extra power used for processing a job. For example, an Intel Xeon E5520 server consumes 150W of energy when idling and 240W when working. To reduce energy usage, some machines can be put to the sleep state in which energy consumption is reduced to essentially zero. However, transiting a machine from the sleep state to the awake state requires an extra amount of wake-up energy, and it is energy inefficient to frequently switch the machines on and off. In this paper, we are interested in online algorithms for job scheduling that can optimize the energy consumption of a pool of machines as well as certain QoS (Quality of Service) measure of the schedule.

Weighted flow time. A commonly used QoS measure is the total weighted flow time (or simply weighted flow) of the jobs. Formally speaking, jobs arrive online at different times in an unpredictable manner. Each job $j$ has a size $p(j)$ and weight $w(j)$. The weight $w(j)$ reflects the importance of job $j$. The flow time of a job is the length of the duration from its arrival until its completion, and its weighted flow time is simply its flow time multiplied by its weight. We assume that all jobs are sequential, i.e., a job can be processed by at most one machine at any time. We consider preemptive scheduling and a preempted job can be resumed at the point of preemption. Migration incurs overhead in practice, so we only consider non-migratory schedules in which each job is dispatched to one machine and is run entirely on that machine.

Minimizing weighted flow is fundamentally harder than minimizing unweighted flow even when energy usage is not a concern: On a single machine, it is well-known that the algorithm SRPT (Shortest Remaining Processing Time) is 1-competitive for unweighted flow time, while Bansal and Chan (Bansal et al. 2009) showed that no online algorithm can be $O(1)$-competitive for weighted flow time. To overcome the hardness, most studies on weighted flow time (without energy concern) consider resource augmentation where the online algorithm is given faster machines. Precisely, an $s$-speed machine can process $s$ units of work in one unit of time. On a single machine, Becchetti et al. (Becchetti et al. 2006) showed that the greedy algorithm HDF (Highest Density First) is $(1 + \epsilon)$-speed $(1 + \frac{1}{2})$-competitive for any $\epsilon > 0$, i.e., its total weighted flow time when given a $(1 + \epsilon)$-speed machine is at most $(1 + \frac{1}{2})$ times that of the optimal offline schedule on a 1-speed machine. Note that the density of a job is its weight divided by its size. HDF always schedules the jobs with the highest density and it requires migration when there are more than one machine. On multiple machines, Becchetti et al. (Becchetti et al. 2006) also showed that HDF is $(2 + \epsilon)$-speed $(1 + \frac{1}{2})$-competitive when migration is allowed. The first non-migratory result in the multiple machine setting was given by Chadha et al. (Chadha et al. 2009) who gave a $(1 + \epsilon)$-speed $O(\frac{1}{\epsilon})$-competitive algorithm. The competitive ratio was improved to $O(\frac{1}{\epsilon^2})$ recently by Anand et al. (Anand et al. 2012).

Energy-efficient scheduling. All the above results assume energy usage is not a concern and the ma-
Chen et al. (2009) studied the single machine setting and the total energy usage. Following the literature, we assume the reverse takes zero energy. We are given \( m \geq 1 \) machines. At any time, a machine can be in the sleep or awake state. A machine can process a job only when it is awake. Let \( \nu \geq 0 \) and \( \sigma \geq 0 \) be the static and dynamic power of a machine, respectively. Then, an awake machine consumes energy at a rate of \( \sigma \) if it is idle and at a rate of \( \sigma + \nu \) when it is processing a job. For convenience, let \( \mu = \sigma + \nu \). A sleeping machine does not consume energy, but \( \lambda > 0 \) units of wake-up energy is needed to switch a machine from the sleep state to the awake state (we assume the reverse takes zero energy). Following the literature, we assume that state transition is immediate. Our objective is to minimize the total weighted flow time plus the total energy usage.

When all jobs have unit weight, Lam et al. (Lam et al. 2009) studied the single machine setting and they gave an \( O(1) \)-competitive algorithm for minimizing total unweighted flow plus energy usage. Their algorithm may put the machine to sleep even when there are unfinished jobs. Sleep management in the multiple machine setting is more complicated. It is natural to balance the workload evenly on all machines for minimizing total flow time, yet it may be more energy-efficient to overload some machine so as to let others sleep. Chan et al. (Chan et al. 2011) showed a multi-machine algorithm \( \text{POOL} \) that is \( (1+\epsilon) \)-speed \( O(1) \)-competitive for minimizing total unweighted flow plus energy usage. They also showed that without resource augmentation, any algorithm is \( \Omega(m) \)-competitive, where \( m \) is the number of machines.

Our contributions. It has been conjectured that \( \text{POOL} \) can be generalized to the weighted setting with static power. When using \( \text{HDF} \) to schedule jobs on each machine, the potential analysis of flow time given in (Chan et al. 2011) does not work in the weighted setting. In this paper, we resolve this conjecture in the affirmative by giving a weighted version of \( \text{POOL} \) which together with \( \text{HDF} \) gives a new algorithm (to be called \( \text{WPOOL} \)) that is \( (1+\epsilon) \)-speed \( O(1) \)-competitive for minimizing total weighted flow time plus energy usage. The main difficulty lies on the analysis, in which a new potential function is given to relate the weighted flow time plus energy usage of the online algorithm and the optimal schedule. This involves comparing the fractional flow time (which is a relaxed notion of weighted flow time).

Remarks on speed scaling. The above result can be extended to the dynamic speed scaling model where an awake machine can run at any speed between \([0, \infty)\) and each machine can scale its speed individually. The static power of an awake machine remains \( \sigma \), yet the dynamic power \( \nu(s) \) becomes a function of the current speed \( s \). Following the literature (see (Albers 2010) for a \( \nu(s) \)), we assume \( \nu(s) = s^\alpha \) where \( \alpha > 1 \) is a small constant (typically 3 for CMOS-based devices (Brooks et al. 2000)). When all jobs have the same weight, Chan et al. (Chan et al. 2011) showed that the algorithm \( \text{POOL} \) can be extended to the speed scaling model. \( \text{POOL} \) exploits the speed scaling algorithm \( \text{AJC} \) (Lam et al. 2008), which sets the processor speed based on the number of the unfinished jobs, and is \( O(1) \)-competitive for unweighted flow plus energy. In this paper, we can extend \( \text{WPOOL} \) to the speed scaling model. We generalize the speed scaling algorithm \( \text{BPS} \) (Bansal et al. 2009) with the static power \( \sigma \) taken into consideration (precisely, the processor speed depends on the remaining fraction of the weight of unfinished jobs and the value of \( \sigma \)). This results in an \( O(\frac{\sigma}{\ln p}) \)-competitive algorithm for minimizing total weighted flow time plus energy usage. Due to the space limitation, we leave the details of the speed scaling result to the full paper.

Notations. Given a schedule \( S \), let \( F(S) \) denote the total weighted flow time of all jobs. In the fixed-speed setting, we assume that the online scheduler is using \((1+\epsilon)\)-speed machines, each can process \((1+\epsilon)\) units of work in one unit of time while consuming energy at the same rate as a 1-speed machine used by the optimal offline algorithm (i.e., \( \mu \)). The energy usage of a machine is the total energy usage when it is awake plus the total wake-up energy used for state transition. Let \( E(S) \) denote the total energy usage of all machines. The objective is to minimize \( G(S) = F(S) + E(S) \). A job is said to be active at time \( t \) if it has been released but not yet completed by \( t \).

2 The Algorithm \( \text{WPOOL} \)

This section presents the algorithm \( \text{WPOOL} \) for scheduling jobs with arbitrary weights. We first show some basic facts related to the notions of fractional weight and remaining working cost. Below \( \text{Off} \) denotes the optimal offline algorithm that minimizes the total weighted flow plus energy for any input.

2.1 Fractional weighted flow and remaining working cost

At any time \( t \), for any job \( j \), let \( p(j, t) \) be the unfinished size of \( j \) at time \( t \). The fractional weight of \( j \) at time \( t \), denoted \( \tilde{w}(j, t) \), is its weight multiplied by the fraction of its unfinished size. I.e., \( \tilde{w}(j, t) = \frac{w(j)}{p(j, t)} \cdot w(j) \). The fractional weighted flow of \( j \) is the integral of its fractional weight over time from its arrival until its completion. For any schedule \( S \), denote \( \tilde{F}(S) \) as the total fractional weighted flow of all jobs. Obviously, \( \tilde{F}(S) \leq F(S) \). We first notice that it is sufficient to design an algorithm that is \( 1+\epsilon \)-competitive in terms of fractional weighted flow. A similar result was proved in (Becchetti et al. 2006) for the setting without energy concern.

Lemma 1. Let \( A \) be any algorithm with sleep management using \( s \)-speed machines. For any \( \delta > 0 \), we can transform \( A \) into another algorithm \( A' \) using \((1+\delta)s\)-speed machines such that \( F(A') \leq (1+\frac{\epsilon}{2})F(A) \) and \( E(A') \leq E(A) \).

Proof. \( A' \) schedules each machine and manages its state identically as \( A \), except that if a job \( j \) is already completed by \( A' \) but not \( A \), then \( A' \) will remain idle when \( A \) is processing \( j \). Obviously, \( E(A') \leq E(A) \). At any time \( t \), if a job \( j \) is not completed by \( A' \), the remaining size of \( j \) in \( A \) is at least \( p(j) = \frac{w(j)}{1+\delta}s \). Hence the fractional weight of \( j \) in \( A \) is at least \( \frac{\tilde{w}(j)}{1+\delta} \). It implies that the total weighted flow of active jobs in \( A' \) is at most \((1+\frac{\epsilon}{2})\) times the total weighted flow of active jobs in \( A \), and hence \( F(A') \leq (1+\frac{\epsilon}{2})F(A) \).

Hence, we first design an algorithm \( A \) with bounded total fractional weighted flow plus energy usage. Then by increasing the speed of \( A \) further by
a factor of $(1 + \delta)$, we obtain an algorithm that is competitive with respect to the total weighted flow plus energy usage.

The density of a job $j$ is the ratio of its weight to size, i.e., $w(j)/p(j)$. It is known that the algorithm HDF (Highest Density First) minimizes the total fractional weighted flow on a single machine when there is no energy concern (Becchetti et al. 2006). In fact, our algorithm WPOOL schedules jobs by HDF for each machine whenever it is awake. WPOOL dispatches a job to a machine once the job arrives. To decide which machine a job is dispatched to, WPOOL uses the following definition.

**Definition 2.** At any time $t$, the remaining working cost (rwc) for a machine $i$, denoted $rwc_i(t)$, is the total fractional weighted flow plus energy to be incurred after time $t$ by machine $i$ while it is processing a job, assuming no more jobs arrive after time $t$ and assuming it processes jobs by HDF whenever it is working.

We can compute $rwc_i(t)$ as follows. Let $T$ be the intervals of time after $t$ during which machine $i$ is processing a job. Let $\tilde{w}_i(x)$ be the total fractional weight of all jobs dispatched to machine $i$ that are not yet completed by time $x$. Then $rwc_i(t) = \int_0^t \tilde{w}_i(x) dx + |T|\mu$, where $|T|$ is the total length of the time intervals in $T$. Note that $rwc_i(t)$ depends only on the time when machine $i$ is working, not the time when it is idle or sleeping. Furthermore, it assumes that the machine processes jobs by HDF whenever it is working.

The following lemma gives a more useful formula for calculating $rwc_i(t)$. The inverse density of a job $j$, denoted $q(j)$, is the inverse of its density, i.e., $q(j) = p(j)/w(j)$. At any time $t$, for any real $q \geq 0$, let $r_i(q,t)$ be the number of active jobs in machine $i$ with inverse density at least $q$, and let $\bar{w}_i(q,t)$ be the total fractional weight of those jobs. Let $\bar{w}_i(t) = \bar{w}_i(0,t)$, i.e., the total fractional weight of all active jobs in machine $i$ at time $t$.

**Lemma 3.** Suppose that machine $i$ uses HDF to schedule the jobs whenever awake. Then, at any time $t$, $\text{rwc}_i(t) = \int_0^t \int_{\bar{w}_i(q,t)}^{\frac{\mu}{\mu}} \tilde{w}_i(q,t) dq dx$, where $s$ is the speed of machine $i$. Moreover, if a job $j$ of inverse density $x$ arrives at time $t$ and is dispatched to machine $i$ immediately, the increase in rwc due to $j$ equals $\int_0^{q(j,t)} \int_{\bar{w}_i(q',t)}^{\frac{\mu}{\mu}} \tilde{w}_i(q',t) dq dx$ (note that $\bar{w}_i(q,t)$ refers to the total fractional weight just before $j$ arrives).

**Proof.** Let $j_1, j_2, \ldots, j_n$ be the active jobs in machine $i$ at time $t$, ordered in non-decreasing order of density, i.e., $w(j_1)/p(j_1) \leq w(j_2)/p(j_2) \leq \cdots \leq w(j_n)/p(j_n)$. If no job arrives after time $t$, then HDF executes the active jobs in the order of $j_n, j_{n-1}, \ldots, j_1$. Since $rwc_i(t)$ only includes the cost incurred when machine $i$ is working, we assume w.l.o.g. that machine $i$ does not sleep after time $t$.

Consider the maximal period $[t_k, t'_k]$ of time that HDF is processing the job $j_k$. Let $\bar{G}(t_k, t'_k)$ be the fractional weighted flow plus energy incurred during $[t_k, y]$ for some time $y \in [t_k, t'_k]$. Then, $\frac{d\bar{G}(t_k, y)}{dy} = \bar{w}(y) + \mu$ and $\bar{w}(y)$ is decreasing (w.r.t. $y$) at a rate of $s\frac{\bar{w}(y)}{\bar{w}(y)^2} = \frac{s}{\bar{w}(y)}$ (i.e., $\frac{d\bar{w}(y)}{dy} = -\frac{s}{\bar{w}(y)}$). Thus, $\frac{d\bar{G}(t_k, y)}{dy} = -\bar{w}(y) + \mu \frac{d\bar{w}(y)}{dy}$. At time $t_k$, $\bar{w}_i(t_k) = \bar{w}(j_1, t) + \cdots + \bar{w}(j_k, t)$; also, at time $t'_k$, $\bar{w}_i(t'_k) = \bar{w}(j_1, t) + \cdots + \bar{w}(j_{k-1}, t)$. Thus, $\bar{G}(t_k, t'_k) = \int_{\bar{w}(j_1, t)}^{\bar{w}(j_{k-1}, t)} \bar{w}(j, t) + (z + \mu) \frac{d\bar{w}(j, t)}{dy} dz$.

Notice that $\text{rwc}_i(t) = \sum_{k=1}^n \bar{G}(t_k, t'_k) = \sum_{k=1}^n (\bar{w}(j_k, t) + (q(j_k) - q(j_{k-1})) \cdot \mu + \int_{\bar{w}(j_{k-1}, t)}^{\bar{w}(j_k, t)} \bar{d}(j, t) dz) + \sum_{k=1}^{n-1} (q(j_k) - q(j_{k-1})) \cdot \int_{\bar{w}(j_{k-1}, t)}^{\bar{w}(j_k, t)} \bar{d}(j, t) dz$.

If a job $j$ with $q(j) = x$ arrives at time $t$ and is dispatched to machine $i$, then $\tilde{w}_i(q, t)$ increases by $w(j)$ for $q \in [0, x]$. Thus, the increase in rwc due to $j$ equals $\int_0^x \int_{\bar{w}_i(q', t)}^{\frac{\mu}{\mu}} \tilde{w}_i(q', t) dq dx$.

**2.2 Algorithm definition**

WPOOL attempts to maintain a small pool $P$ of dispatchable machines; $P$ contains one sleeping machine initially and is always non-empty. At any time, machines in $P$ are either all asleep or all awake, and $P$ is said to be asleep or awake, respectively. Machines not in $P$ are always asleep and do not have active jobs. WPOOL would gradually include more machines into $P$ as jobs arrive, and they are immediately put into the same state as $P$.

WPOOL maintains three real-value counters $X_{AF}$, $X_{AI}$, and $X_{WC}$ to keep track of the recent increase to the accumulated flow (when $P$ is asleep), idle energy, and working cost, respectively. Initially, all counters equal 0. As detailed below, they each keep increasing until they trigger certain events, then they will be reset.

- When $P$ is asleep, $X_{AF}$ increases continuously at the rate of the total fractional weight of active jobs. Once $X_{AF}$ reaches $\lambda$, we wake up all machines of $P$ and reset $X_{AF}$ to zero.

- When $P$ is awake, $X_{AI}$ increases continuously at the rate of $\sigma$ times the number of idle machines. Once $X_{AI}$ reaches $\lambda$, if $P$ has two or more idle machines, we remove one idle machine from $P$, put it to sleep and reset $X_{AI}$ to zero. See the algorithm below for the details of some boundary cases.

- $X_{WC}$ increases only when a job arrives. Intuitively, whenever $X_{WC}$ reaches a value at least $\lambda$, we try to include one machine into $P$ and decrease $X_{WC}$ by $\lambda$. Specifically, when a job $j$ arrives, WPOOL first assumes that $j$ is dispatched to a machine with no active jobs and calculates the increase in rwc. Denote this amount of increase as $\text{null}_{IWC}(j)$. If $X_{WC} + \text{null}_{IWC}(j) \geq \lambda$ and $|P| < m$, WPOOL will add one machine into $P$ and dispatch $j$ to this machine (even if $P$ already has idle or sleeping machines), and $X_{WC}$ is set to $X_{WC} + \text{null}_{IWC}(j) - \lambda$. Otherwise, WPOOL dispatches $j$ to a machine in $P$ that minimizes the increase in rwc; below we denote
machine $i$ as $\ell(j, P)$, and the amount of increase in $rwc$ as $\min \text{Inc}_r \cdot rwc(j, P)$. $X_{wc}$ increases by $\min \text{Inc}_r \cdot rwc(j, P)$ (which is at least $\min \text{Inc}_r \cdot rwc(j)$). At the end, if $X_{wc}$ is at least $\lambda$, we repeatedly try to add a machine into $P$ and decrease $X_{wc}$ by $\lambda$.

Below is a complete description of the algorithm $WPOOL$.

**Increase $X_{wc}$ and wake up $P$**: At any time, if $P$ is asleep,

- $X_{wc}$ increases at the rate of the total fractional weight of active jobs;
- If $(X_{wc} = \lambda)$, then wake up all machines in $P$; reset $X_{wc} = 0$.

**Increase $X_{wc}$, remove & sleep an idle machine of $P$**: At any time, if $P$ is awake,

- If $(X_{wc} < \lambda)$, then $X_{wc}$ increases at the rate of $\sigma$ times the number of idle machines.
- If $(X_{wc} = \lambda)$, then
  - If $P$ has already $\geq 2$ idle machines, remove and sleep one idle machine from $P$; reset $X_{wc} = 0$;
  - If $P$ has only one idle machine and $|P| = 1$, put $P$ to sleep; reset $X_{wc} = X_{wc} = 0$;
  - Otherwise, $X_{wc}$ remains equal to $\lambda$.

**Dispatch a new job, increase $X_{wc}$ and expand $P$**: When a job $j$ arrives,

- If $((|P| < m) \land (X_{wc} + \text{null}_r \cdot rwc(j) \geq \lambda))$,
  - then add a machine to $P$; dispatch $j$ to this machine;
  - else dispatch $j$ to $\ell(j, P)$;
  - $X_{wc} = X_{wc} + \text{null}_r \cdot rwc(j) - \lambda$;
- While $((X_{wc} \geq \lambda) \land (|P| < m))$ do
  - { add a machine to $P$; $X_{wc} = X_{wc} - \lambda$ }
- If $X_{wc} > \lambda$, then $X_{wc} = \lambda$.

**Job scheduling in each machine of $P$**: When awake, use HDF policy.

**Property 4.** Every time after $WPOOL$ executes the job dispatching procedure, it maintains the invariant that if $|P| < m$, there exists a machine with $rwc < \lambda$.

**Proof.** Suppose, for the sake of contradiction, that there exists a time $t$, such that immediately after some job is dispatched at time $t$, $|P| < m$, and all machines in $|P|$ have $rwc \geq \lambda$. Note that $\lambda > 0$. Let $t_0 \leq t$ be the latest time not later than $t$, such that at $t_0$ all machines in $P$ have $rwc > 0$, yet immediately before $t_0$ at least one machine in $P$ has $rwc = 0$. Hence, from time $t_0$ to time $t$, all machines in $P$ have $rwc > 0$ and no machine is removed from $P$ (since $WPOOL$ removes a machine from $P$ only when it is idle, which implies $rwc = 0$). Now, consider all zero-$rwc$ machines in $P$ immediately before $t_0$. By the definition of $WPOOL$, at most one such machine can be removed from $P$ at time $t_0$, and it cannot be removed if it is the only idle machine in $P$. Hence, there exists a zero-$rwc$ machine $i_0$ in $P$ immediately before $t_0$, such that $i_0$ is still in $P$ immediately after $t_0$ but with $rwc > 0$ (due to some jobs dispatched to it). Assume that there are $x$ new machines added to $P$ from $t_0$ to $t$ (including the ones added at time $t$). As mentioned above, no machine will be removed from $P$ from $t_0$ to $t$. Hence, $i_0$ and those $x$ machines (totally $x + 1$ machines) will remain in $P$ at least until time $t$, and each of them has $rwc \geq \lambda$ at time $t$. So, each of them has an increase of $rwc$ at least $\lambda$ since $t_0$.

By the definition of $WPOOL$, at least $x + 1$ machines are added to $P$ from $t_0$ to $t$, which contradicts the assumption that only $x$ machines are added.

Our main result is as follows, which will be proved in the next section.

**Theorem 5.** When $WPOOL$ is given $(1 + \epsilon)$-speed machines, the total fractional weighted flow plus energy usage of $WPOOL$ is at most $O(1 + \frac{1}{\epsilon})$ times the total weighted flow plus energy usage of $Off$.

Then, together with Lemma 1, we have the following corollary.

**Corollary 6.** There exists an algorithm $WPOOL'$ that is $(1 + \epsilon)$-speed $O(\frac{1}{\epsilon})$-competitive for total weighted flow plus energy usage.

**Proof.** By Lemma 1 and Theorem 5, we can obtain an algorithm $WPOOL'$ that is $(1 + \epsilon')(1 + \delta)$-speed $O((1 + \frac{1}{\epsilon})(1 + \frac{1}{\delta}))$-competitive. By putting $\epsilon' = \delta = \frac{\epsilon}{2}$, the result follows.

## 3 Analysis of $WPOOL$

This section analyzes $WPOOL$ and hence proves Theorem 5. Consider a certain input sequence of jobs. Recall that $Off$ is the optimal offline schedule. W.L.O.G., we can assume that $Off$ dispatches a job to a machine once the job arrives. A machine $i$ in $Off$ is said to be procrastinating at time $t$ if some job dispatched to machine $i$ is unfinished by time $t$ yet machine $i$ is asleep at time $t$. Instead of analyzing $WPOOL$ with respect to $Off$ directly, we analyze $WPOOL$ with another offline schedule $Opt$ with the following property.

**Lemma 7.** We can transform $Off$ into another schedule $Opt$ such that $Opt$ has at most one procrastinating machine at any time and $Opt$ schedules a machine by HDF whenever the machine is working. Furthermore, $F'(Opt) \leq F(Off)$ and $E(Off) \leq 3E(Off)$.

**Proof.** Chan et al. (Chan et al. 2011) showed that we can transform $Off$ into another schedule $Opt$ such that $Opt$ has at most one procrastinating machine at any time, by changing some machines from the sleep state to awake state when needed. The energy usage of $Opt$ is at most 3 times that of $Off$, i.e. $E(Off) \leq 3E(Off)$. Furthermore, the transformation guarantees that the completion time of every job $j$ in $Opt$ is no later than its completion time in $Off$, so the total weighted flow of $Opt$ is at most that of $Off$, i.e. $F'(Opt) \leq F(Off)$). Then, we further transform $Opt$ into the targeted schedule $Opt$ as follows: for each job, $Opt$ dispatches it to the same machine as $Opt$ does. For each machine $i$, at any time $t$, the state (sleep, idle or working) and speed of $i$ in $Opt$ are the same as the state and speed of $i$ in $Off$. However, for each machine $i$ in $Opt$, $i$ schedules jobs dispatched to it by HDF whenever working. Obviously, $Opt$ is a valid schedule, and $E(Off) = E(Off')$. It is well known that, on single machine, HDF minimizes the fractional weighted flow when speed function is fixed, hence, $F'(Opt) \leq F(Off')$.

In the following, we show that the total fractional weighted flow plus energy usage of $WPOOL$ is at most $O(1 + \frac{1}{\epsilon})$ times that of $Opt$. Let $\hat{F}$ and $\hat{E}$ be the total fractional weighted flow and energy usage of $WPOOL$, respectively. We divide $\hat{F}$ into two parts: the working flow $\hat{F}_w$ and the sleeping flow $\hat{F}_s$, which refer to
the total fractional weighted flow incurred by the machines when they are working and sleeping, respectively. Note that \( F = E_w + E_s \). We also divide \( E \) into three parts: \( E_s \) is the idling energy (static energy usage during the idle state), \( E_w \) the working energy, and \( E_{aw} \) the wake-up energy. Then \( E = E_s + E_w + E_{aw} \). Let \( \bar{G} = \bar{F} + E \) be the cost of WPOOL. We use the same notations with an extra asterisk to denote the corresponding quantity in Opt. For example, \( \bar{F}^* \) is the total fractional weighted flow of Opt, and \( \bar{G}^* \) is the cost of Opt.

### 3.1 Sleeping flow and energy usage

It is relatively easy to upper bound the sleeping flow \( \bar{F}_s \) and the energy usage in terms of \( \bar{G}^* \) and \( \bar{F}_w \). They are summarized by the following lemma. The proof is similar to that in (Chen et al. 2011).

**Lemma 8.** (i) \( \bar{F}_s \leq \bar{G}^* \); (ii) \( E_w \leq E_w^* \); (iii) \( E_v \leq \bar{F}_w + E_w + \bar{G}^* \); (iv) \( E \leq E_v + E_w \).

**Proof.** We divide the timeline into intervals called \( P \)-intervals, each of which consists of a maximal asleep period of \( P \), followed by a maximal awake period of \( P \). Within a \( P \)-interval, we use the corresponding notation with a suffix (I) to denote the flow or energy within this interval. Note that within each \( P \)-interval, the sleeping fractional weighted flow accumulated during asleep period is \( \lambda \), and the idling energy accumulated during awake period is at least \( \lambda \).

To make the discussion easy, we charge the wake-up energy of Opt to the time when it sleeps a machine. We will first show that within each \( P \)-interval, the cost of Opt is at least \( \lambda \), i.e. \( \bar{G}^*(I) \geq \lambda \). The argument is as follows. If Opt sleeps a machine during \( I \), then it is obvious that \( \bar{G}^*(I) \geq \lambda \). If Opt never sleeps a machine during \( I \), there are two cases: Case (1), if Opt has all machines asleep at the beginning of \( I \) and it does not wake up a machine during the asleep period of \( I \), then Opt would let the jobs arrive during the asleep period of \( I \) to accumulate a fractional weighted flow at least \( \lambda \), thus \( \bar{G}^*(I) \geq \lambda \). Case (2), if Opt has at least one machine awake at the beginning of \( I \) or it wakes up a machine during the asleep period of \( I \), then the static energy incurred by this machine during \( I \) is at least the idling energy incurred by WPOOL during \( I \) when \( P \) has only one idle machine, which is at least \( \lambda \). We are now ready to prove items (i) to (iv).

(i) At the beginning of a \( P \)-interval \( I \), all machines in \( P \) are asleep and there are no active jobs. POOL wakes up machines in \( P \) as soon as the accumulated sleeping fractional flow increases to \( \lambda \), and later, no sleeping fractional flow is accumulated till the end of \( I \). Hence, \( \bar{F}_s(I) = \lambda \leq \bar{G}^*(I) \). Summing over all \( P \)-intervals, we obtain \( \bar{F}_s \leq \bar{G}^* \).

(ii) Note that WPOOL and Opt process the same amount of work and WPOOL is using machines with speed \( s \geq 1 \). Hence, the total amount of time that WPOOL is working is at most that of Opt. It implies that the working energy of WPOOL is at most that of Opt.

(iii) Note that at the beginning of a \( P \)-interval \( I \), there is only one machine in \( P \) and this machine is asleep. Suppose that there are totally \( x \) machine additions within \( I \). Hence, the wake-up energy of POOL within \( I \) is \( E_w(I) = (x + 1)\lambda \). Note that a machine is added into \( P \) only when the accumulated increase of \( rwc \) (i.e. the counter \( X_{wc} \)) is at least \( \lambda \). When \( I \) ends, all jobs are completed and all the accumulated increase in \( rwc \) have become part of \( E_w(I) + E_{aw}(I) \). Hence, \( x \lambda \leq E_w(I) + E_{aw}(I) \). It follows that \( E_w(I) = (x + 1)\lambda = x\lambda + \lambda \leq E_w(I) + E_{aw}(I) + \bar{G}^*(I) \). Summing over all \( P \)-interval, we obtain \( E_v \leq \bar{F}_w + E_w + \bar{G}^* \).

(iv) \( E \) can be divided into two types. The first type is incurred when the counter \( X_{ie} < \lambda \) and the other type is incurred when the counter \( X_{ie} = \lambda \). For the first type \( E_1 \), it increases at the same rate as \( X_{ie} \). At any time \( t \) that \( X_{ie} \) reaches \( \lambda \), WPOOL would sleep a machine at time \( t \) or later. Hence, the total of the first type \( E_1 \) equals \( \lambda \) times the total number of times WPOOL sleeps a machine, which is exactly \( E_v \).

For the second type \( E_2 \), it accumulates only when \( P \) has only one idle machine and \(|P| > 1 \). So, there exists another working machine in \( P \) when this type of \( E_2 \) is accumulated. Note that a working machine consumes energy at a rate more than an idle machine. Hence, this type of \( E_2 \) is at most \( E_w \).

### 3.2 Potential analysis of \( \bar{F}_w \)

It remains to analyze the working flow \( \bar{F}_w \), which will be bounded by the cost \( \bar{G}^* \) of Opt and the sleeping flow \( \bar{F}_s \) of WPOOL. Precisely, we will show the following lemma.

**Lemma 9.** \( \bar{F}_w \leq (12 + \frac{12}{\epsilon})\bar{G}^* + \frac{1}{\epsilon} \bar{F}_s \).

Notice that Lemmas 8 and 9 together imply that \( \bar{G} \leq (43 + \frac{24}{\epsilon})\bar{G}^* \). By Lemma 7, \( \bar{G}^* \leq 3G(Off) \).

Hence, Theorem 5 follows.

The rest of the section proves Lemma 9 using a potential function that allows different match-ups between machines of WPOOL and Opt. Let \( \bar{F}_w(t) \) and \( \bar{F}_s(t) \) denote respectively the working flow and sleeping flow incurred by WPOOL up to time \( t \). Let \( \bar{G}^*(t) \) be the cost incurred by Opt up to time \( t \). Assume that machines are labeled with integers from 1 to \( m \).

At any time, we match each machine in WPOOL with a certain machine in Opt. Below we denote \( x(i) \) as the machine in Opt currently matched with machine \( i \) in WPOOL. This matching is only for the purpose of analysis and not known to the algorithms. Initially \( x(i) = i \) for all \( i \). To show Lemma 9, we define a potential function \( \Phi(t) \) that reflects WPOOL’s remaining working cost discounted in view of Opt’s workload in the corresponding machines. Technically, we want \( \Phi(t) \) to satisfy the following conditions: (i) Boundary condition: \( \Phi = 0 \) before any job is released and after all jobs are completed. (ii) Job completion and state transition condition: \( \Phi \) may then increase, yet the total increase due to all job arrivals is upper bounded by \( O(1 + \frac{1}{\epsilon})\bar{G}^* \) (precisely, \( (11 + \frac{12}{\epsilon})\bar{G}^* \)). (iv) Running condition: At any other time \( t \), \( \frac{d\Phi(t)}{dt} \) is upper bounded by \( O(1 + \frac{1}{\epsilon}) \cdot \frac{d\Phi(t)}{dt} + \frac{1}{\epsilon} \cdot \frac{d\Phi(t)}{dt} \).

By integrating the above condition over time, \( \bar{F}_w \leq (11 + \frac{12}{\epsilon} + 1 + \frac{1}{\epsilon})\bar{G}^* + \frac{1}{\epsilon} \bar{F}_s = (12 + \frac{12}{\epsilon})\bar{G}^* + \frac{1}{\epsilon} \bar{F}_s \), and Lemma 9 follows.

**Potential function \( \Phi \).** For any machine \( i \) of WPOOL, for any \( q \geq 0 \), recall that \( \tilde{w}(q, t) \) denotes the total fractional weight of active jobs dispatched
to machine $i$ with inverse density at least $q$ at time $t$. Define $\bar{w}_i^*(q,t)$ similarly for Opt. We will drop the parameter $t$ when $t$ refers clearly to the current time. Let $\gamma_\ell = \max(\cdot,0)$. Below is the definition of the potential function $\Phi(t)$.

$$\Phi(t) = \sum_{i=1}^m \Phi_i(t), \text{ where}$$

$$\Phi_i(t) = \frac{1}{\epsilon} \int_0^\infty \int_0^\infty (z - \bar{w}_x^{i}(q) + \mu)_+ dz dq.$$

Machine re-matching. Recall that machine re-matching is for the sake of analysis and not part of WPOOL or Opt. We allow it to operate based on a rather intricate view of the machine states. Details are as follows. At any time $t$, we define three different views of machine states in WPOOL and Opt, namely, $H_0(t)$, $H_1(t)$ and $H_2(t)$. The first two are from an operational viewpoint, and the last one is for the purpose of re-matching and analysis only. W.L.O.G., we assume that Opt, at any time, first performs the required wake-up operations before moving to other operations.

- $H_0(t)$ refers to states of the machines in WPOOL or Opt just before time $t$;
- $H_1(t)$ refers to states of the machines WPOOL or Opt immediately after time $t$; and
- $H_2(t)$ is something in between $H_0(t)$ and $H_1(t)$, it refers to the states after WPOOL and Opt have performed any operations at time $t$: WPOOL has removed all idle machines from $P$ that should sleep at time $t$, but WPOOL has not yet handled any job arriving at $t$ (and has not included more machine into $P$); and Opt has woken up all machines that are scheduled to wake up at time $t$, but Opt has not yet put any machine to sleep and has not dispatched any job arriving at $t$. At any time $t$ that a job $j$ arrives, we re-match the machines of WPOOL and Opt with respect to $H_2(t)$. Re-matching actually means computing a new matching function $x(i)$ as follows.

Let $x(i)$ be the current matching function. Note that $x(1),\ldots,x(m)$ is a permutation of $1,2,\ldots,m$. With respect to $H_2(t)$, we figure out whether a machine is in $P$ or not in accordance with WPOOL, as well as whether a machine is awake or sleep in accordance with Opt. Then, as long as we find machines $i$ and $i'$ satisfying the following conditions, we swap $x(i)$ and $x(i')$.

- WPOOL has $i$ not in $P$ and Opt has $x(i)$ awake, or procrastinating, or to be procrastinating (sleeping w.r.t. $H_2(t)$), but $j$ to be dispatched to $x(i)$; and
- WPOOL has $i'$ in $P$ and Opt has $x(i')$ sleeping and not procrastinating.

Note that $\Phi_i$, $\Phi_j$ and hence $\Phi$ may change after a swapping. Interestingly, we can show that this change is always non-increasing.

**Lemma 10.** After some $x(i)$ and $x(i')$ are swapped, $\Phi_i$ and $\Phi_j$ do not increase.

**Proof.** Machine $i$ is not in the pool $P$ and has no active jobs, so $\Phi_i = 0$ before and after the swapping. Next, we consider $\Phi_j$. Before swapping, machine $x(i')$ was sleeping and not procrastinating in Opt, thus $\bar{w}_{x(i')}(q) = 0$ for all $q$. Since $\Phi_j$ changes from $\frac{1}{\epsilon} \int_0^\infty \int_0^\infty (z - \bar{w}_{x(i')}(q) + \mu)_+ dz dq$ to $\frac{1}{\epsilon} \int_0^\infty \int_0^\infty (z - \bar{w}_{x(i')}(q) + \mu)_+ dz dq$, it can only decrease after the swapping. \hfill \square

We are ready to consider the various conditions imposed on $\Phi$. The boundary condition trivially holds. The job completion and state transition condition also hold as follows. When a job is completed by WPOOL or Opt, $\bar{w}_x(q)$ and $\bar{w}_{x(i')}(q)$ are unchanged for all $i$ and all $q$, so $\Phi$ is unchanged. Furthermore, a state transition does not affect $\Phi$.

The running condition depends solely on the job scheduling policy (HDF) and can be analyzed independently for each matched pair of machines using similar techniques for the single-machine analysis (Bansal et al. 2009); details will be given in Section 3.4.

The arrival condition depends on both sleep management and job dispatching policies. In the next subsection, we show that when a job $j$ arrives, after machine re-matching and job dispatching, the increase of $\Phi$ due to $j$ can be bounded in terms of some non-overlapping cost of Opt.

### 3.3 Arrival Condition

When a job arrives, machines may be re-matched, and then the job gets dispatched by WPOOL and Opt. $\Phi$ would possibly increase. This section is devoted to upper bounding such increase.

**Lemma 11.** The sum over all jobs of the increase in $\Phi$ due to a job arrival (after machine re-matching and job dispatching) is at most $(11 + \frac{14}{\epsilon}) \cdot \tilde{G}^*$. By Lemma 10, machine re-matching can not increase $\Phi$, hence, in the following, our analysis bases on the assumption that machine re-matching is already done.

Recall that $H_0(t)$, $H_1(t)$ and $H_2(t)$ are different views of machine states at time $t$. We define $h_i(t)$, $h_i^*(t)$ and $h_i^{+}(t)$ to be the number of machines in $P$ with respect to $H_0(t)$, $H_1(t)$ and $H_2(t)$, respectively. Define $h_i^*(t)$, $h_i^+(t)$ and $h_i^{+}(t)$ similarly for the number of awake machines in Opt.

**Type-0, Type-1 and Type-2 jobs.** To prove Lemma 11, we divide the jobs into three types and analyze them separately. Define Type-0 jobs to be the jobs which WPOOL dispatches to a zero-rwc machine (i.e. machine with no active jobs). For any other job $j$, if at its arrival time $t$, $h_j^+(t) \leq h_j^*(t)$ and $h_j^+(t) < m$, $j$ is Type-1; otherwise, $j$ is Type-2. Roughly speaking, Type-2 jobs arrive when WPOOL is using more machines than Opt (or using all the $m$ machines). It is relatively easy to show that for each Type-2 job $j$, $\Phi$ has limited increase (see Lemma 12).

For any Type-0 or Type-1 job $j$, we first observe that once $j$ is dispatched, the increase in $\Phi$ can be upper bounded in terms of WPOOL’s increase in $rwc$ due to $j$. Let $\Delta rwc(j)$ be the increase in $rwc$ to WPOOL due to $j$. $\Phi$ only increases due to the increase in $\Phi_j$, where $i$ is the machine to which WPOOL assigns $j$. 
This increase, by definition of $\Phi$, is at most
\[
\frac{1}{\epsilon} \int_0^{q(j)} \int_{\tilde{w}(q)} (z - \tilde{w}_{i}(q) + \mu) d \zeta dq
\]
\[
\leq (1 + \frac{1}{\epsilon}) \int_0^{q(j)} \int_{\tilde{w}(q)} (z + \mu) d \zeta dq
\]
\[
= (1 + \frac{1}{\epsilon}) \Delta rwc(j)
\]
where the last equality follows from Lemma 3. In other words, consider all Type-0 and Type-1 jobs, the total increase in $\Phi$ is at most $(1 + \frac{1}{\epsilon})$ times the total increase in $rwc$ of WPOOL. Lemmas 13 and 14 below give upper bounds of the increase in $rwc$ to WPOOL due to Type-0 and Type-1 jobs, respectively. Lemma 13 is relatively simple as we can show that for each Type-0 job, WPOOL’s increase in $rwc$ cannot exceed that of Opt. For Type-1 jobs, WPOOL might be using only few machines and WPOOL’s increase in $rwc$ can be way larger than Opt’s. In Lemma 14, we analyze Type-1 jobs interval by interval (instead of job by job) and show that WPOOL’s increase in $rwc$ is bounded by the static and wakeup energy of Opt. The proofs of Lemmas 12 and 13 will be shown in Appendix A.

**Lemma 12.** The total increase in $\Phi$ due to Type-2 jobs is at most $\frac{1}{\epsilon} \cdot G^*.$

**Lemma 13.** WPOOL’s total increase in $rwc$ due to Type-0 jobs is at most $G^*.$

**Lemma 14.** WPOOL’s total increase in $rwc$ due to Type-1 jobs is at most $10G^*.$

To analyze Type-1 jobs, we define lazy intervals below, which would include all arrival times of Type-1 jobs. Roughly speaking, inside a lazy interval, WPOOL is lazy in the sense that WPOOL is using no more machines than Opt.

**Lazy intervals.** A lazy interval $I = \{t_1, t_2\}$, where $t_1 \leq t_2$, satisfies the following property. Consider any view of machine status $H(t)$ where $t \in [t_1, t_2]$ and $\gamma \in \{-, +, R\}$. If $H(t) \equiv H_0(t_1)$ or $H_0(t)$, then with respect to $H_0(t)$, the number of machines in $P$ is greater than the number of awake machines of Opt (i.e., $h_0(t) > h^*_0(t)$); for any other view $H_\gamma(t)$, the number of machines in $P$ is at most the number of awake machines of Opt (i.e., $h_\gamma(t) \leq h^*_\gamma(t)$).

Before proving Lemma 14, we observe the following properties of WPOOL.

**Property 15.** When a job $j$ arrives, if WPOOL dispatches it to a machine with non-zero $rwc$, then at most two machines are added to $P$.

**Proof.** Suppose a job $j$ arrives at time $t$. Consider the moment just before WPOOL dispatches $j$. If $|P| = h_0(t) \geq m - 2$, at most two machines can be added to $P$ and the lemma holds. Now assume that $|P| < m - 2$. First note that
\[
null_{Inc,RWC}(j) = \int_0^{q(j)} \int_{\tilde{w}(q)} (\frac{1}{2} (w(j))^2 + \mu w(j)) dq.
\]
Since $j$ is dispatched to a machine with non-zero $\mu w(j)$, by the definition of WPOOL, $X_{WC} + null_{Inc,RWC}(j) < \lambda$. Just before $j$ is dispatched, let $i$ be the machine in WPOOL with the smallest $\mu w(j)$, and denoted the $rwc$ of it by $rwc(i)$. Hence $rwc(i) = \int_0^{q(i)} \int_{\tilde{w}(q)} (z + \mu) d \zeta dq = \int_0^{q(i)} (\frac{1}{2} (\tilde{w}(q))^2 + \mu \tilde{w}(q)) dq$. Since $|P| < m$, by Property 4, $rwc(i) < \lambda$.

Let $Inc, RWC(j, i, P)$ be the increase of $rwc$ if $j$ is dispatched to machine $i$. In the following, we will show that $X_{WC} + Inc, RWC(j, i, P) < 3\lambda$. Hence $X_{WC} + min_{Inc,RWC}(j, P) < 3\lambda$ and at most two machines are added to $P$.

\[
\begin{align*}
Inc, RWC(j, i, P) &= \frac{1}{\epsilon} \int_0^{q(j)} \int_{\tilde{w}(q)} (z + \mu) d \zeta dq \\
&= \frac{1}{\epsilon} \int_0^{q(j)} \int_{\tilde{w}(q)} (w(j) + \frac{1}{2} (w(j))^2 + \mu w(j)) dq \\
&= \frac{1}{\epsilon} \int_0^{q(j)} \int_{\tilde{w}(q)} (w(j)) dq + null_{Inc,RWC}(j) \\
&\leq \frac{1}{\epsilon} \int_0^{q(j)} \frac{1}{2} (w(j))^2 dq + null_{Inc,RWC}(j) \\
&\leq rwc(i) + 2 \cdot null_{Inc,RWC}(j).
\end{align*}
\]
Therefore, $X_{WC} + Inc, RWC(j, i, P) \leq rwc(i) + 2 |X_{WC} + null_{Inc,RWC}(j)| < 3\lambda$.

**Property 16.** Every Type-1 job must arrive within a lazy interval.

**Proof.** Suppose a Type-1 job $j$ arrives at time $t$. By definition, $h_0(t) \leq h^*_0(t)$. We find the largest $t_1 \leq t$ and the smallest $t_2 \geq t$ satisfying the boundary conditions of a lazy interval (the boundaries $t_1$ and $t_2$ always exist because at time 0 and at the time $t_e$ when both schedules of WPOOL and Opt end, $|P| = 1$ and the number of awake machines in Opt is 0). Thus, $h_0(t_1) = 1 > h^*_0(t_1)$, and $h_0(t_2) = 1 > h^*_0(t_2)$.

Consider any time $t' \in [t_1, t_2]$. If $h_0(t') < h^*_0(t')$ or $h_0(t') \leq h^*_0(t')$ then $h_0(t') \leq h^*_0(t')$. It follows that with respect to any view $H_0(t')$, except the two boundary views, we have $h_0(t') \leq h^*_0(t')$. Therefore, $[t_1, t_2]$ is a lazy interval covering time $t$.

Within a lazy interval $I = [t_1, t_2]$, let $O_I$ be the number of times WPOOL has removed a machine from $P$, let $I_t$ be the number of times WPOOL has added a machine into $P$ except those added at $t_2$, and let $W_I$ be the number of times Opt has waken up a machine. The above definitions imply another useful property of a lazy interval.

**Property 17.** For any lazy interval $I$, $I_t + 1 \leq W_I + O_I$.

**Proof.** By definition of a lazy interval, $h_0(t_1) > h^*_0(t_1)$ and $h_0(t_2) \leq h^*_0(t_2)$. By definition of $O_I$, $I_t$, and $W_I$, $h_0(t_1) = O_I + I_t \leq h_0(t_2)$, and $h^*_0(t_2) \leq h^*_0(t_1) + W_I$. Therefore, $h_0(t_1) - O_I \leq h^*_0(t_2) \leq h^*_0(t_1) + W_I$. Recall that $h_0(t_1) > h^*_0(t_1)$. Therefore, $-O_I + I_t < W_I$, or equivalently, $I_t < W_I + O_I$. The lemma then follows since $I_t$, $W_I$ and $O_I$ are integers.

We now prove Lemma 14. Note that we consider only the lazy intervals in which at least one Type-1 job arrives, and ignore those lazy intervals without any Type-1 job. Let $\Delta rwc'\ell$ be the increase in $rwc$ due to Type-1 jobs arriving in a lazy interval $\ell$, and let $\Delta rwc'\ell$ be the increase in $rwc$ to WPOOL due to all Type-1 jobs, respectively. Hence summing $\Delta rwc'\ell$ over all $\ell$ gives $\Delta rwc'$. Let $L$ be the set of all lazy intervals. Then $|L|$ is the total number of
lazy intervals. Define \( L = \sum_{\ell \in L} I_\ell \) and similarly for \( O_L \) and \( W_L^c \). It is useful to define \( E_\ell^c \) to be the total wake-up energy plus the total static energy used by OPT during \( \ell \), and \( E_\ell^s \) to be the sum of \( E_\ell^c \) over all \( \ell \in L \). Obviously, \( E_\ell^c \leq \bar{G}^* \). We will prove the following three relationships.

\[
\begin{align*}
(A) & \quad \Delta rwc' < (I_L + 3[L])\lambda; \\
(B) & \quad (W_L^c + O_L - |L|)\lambda \leq E_\ell^c; \\
(C) & \quad |L|\lambda \leq E_\ell^c + 2\bar{G}^*.
\end{align*}
\]

(A), (B) and (C), together with Property 17, would imply Lemma 14 immediately. The argument is as follows. By Property 17, for each \( \ell \in L \), \( I_\ell + 1 \leq W_\ell^c + O_\ell \). Summing over all \( \ell \in L \) gives \( I_L + |L| \leq W_L^c + O_L \). Therefore,

\[
\Delta rwc' < (I_L + 3[L])\lambda \leq (W_L^c + O_L + 2[L])\lambda \leq E_\ell^c + 3[L]\lambda \leq 4E_\ell^c + 6\bar{G}^* \leq 10\bar{G}^*.
\]

We come to the conclusion of this section.

Proof of (B). Consider a lazy interval \( \ell \). Note that WPOOL adds a machine to \( P \) whenever the accumulated increase in \( rwc \) due to arrived jobs exceeds \( \lambda |P| < m \), and when \( |P| = m \), arriving jobs are not Type-1. By the assumption that job arrival times are distinct, at most one job arrives at \( t_2 \). If that job is Type-1, by Property 15, it can cause at most two machines to be added to \( |P| \). Thus, the accumulated increase in \( rwc \) due to Type-1 jobs arriving in \( \ell \) is only enough to cause at most \( I_\ell + 2 \) machine additions to \( P \), i.e., \( \Delta rwc'_\ell \leq I_\ell + 2 \). Hence \( \frac{\Delta rwc'_\ell}{\lambda} \leq I_\ell + 2 \). Summing over all \( \ell \in L \), \( \Delta rwc' \leq (I_L + 3[L])\lambda \).

Proof of (C). Consider a lazy interval \( \ell \). For each time \( \ell \) wakes up a machine, it charges its wakeup energy \( \lambda \). For each time except the first that WPOOL removes a machine from \( P \), \( \lambda \) units of idling energy (counted in the counter \( X_{id} \)) must be accumulated inside \( \ell \) by WPOOL. During the lazy interval \( \ell \), OPT has at least the same number of awake machines as WPOOL and must also have incurred \( \lambda \) units of static energy when WPOOL accumulates the \( \lambda \) units of idling energy. Therefore, \( W_\ell^c + O_\ell - |L|\lambda \leq E_\ell^c \). Summing over all \( \ell \in L \), \( W_L^c + O_L - |L|\lambda \leq E_L^c \).

Proof of (A). For each lazy interval \( \ell \), we wish to show that either \( E_\ell^c \geq \lambda \) or we can charge \( \lambda \) non-overlapping units from \( 2\bar{G}^* \). Then (C) follows. Let us first consider two trivial cases of \( \ell \). Suppose \( W_\ell^c + O_\ell \geq 2 \), then the charging scheme in the proof of (B) implies \( E_\ell^c \geq \lambda \). Next, if \( W_\ell^c = 1 \) and \( O_\ell = 0 \), then we can charge the wakeup energy of OPT and \( E_\ell^c \geq \lambda \). Note that the costs of \( E_\ell^c \) charged in the two cases are non-overlapping.

It remains to consider the case when \( W_\ell^c = 0 \) and \( O_\ell = 1 \), which is indeed non-trivial. We call \( \ell \) a lazy-01 interval. We will use another charging scheme to charge \( \lambda \) units from \( 2\bar{G}^* \); in other words, any cost of OPT is charged at most once. Let \( \ell = [t_1, t_2] \). We first show that \( t_1 \neq t_2 \). Suppose, for the sake of contradiction, \( t_1 = t_2 \). Since we only consider lazy intervals in which Type-1 job arrives, by the assumption that job arrival times are distinct, one job arrives at \( t_1 \) and this job is a Type-1 job. By definition, \( h_\ell(t_1) > h_\ell^*(t_1) \) and \( h_\ell(t_1) \leq h_\ell^*(t_1) \). Together with the assumption that \( W_\ell^c = 0 \), WPOOL must have removed a machine from \( P \) at \( t_1 \). By the definition of WPOOL, there exists another idle machine in \( P \), and WPOOL would dispatch the job that arrives at \( t_1 \) to this zero-rwc machine, so the job is Type-0, contradicting that it is Type-1. Therefore, we must have \( t_1 \neq t_2 \).

By the definition of lazy interval, we have \( h_\ell(t_1) > h_\ell^*(t_1) \), and for any \( t \in (t_1, t_2] \), \( h_\ell(t) \leq h_\ell^*(t) \). Therefore, \( \phi_{t_1} \leq \phi_{t_2} \).

For each time except the first that WPOOL removes a machine from \( P \), and before another lazy-01 interval, we will prove the following running condition of the potential, say \( \phi_{t_2} \).

\[
\sum_{t_1 \leq t < t_2} \phi_{t_2} = \sum_{t_1 \leq t < t_2} \phi_{t_2} \leq \sum_{t_1 \leq t < t_2} \phi_{t_2} \leq \sum_{t_1 \leq t < t_2} \phi_{t_2}.
\]

Next, we show how to charge for lazy-01 intervals. By the definition of the subsequence, \( h_\ell^*(t) \) remains a constant, say \( M \), which is at least 1, from the start of \( t_1 \) to the end of \( \ell_p \); and inside each \( \ell_i \), \( h_\ell(t) = h_\ell^*(t) = M \). As shown before, a lazy-01 interval starts with that WPOOL removes a machine from \( P \). Thus, at some time after \( \ell_i \), \( h_\ell(t) \leq h_\ell^*(t) = M \). In each such interval, any job that arrives or completes, and both WPOOL and OPT have no change in machine state. Therefore, we have charged OPT a cost of at most \( 2\bar{A} \) for each lazy-01 interval.

Summing over the three types of intervals, \( |L|\lambda \leq E_L^c + 2\bar{G}^* \).

3.4 Running condition

We show the following running condition of the potential analysis of \( F_{\lambda} \), which considers how \( \phi \) changes in an infinitesimal amount of time \([t, t + \text{dt})\] when there is no job arrival or completion, and both WPOOL and OPT have no change in machine state.

Lemma 18. Consider any time \( t \) without job arrival, completion, and machine state transition in both WPOOL and OPT. Then

\[
\frac{\Delta \phi_{t+\text{dt}}}{\text{dt}} + \frac{\Delta \phi_{t+\text{dt}}}{\text{dt}} \leq (1 + \epsilon) \left( \frac{\Delta \phi_{t+\text{dt}}}{\text{dt}} + \frac{\Delta \phi_{t+\text{dt}}}{\text{dt}} \right).
\]

At least one sleep must exist since \( h_\ell^*(t) \geq 1 \) and OPT must put this machine to sleep after \( \ell_p \).
Since $\Phi(t) = \sum_{i=1}^{m} \Phi_i(t)$, we analyze on a per-machine basis. We focus on analyzing a certain machine $i$ in WPOOL and the matching machine $x(i)$ in $\text{Opt}$, respectively (0 if no active jobs). The processing of $w_i$ is trivial that $(d \tilde{w}_i(t) + \frac{d\Phi}{dt}) \leq 0$. Similarly, we can also divide the analysis into subcases and verify in each subcase that $(d \tilde{w}_i(t) + \frac{d\Phi}{dt}) \leq (1 + \frac{1}{\varepsilon}) \cdot (\frac{dG^*}{dt})(x(i)) + \frac{1}{\varepsilon} (\frac{dF}{dt})(x(i))$.}

References


Bansal, N. & Chan, H.L. (2009), Weighted flow time does not admit $O(1)$-competitive algorithms, in *Proc. SODA*, pp. 1238–1244.


Belady, C. (2007), ‘In the data center, power and cooling costs more than the IT equipment it supports’, *Electronics Cooling Magazine*, 13(1), 24–27.


Appendix A: Omitted Proofs

In this appendix, we give the proofs omitted from Section 3.

Lemma 12. The total increase in $\Phi$ due to Type-2 jobs is at most $\frac{2}{3} \hat{G}^*$. 

Proof. We analyze job by job. Consider a Type-2 job $j$ arriving at time $t$. Let $\Delta Wrc$ and $\Delta Wrc^*$ be the increase in the $wrc$ due to $j$ in WPOOL and Opt, respectively. We will show that after WPOOL and OPT dispatch $j$, the change of $\Phi$, denoted $\Delta \Phi$, is at most $\frac{2}{3} \Delta Wrc^*$ over all jobs equals the working cost of Opt, which is at most $\hat{G}^*$. Hence, the total increase in $\Phi$ due to Type-2 jobs is at most $\frac{2}{3} \hat{G}^*$.

Suppose WPOOL and Opt dispatch $j$ to machines $i$ and $k$, respectively. Let $x(i)$ be the matching function updated just before dispatching $j$. Also define $u$ such that $x(u) = k$. Both $\tilde{w}(i)$ and $\tilde{w}(k)$ increase by $w(j)$ for $q \in [0, q(j)]$. We consider two cases depending on whether $x(i) = k$.

Case 1. When $x(i) \neq k$, i.e. $i \neq u$. We can show that when $j$ arrives at time $t$, $u$ is in $P$ just before WPOOL dispatches $j$. The argument is as follows. Consider $H_n(t)$, since $j$ is a Type-2 job, $h_u(t) > h_q(t)$ or $h_u(t) = m$. By definition, Opt has at most one procrastinating machine at time $t$, or precisely, with respect to $H_n(t)$, $H_u(t)$ and hence $H_n(t)$. Thus, at $H_u(t)$, $|P| = h_u(t)$ is at least the number of awake or procrastinating machines in Opt. All those machines in Opt, including $k$, must be matched with a machine in $P$. As $x(u) = k$, we have $u \in P$.

By Lemma 3, $\Delta Wrc^* = \int_0^{q(j)} w(j) (\tilde{w}(i)(q) + \frac{1}{2} w(j) + \mu) dq$. Note that $\Delta \Phi = \hat{\Phi}_f + \hat{\Phi}_u$. First consider $\hat{\Phi}_f$. $\hat{\Phi}_f = \frac{1}{\epsilon} \int_0^{q(j)} \tilde{w}(i)(q) w(j) dq \leq \frac{1}{\epsilon} \int_0^{q(j)} \tilde{w}(i)(q) w(j)(z + \mu) dq$. As shown above, $u \in P$ when WPOOL handles $j$, but WPOOL assigns $j$ to machine $i$ instead of $u$. By definition of $\Delta Wrc$, $\Delta Wrc = \int_0^{q(j)} \tilde{w}(u)(q) w(j)(z + \mu) dq$. Hence, we have

$$\Delta \Phi_f \leq \frac{1}{\epsilon} \int_0^{q(j)} \tilde{w}(u)(q) w(j)(z + \mu) dq = \frac{1}{\epsilon} \int_0^{q(j)} w(j) (\tilde{w}(u) + \frac{1}{2} w(j) + \mu) dq$$

Then consider $\Delta \Phi_u$. For any $q \in [0, q(j)]$, let $\delta(q) = \int_0^{q(j)} (z - \tilde{w}(u)(q) + \mu) dq = (z - \tilde{w}(u)(q) - w(j) + \mu) dq$. Hence, we have

$$\Delta \Phi_u \geq \frac{1}{\epsilon} \int_0^{q(j)} w(j) (\tilde{w}(u)(q) + \frac{1}{2} w(j) + \mu) dq$$

It follows from the bounds of $\Delta \Phi_f$ and $\Delta \Phi_u$ that,

$$\Delta \Phi = \Delta \Phi_f + \Delta \Phi_u \leq \frac{2}{3} \Delta Wrc^*$$

Case 2. When $x(i) = k$, i.e. $i = u$. By Lemma 3, $\Delta Wrc^* = \int_0^{q(j)} w(j) (\tilde{w}(i)(q) + \frac{1}{2} w(j) + \mu) dq$. Note that $\Delta \Phi = \hat{\Phi}_f$. For any $q \in [0, q(j)]$, let $\delta(q) = \int_0^{q(j)} ((z - \tilde{w}(i)(q) + \mu) + w(j) + \mu) dq$. Then $\Delta_1 = \frac{1}{\epsilon} \int_0^{q(j)} \tilde{w}(i)(q) w(j)(z - \tilde{w}(i)(q) - w(j) + \mu) dq$, and $\Delta_2 = \frac{1}{\epsilon} \int_0^{q(j)} \delta(q) dq$. Then we have $\Delta \Phi = \Delta_1 + \Delta_2$. Observe that $\Delta_1$ and $\Delta_2$, respectively, plays the same role as $\Delta \Phi_f$ and $\Delta \Phi_u$ in Case 1. By a similar calculation, we have $\Delta \Phi = \Delta_1 + \Delta_2 \leq \frac{2}{3} \Delta Wrc^*$.

Lemma 13. WPOOL’s total increase in $wrc$ due to Type-0 jobs is at most $\hat{G}^*$.

Proof. Recall that WPOOL dispatches a Type-0 job to a zero-wrc machine. Suppose Opt dispatches $j$ to machine $k$. By Lemma 3, the increase in $wrc$ to WPOOL due to $j$ is $\int_0^{q(j)} \frac{w(j)}{1 + \epsilon} dq$, which must be less than $\int_0^{q(j)} \tilde{w}(i)(q) w(j)(z + \mu) dq$ (the increase in $wrc$ to Opt due to $j$). Summing over all Type-0 jobs, the total increase in $wrc$ to WPOOL due to dispatching all Type-0 jobs is at most $\hat{G}^*$. □
Rotated Library Sort

Franky Lam1

Raymond K. Wong1,2

1 School of Computer Science & Engineering
University of New South Wales
NSW 2052, Australia

2 National ICT Australia, Australia

Abstract

This paper investigates how to improve the worst case runtime of INSERTION SORT while keeping it in-place, incremental and adaptive. To sort an array of n elements with w bits for each element, classic INSERTION SORT runs in \( O(n^2) \) operations with \( wn \) bits space. GAPPED INSERTION SORT has a runtime of \( O(n \log n) \) with a high probability of only using \( (1+\epsilon)wn \) bits space. This paper shows that ROTATED INSERTION SORT guarantees \( O(\sqrt{n} \log n) \) operations per insertion and has a worst case sorting time of \( O(n^{1.5} \log n) \) operations by using optimal \( O(w) \) auxiliary bits. By using extra \( \Theta(\sqrt{n} \log n) \) bits and recursively applying the same structure \( l \) times, it can be done with \( O(2n^{1.5}) \) operations. Apart from the space usage and time guarantees, it also has the advantage of efficiently removing the \( i \)-th element in constant time. This paper presents ROTATED LIBRARY SORT that combines the advantages of the above two improved approaches.

1 Introduction

In this paper, given the universe \( U = \{1, \ldots, u\} \), we use the transdichotomous machine model (Fredman & Willard 1994). The word size \( w \) of this machine model is \( w = O(\log u) \) bits and each word operation of this model takes \( O(1) \) time (this paper defines \( \log \) as \( \log_2 \)). This paper assumes all the \( n \) elements, within the universe \( U \), stored in the array \( A \). This means that each element takes exactly \( w \) bits and the array \( A \) uses \( wn \) bits in total. The traditional INSERTION SORT algorithm belongs to the family of exchange sorting algorithms (Knuth 1998) which is based on element comparisons. It is similar to how humans sort data and its advantage over other exchange sorting algorithms is that it can be done incrementally. The total order of all elements are maintained at all times, traversal and query operations can be performed on \( A \) as INSERTION SORT never violates the invariants of a sorted array. It is also adaptive, as its runtime is proportional to the order of the insertion sequence. During insertion of a new element \( x \) to an existing sorted array \( A \), INSERTION SORT finds the location of \( x \) for insertion and creates a single gap by right-shifting all the elements larger than \( x \) by one position. Obviously, its worst case \( \log n \) \( + n \) comparisons combined with its worst case \( \Omega(n^2) \) element moves and a total of \( O(n^2) \) operations makes it impractical in most situations, except for sorting with a small \( n \) or when the insertion sequence is mostly sorted. This paper investigates how to improve INSERTION SORT while keeping its nice incremental and adaptive properties.

This paper is organized as follows. Section 2 discusses previous work related to this paper. We then present the rotated sort algorithm in Section 3 that achieves the \( O(n^{1.5} \log n) \) operations. After that we discuss the time and space complexity as well as their tradeoffs in Section 4 and Section 5. Section 6 shows how to achieve \( O(2n^{1.5}) \) operations by applying the idea recursively and Section 7 combines the idea of both LIBRARY SORT and ROTATED SORT. Finally, Section 8 concludes the paper.

2 Background

2.1 Incremental Sorting Problem

First we define the incremental sorting problem as maintaining a sequence \( S \) (not necessarily an array \( A \)) of \( n \) elements in universe \( U \) subject to the following functions:

- \text{insert}(x, S): insert \( x \) into \( S \).
- \text{member}(x, S): return whether element \( x \in S \).
- \text{select}(j, S): return the \( j \)-th smallest element where \( \text{select}(1, S) \) is the smallest element in \( S \) and \( \text{select}(j, S) < \text{select}(k, S) \) if \( j < k \).
- \text{predecessor}(j, S): special case of \( \text{select}(j-1, S) \), but \( \text{select}(j, S) \) is already known.
- \text{successor}(j, S): special case of \( \text{select}(j+1, S) \), but \( \text{select}(j, S) \) is already known.

This model defines incremental sorting as a series of \( \text{insert}(x, S) \) from the input sequence \( X = \{x_1, \ldots, x_n\} \), such that we can query the array \( S \) using \( \text{select} \) and \( \text{member} \) between insertions; or we can traverse \( S \) using \( \text{predecessor} \) and \( \text{successor} \) between insertions. The traversal functions might seem to be redundant, but in fact they are only redundant when \( \text{select} \) can be done in \( O(1) \) operations, which Corollary 1 shows that we have to relax this requirement. For most cases, when \( \text{select} \) cannot be done in constant time, \( \text{predecessor} \) and \( \text{successor} \) can still be done in \( O(1) \) operations. It is possible that some incremental sorting algorithms can be done in-place if they reuse the same space of the input sequence \( X \).

Although there is no strict guidelines, but similar to most other definition of incremental algorithms, we only consider a particular algorithm is an incremental sorting algorithm if the runtime of its query functions after every individual insertion is comparable to the runtime of the same query functions of normal sorting algorithm after \( n \) insertions.
2.2 Adaptive Sorting Problem

The adaptive sorting problem is defined as any sorting algorithm that uses only $O(w)$ temporary space and achieves $O(1)$ operations for select requires at least $O(\text{inv}(X))$ swaps.

It is trivial that with the above scenario, INSERTION SORT is the only optimal sorting algorithm as there are no other possible alternative approaches that can accomplish all the above constraints, thus we have to relax some of the requirements — this paper assumes select does not need to run in $O(1)$ time, meaning partial order is tolerable until all elements in $X$ are inserted. It is essential that select should still run reasonably fast, or it will lose the purpose of being incremental.

2.3 Variants of Insertion Sort

2.3.1 Fun Sort

Biedl et al (Biedl et al. 2004) have shown an in-place variant of INSERTION SORT called FUN SORT that achieves worst case $O(n^2 \log n)$ operations. They achieve the bound by applying binary search to an unsorted array to find an inversion and reduce the total number of inversions by swapping them. By picking two random elements $A[i]$ and $A[j]$, $i < j$ and swapping them if it is an inversion, the total number of inversions is reduced by at least one, up to $2(j-i)-1$ reductions; because for $\forall k, i < k < j$, if $(i, j)$ is an inversion, either $(i, k)$ or $(k, j)$ is an inversion, or both. As stated before, any algorithm that can maintain after $O(n^2)$ element swaps. By observation, its performance is rather poor in the worst case, as $(j-i)$ swaps are required, but its average runtime seems rather fast. Strictly speaking, FUN SORT does not belong to a variant of INSERTION SORT as it is not strictly incremental, but it is an interesting adaptive approach.

2.3.2 Library Sort

Bender et al (Bender et al. 2004) have shown that by having a $\epsilon n \log n$ bits space overhead as gaps, and keeping gaps evenly distributed by redistributing the gaps when the $2^{\epsilon}$-th element is inserted, GAPPED INSERTION SORT, or LIBRARY SORT for short, has a high probability of achieving $O(n \log n)$ operations. As most sorting algorithms can be done in-place, we can make a fair assumption that the sorted result must use the same memory location. The auxiliary space cost of LIBRARY SORT is $(1 + \epsilon)n \log n$ bits as it needs to create a temporary continuous array $A'$. Alternatively, their approach can be improved by tagging a temporary auxiliary $\epsilon n \log n$ size space to $A$, thus creating a virtual $(1 + \epsilon)n \log n$ size array $A'$, making the algorithm less elegant but not affecting the time bound or space bound.

Unfortunately, $\epsilon$ needs to be chosen beforehand, and large $\epsilon$ does not guarantee $O(n \log n)$ operations as they have made an assumption that $A$ is a randomly permuted sequence within $U$. To describe it in another way, the algorithm can randomly permute the input with $O(n)$ time before sorting. By permuting the input, the algorithm becomes insensitive to the input sequence, which by definition, LIBRARY SORT is not an adaptive algorithm that can take advantage of a nearly sorted sequence. Under the incremental sorting model, input comes individually, permuting the future input is impossible. With the incremental sorting model, without the permutation of input and with adversary insertion (such as reverse sorted order that can happen fairly often in real life scenarios), the performance of this algorithm degrades to amortized $\Omega(\sqrt{n})$ operations per insertion, regardless of the $\epsilon$. This makes the worst case $O(n^{1.5})$ operations, although it might be possible to improve the runtime cost to worst case amortized $O(\log^2 n)$ per insertion (Bender et al. 2002). Although in their assumptions the time bound is amortized per insertion, regardless of the disorder of the input sequence, as their algorithm needs to rebalance the gaps on the $2^{\epsilon}$-th insertion.

Finally, Bender et al did not address that their approach takes worst case $O(n + \epsilon n)$ operations to perform select$(j, A)$, which finds the $j$-th smallest element in an array $A$. This is because the $j$-th smallest element does not locate at $A[j-1]$. It locates at somewhere between $A[j-1]$ to $A[j-1 + \epsilon j / n]$ depending on the distribution of the gaps. Without knowing the location of the gaps, a linear scan is required to determine the rank of the $j$-th smallest element between insertions. It is possible to improve select by using more space to maintain the locations of gaps, using a similar structure like the weight-balanced $B$-tree by Dietz (Dietz 1989).

2.3.3 Rotated Sort

ROTATED INSERTION SORT, or just Rotated SORT for short, is based on the idea of the implicit data structure called rotated list (Munro & Suwedanda 1979). Implicit data structure is where the relative ordering of the elements is stored implicitly in the pattern of the data structure, rather than explicitly storing the relative ordering using offsets or pointers. Rotated list achieves $O(n \log n)$ operations using constant $O(w)$ bits temporary space, or $O(n^{1.5})$ operations with extra $\Theta(\sqrt{n} \log n)$ bits temporary space, regardless of $w$. It is adaptive as its runtime depends on $\text{inv}(X)$. It is incremental as select can be done in constant time.

3 Rotated Sorted

In essence, the sorted list is done by controlling the number of element shifts from $O(n)$ shifts per insertion to a smaller term, such as $O(\sqrt{n})$ shifts or even $O(\log n)$ shifts, by virtually dividing $A$ into an alternating singleton elements and rotated lists that satisfies a partial order. By having an increasing function that controls the size of all the rotated lists, we only need to push the smallest element and pop the largest element between a small sequence of rotated lists per insertion.

3.1 Rotated List

A rotated list or sorted circular array, is an array $L = [0, \ldots, n-1]$ with a largest element $L[m] > L[i]$, $0 \leq i < n$ and $L[i \mod m] < L[i + 1 \mod n]$, $i \neq m$.

We need $\lfloor \log n \rfloor$ comparisons to find the positions of the minimum and maximum elements in the array, or constant time if we have maintained a $\lfloor \log n \rfloor$ bits pointer to store $m$ explicitly for $L$.

This paper uses the same terminologies from Frederickson (Frederickson 1983), where the rotated list $L$ has
two functions — easyExchange, where the smallest element \( x < L[i] \), \( 0 \leq i < n \) replaces the largest element \( L[m] \) and returns \( L[m] \); hardExchange is identical to easy exchange, but \( x \) can be any number. This paper defines an extra function normalize that transform the rotated list to a sorted array.

As described in (Frederickson 1983), easy exchange can be done in \( O(1) \) operations once \( L[m] \) is found, as the operation only needs to replace \( L[m] \) with the new smallest element \( x \). Array \( L \) still satisfies as a rotated list, but the position \( m' \) of the new largest element \( L[m'] \) is left-circular-shifted by one \( (m' = m - 1, \text{or } m' = n - 1 \text{ if } m = 0) \). Hard exchange is \( O(n) \) since it needs to shift all the elements larger than \( x \) in the worst case. Figure 1 shows easy exchange and hard exchange examples on a rotated list.

Normalization can be done in \( O(n) \) time, an obvious way to achieve this is by having a temporary duplicate but the exact bound can also be achieved in-place recursively by using Algorithm 1, which has exactly optimal \( 2n \) words read and \( 2n \) words write for the array \( L \). The same algorithm can also be done iteratively.

Algorithm 1 Transformation of a rotated list \( L \) to a sorted list \( L' \) with \( 2n \) words read and \( 2n \) words write. \( L[m] \) is the largest element and \( n = |L| \).

```plaintext
1. normalize(m, L):
   1. if \( m < \frac{n}{2} - 1 \) then
   2. swap(L[0],...,m], L[m+1,...,2m+1])
   3. normalize(2m+1, L[m+1,...,n-1])
   4. else \( m > \frac{n}{2} - 1 \) then
   5. swap(L[0],...,n-m-2], L[m+1,...,n-1])
   6. normalize(m, L[n-m-1,...,m])
   7. else
   8. swap(L[0],...,m], L[m+1,...,n-1])
```

3.2 Implicit Dynamic Dictionary

The dynamic dictionary problem is defined as follows. Given a set \( D \subseteq U, |D| = n \), we need to implement efficiently \( \text{member}(x, D) \) to determine whether \( x \in D \) and insert \( x, D \) that insert \( x \) into \( D \). It is a subset of the incremental sorting problem. Given a monotonic (strictly) increasing integer function \( f: \mathbb{Z}^+ \rightarrow \mathbb{Z}^+ \), dynamic dictionary can be implemented implicitly by using an array \( A \), and be visualized as a 2-level rotated lists. We divide \( A \) into a list of \( r \) pairs \( D = \langle P_1, ..., P_r \rangle \), each pair \( P_i \) consists of a singleton element \( e_i \) and a sub-array \( L_i \) of size \( f(i) \) that is used as a rotated list. For an array of size \( n \), we have \( n \leq \sum_{i=1}^{r} f(i) + 1 \). The purpose of having a monotonic increasing integer function is that the number of blocks will always be proportional to the array size, regardless of the number of insertions. This also avoids amortized runtime cost as it requires no re-dividing when the array grows. This invariant needs to be guaranteed in order to have the runtime guarantee as it controls the number of soft exchanges performed per insertion.

4 Analysis

**Lemma 1.** The total number of rotated lists, or the total number of singleton elements in the implicit dictionary structure of size \( n \) is at most \( \lceil \sqrt{2n} \rceil \), regardless of the increasing function \( f \).

**Proof.** To make it simpler, we can increase \( n \) to \( n' = \sum_{i=1}^{r} (i+1) \geq 2r \), and if we use the slowest increasing function on \( \mathbb{Z}^+ \) where \( f(i) = i \), then:

\[
\begin{align*}
n' &= \sum_{i=1}^{r} (i+1) \\
2n' &= r^2 + 3r \\
2n' + \frac{9}{4} &= (r + \frac{3}{2})^2 \\
r &= \sqrt{2n' + \frac{9}{4}} - \frac{3}{2} \\
r &\leq \sqrt{2n'}
\end{align*}
\]

We can now analyze the total runtime cost of maintaining the offset \( m \) for the largest element \( L_k[m] \) on all rotated lists \( L_k \).

**Lemma 2.** The total space cost of maintaining \( M = \langle m_1, ..., m_r \rangle \), where \( m_k \) is the position of the largest element for the rotated lists \( L_k \), is \( \sum_{i=1}^{r} \lg f(i) \) bits, or it can be done in \( \Theta(\sqrt{n} \lg n) \) bits.

**Proof.** Using \( f(i) = i \) and Lemma 1, we have the list of rotated lists \( \langle L_1, ..., L_r \rangle \) of size \( 1, ..., \sqrt{2n} \). The sum of the bits required is \( \sum_{i=1}^{r} \lg i = \lg(\sqrt{2n})! \). By Stirling’s approximation, it is reduced to approximately \( \sqrt{2n} \lg \sqrt{2n} - 2n + 1 = \Theta(\sqrt{n} \lg n) \).

**Lemma 3.** select takes \( O(1) \) operations using extra \( \Theta(r \lg n) \) bits space, or it can be done in extra \( \Theta(\sqrt{n} \lg n) \) bits space. On optimal space, select takes \( O(\lg f(r)) \) operations.

**Proof.** To calculate select \((j, S)\), we need to find which rotated list \( L_k \) that it is located, meaning we need to find the smallest \( k \) such that \( \sum_{i=1}^{k} (f(i) + 1) \gg \). When \( L_k \) is found, we can get \( m_k \) in \( O(1) \) operations with \( \Theta(\sqrt{n} \lg n) \) bits space using Lemma 2. From Lemma 1, we need at most \( r \leq \sqrt{2n} \) rotated lists and storing the beginning offset of any rotated list takes at most \( |L_k| \) bits. Therefore, we can hold the whole offset table in \( \Theta(r \lg n) = \Theta(\sqrt{n} \lg n) \) bits. If there exists a function \( g(x) = \int f(x) \, \text{d}x \), then \( O(1) \) operation can be done without the offset table. For example, using \( f(i) = i \), \( g(x) = \frac{3x^2}{2} \), \( L_k \) can be found by doing \( k = \sqrt{2j + 9/4 - 3/2} \).

Without maintaining \( m_k \), it takes \( O(1) \) time to find \( L_k \), along with an extra \( \lfloor \lg f(k) \rfloor \) comparisons to find \( m_k \), in worst case where \( k = r \), the time complexity becomes \( O(\lg f(r)) \).

**Lemma 4.** member can be done in \( O(\lg r + \lg f(r)) \) operations. On optimal space, we need no more than \( \frac{\lg f(r)}{2} \) comparisons, or no more than \( |L_k| + O(1) \) comparisons using \( \Theta(\sqrt{n} \lg n) \) bits.
Proof. We perform member \((x, D)\) by doing a binary search on all the singletons \(\{e_0, \ldots, e_r\}\) to determine which rotated list \(L_k\) does \(x\) belong to (or returns the position of singleton element \(e_k = x\) itself if we are lucky), then followed by a binary search on the rotated list \(L_k[0, \ldots, f(k) - 1]\) to find the largest element \(m_k\) and finally perform another binary search on \(L_k\) to find \(x\). The total number of comparisons is \(|\lfloor \lg r \rfloor + 2\lfloor \lg f(k) \rfloor|\). In the worst case where \(k = r\), the search is within the last (and largest) rotated list \(L_r\). Let \(f(i) = i\) then the worst case cost is \(|\lfloor \lg r \rfloor + 2\lfloor \lg f(i) \rfloor| = 3\lfloor \lg \sqrt{2n} \rfloor = \Theta(\lg n) + O(1)\).

Using Lemma 2, we eliminate one binary search and the cost is reduced to \(|\lfloor \lg r \rfloor + \lfloor \lg f(r) \rfloor| = |\lfloor \lg n \rfloor| + O(1)\).

The probability of finding a singleton element is \(P(x \in \{e_0, \ldots, e_r\}) = \frac{r}{n}\) and the probability of finding an element in the rotated list \(L_k\) is \(P(x \in L_k) = \frac{k}{n}\).

Lemma 5. With optimal space, insert is \(O(\lg r + \sum_{i=1}^{r-1} \lg f(i) + f(r))\) operations, or it can be done in no more than \(O(\sqrt{n} \lg n)\) operations. Using \(\Theta(\sqrt{n} \lg n)\) bits, the time complexity is \(O(r + f(r))\), or it can be done in \(O(\sqrt{n})\) operations.

Proof. To perform insertion, first we need to locate the rotated list \(L_k\) for insertion by performing a \(|\lfloor \lg r \rfloor|\) search on the singleton elements, then a hard exchange is performed on \(L_k\), which is followed by a sequence of soft exchanges will be done from \(L_{k+1}\) to \(L_{r-1}\) and terminated with either a hard exchange or append to \(L_r\). The total cost is \(|\lfloor \lg r \rfloor + f(k) + \sum_{i=k+1}^{r-1} \lg f(i) + f(r) + O(1)|\) or \(O(\lg r + \sum_{i=1}^{r-1} \lg f(i) + f(r))\) as \(f(k) \ll f(r)\). In worst case, where \(k = 1\), using \(f(i) = i\), the cost is \(O(\lg \sqrt{2n} + (\sqrt{2n} \lg \sqrt{2n} - 2\sqrt{2n} - 2\sqrt{2} - 2 + 1) + \sqrt{2n}) = O(\sqrt{n} \lg n)\) operations.

From Section 3.1, with space specified in Lemma 2, soft exchange takes \(O(1)\) time. In worst case, where \(k = 1\), we need to perform soft exchange on all rotated lists, except \(L_{k+1}\) and \(L_{r-1}\), thus \(r - 2\) rotated lists in total. Therefore, the total time complexity of \(|\lfloor \lg r \rfloor|\) binary search, initial hard exchange on \(L_1\), sequence of \(r - 2\) soft exchanges and the final hard exchange on \(L_{r-1}\) is \(|\lfloor \lg r \rfloor + f(k) + r + f(r) + O(1)| = O(r + f(r))\). Using \(f(i) = i\), we have \(O(\sqrt{2n} + 2\sqrt{2n}) = O(\sqrt{n})\) operations.

Theorem 1. Rotated Sort can be done in worst case \(O(n^{1.5} \lg n)\) operations with only \(O(w)\) bits space, or in worst case \(O(n^{1.5})\) operations with \(\Theta(\sqrt{n} \lg n)\) bits space.

Proof. First, visualize an array \(A\) as a concatenation of an implicit dictionary \(D\) with size \(0\) with the input sequence \(X\) with \(n\) remaining elements. We increase \(D\) by inserting \(A[i]\) into \(D\) at every step \(i\). Using \(f(i) = i\), from Lemma 5 where each insertion takes worst case \(O(\sqrt{\lg i})\), the total can be done in \(\sum_{i=1}^{n} \sqrt{\lg i} \approx \int_{1}^{n} (\sqrt{\lg x}) = O(n^{1.5} \lg n)\).

Theorem 2. select\((j, A)\) can be done adaptively in constant time with extra \(\Theta(\sqrt{n} \lg n)\) bits space and it can be done in \(O(1)\) operations after \(n\) insertions for Rotated Sort without using extra space.

Proof. Using the \(O(1)\) time function \(g(k) = \frac{k^2+k}{2}\) for \(f(i) = i\), from the proof at Lemma 3, \(L_k\) can be found in \(O(1)\), select\((j, A)\) can be implemented simply using \(A[m_j + j - g(k)] \bmod f(k)) + g(k))\). Once after \(n\) insertions, we only need to perform normalization, in which the runtime \(O(n)\) takes the lower term of the sort. Now we simplify the function select\((j, A)\) = \(A[j - 1]\). The above leads to the proof of the following:

Corollary 2. predecessor and successor can be done in \(O(1)\) operations adaptively with \(O(\sqrt{n} \lg n)\) space if \(g(i)\) exists and it can be done in \(O(1)\) operations after \(n\) insertions for Rotated Sort without using extra space.

Proof. Trivial. They are both special cases of select. Alternatively, successor can be performed even faster by checking \(m_i\) and \(m_{i+1}\), where \(L_i\) is the rotated list that select\((j, S)\) belongs to.

5 Choosing the Increasing Function

The increasing function \(f\) affects the time complexity of the insertion and thus the sorting time. We have shown in Theorem 1 that using the slowest increasing integer function, Rotated Sort takes worst case \(O(n^{1.5})\) operations.

Note that the dominant time is spent on performing easy exchange on \(O(\sqrt{n})\) rotated lists for every insertion. One idea to improve this is to reduce \(r\) from \(O(\sqrt{n})\) to \(O(\lg n)\) by using an exponential growing function. However, the larger the ratio of \(f(i+1)/f(i)\), the more expensive it is to perform a hard exchange on the rotated list. In the case where \(f(i) = 2^i\), hard exchange takes worst case \(n/2\) right-shifts on the last rotated list \(L_{r-1}\). We need to minimize the insertion cost \(O(\lg r + f(r))\) from Lemma 5 by choosing the appropriate increasing function.

Theorem 3. The function \(f(i) = i\) is optimal, up to a constant factor, to control the increasing size for the \(2\)-levels rotated lists in Rotated Sort.

Proof. If we make \(r\) as the \(x\)-axis and \(f(r)\) as the \(y\)-axis, and we limit the maximum range of both axes to \(n\), then from Lemma 1, we know the area \(\int_{0}^{n} (f(x) + 1) = n\). Even if we assume \(n\) does not grow (thus we allow the change of rate of \(f\) to be 1) the optimal function is where \(r = f(r) = \sqrt{n}\), as the problem is equivalent to minimizing the circumference of a fixed rectangular area. With those values, insertion takes \(O(\lg r + f(r)) = O(\sqrt{n})\) operations. Therefore, from Lemma 5, the slowest increasing integer function \(f(i) = i\) is already close to the optimal up to a constant factor.

6 Multi-Level Rotated List

To reduce the number of hard and easy exchanges, we can apply the idea of rotated list divisions recursively on each rotated list itself. Each sub-array \(L_i\) within \(A\) is further divided up recursively for \(l\) number of times; we can see that even for the fast growing function \(f(i) = 2^i\), an array of size \(n\) will consist of at most \(l = \lfloor \lg n \rfloor\) rotated lists
with exponential growing size, and the maximum number of levels \( l \) is at most \( \lg n \).

**Lemma 6.** \( \text{insert}(x,S) \) can be done in \( O(2^{\sqrt{\lg n}}) \) operations by using an \( l \)-levels rotated list, showed by Raman et al (Raman et al. 2001).

With Lemma 6, ROTATED SORT can be done in \( \sum_{i=1}^{n} (2^{i+1} + \frac{1}{2}) \) operations; we know that to minimize the sorting cost, \( l \) should be chosen to minimize \( 2^{\sqrt{\lg n}} \). We can always choose the perfect \( l \) but make the cost amortized, by performing normalization that takes \( O(n) \) operations whenever the array grows until \( l \) is not optimal. A perfectly sorted array can be visualized as an \( l \)-levels rotated list, regardless of \( l \). We can maintain the optimal value of \( l \) by normalization, with the amortized constant cost. Therefore, the overall sorting cost can remain the same.

**Corollary 3.** The optimal number of levels on the multi-levels rotated list is \( l = \sqrt{\lg n} \). As \( 2^l = n^\frac{1}{l} \implies l = \lg n \).

**Theorem 4.** ROTATED SORT can be done in \( O(2^{\sqrt{\lg n}}) \) operations.

**Proof.** From Lemma 6 and Corollary 3, we know that the above time bound can be achieved, amortized, by doing normalization on every \( 2^{2l} \)-th insertion. The same bound can be de-amortized easily, simply by having \( (i+1) \)-level rotated list for rotated lists \( L_i \), regardless of \( i \).

The runtime on Theorem 4 is smaller than \( O(n \sqrt{\lg n}) \) but larger than \( O(n \sqrt{\lg n}) \), and they are all growing in a decreasing rate with respect to \( n \).

The advantage of INSERTION SORT is that not only it is incremental, but also adaptive, where traditional INSERTION SORT performs exactly \( \text{in}[X] + 2n - 1 \) data moves (Estivill-Castro & Wood 1992). Knuth (Knuth 1998) and Cook et al (Cook & Kim 1980) showed that INSERTION SORT is best for nearly sorted sequences. The same adaptive property can also apply for ROTATED SORT.

**Lemma 7.** ROTATED SORT can be done in best case \( O(n) \) operations.

**Proof.** During \( \text{insert} \), changing the worst case cost by only a constant, we can perform binary search by searching from the last singleton element \( e_r \) instead of \( e_{r/2} \). This only increases the number of comparison by \( 1 \) but reduces the \( \lceil \lg r \rceil \) comparisons of singleton elements to only \( 1 \). In the best situation, no hard exchange or soft exchange is performed, making the time complexity \( O(n) \).

**Theorem 5.** ROTATED SORT can be adaptive according to the inversion of \( X \).

**Proof.** Instead of optimizing for just the best case, we want to generalize it for any nearly sorted sequence \( S \), where the total cost is proportional to \( \text{in}[X] \). We need to perform a sequence of exponential searches of \( x \) from the tail of \( \langle e_{r-1}, e_{r-2}, \ldots, e_{r-2k} \rangle \) until \( e_{r-2k} < x \) and \( e_{r-2k+1} > x \), then we begin a binary search of \( x \) between \( e_{r-2k} \) and \( e_{r-2k+1} \).

### 7 The Best of Both Worlds — Rotated Library Sort

Instead of using multi-level rotated list, an alternative way to minimize the total number of soft exchanges and hard exchanges is to combine the concept of gaps from LIBRARY SORT with ROTATED SORT. For every rotated list \( L_1 \), we maintain an extra array \( K_i \) with the size \( \epsilon f(i) \). We now treat \( J_i = \langle K_i, L_i \rangle \) as one single array that acts as a rotated list. We maintain the total number of gaps (its total value) and the position offset of the largest element \( n_i \), for \( J_i \) instead of \( L_i \). In this setting, the gaps of \( J_i \) are always located between the smallest element and the largest element.

During insertion, if the initial rotated list \( J_1 \) contains gaps, only the initial hard exchange on \( J_0 \) is performed. No soft exchange nor hard exchange on the final rotated list \( J_{n-1} \) is required. If \( J_0 \) is full before the insertion, we still need to perform soft exchanges from the rotated list \( J_{n-1} \) up to the rotated list \( J_{n+1} \). However, these soft exchanges will stop at the first rotated list that contains at least one gap. This can also be seen as an improved version of LIBRARY SORT — by clustering the gaps into \( r \) blocks in order to find them quickly without right-shifting all the elements between gaps.

**Lemma 8.** For a given input sequence of size \( n \), the cost of all re-balancing is \( O(n) \) and the amortized re-balancing cost is \( O(1) \) per insertion (Bender et al. 2004).

Similar to the LIBRARY SORT, after the \( 2^i \)-th element insertion, the array \( A \) need to be relabeled with the cost specified in Lemma 8, but we can save the cost of normalization, i.e., if we do apply the rotated list divisions recursively, from Theorem 4, the optimal level (i.e., \( l \)) grows after the \( 2^{2l} \)-th element insertion. As a result, relabeling that includes the effect of normalization will automatically adjust the optimal recursion level. Since array relabeling is needed regularly during element insertions and relabeling does has the normalization effect, the frequency of normalization is less than the frequency of relabeling. Note that it is possible to improve the cost of rebalancing from \( O(n) \) to \( O(r) \). However, this improvement will not affect the \( O(1) \) amortized relabeling cost and it will not include the effect of normalization, we will omit its discussions here.

**Lemma 9.** It is possible to query the sum of all previous gaps before the rotated list \( L_i \) in constant worst case time and updates in \( O(r^c) \) worst case time, with only extra \( O(r \lg(\epsilon f(r))) \) bits space.

**Proof.** For simplicity, we do not consider gaps after the last element. Each rotated list \( L_i \) has at most \( \epsilon f(i) \) gaps, the largest rotated list \( L_r \) has at most \( \epsilon f(r) \) gaps. The problem is identical to the problem of partial sum with \( r \) elements with the universe \( \{1, \ldots, f(r)\} \) that Raman et al (Raman et al. 2001) solved in the above bounds.

**Lemma 10.** \( \text{select} \) can be done in \( O(1) \) with extra \( O(\sqrt{\lg n}) \) space in ROTATED LIBRARY SORT.

**Proof.** Trivial. We perform \( \text{select}(j, S) \) similar to ROTATED SORT, but we need to add the sum of all previous gaps to \( j \) using Lemma 9, which also takes \( O(1) \) time.

It is possible to avoid re-balancing on the \( 2^i \)-th element insertion. Instead of performing a sequence of soft exchanges with each soft exchange inserting a single smallest element and returning a single largest element, we can perform the soft exchange with \( \epsilon f(k) \) elements. When \( J_{n-1} \) is full after a hard exchange, we pop the largest \( \delta = \epsilon f(k) \) elements after the hard exchange and
perform the soft exchange of $\delta$ elements on the rotated lists $(J_{k+1}, \ldots, J_{r-2})$. $\delta$ will get smaller and eventually becomes zero. If we assume the elements of the input sequence are randomly distributed, $\delta$ will decrease in an increasing rate as the size of the extra arrays $K$ increases monotonically according to the function $f$. Soft exchange will then cost $O(\delta)$ operations, while the worst case cost for hard exchange remains unchanged.

**Theorem 6.** insert in ROTATED LIBRARY SORT can be done in amortized $O(\lg r + f(r))$ operations.

**Proof.** For insert in ROTATED LIBRARY SORT, hard exchanges on $J_k$ are unavoidable initially. However, the larger the $r$ is, fewer hard exchanges on $J_{r-1}$ will be required at the end. Therefore, the worst case scenario happens when insertion occurs at $J_k$ where $k = r/2$. Each insertion consists of a binary search with $O(\lg r)$ operations. The first $(\epsilon f(k) - 1)$ insertions include a hard exchange, that costs worst case $O(f(k))$ operations, because of the empty gaps. The $(\epsilon f(k))$-th insertion will incur the initial hard exchange plus a soft exchange on $J_{k+1}$ that costs $O(\epsilon f(k))$ operations. It terminates because the number of gaps in $J_{k+1}$ is greater than in $J_k$. Since $J_k$ contains $\epsilon f(k)$ gaps, the $(\epsilon f(k))$-th insertion till the $(2\epsilon f(k) - 1)$th insertion require only $O(\lg r + f(k))$ operations. Then the $(2\epsilon f(k))$-th insertion needs to perform more soft exchanges. The difference between the numbers of soft exchanges of the $(\epsilon f(k))$-th and $(\epsilon f(k) + 1)$-th insertions will increase by at most one (i.e., the difference will be either zero or one). The difference decreases until the number of soft exchanges hits its bound $r - k$. When the bound of $r - k$ soft exchanges is reached, we need the final hard exchange with worst case cost $O(f(r))$ operations. We can clearly see the pattern here, i.e., every $\epsilon f(k)$ insertions require $O(\lg r + f(k))$ operations, then followed by a single insertion that requires $O((r - k) + f(r))$ operations. From this observation, we can approximate that the amortized cost is $O(\lg r + f(k) + (r - k + f(r))/\epsilon)$. So with a large enough $\epsilon$, the insertion cost in the worst case scenario is close to amortized $O(\lg r + f(k)) \leq O(\lg r + f(r))$ operations, instead of $O(r + f(r))$ from Lemma 5, which is clearly an improvement. \hfill $\Box$

**8 Conclusions**

This paper presents an alternative approach called ROTATED INSERTION SORT to solve the high time complexity of INSERTION SORT. The approach is incremental yet adaptive, it uses less space than GAPPED INSERTION SORT (Bender et al. 2004) and does not rely on the distribution of input. It shows that the ROTATED INSERTION SORT can be done in $O(n^{1.5} \lg n)$ time with $O(w)$ temporary space, which is tight space bound; or it can be run in $O(2^kn^{1+\epsilon})$ operations, using only a lower order $\Theta(\sqrt{n} \lg n)$ bits space. This paper further shows a possible combined approach called ROTATED LIBRARY SORT.

There are several problems remain open — first, which function is the best function for the ROTATED LIBRARY SORT to virtually divide the array? Are there any other in-place, incremental and adaptive approaches that outperform ROTATED LIBRARY SORT? What are the time bound, space bound and their tradeoffs between the extra space use, member, insert and select?

**References**

Improved Inapproximability Results for the Shortest Superstring and Related Problems

Marek Karpinski¹  Richard Schmied²

Dept. of Computer Science
University of Bonn
Email: {marek, schmied}@cs.uni-bonn.de

Abstract

We develop a new method for proving explicit approximation lower bounds for the Shortest Superstring problem, the Maximum Compression problem, the Maximum Asymmetric TSP problem, the (1,2)–ATSP problem and the (1,2)–TSP problem improving on the best known approximation lower bounds for those problems.

Keywords: Inapproximability, Explicit Lower Bounds, Asymmetric Traveling Salesperson Problem, Bounded Metrics, Shortest Superstring Problem, MAX-ATSP Problem, Maximum Compression Problem

1 Introduction

In the Shortest Superstring (SSP) problem, we are given a finite set $S$ of strings and we would like to construct their shortest superstring, which is the shortest possible string such that every string in $S$ is a proper substring of it.

The task of computing a shortest common superstring appears in a wide variety of applications related to computational biology [L90]. Vasilyevska [V05] proved that approximating the SSP problem with approximation factor less than $1217_{32}$ is NP-hard. The currently best known approximation algorithm is due to Mucha [M12] and yields an approximation factor of $\frac{11}{8}$.

In this paper, we prove that the Shortest Superstring problem is NP-hard to approximate within any constant approximation factor better than $\frac{333}{32}$. In the Traveling Salesperson (TSP) problem, we are given a metric space $(V,d)$ and the task consists of constructing a shortest tour visiting each vertex exactly once.

The TSP problem in metric spaces is one of the most fundamental NP-hard optimization problems. The decision version of this problem was shown early to be NP-complete by Karp [K72]. Christofides [C76] gave an algorithm approximating the TSP problem within $3/2$, i.e., an algorithm that produces a tour with length being at most a factor $3/2$ from the optimum. As for lower bounds, a reduction due to Papadimitriou and Yannakakis [PY93] and the PCP Theorem [ALM+98] together imply that there exists some constant $c$, not less than $1 + 10^{-6}$, such that it is NP-hard to approximate the TSP problem with distances either one or two with approximation factor less than $c$. For discussion of bounded metrics TSP, see also [T00]. The best known approximation lower bound for the general version of this problem is due to Lampis [L12]. He proved that the TSP problem is NP-hard to approximate with approximation factor less than $\frac{185}{184}$. The restricted version of the TSP problem, in which the distance function takes values in $\{1,\ldots, B\}$, is referred to as the $(1,B)$–TSP problem. The $(1,2)$–TSP problem can be approximated in polynomial time with approximation factor $8/7$ due to Berman and Karpinski [BK06]. On the other hand, Engenbetzen and Karpinski [EK06] proved that it is NP-hard to approximate the $(1,B)$–TSP problem with approximation factor less than $\frac{741}{740}$ for $B = 2$ and $\frac{389}{388}$ for $B = 8$.

In this paper, we prove that it is NP-hard to approximate the $(1,2)$–TSP problem with approximation factor less than $\frac{535}{534}$.

In the Asymmetric Traveling Salesperson (ATSP) problem, we are given an asymmetric metric space $(V,d)$, i.e., $d$ is not necessarily symmetric, and we would like to construct a shortest tour visiting every vertex exactly once. The best known algorithm for the ATSP problem approximates the solution within $O(\log n / \log \log n)$, where $n$ is the number of vertices in the metric space [AGM’10]. On the other hand, Papadimitriou and Vempala [PV06] proved that the ATSP problem is NP-hard to approximate with approximation factor less than $\frac{117}{116}$. It is conceivable...

---

¹ Supported in part by DFG grants and the Hausdorff Center grant EXC59-1.
² Work supported by Hausdorff Doctoral Fellowship

Copyright ©2013, Australian Computer Society, Inc. This paper appeared at the 19th Computing: Australasian Theory Symposium (CATS 2013), Adelaide, South Australia, January-February 2013. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 141, Anthony Wirth, Ed. Reproduction for academic, not-for-profit purposes permitted provided this text is included.
that the special cases with bounded metric are easier to approximate than the cases when the distance between two points grows with the size of the instance. Clearly, the \((1, B)\)-ATSP problem, in which the distance function is taking values in the set \(\{1, \ldots, B\}\), can be approximated within \(B\) by just picking any tour as the solution. When we restrict the problem to distances one and two, it can be approximated within \(5/4\) due to Bläser [B04]. Furthermore, it is NP-hard to approximate this problem with approximation factor less than \(321/320\) [EK06]. For the case \(B = 8\), Engebretsen and Karpinski [EK06] constructed a reduction yielding the approximation lower bound \(135/134\) for the \((1, 8)\)-ATSP problem.

In this paper, we prove that it is NP-hard to approximate the \((1, 2)\)-ATSP problem with approximation factor less than \(207/206\).

In the Maximum Compression (MAX–CP) problem, we are given a collection of strings \(S = \{s_1, \ldots, s_n\}\). The task is to find a superstring for \(S\) with maximum compression, which is the difference between the sum of the lengths of the given strings and the length of the superstring.

In the exact setting, an optimal solution to the Shortest Superstring problem is an optimal solution to this problem, but the approximate solutions can differ significantly in the sense of approximation ratio. The Maximum Compression problem arises in various data compression problems (cf. [S88]). The best known approximation upper bound is \(3/2\) [KLS’05] by reducing it to the MAX–ATSP problem, which is defined below.

On the approximation lower bound side, Vasilevska [V05] proved that it is NP-hard to approximate this problem with approximation factor better than 1072/1071.

In this paper, we prove that approximating the Maximum Compression problem with approximation factor less than 204/203 is NP-hard.

In the Maximum Asymmetric Traveling Salesperson (MAX–ATSP) Problem, we are given a complete directed graph \(G\) and a weight function \(w\) assigning each edge of \(G\) a nonnegative weight. The task is to find a tour of maximum weight visiting every vertex of \(G\) exactly once.

This problem is well-known and motivated by several applications (cf. [BGS02]). A good approximation algorithm for the MAX–ATSP problem yields a good approximation algorithm for many other optimization problems such as the Shortest Superstring problem, the Maximum Compression problem and the \((1, 2)\)-ATSP problem. In particular, an \(\alpha\)-approximation algorithm for the MAX–ATSP problem implies an \(\alpha\)-approximation algorithm for the Maximum Compression problem (cf. [KLS’05]).

The MAX–(0, 1)–ATSP problem is the restricted version of the MAX–ATSP problem, in which the weight function \(w\) takes values in the set \(\{0, 1\}\). Vishwanathan [V92] constructed an approximation preserving reduction proving that any \((1/\alpha)\)-approximation algorithm for the MAX–(0, 1)–ATSP problem transforms in a \((2–\alpha)\)-approximation algorithm for the \((1, 2)\)-ATSP problem. Due to the explicit approximation lower bound for the \((1, 2)\)-ATSP problem given in [EK06], it is NP-hard to approximate the MAX–(0, 1)–ATSP problem with approximation factor less than \(320/319\).

The best known approximation algorithm for the restricted version of this problem is due to Bläser [B04] and achieves an approximation factor \(5/4\).

For the general problem, Kaplan et al. [KLS’05] designed an algorithm for the MAX–ATSP problem yielding the best known approximation upper bound of \(3/2\). Elbassioni, Paluch and v. Zuylen [EPZ12] gave a simpler approximation algorithm for the problem with the same approximation factor.

In this paper, we prove that approximating the MAX–ATSP problem with approximation factor less than 204/203 is NP-hard.

2 Preliminaries

Throughout, for \(i \in \mathbb{N}\), we use the abbreviation \([i]\) for the set \(\{1, \ldots, i\}\). Given an alphabet \(\Sigma\), a string is an element of \(\Sigma^*\). Given a string \(v\), we denote the length of \(v\) by \(|v|\). For two strings \(x\) and \(y\), we define the overlap of \(x\) and \(y\), denoted \(ov(x, y)\), as the longest suffix of \(x\) that is also a prefix of \(y\). Furthermore, we define the prefix of \(x\) with respect to \(y\), denoted \(\text{pref}(x, y)\), as the string \(u\) with \(x = u \text{ ov}(x, y)\).

In this paper, an instance \((V, d)\) of the \((1, 2)\)-ATSP problem is specified by means of a directed graph \(D_V = (V, A)\), where \((x, y) \in A\) if and only if \(d(x, y) = 1\). In addition, we refer to an arc \((x, y) \in V \times V\) as a \(z\)-arc if \(d(x, y) = z \in \{1, 2\}\).

In order to specify an instance of the \((1, 2)\)-TSP problem, we will use undirected graphs.

3 Hybrid Problem

Berman and Karpinski [BK99] introduced the following Hybrid problem and proved that this problem is NP-hard to approximate with some constant.

Definition 3.1 (Hybrid problem). Given a system of linear equations mod 2 containing \(n\) variables, \(m_2\) equations with exactly two variables, and \(m_3\) equations with exactly three variables, find an assignment to the variables that satisfies as many equations as possible.

The following result is due to Berman and Karpinski [BK99].

Theorem 3.1 ([BK99]). For every constant \(\delta \in (0, 1/2)\), there exists instances of the Hybrid problem \(H(\nu)\) with \(4\nu\) variables, \(60\nu\) equations with exactly two variables, and \(2\nu\) equations with exactly three variables such that: (i) Each variable occurs exactly three times. (ii) Either there is an assignment to the variables that leaves at most \(\delta \cdot \nu\) equations unsatisfied, or else every assignment to the variables leaves at least \((1 – \delta)\nu\) equations unsatisfied. (iii) It is NP-hard to decide which of the two cases in item (ii) above holds. (iv) An optimal assignment to the
variables in $H(\nu)$ can be transformed in polynomial time into an optimal assignment satisfying all $60\nu$ equations with two variables in $H(\nu)$.

The instances of the Hybrid problem produced in Theorem 3.1 have an even more special structure, which we are going to describe. The equations containing three variables are of the form $x \oplus y \oplus z = \{0, 1\}$. These equations stem from the Theorem of Håstad [H01] dealing with the hardness of approximating equations with exactly three variables. We refer to it as the MAX-E3-LIN problem, which can be seen as a special instance of the Hybrid problem.

**Theorem 3.2 ([H01]).** For every constant $\delta \in (0, 1/2)$, there exists systems of linear equations mod 2 with $2\nu$ equations and exactly three unknowns in each equation such that:

(i) Each variable in the instance occurs a constant number of times, half of them negated and half of them unnegated.

(ii) Either there is an assignment satisfying all but

$2 \delta$ $\nu$ equations, or every assignment leaves at least $1 - (1 - \delta)\nu$ equations unsatisfied.

(iii) It is $\text{NP}$-hard to distinguish between these two cases.

Let us describe briefly the reduction from the MAX-E3-LIN problem to the Hybrid problem. For a detailed description, we refer to [BK99], [BK01] and [K01]. For every variable $x$ of the original instance $I$ of the MAX-E3-LIN problem, we introduce a corresponding set of variables $V_x$. If the variable $x$ occurs $t_x$ times in $I$, then, $V_x$ contains $n = 7t_x$ new variables $x_1, \ldots, x_n$. The variables contained in $\{x_i \mid i \in \{t_x\}\}$ are called contact variables, whereas the remaining variables in $V_x$ are called checker variables. All variables in $V_x$ are connected by equations of the form $x_i \oplus x_{i+1} = 0$ with $i \in [n-1]$ (circle equations) and $x_1 \oplus x_n = 0$ (circle border equation). In addition, there exists equations of the form $x_i \oplus x_j = 0$ with $(i, j) \in M_x$ (matching equations), where the set $M_x$ induces a perfect matching on the index set of checker variables. In the remainder, we refer to this construction as the circle $C_x$ containing the variables $x_1, \ldots, x_n$. For each variable $x$ in $I$, we replace the $t_x$ occurrences of $x$ in equations with three variables in $I$ by the $t_x$ contact variables in $V_x$ and add all equations with two variables associated with $C_x$. Accordingly, every variable in the corresponding instance $I_H$ of the Hybrid problem occurs exactly three times.

4 Our Results

We now formulate our results.

**Theorem 4.1.** Let $H$ be an instance of the Hybrid problem with $n$ circles, $60\nu$ equations with two variables and $2\nu$ equations with exactly three variables satisfying the properties described in Theorem 3.1. Then:

(i) If there exists an assignment $\phi$ to the variables of $H$ which leaves at most $\delta \nu$ equations unsatisfied for some $\delta \in (0, 1)$, then, there exists a tour with length at most $3 \cdot 60\nu + 13 \cdot 2\nu + \nu + 1 + \delta \nu$.

(ii) From every tour in $(V_H, d_H)$ with length $206 \cdot \nu + n + 1 + \delta \nu$, we can construct in polynomial time an assignment that leaves at most $\delta \cdot \nu$ equations in $H$ unsatisfied.

2. It is possible to construct in polynomial time an instance $(V_H, d_H)$ of the $(1, 2)$-TSP problem such that:

(i) If there exists an assignment $\phi$ to the variables of $H$ which leaves at most $\delta \nu$ equations unsatisfied for some $\delta \in (0, 1)$, then, there exists a tour with length at most $8 \cdot 60\nu + 27 \cdot 2\nu + 3\nu(n+1) + 1 + \delta \nu$.

(ii) From every tour $\sigma$ in $(V_H, d_H)$ with length $534 \cdot \nu + 3\nu(n+1) + 1 + \delta \nu$, we can construct in polynomial time an assignment that leaves at most $\delta \cdot \nu$ equations in $H$ unsatisfied.

3. It is possible to construct in polynomial time an instance $S_H$ of the Shortest Superstring problem such that:

(i) If there exists an assignment $\phi$ to the variables of $H$ which leaves at most $\delta \nu$ equations unsatisfied for some $\delta \in (0, 1)$, then, there exists a superstring $s_\phi$ for $S_H$ with length $5 \cdot 60\nu + 16 \cdot 2\nu + 7\nu + 60\nu$.

(ii) From every superstring $s$ for $S_H$ with length $332 \cdot \nu + 7\nu + 60\nu$, we can construct in polynomial time an assignment to the variables of $H$ that leaves at most $\delta \cdot \nu$ equations in $H$ unsatisfied.

4. It is possible to construct in polynomial time an instance $S_H$ of the Maximum Compression problem such that:

(i) If there exists an assignment $\phi$ to the variables of $H$ which leaves at most $\delta \nu$ equation unsatisfied for some $\delta \in (0, 1)$, then, there exists a superstring $s_\phi$ for $S_H$ with compression at least $3 \cdot 60\nu + 12 \cdot 2\nu + 5\nu - \delta \nu$.

(ii) From every superstring $s$ for $S_H$ with compression $204\nu + 5\nu - \delta \nu$, we can construct in polynomial time an assignment to the variables of $H$ that leaves at most $\delta \cdot \nu$ equations in $H$ unsatisfied.

The former theorem can be used to derive an explicit approximation lower bound for the $(1, 2)$-ATSP problem.

**Corollary 4.1.** For every $\epsilon > 0$, it is $\text{NP}$-hard to approximate the $(1, 2)$-ATSP problem within any constant approximation ratio better than $207/206 - \epsilon$.

**Proof.** First of all, we choose $k \in \mathbb{N}$ and $\delta > 0$ such that $\frac{207^k - 1}{207^k} \geq 207^{1/2} - \epsilon$ holds. Given an instance $E_3$ of the MAX-E3-LIN problem, we generate another instance $E_3^k$ consisting of $k$ copies of $E_3$. This construction ensures that the number of occurrences of a variable is bounded from below by $k$ without introducing new variables. Then, we construct the corresponding instance $H$ of the Hybrid problem and $(V_H, d_H)$ of the $(1, 2)$-ATSP problem with the properties described in Theorem 4.1.1. We conclude according to Theorem 3.1 that there exist a tour in $(V_H, d_H)$ with length at most $206kv + 6kv + (n+1) \leq (206 + 1 - \delta)kv \leq (206 + \delta + \frac{4\delta}{1+\delta})kv$ or the length of a tour in $(V_H, d_H)$ is bounded from below by $206kv + (1 - \delta)kv + n + 1 \geq (206 + (1 - \delta)kv \geq (207 - \delta)kv$. From Theorem 3.1, we know that the two cases above are $\text{NP}$-hard to distinguish.
Hence, for every $\epsilon > 0$, it is NP-hard to find a solution to the Shortest Superstring problem with an approximation ratio $\frac{207 - \delta}{206 + \delta + 1227} \geq \frac{207}{206} - \epsilon$.

Analogously, Theorem 4.1 can be used to derive approximation lower bounds for the other problems summarized in Figure 1. The explicit approximation lower bound for the Max–ATSP problem is obtained by using a well-known approximation preserving reduction from the Maximum Compression problem to the Max–ATSP problem (cf. [KLS’05]).

<table>
<thead>
<tr>
<th>Problem</th>
<th>Our Results</th>
<th>Previously known</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, 2)–ATSP</td>
<td>207/206</td>
<td>321/321 [EK06]</td>
</tr>
<tr>
<td>(1, 2)–TSP</td>
<td>535/534</td>
<td>741/740 [EK06]</td>
</tr>
<tr>
<td>MAX–ATSP</td>
<td>204/203</td>
<td>320/319 [EK06]</td>
</tr>
<tr>
<td>MAX–CP</td>
<td>204/203</td>
<td>1072/1071 [V05]</td>
</tr>
<tr>
<td>SSP</td>
<td>333/332</td>
<td>1217/1216 [V05]</td>
</tr>
</tbody>
</table>

Figure 1: Comparison of our results to previously known explicit approximation lower bounds.

For other details and explicit approximation lower bounds for related problems, see [KS11] and [KS12].

5 The (1, 2)–ATSP problem

Given an instance of the Hybrid problem $\mathcal{H}$, we want to transform $\mathcal{H}$ into an instance of the (1, 2)–ATSP problem. Fortunately, the special structure of the linear equations in the Hybrid problem is particularly well-suited for our reduction, since a part of the equations with two variables form a cycle and every variable occurs exactly three times. The main idea of our reduction is to make use of the special structure of the circles in $\mathcal{H}$. Every cycle $C_l$ in $\mathcal{H}$ corresponds to a subgraph $D_l$ in the instance $D_\mathcal{H}$ of the (1, 2)–ATSP problem. Moreover, $D_l$ forms almost a cycle. An assignment to the variable $x_l$ will have a natural interpretation in this reduction. The parity of $x_l$ corresponds to the direction of movement in $D_l$ of the underlying tour. The circle graphs $D_1, \ldots, D_n$ of $D_\mathcal{H}$ are connected and build together the inner loop of $D_\mathcal{H}$ (Figure 2). Every variable $x_l$ in a circle $C_l$ possesses an associated parity graph $P_{e_l}$ (Figure 3(a)) in $D_l$ as a subgraph. The two natural ways to traverse a parity graph will be called 0/1-traversals (Figure 3(b) & (c)) and correspond to the parity of the variable $x_l$. Some of the parity graphs in $D_l$ are also contained in graphs $D_\mathcal{H}$ (Figure 5 and Figure 6 for a more detailed view) corresponding to equations with three variables of the form $g^3 = x \oplus y \oplus z = 0$. We may assume that equations with three variables are of the form $x \oplus y \oplus z = 0$ or $\overline{x} \oplus y \oplus z = 0$ due to the transformation $\overline{x} \oplus y \oplus z = 0 \equiv x \oplus y \oplus z = 1$. These graphs are connected and build the outer loop of $D_\mathcal{H}$. The outer loop of the tour checks whether the 0/1-traversals of the parity graphs correspond to a satisfying assignment of the equations with three variables. If an underlying equation is not satisfied by the assignment defined via 0/1-traversals of the associated parity graphs, it will be punished by using a costly arc with distance 2.

Figure 2: An illustration of $D_\mathcal{H}$ and a tour in $D_\mathcal{H}$.

Figure 3: Traversals of the parity graph $P_{e_l}$. Traversed arcs are illustrated by thick arrows.

Constructing $D_\mathcal{H}$ from the Instance $\mathcal{H}$

Given an instance of the Hybrid problem $\mathcal{H}$, we are going to construct the corresponding instance $D_\mathcal{H}$ of the (1, 2)-ATSP problem. For every type of equation in $\mathcal{H}$, we will introduce a specific graph or a specific way to connect the so far constructed subgraphs. In particular, we will distinguish between graphs corresponding to circle equations, matching equations, circle border equations and equations with three variables. First of all, we introduce graphs corresponding to the variables in $\mathcal{H}$.

Variable Graphs: For every variable $x_l$ in $\mathcal{H}$, we introduce the parity graph $P_{e_l}$ consisting of the vertices $\{v_{i_l}, v_{i_l+1}, v_{i_l+2}\}$ and is displayed in Figure 3(a).

Matching and Circle Equations: Let $\mathcal{H}$ be an instance of the hybrid problem, $C_l$ a circle in $\mathcal{H}$ and $M_i$ the associated perfect matching. Furthermore, let $x_l \oplus x_j = 0$ with $e = \{i, j\} \in M_i$ and $i < j$ be a matching equation. Due to the construction of $\mathcal{H}$, the circle equations $x_l \oplus x_{j+1} = 0$ and $x_{i+1} \oplus x_j = 0$ are both contained in $C_l$. Then, we introduce the associated parity graph $P_{e_l}$ consisting of the
Let \( x \) be an equation with three variables in \( H \). We introduce the vertex \( x_1, x_{i+1} \) and the parity graph \( P^i_1, P^i_{i+1}, P^j_1, P^j_{j+1} \) and \( P^k \) as depicted in Figure 4.

**Equations with Three Variables:** Let \( g^3_1 \equiv x^3_1 \oplus x^3_i \oplus x^3_k = 0 \) be an equation with three variables in \( H \). Then, we introduce the graph \( D^3 \) (Figure 5) corresponding to the equation \( g^3_1 \). The graph \( D^3 \) includes the vertices \( s_1, v_i^1, v_i^2, v_i^3, s_{i+1} \). Engerbretsen and Karpinski [EK06] used this graph in their reduction and proved the following statement.

**Proposition 5.1 ([EK06]).** There is a Hamiltonian path from \( s_1 \) to \( s_{i+1} \) in the graph displayed in Figure 5 if and only if an even number of dashed arcs is traversed.

This construction is extended by replacing the dashed arcs with the parity graphs \( P^i, P^k_1, P^j \) and \( P^k_1 \), where \( e = \{ i, i+1 \}, b = \{ j, j+1 \} \) and \( a = \{ t, t+1 \} \). In Figure 6, we display \( D^3_1 \) with its connections to the graph corresponding to \( x^2_1 \oplus x^2_i \oplus x^2_k = 0 \). (In case of \( g^3_1 \equiv x^3_1 \oplus x^3_i \oplus x^3_k = 0 \), we create \((v^1_1, v^1_{i+1}), (v^0_1, v^0_{i+1}), (v^1_i, v^1_k)\) and \((v^0_i, v^0_k)\) instead.)

**Figure 5:** Gadget for \( x \oplus y \oplus z = 0 \).

**Figure 6:** The graph \( D^3 \) corresponding to \( g^3_1 \equiv x^3_1 \oplus x^3_i \oplus x^3_k = 0 \) connected to graphs corresponding to \( x^2_1 \oplus x^2_i \oplus x^2_k = 0 \).

**Circle Border Equations:** Let \( C_t \) and \( C_{t+1} \) be circles in \( H \). In addition, let \( x^1_1 \oplus x^1_n = 0 \) be the circle border equation of \( C_t \). Recall that \( x^1_1 \) also occurs in an equation \( g^3_1 \) with three variables in \( H \).

Assuming \( g^3_1 \equiv x^3_1 \oplus y \oplus z = 0 \), we introduce the vertex \( b_1, b_{i+1} \) and the parity graph \( P^i_{(n,1)}, P^j_{(n,1)} \). Then, we create \((b_1, v^0_{(n,1)}), (v^0_{(n,1)}, b_{n+1}), (v^1_{(n,1)}, b_{i+1}) \) and \((v^1_{(n,1)}, b_{i+1}) \). (In case of \( g^3_1 \equiv x^3_1 \oplus y \oplus z = 0 \), we add \((b_1, v^0_{(n,1)}), (v^0_{(n,1)}, b_{n+1}), (v^1_{(n,1)}, v^0_{(n,1)}) \) and \((v^1_{(n,1)}, b_{i+1}) \). Finally, we set \( b_{n+1} = s_1 \), where \( s_1 \) is the starting vertex of \( D^3_1 \).

**Constructing a Tour from an Assignment**

Let \( H \) be an instance of the Hybrid problem consisting of the circles \( C_1, C_2, \ldots, C_m, 60\nu \) equations with 2 variables and 2\( \nu \) equations with three variables. Given an assignment \( \phi \) to the variables of \( H \) leaving \( \delta \cdot \nu \) equations unsatisfied for a constant \( \delta \in (0, 1) \), we are going to construct the associated Hamiltonian tour \( \sigma_\phi \) in \( D_H \).

According to Theorem 3.1, we may assume that all equations with 2 variables in \( H \) are satisfied by \( \phi \). Thus, all variables associated to a circle have the same value. Then, the Hamiltonian tour \( \sigma_\phi \) starts at the vertex \( b_1 \). From a high-level view, \( \sigma_\phi \) traverses all graphs corresponding to the equations with the circle \( C_1 \) using the \( \phi(x^1_1) \)-traversal of all parity graphs corresponding to circle equations of \( C_1 \), until it reaches \( b_2 \). Then, \( \sigma \) restarts at \( b_2 \) and visits all vertices associated to \( C_2 \). Successively, it passes all graphs for each circle in \( H \) until it reaches the vertex \( b_{n+1} = s_1 \) as \( s_1 \) is the starting vertex of the graph \( D^3_1 \).

At this point, the tour begins to traverse the remaining graphs \( D^3_1 \), which are simulating the equations with three variables in \( H \). By now, some of the parity graphs appearing in graphs \( D^3_1 \) already have been traversed in the inner loop of \( \sigma_\phi \). The outer loop checks whether for each graph \( D^3_1 \), an even number of parity graphs has been traversed in the inner loop. In every situation, in which \( \phi \) does not satisfy the underlying equation, the tour needs to use a 2-arc.

**Constructing an Assignment from a Tour**

Let \( H \) be an instance of the Hybrid problem, \( D_H = (V_H, A_H) \) the associated instance of the \((1, 2)\)-ATSP problem and \( \sigma \) a tour in \( D_H \). We are going to define the corresponding assignment \( \psi_\sigma \) to the variables in \( H \). In addition, we establish a connection between the length of \( \sigma \) and the number of satisfied equations by \( \psi_\sigma \). First of all, we introduce the notion of consistent tours.

**Definition 5.1 (Consistent Tour).** Let \( H \) be an instance of the Hybrid problem and \( D_H \) the associated instance of the \((1, 2)\)-ATSP problem. A tour in \( D_H \) is called consistent if every parity graph in \( D_H \) is traversed by means of its corresponding 0/1-traversal.

Due to the following proposition, we may assume that the underlying tour is consistent.

**Proposition 5.2.** Let \( H \) be an instance of the Hybrid problem and \( D_H \) the associated instance of the \((1, 2)\)-ATSP problem. Any tour \( \sigma \) in \( D_H \) can be transformed in polynomial time into a consistent tour with at most the same length as \( \sigma \).
Proof. For every parity graph contained in $D_H$, it can be seen by considering all possibilities exhaustively that any tour in $D_H$ that is not using the corresponding 0/1-traversal can be modified into a tour with at most the same number of 2-arcs. The less obvious cases are shown in the full version [KS12].

Let us define the corresponding assignment $\psi_\sigma$ given a tour $\sigma$ in $D_H$.

Definition 5.2 (Assignment $\psi_\sigma$). Let $H$ be an instance of the Hybrid problem, $D_H = (V_H, A_H)$ the associated instance of the (1,2)-ATSP problem. Given a consistent tour $\sigma$ in $D_H$, the corresponding assignment $\psi_\sigma$ is defined as $\psi_\sigma(x_i^1) = 1$ if $\sigma$ uses a 1-traversal of $P_i^1$, and 0 otherwise.

Let us start with the analysis. In the remainder, we assume that the underlying tour is consistent.

Matching Equations: Given the equations

$$x_i \oplus x_{i+1} = 0,$$

$$x_i \oplus x_j = 0,$$

$$x_i \oplus x_{j+1} = 0$$

and a tour $\sigma$, we are going to analyze the relation between the length of the tour and the number of satisfied equations by $\psi_\sigma$.

1. Case $\psi_\sigma(x_i) \oplus \psi_\sigma(x_{i+1}) = 0$, $\psi_\sigma(x_i) \oplus \psi_\sigma(x_j) = 0$ & $\psi_\sigma(x_j) \oplus \psi_\sigma(x_{j+1}) = 0$: Given $\psi_\sigma(x_i) = \psi_\sigma(x_{i+1}) = \psi_\sigma(x_j) = \psi_\sigma(x_{j+1}) = 1$, the cost of a tour traversing this part of $D_H$ can be bounded from below by 5. In this case, $\sigma$ contains $(u^1i^1, v^0j^0)$, $(v^1j^1, v^0i^0)$, $(v^2j^2, v^0i^0)$, $(v^3j^3, v^0i^0)$, and $(u^1i^1, v^0j^0)$. The case $\psi_\sigma(x_i) \psi_\sigma(x_{i+1}) = \psi_\sigma(x_j) = \psi_\sigma(x_{j+1}) = 0$ can be discussed analogously. In both cases, we obtain the local length 5 for this part of $\sigma$ while $\psi_\sigma$ satisfies all 3 equations.

Figure 7: 5 Case with $\psi_\sigma(x_i) = \psi_\sigma(x_{i+1}) = 1$ and $\psi_\sigma(x_j) \neq \psi_\sigma(x_{j+1}) = 1$.

2. Case $\psi_\sigma(x_i) \oplus \psi_\sigma(x_{i+1}) = 0$, $\psi_\sigma(x_i) \oplus \psi_\sigma(x_j) = 1$ & $\psi_\sigma(x_j) \oplus \psi_\sigma(x_{j+1}) = 0$: In both cases, we associate only the cost of one 2-arc yielding a lower bound of 6 on the local length, which corresponds to the fact that $\psi_\sigma$ leaves the equation $x_i \oplus x_j = 0$ unsatisfied. Note that a similar situation holds in case of $\psi_\sigma(x_i) = \psi_\sigma(x_{i+1}) = 0$ and $\psi_\sigma(x_j) = \psi_\sigma(x_{j+1}) = 1$.

3. Case $\psi_\sigma(x_i) \oplus \psi_\sigma(x_{i+1}) = 0$, $\psi_\sigma(x_i) \oplus \psi_\sigma(x_j) = 0 \& \psi_\sigma(x_j) \oplus \psi_\sigma(x_{j+1}) = 1$: Given $\psi_\sigma(x_i) = \psi_\sigma(x_{i+1}) = 1$ and $\psi_\sigma(x_j) \neq \psi_\sigma(x_{j+1}) = 0$, we are forced to use two 2-arcs increasing the cost by 2. Thus, we obtain a lower bound of 4 + 2. The case $\psi_\sigma(x_i) = \psi_\sigma(x_{i+1}) = 0$ and $\psi_\sigma(x_j) \neq \psi_\sigma(x_{j+1}) = 1$ can be analyzed analogously. A similar argumentation holds for $\psi_\sigma(x_i) \oplus \psi_\sigma(x_{i+1}) = 1$, $\psi_\sigma(x_i) \neq \psi_\sigma(x_j) = 0$ and $\psi_\sigma(x_j) \oplus \psi_\sigma(x_{j+1}) = 0$.

4. Case $\psi_\sigma(x_i) \oplus \psi_\sigma(x_{i+1}) = 1$, $\psi_\sigma(x_i) \oplus \psi_\sigma(x_j) = 0 \& \psi_\sigma(x_j) \oplus \psi_\sigma(x_{j+1}) = 1$: Given $\psi_\sigma(x_i) \neq \psi_\sigma(x_{i+1}) = 0$ and $\psi_\sigma(x_j) \neq \psi_\sigma(x_{j+1}) = 0$, we are forced to use four 2-arcs in order to connect all vertices. Consequently, it yields the lower bound of 7. The case, in which $\psi_\sigma(x_i) \neq \psi_\sigma(x_{i+1}) = 0$ and $\psi_\sigma(x_j) \neq \psi_\sigma(x_{j+1}) = 0$ holds, can be discussed analogously.

5. Case $\psi_\sigma(x_i) \oplus \psi_\sigma(x_{i+1}) = 0$, $\psi_\sigma(x_i) \oplus \psi_\sigma(x_j) = 1 \& \psi_\sigma(x_j) \oplus \psi_\sigma(x_{j+1}) = 1$: Let the tour $\sigma$ be characterized by $\psi_\sigma(x_i) = \psi_\sigma(x_{i+1}) = 1$ and $\psi_\sigma(x_j) \neq \psi_\sigma(x_{j+1}) = 0$. Let us assume that $\sigma$ uses the arc $(u^1i^1, v^0j^0)$. The corresponding situation is illustrated in Figure 7(a). We transform $\sigma$ such that it traverses the parity graph $P_i^1$ in the other direction and obtain $\psi_\sigma(x_i) = 1$. This transformation induces a tour with at most the same cost. On the other hand, the corresponding assignment $\psi_\sigma$ satisfies at least 2 - 1 = 1 more equations since $x^1_i \oplus x^1_{i+1} = 0$ might get unsatisfied. In this case, we associate the local costs of 6 with $\sigma$. In the other cases, in which $\psi_\sigma(x_i) = \psi_\sigma(x_{i+1}) = 0 \& \psi_\sigma(x_j) \neq \psi_\sigma(x_{j+1}) = 0$ or $\psi_\sigma(x_i) \oplus \psi_\sigma(x_{j+1}) = 1$, $\psi_\sigma(x_i) \neq \psi_\sigma(x_j) = 0 \& \psi_\sigma(x_j) \oplus \psi_\sigma(x_{j+1}) = 0$, we may argue similarly.

6. Case $\psi_\sigma(x_i) \oplus \psi_\sigma(x_{i+1}) = 1$, $\psi_\sigma(x_i) \oplus \psi_\sigma(x_j) = 1$ and $\psi_\sigma(x_j) \oplus \psi_\sigma(x_{j+1}) = 1$: Given a tour $\sigma$ with $\psi_\sigma(x_i) \neq \psi_\sigma(x_{i+1}) = 1$ and $\psi_\sigma(x_j) \neq \psi_\sigma(x_{j+1}) = 1$, we transform $\sigma$ such that it traverses the parity graph $P^1_i$ in the opposite direction meaning $\psi_\sigma(x_j) = 0$. This transformation enables us to use the arc $(v^0j^0, u^1i^1)$. Furthermore, it yields at least one more satisfied equation in $H$. In order to connect the remaining vertices, we are forced to use at least two 2-arcs. In summary, we associate the local length 7 with this situation in conformity with the at most 2 unsatisfied equations by $\psi_\sigma$. The case, in which $\psi_\sigma(x_i) \neq \psi_\sigma(x_{i+1}) = 0$ & $\psi_\sigma(x_j) \neq \psi_\sigma(x_{j+1}) = 1$ holds, can be discussed analogously.

In summary, we obtain the following statement.

Proposition 5.3. Let $E = \{x^1_i \oplus x^1_{i+1} = 0, x^1_i \oplus x^2_j = 0, x^1_j \oplus x^1_{j+1} = 0\}$ be a subset of $H$ with $\{i,j\} \in M$. Then, it is possible to transform in polynomial time a given tour $\sigma$ passing through the graphs corresponding to $g \in E$ into a tour $\pi$ that has local cost $(5 + \alpha)$ and the number of unsatisfied equations in $E$ by $\psi_\sigma$ is at most $\alpha$.
Equations with Three Variables: Let \( q^i_j \equiv x^i_j \oplus x^i_j \oplus x^i_j \) be an equation with three variables in \( H \). Furthermore, let \( C_i \) be a circle in \( H \) and \( x^i_j \oplus x^i_{j+1} \oplus x^i_j = 0 \) a circle equation. For notational simplicity, we set \( e = \{ i, i+1 \} \). We are going to analyze the number of satisfied equations by \( \psi_\sigma \) in dependence to the local length of \( \sigma \) in the graphs \( P^M, P^{e_1}, P^e \) and \( D^3 \). First, we transform the tour traversing the graphs \( P^M, P^{e_1}, P^e \) and \( D^3 \) such that it uses the \( \psi_\sigma \)-traversal of \( P^e \). Afterwards, due to the construction of \( D^3 \) and Proposition 5.1, the tour can be transformed such that it has local length of \( 3 \cdot 3 + 4 \) if it passes an even number of parity graphs \( P \in \{ P^M, P^{e_1}, P^e \} \). Using a simple path through \( D^3 \). Otherwise, it yields a local length of \( 13 + 1 \).

Let us start to analyze the local cost of \( \sigma \) in the graph corresponding to \( x^i_1 \oplus x^i_{i+1} = 0 \):

1. **Case \( (\psi_\sigma(x^i_1) \oplus \psi_\sigma(x^i_{i+1}) = 0) \):** In both cases, we transform the tour such that it uses the \( \psi_\sigma \)-traversal of \( P^e \) without increasing its length. Exemplary, we display such a scenario for the case \( (\psi_\sigma(x^i_1) = 1 \land \psi_\sigma(x^i_{i+1}) = 1) \) in Figure 8(a) and (b) (transformed tour in Figure 8(b)). For both cases, we associate a lower bound of 1 on the local cost.

2. **Case \( (\psi_\sigma(x^i_1) = 1 \land \psi_\sigma(x^i_{i+1}) = 0) \):** Let us assume that \( \psi_\sigma(x^i_1) \oplus \psi_\sigma(x^i_1) \oplus \psi_\sigma(x^i_{i+1}) = 0 \) holds. Due to Proposition 5.1, it is possible to transform the tour such that it uses the \( 0 \)-traversal of the parity graph \( P^e \) without increasing the length. In the other case, i.e. \( \psi_\sigma(x^i_1) \oplus \psi_\sigma(x^i_{i+1}) \oplus \psi_\sigma(x^i_{i+1}) = 1 \), we will change the value of \( \psi_\sigma(x^i_1) \) achieving in this way at least \( 2 - 1 \) more satisfied equation. Let us examine the scenario and the corresponding transformation in Figure 9(a) and (b), respectively. Accordingly, the tour uses the \( 0 \)-traversal of the parity graph \( P^e \), which enables \( \sigma \) to pass the parity check in \( D^3 \). In both cases, we obtain the local length of 2 in conformity with the at most one unsatisfied equation by \( \psi_\sigma \).

3. **Case \( (\psi_\sigma(x^i_1) = 0 \land \psi_\sigma(x^i_{i+1}) = 1) \):** Assuming \( \psi_\sigma(x^i_1) \oplus \psi_\sigma(x^i_{i+1}) \oplus \psi_\sigma(x^i_{i+1}) = 0 \), the tour will be modified such that the parity graphs \( P^M \) and \( P^e \) are traversed in the same direction. Since we have \( \psi_\sigma(x^i_1) \oplus \psi_\sigma(x^i_{i+1}) \oplus \psi_\sigma(x^i_{i+1}) = 0 \), we are able to uncouple the parity graph \( P^e \) from the tour \( \sigma \) through \( D^3 \) without increasing the length of \( \sigma \). Assuming \( \psi_\sigma(x^i_1) \oplus \psi_\sigma(x^i_{i+1}) \oplus \psi_\sigma(x^i_{i+1}) = 1 \), we transform \( \sigma \) such that the parity graph \( P^e \) is traversed when \( \sigma \) is passing through \( D^3 \) meaning \( v_c^3 \rightarrow v_{c+1}^0 \rightarrow v_{c+1}^1 \rightarrow v_{c+1}^2 \rightarrow v_c^3 \) is a part of the tour. In addition, we change the value of \( \psi_\sigma(x^i_1) \) yielding at least \( 2 - 1 \) more satisfied equations. In both cases, we associate the local length of 2 with \( \sigma \). On the other hand, \( \psi_\sigma \) leaves at most one equation unsatisfied.

The construction for \( x^i_1 \oplus x^i_{i+1} = 0 \) and \( x^i_1 \oplus x^i_{i+1} = 0 \) can be analyzed analogously yielding the following statement.

**Proposition 5.4.** Let \( E = \{ x^i_1 \oplus x^i_1 \oplus x^i_k = 0, x^i_1 \oplus x^i_{i+1} = 0, x^i_1 \oplus x^i_{i+1} = 0, x^i_1 \oplus x^i_{i+1} = 0 \} \) be a subset of \( H \). Then, it is possible to transform in polynomial time a given tour \( \sigma \) passing through the graph corresponding to \( g \in E \) into a tour \( \tau \) that has local length \( (4 + 3 + 3 + 3 + \alpha) \) and the number of unsatisfied equations in \( E \) by \( \psi_\tau \) is at most \( \alpha \).

The construction for circle border equations can be analyzed similarly to the the construction for equations with three variables. We obtain the following statement.

**Proposition 5.5.** Let \( x^i_1 \oplus x^i_n = 0 \) be a circle border in \( H \). Then, it is possible to transform in polynomial time a given tour \( \sigma \) passing through the graph corresponding to \( x^i_1 \oplus x^i_n = 0 \) into a tour \( \tau \) that has local length at least 2 if \( x^i_1 \oplus x^i_n = 0 \) is satisfied by \( \psi_\tau \), and at least 3 otherwise.

Thus far, we are ready to prove Theorem 4.1.1.

**Proof of Theorem 4.1.1.** Let \( H \) be an instance of the Hybrid problem consisting of \( n \) circles, \( 60 \nu \) equations with two variables and \( 2 \nu \) equations with three variables. Then, we construct in polynomial time the corresponding instance \( D_H \) of the (1,2)-ATSP problem.

(i) Let \( \phi \) be an assignment to the variables in \( H \) leaving \( 0 \nu \) equations in \( H \) unsatisfied for a constant \( \delta \in (0.1) \). Then, it is possible to construct in polynomial time a tour \( \tau \) with length at most \( 3 \cdot 60 \nu + (4 + 3 \cdot 3) \cdot 2 \nu + n + 1 + \delta \nu \).
Let $\sigma$ be a tour in $D_H$ with length $206\nu + n + 1 + \delta\nu$. Due to Proposition 5.2 we may assume that $\sigma$ uses only $0/1$-traversals of every parity graph included in $D_H$. According to Definition 5.2, we associate the corresponding assignment $\psi_\sigma$ with the underlying tour $\sigma$. Recall from Proposition 5.3 – 5.5 that it is possible to convert $\sigma$ in polynomial time into a tour $\pi$ without increasing the length such that $\psi_\pi$ leaves at most $\delta\nu$ equations in $H$ unsatisfied. 

6 The $(1,2)$-TSP Problem

In order to prove Theorem 4.1.2, we apply the reduction method used in the previous section to the $(1,2)$–TSP problem. As for parity gadget, we use the graph displayed in Figure 10 with its corresponding traversals. The traversed edges are illustrated by thick lines. Let $H$ be an instance of the hybrid problem and $x_1 \oplus x_2 = 0$ a contained matching equation. Let $x_i \oplus x_{i+1} = 0$ and $x_i \oplus x_{i+1} = 0$ be the corresponding circle equations. Then, we connect the associated parity graphs $P_1, P_1', P_{i+1}$, $P_j$, and $P_j'$ as displayed in Figure 11. For equations with three variables $y_1 \oplus x \oplus z = 0$ in $H$, we use the graph $G_3^y$ depicted in Figure 12. For this graph, Engebretsen and Karpinski [EK06] proved the following statement.

**Proposition 6.1 ([EK06]).** There is a simple path from $s_i$ to $s_{i+1}$ in Figure 12 containing $v_i^1$ and $v_i^2$ if and only if an even number of parity graphs is traversed.

Let $C_i$ and $C_{i+1}$ be circles in $H$. Let $x_1, \ldots, x_n$ be the variables contained in $C_i$. For the circle border equation of $C_i$, we introduce the path $p_i = b_i^1 - b_i^2 + b_i^1$ and the parity graph $P_{i,1}^l$. In addition, we connect $b_i^1$ and $b_{i+1}^1$ to the parity graphs $P_{i}^l$, $P_i$ and $P_{i,1}^l$ in a similar way as in the reduction from the Hybrid problem to the $(1,2)$–ATSP problem. Let $C_n$ be the last circle in $H$. Then, we introduce the path $p_{n+1} = b_{n+1}^1 - b_{n+1}^2 - s_1$, where $s_1$ is a vertex of the graph $G_3^k$ associated to the equation $y_1^k$ with three variables in $H$. This is the whole description of the corresponding graph $G_{3,H}$.

![Figure 12: Graph $G_3^y$ corresponding to $x \oplus y \oplus z = 0$.](image)

We are ready to give the proof of Theorem 4.1.2.

**Proof of Theorem 4.1.2.** Given $H$ an instance of the Hybrid problem consisting of $n$ circles, $60\nu$ equations with two variables and $2\nu$ equations with three variables, we construct in polynomial time the associated instance $G_H$ of the $(1,2)$–TSP problem.

(i) Given an assignment $\phi$ to the variables of $H$ leaving $\delta\nu$ equations unsatisfied in $H$ for a constant $\delta \in (0, 1)$, then, there is a tour in $G_H$ with length at most $8 \cdot 60\nu \cdot (3 \cdot 8 + 3) \cdot 2\nu \cdot 3 \cdot (n + 1) + 1 + \delta\nu$.

(ii) On the other hand, if we are given a tour $\sigma$ in $G_H$ with length $534\nu \cdot 3(n + 1) + 1 + \delta\nu$, it is possible to transform $\sigma$ in polynomial time into a tour $\sigma'$ such that it uses only $0/1$-traversals of all contained parity graphs in $G_H$ without increasing the length. Some cases are displayed in in the full version [KS12]. The remaining transformations described in the previous section can be straightforwardly adapted to the symmetric case since they only work with the connection edges of the parity graphs. Moreover, we are able to construct in polynomial time an assignment to the variables of $H$, which leaves at most $\delta\nu$ equations in $H$ unsatisfied.

7 The Shortest Superstring Problem

In order to apply the arguments given in Section 5, we first describe a well-known reduction from the SSP problem to the ATSP problem. Let $S$ be a collection of strings over $\Sigma$ such that no string is a proper substring of another string in $S$. Then, we define an instance of the ASTP problem by $(V_S, d_S)$, where $V_S = S \cup \{\Gamma\}$ with $\Gamma \notin \Sigma$ and $d_S(s_i, s_j) = |\text{pref}(s_i, s_j)|$ for all $s_i, s_j \in V_S$. Note that we can construct from a shortest tour in $(V_S, d_S)$ of length $\ell$ a shortest superstring for $S$ of length $\ell$.

We first give a high-level view of the reduction in order to build some intuition. Let $x_i \oplus x_{i+1} = 0$ be a circle equation of an instance $H$ of the Hybrid problem such that $x_i$ and $x_{i+1}$ appears only in equations with two variables. The parity gadget of $x_i \oplus x_{i+1} = 0$ consists of two strings $s_i^1$ and $s_i^2$, which can be overlapped by two letters in two
different ways. These two alignments, called $0/1$-alignments, define the assigned value to $x_i$. For any other string $s$ in the corresponding instance $S_M$, both $s^1$ and $s^2$ can be aligned with $s$ by at most 1 letter. Then, a tour in $(V_{S_M}, d_{S_M})$ is called consistent with the parity gadget for $x_i \oplus x_{i+1} = 0$ if the tour contains the arc $(s^1, s^2)$ or $(s^2, s^1)$, i.e. a $0/1$-alignment of the strings $s^1$ and $s^2$. Moreover, it is not hard to see that a tour $\sigma$ in $(V_{S_M}, d_{S_M})$ can be transformed into a tour $\pi$ that is consistent with the parity gadget for $x_i \oplus x_{i+1} = 0$ without increasing the length.

Let us start with the description of $S_H$. For every equation $g \in H$, we define a set $S(g)$ of corresponding strings.

**Strings for Circle Border Equations:** Given a circle $C_i$ and its border equation $x_i^1 \oplus x_i^2 = 0$, we introduce six associated strings. Recall that $x_i^j$ appears in an equation $g_i^j$ with three variables. The strings differ by the type of equation $x_i \oplus y \oplus z = \{0, 1\}$. We begin with the case $x_i \oplus y \oplus z = 0$: The string $L_{C_i} C_i^1$ is used as the initial part of the superstring corresponding to this circle, whereas $C_i^0 R_i$ is used as the end part. Furthermore, we introduce strings that represent an assignment that sets either the variable $x_i^1$ to 0 or the variable $x_i^2$ to 1. The corresponding two strings are $C_i x_i^1 m_0 C_i^0$ and $x_i^1 C_i R_i C_i^0 x_i^1$. Finally, we introduce $C_i x_i^1 m_0 C_i x_i^0$ and $x_i^0 C_i^0 C_i x_i^1$ having a similar interpretation. The following two alignments are called the $0$-alignment of the four strings. 

\[
C_i^1 x_i^1 m_0 C_i^1 C_i^1 x_i^0 m_0 C_i^0 C_i x_i^1 m_0 C_i^0 .
\]

(The bars indicate the position of the original strings in the corresponding alignment. For example, the bars in $C_i x_i^1 m_0 C_i^1 C_i^1 x_i^0 m_0 C_i^0 C_i x_i^1 m_0 C_i^0$ indicate the positions of $C_i^1 x_i^1 m_0 C_i^1$ and $C_i^1 C_i^1 x_i^0 m_0 C_i^0$.) On the other hand, we define the $1$-alignment as

\[
x_i^1 C_i^0 m_0 C_i^0 C_i C_i^1 x_i^1 m_0 C_i^0 C_i x_i^1 m_0 C_i^0 .
\]

For equations of the form $g_i^j \equiv x_i \oplus y \oplus z = 1$, we use $L_{C_i} C_i^1$, $C_i^0 R_i$, $C_i^1 x_i^0 m_0 C_i x_i^1 m_0 C_i^1 x_i^1 x_i^0 C_i^1 x_i^1 x_i^0 C_i^0 C_i x_i^1 m_0 C_i^0 C_i^0 C_i x_i^1 m_0 C_i^0 C_i x_i^1 m_0 C_i^0$, and $x_i^0 C_i^0 C_i x_i^1 m_0 C_i^0 C_i x_i^1 m_0 C_i^0$.

**Strings Corresponding to Matching Equations:** Let $x_i \oplus x_j = 0$ be a matching equation in $H$ with $i < j$. Then, we introduce $x_i^0 x_j^0 x_i^1 x_j^1$ and $x_i^1 x_j^1 x_i^0 x_j^0$. We define the $0$-alignment and $1$-alignment as

\[
x_i^0 x_j^0 x_i^1 x_j^1 x_i^0 x_j^0 x_i^1 x_j^1 ,
\]

respectively.

**Strings for Equations with Three Variables:** Let $g_i^j$ be an equation with three variables in $H$. For every equation $g_i^j$, we define two corresponding sets $S^0(g_i^j)$ and $S^1(g_i^j)$, both containing three strings. Finally, the set $S(g_i^j)$ is defined as the union $S^0(g_i^j) \cup S^1(g_i^j)$. An equation of the form $x \oplus y \oplus z = 0$ is represented by $S^0(g_i^j)$ containing the strings $x_i^1 x_j^1 x_i^0 x_j^0$, $y_i^1 y_j^0 m_0 x_i^0 C_i^1 x_i^1 m_0 C_i^0$, $x_i^0 m_0 C_i^1 x_i^1 m_0 C_i^0$, and $y_i^1 y_j^0 m_0 x_i^0 C_i^1 x_i^1 m_0 C_i^0$. The strings in $S^0(g_i^j)$ can be overlapped by two letters in a cyclic fashion to obtain three different constellations. A suitable constellation can be used to connect with $0/1$-alignments corresponding to circle equations.

The string $x_i^1 x_j^1 x_i^0 x_j^0 C_i^1 x_i^1 m_0 C_i^0 C_i x_i^1 m_0 C_i^0$ represents the assignment $x = 1$, whereas the constellation $y_i^1 y_j^0 m_0 x_i^0 C_i^1 x_i^1 m_0 C_i^0$ is representing $y = 1$. Finally, the string $x_i^1 x_j^1 x_i^0 x_j^0 C_i^1 x_i^1 m_0 C_i^0 C_i x_i^1 m_0 C_i^0$ can be used to overlap with $C_i x_i^1 m_0 C_i^0 C_i x_i^1 m_0 C_i^0$ consisting of the strings in $S^2(g_i^j)$ in the case $(x = 0, y = 0, z = 0)$. $x_i^1 x_j^1 x_i^0 x_j^0 C_i^1 x_i^1 m_0 C_i^0 C_i x_i^1 m_0 C_i^0$ is used in the case $z = 1$. The sets $S^0(g_i^j)$ and $S^1(g_i^j)$ representing equations of the form $g_i^j \equiv x \oplus y \oplus z = 1$ can be constructed analogously.

**Strings for Circle Equations:** Let $C_i$ be a circle in $H$ and $M_i$ its associated matching. Furthermore, let $\{i, j\}$ and $\{i + 1, j\}$ be both contained in $M_i$. We assume that $i < j$. Then, we introduce the corresponding strings for $x_i \oplus x_{i+1} = 0$. If $i + 1 < j'$, we have $x_i^0 x_{i+1}^0 x_{i+1}^1 x_i^1$ and $x_{i+1}^1 x_i^1 x_{i+1}^0 x_i^0$. We define the $0$-alignment and $1$-alignment as

\[
x_i^1 x_{i+1}^0 m_0 C_i x_{i+1}^1 x_{i+1}^1 x_i^0 m_0 C_i C_i^1 x_{i+1}^1 x_{i+1}^1 x_i^0 m_0 C_i^0 C_i x_{i+1}^1 m_0 C_i^0 C_i x_{i+1}^1 m_0 C_i^0 .
\]

In the case $(i + 1 > j')$, we use $x_i^0 x_{i+1}^0 x_{i+1}^1 x_i^1$ and $x_{i+1}^1 x_i^1 x_{i+1}^0 x_i^0$. The strings for the remaining cases can be defined analogously. If the variable $x_i$ is contained in an equation $x_i \oplus y \oplus z = 0$, we introduce three strings for the equation $x_{i-1} \oplus x_i = 0$: $x_{i-1}^1 x_i^1 x_{i-1}^0 x_i^0$, $x_{i-1}^1 x_i^1 x_{i-1}^0 x_i^0$ and $x_{i-1}^1 x_i^1 x_{i-1}^0 x_i^0$. The strings for the case $g_i^j \equiv x \oplus y \oplus z = 1$ can be constructed analogously.

We are ready to give the proof of Theorem 4.1.3 and 4.1.4.

**Proof of Theorem 4.1.3 and 4.1.4.** Given $H$ an instance of the Hybrid problem consisting of $n$ circles, $60\nu$ equations with two variables and $2\nu$ equations with three variables, we construct in polynomial time the associated instance $S_{H_M}$.

(i) Given an assignment $\phi$ to the variables of $H$ leaving $\nu$ equations with three variables unsatisfied for a constant $\delta \in (0, 1)$, we are going to construct a superstring for $S_{H_M}$. Since we may
assume that $\phi$ assigns to every variable $x_i$ associated to a circle $C_i$ the same value, we use the $\phi(x_i)$-alignment of the strings corresponding to equations contained in $C_i$. These fragments can be overlapped by one letter from both sides. For equations with three variables, we use the appropriate constellations. It yields an overlap of 5 character if the underlying equation is satisfied, and 4 otherwise. Therefore, the resulting superstring has a length at most $60\nu - 5 + 7\cdot n + 16\cdot 2\nu + 5\nu$ and a compression at least $60\nu - 8 + 12\cdot n + 28\cdot 2\nu - (7\cdot n + 332\nu + 6\nu) = 5n + 60\nu - 3 + 12\cdot 2\nu - 5\nu$.

(ii) Let $s$ be a superstring for $S_H$ having length $7\cdot n + 332\nu + 6\nu$ or compression $5n + 60\nu + 12\cdot 2\nu - 5\nu$. Recall that $s$ can be transformed into a superstring for $S_H$ using 0/1-alignments without increasing its length. The argumentation given in Section 5 for the (1,2)-ATSP problem can be adapted to analyze these fragments (0/1-alignments) and the corresponding instance $(V_{s_H}, d_{s_H})$ of the ATSP problem. Therefore, we define an assignment to the variables in $H$ according to the 0/1-alignments used in $s$ leaving at most $5\nu$ equations in $H$ unsatisfied.

References


Complexity of Counting Output Patterns of Logic Circuits

Kei Uchizawa\textsuperscript{1}  \quad Zhenghong Wang\textsuperscript{2}  \quad Hiroki Morizumi\textsuperscript{3}  \quad Xiao Zhou\textsuperscript{2}

\textsuperscript{1} Graduate School of Science and Engineering
Yamagata University,
Jonan 4-3-16, Yonezawa-shi Yamagata, 992-8510 Japan,
Email: uchizawa@yz.yamagata-u.ac.jp

\textsuperscript{2} Graduate School of Information Sciences
Tohoku University,
Aramaki-aza Aoba 6-6-05, Aoba-ku, Sendai, 980-8579, Japan,
Email: wang@nishizeki.ecei.tohoku.ac.jp, zhou@ecei.tohoku.ac.jp

\textsuperscript{3} Interdisciplinary Faculty of Science and Engineering
Shimane University,
Nishikawatsu-cho 1060, Matsue, Shimane, 690-8504, Japan,
Email: morizumi@cis.shimane-u.ac.jp

Abstract

Let $C$ be a logic circuit consisting of $s$ gates $g_1, g_2, \ldots, g_s$, then the output pattern of $C$ for an input $x \in \{0,1\}^n$ is defined to be a vector $(g_1(x), g_2(x), \ldots, g_s(x)) \in \{0,1\}^s$ of the outputs of $g_1, g_2, \ldots, g_s$ for $x$. For each $f : \{0,1\}^2 \to \{0,1\}$, we define an $f$-circuit as a logic circuit where every gate computes $f$, and investigate computational complexity of the following counting problem: Given an $f$-circuit $C$, how many output patterns arise in $C$? We then provide a dichotomy result on the counting problem: We prove that the problem is solvable in polynomial time if $f$ is PARITY or any degenerate function, while the problem is #P-complete even for constant-depth $f$-circuits if $f$ is one of the other functions, such as AND, OR, NAND and NOR.

Keywords: Boolean functions, counting complexity, logic circuits, minimum AND-circuits problem.

1 Introduction

Neural circuits in the brain consist of computational units, called neurons. Neurons communicate with each other by firing in order to perform various information processing. Many theoretical models of neurons are proposed in the literature, and a circuit consisting of such particular model of neurons is intensively studied (See, for example, a survey (Sima & Orponen 2003)). Among these models, a logic circuit (i.e., a combinatorial circuit consisting of gates, each of which computes a Boolean function) plays a fundamental role: a threshold circuit is an example of such important theoretical models (Parberry 1994, Siu et al. 1995). In these models, gates in a logic circuit $C$ output 0 or 1 for a given input assignment to $C$, and an output “1” of a gate is considered to represent a “firing” of a neuron. From the viewpoint of neuroscience, such a computation of a logic circuit shows an information coding carried out by a neural circuit: a stimulus to a neural circuit is coded to a set of firing and non-firing neurons, that is, an output pattern. More formally, an output pattern of a logic circuit is defined as follows: For a logic circuit $C$ consisting of $s$ gates $g_1, g_2, \ldots, g_s$ and $n$ input variables, an output pattern of $C$ for a circuit input $x \in \{0,1\}^n$ is defined to be a vector $(g_1(x), g_2(x), \ldots, g_s(x)) \in \{0,1\}^s$ of the outputs of $g_1, g_2, \ldots, g_s$ for $x$. In previous research, it turns out that the number of output patterns that arise in a circuit closely related to its computational power: a threshold circuit of $n + 1$ patterns can compute the Parity function of $n$ variables, while any threshold circuit of $n$ output patterns cannot (Uchizawa et al. 2011); thus, there exists a gap of computational power between threshold circuits of $n + 1$ patterns and those of $n$ patterns. Furthermore, it is known that a threshold circuit $C$ of $\Gamma$ patterns can be simulated by a threshold circuit $C'$ of $\Gamma + 1$ gates (Uchizawa et al. 2006); thus, if we can decide whether a given circuit $C$ has a small number of patterns, we can construct such $C'$ of small size.

In this paper, we focus more on the number of patterns that arise in a logic circuit. In particular, we investigate the following counting problem: Given a circuit $C$, how many output patterns arise in $C$? We show that the counting problem can be intractable even for very simple case. Consider a logic circuit $C$, every gate of which has fan-in two and computes a common function. More specifically, for each $f : \{0,1\}^2 \to \{0,1\}$, we define an $f$-circuit as a logic circuit where every gate computes $f$, and investigate computational complexity of counting output patterns of a given $f$-circuit. An $f$-circuit computes an elementary function, but analyzing its computation is not so trivial if it computes a multi-output function. For example, a multi-output $\wedge$-circuit is extensively studied in a context of automated circuit design, and minimizing size of such an $\wedge$-circuit performing a particular task is known to be an APX-hard problem (Arpe & Manthey 2009, Charikar et al. 2005, Morizumi 2011). Let $B_2$ be a set of all the Boolean functions $f : \{0,1\}^2 \to \{0,1\}$ with two input vari-
ables $z_1$ and $z_2$. Clearly, $|B_2| = 2^{2^2} = 16$. We denote the sixteen functions in $B_2$ as follows:

$0(z_1, z_2) = 0, \quad 1(z_1, z_2) = 1,$
$\oplus(z_1, z_2) = z_1 \oplus z_2, \quad \overline{\oplus}(z_1, z_2) = \overline{z_1} \oplus \overline{z_2},$
$\land(z_1, z_2) = z_1 \land z_2, \quad \lor(z_1, z_2) = z_1 \lor z_2,$
$\land(z_1, z_2) = z_1 \land z_2, \quad \lor(z_1, z_2) = z_1 \lor z_2,$
$\land(z_1, z_2) = z_1 \land z_2, \quad \lor(z_1, z_2) = z_1 \lor z_2.$

We make a complete analysis on the problem of computing the number of the output patterns that arise in an $f$-circuit $C$ for each $f \in B_2$, and provide the dichotomy result as follows: We prove that the problem is solvable in polynomial time if $f \in \{0, 1, \land, \lor, \oplus, \overline{\oplus}\}$; while the problem is $\#P$-complete even for constant-depth $f$-circuits if $f \in \{\land, \lor, \land, \lor, \land, \lor\}$ (See Table 1).

The rest of the paper is organized as follows. In Section 2, we define some terms on logic circuits and counting problems. In Section 3, we consider $f \in \{0, 1, \land, \lor, \oplus, \overline{\oplus}\}$, and show that counting patterns of an $f$-circuit is solvable in polynomial time. In Section 4, we prove $\#P$-completeness for $f \in \{\land, \lor, \land, \lor, \land, \lor\}$. In Section 5, we conclude with some remarks.

### 2 Definitions

Let $B_2$ be a set of all the Boolean functions $f : \{0, 1\}^2 \to \{0, 1\}$ with two input variables. We denote each of the functions in $B_2$ by the letter in (1). For $f \in B_2$, an $f$-gate is a logic gate that computes $f$. An $f$-circuit $C$ of $n$ input variables is expressed as a directed acyclic graph where each node of indegree 0 in $C$ corresponds to one of the input variables $x_1, x_2, \ldots, x_n$, and the other nodes correspond to $f$-gates. Let $C$ be an $f$-circuit with $n$ input variables. Let $s$ be the number of gates in $C$, and let $g_1, g_2, \ldots, g_s$ be the $s$ gates. The output pattern of $C$ for an input $x \in \{0, 1\}^n$ is defined to be the vector $(g_1(x), g_2(x), \ldots, g_s(x))$, where $g_i(x), 1 \leq i \leq s$, is the output of the gate $g_i$ for $x$. We often abbreviate an output pattern as a pattern. We say that a pattern $p = (p_1, p_2, \ldots, p_s) \in \{0, 1\}^s$ arises in $C$ if there exists an input $x \in \{0, 1\}^n$ such that $g_i(x) = p_i$ for every $i, 1 \leq i \leq s$. We denote by $\Gamma(C)$ a set of the patterns that arise in $C$. Clearly, we have $|\Gamma(C)| \leq 2^s$ and $|\Gamma(C)| \leq 2^s$. For each $f \in B_2$, a counting problem $\#Pат(f)$ is to compute $|\Gamma(C)|$ for a given $f$-circuit $C$.

The class $\#P$ is defined to be a set of functions that can be expressed as the number of accepting path of a nondeterministic polynomial-time Turing machine. Let $FP$ be a class of functions that can be computed by a polynomial-time deterministic Turing machine. A function $f$ is $\#P$-hard if $\#P \subseteq FP^f$, where $FP^f$ is a class of functions that can be computed by a polynomial-time deterministic Turing machine with an oracle to $f$. We say that $f$ is $\#P$-complete if $f \in \#P$ and $f$ is $\#P$-hard.

### 3 Tractable cases

In this section, we prove that $\#Pат(f)$ is solvable in polynomial time if $f \in \{0, 1, \land, \lor, \oplus, \overline{\oplus}\}$. That is, we prove the following theorem.

**Theorem 1.** For any $f \in \{0, 1, \land, \lor, \oplus, \overline{\oplus}\}$, $\#Pат(f)$ is in $FP$.

It is trivially true that $\#Pат(0)$ and $\#Pат(1)$ are in $FP$, since any 0-circuit and 1-circuit have only one output pattern. It thus suffices to give proofs for the other six functions $\{\land, \lor, \land, \lor, \land, \lor\}$. We first consider the cases for $\land, \lor, \land, \lor$.

**Lemma 1.** For any $f \in \{\land, \lor, \land, \lor\}$, $\#Pат(f)$ is in $FP$.

**Proof.** In this proof, we verify the lemma only for $\#Pат(\land)$, since the proofs for the other cases are similar.

Let $C$ be a circuit consisting of a number $s$ of $\land$-gates $g_1, g_2, \ldots, g_s$ together with $n$ input variables. Without loss of generality, we assume that $g_1, g_2, \ldots, g_s$ are topologically ordered in the underlying graph of $C$. Then, for each $i, 1 \leq i \leq s$, we inductively label $g_i$ by an index of an input variable as follows: If the left input of $g_i$ is an input variable $x_j$, then the label of $g_i$ is $j$; otherwise, the label of $g_i$ is same as the one for the left input gate of $g_i$. Clearly, the output of every gate labelled by $j$ equals to $x_j$ for every $x = (x_1, x_2, \ldots, x_n) \in \{0, 1\}^n$. Let $l$ be the number of distinct labels by which we give at least a gate in $C$, then we have $|\Gamma(C)| = 2^l$.

We then consider the cases for $\oplus$ and $\overline{\oplus}$.

**Lemma 2.** For any $f \in \{\oplus, \overline{\oplus}\}$, $\#Pат(f)$ is in $FP$.

**Proof.** We first consider $\oplus$-circuits. Let $C$ be a $\oplus$-circuit with $n$ input variables $x_1, x_2, \ldots, x_n$. Without loss of generality, we assume that every input variable is connected to some gate in $C$. Let $A$ be a set of gates whose inputs are both input variables. We denote by $G(C) = (V, E)$ a graph obtained from $C$ as follows:

$$V = \{1, 2, \ldots, n\}$$

and

$$E = \{(i_1, i_2) \in V \times V \mid a \text{ gate } g \in A \text{ receives } x_{i_1} \text{ and } x_{i_2} \text{ s.t. } i_1 < i_2\}.$$
Note that $G(C)$ is an undirected graph. We denote by $r$ the number of connected components in $G(C)$, and by $G_1 = (V_1, E_1), G_2 = (V_2, E_2), \ldots, G_r = (V_r, E_r)$ the $r$ connected components of $G(C)$. Note that some connected component may consist of only a single node. Let $B$ be a set of gates whose inputs are an input variable and an output of a gate. We now define $r$ Boolean values $\beta_1, \beta_2, \ldots, \beta_r \in \{0, 1\}$ as follows:

For each $j, 1 \leq j \leq r$, we have $\beta_j = 1$ if there is a gate $g \in B$ that receives $x_i, i \in V_j$, as its input; and $\beta_j = 0$ otherwise. Below we prove that

$$|\Gamma(C)| = \prod_{j=1}^{r} 2^{|V_j|+\beta_j-1}$$

by induction on $r$; Eq. (2) implies that $\#\text{PAT}(\oplus)$ is in FP, since we can obtain $V_1, V_2, \ldots, V_r$ and $\beta_1, \beta_2, \ldots, \beta_r$ in polynomial time.

For each $j, 1 \leq j \leq r$, let $A_j \subseteq A$ be a set of gates whose input variables are $x_{i_1}, x_{i_2}, \ldots, x_{i_j} \in V_j$. Clearly, $A_1, A_2, \ldots, A_r$ compose a partition of $A$. We use the following simple claim in the inductive proof.

Claim. Let $j, 1 \leq j \leq r$, be a positive integer, and let $\mathbf{a} = (a_1, a_2, \ldots, a_n) = (b_1, b_2, \ldots, b_n) \in \{0, 1\}^n$ be a pair of input assignments to $C$. Then $g(\mathbf{a}) = g(\mathbf{b})$ for every $g \in A_j$ if and only if either $a_i = b_i$ for every $i \in V_j$ or $a_i \neq b_i$ for every $i \in V_j$.

Proof. Sufficiency. Assume that either $a_i = b_i$ for every $i \in V_j$ or $a_i \neq b_i$ for every $i \in V_j$. Then, we have $a_1 + a_2 + b_1 + b_2$ (mod 2) for any pair of $i, j \in V_j$. Since every gate $g \in A_j$ is a $\oplus$-gate, we clearly have $g(\mathbf{a}) = g(\mathbf{b})$ for every $g \in A_j$.

Necessity. Assume that $g(\mathbf{a}) = g(\mathbf{b})$ for every $g \in A_j$.

for every $g \in A_j$, and suppose for the sake of contradiction that there exist indices $i_1, i_2 \in V_j$ such that $a_{i_1} = b_{i_1}$ and $a_{i_2} \neq b_{i_2}$. Consider the $j$th connected component $G_j = (V_j, E_j)$ of $G$, then there is a path $p$ between $i_1 \in V_j$ and $i_2 \in V_j$; we denote by $l$ the length of $p$, and by $(i_1, v_1), (v_1, v_2), \ldots, (v_{l-1}, i_2)$ be the edges on $p$. Since $a_{i_1} = b_{i_1}$, Eq. (3) implies that, for every $z, 1 \leq z \leq l-1$, we have $a_{v_z} = b_{v_z}$, and thus $a_{i_1} = b_{i_2}$. This contradicts the fact that $a_{i_2} \neq b_{i_2}$.

The claim implies that a same output pattern arises for $\mathbf{a}, \mathbf{b} \in \{0, 1\}^n$ only if $\mathbf{a} = \mathbf{b}$ where $\mathbf{b}$ is the bitwise complement of $\mathbf{b}$.

For the basis, we verify that Eq. (2) holds for $r = 1$.

Clearly, the claim implies that, for any pair of input assignment $\mathbf{a}, \mathbf{b} \in \{0, 1\}^n$, $g(\mathbf{a}) = g(\mathbf{b})$ for every gate $g \in A_1 (= A)$ if and only if $\mathbf{a} = \mathbf{b}$. Thus, we have

$$|\Gamma(C)| = 2^{|V_1|+\beta_1-1}$$

Consider the other case where $\beta_1 = 1$. In this case, we have $B \neq \emptyset$. Let $g'$ be an arbitrary gate in $B$. The gate $g'$ receives an input variable, say $x_i$, and an output of a gate, say $g''$. If $\mathbf{a} = \mathbf{b}$, then $g'(^{\mathbf{a}}) = g'(^{\mathbf{b}})$ if and only if $g''(^{\mathbf{a}}) = g''(^{\mathbf{b}})$. Furthermore, by the claim, we have $g(\mathbf{a}) = g(\mathbf{b})$ for every gate $g \in A_1 (= A)$ if and only if $\mathbf{a} = \mathbf{b}$. Thus, we have

$$|\Gamma(C)| = 2^{|V_1|+\beta_1-1}.$$

The inductive hypothesis, assume that Eq. (2) holds for $r - 1$, and that the graph $G(C)$ has $r$ connected components $G_1 = (V_1, E_1), G_2 = (V_2, E_2), \ldots, G_r = (V_r, E_r)$. For each $j, 1 \leq j \leq r$, we define $k_j = |V_j|$, and then relabel the $k_j$ input variables whose indices are in $V_j$ as $x_{i_1}, x_{i_2}, \ldots, x_{i_{k_j}}$.

Consider a circuit $C'$ given by applying any partial input assignment to the variables $x_{i_1}, x_{i_2}, \ldots, x_{i_{k_j}}$, and removing the gates whose outputs are determined. Let $G(C')$ be a graph obtained from $C'$. Clearly, $G(C')$ has $r - 1$ connected components, and hence the induction hypothesis implies that $C'$ has

$$\prod_{j=1}^{r-1} 2^{|V_j|+\beta_j-1}$$

patterns. Note that there are $2^{|V_1|}$ partial input assignments to $x_{i_1}, x_{i_2}, \ldots, x_{i_{k_j}}$. Therefore, if $\beta_j = 0$, the claim and Eq. (4) imply that $C$ has

$$2^{|V_1|} \times \prod_{j=1}^{r-1} 2^{|V_j|+\beta_j-1} = \prod_{j=1}^{r} 2^{|V_j|+\beta_j-1}$$

patterns, as required. On the other hand, if $\beta_j = 1$, we have a gate $g' \in B$ that receives an input variable, say $x_{i_1}, i \in V_r$, and an output of a gate, say $y'$. If $\mathbf{a} = \mathbf{b}$, then $g'(^{\mathbf{a}}) = g'(^{\mathbf{b}})$ if and only if $g''(^{\mathbf{a}}) \neq g''(^{\mathbf{b}})$. By the claim, we have $g(\mathbf{a}) = g(\mathbf{b})$ for every gate $g \in A_r$ if and only if $\mathbf{a} = \mathbf{b}$. Therefore, $C$ has

$$2^{|V_1|} \times \prod_{j=1}^{r-1} 2^{|V_j|+\beta_j-1} = \prod_{j=1}^{r} 2^{|V_j|+\beta_j-1}$$

patterns. Consequently, by Eqs. (5) and (6), Eq. (2) holds for $r$.

We can similarly compute the number of patterns of a $\oplus$-circuit, and so omit the proof.

4 Intractable cases

In this section, we consider eight functions $\land, \lor, \neg, \neg\neg, \land, \lor, \land, \lor, \land, \lor, \land, \lor, \land, \lor, \land, \lor$ and show that every case is intractable even for constant-depth circuits.

Theorem 2. For any $f \in \{\land, \lor, \land, \lor\}$, $\#\text{PAT}(f)$ is $\#P$-complete even for circuits of depth two. For any $f \in \{\land, \lor\}$, $\#\text{PAT}(f)$ is $\#P$-complete even for circuits of depth three.

In the following lemma, we show that counting patterns of a circuit $C$ is contained in $\#P$.

Lemma 3. For any $f \in \{\land, \lor, \land, \lor\}$, $\#\text{PAT}(f)$ is contained in $\#P$.

Proof. Let $f \in \{\land, \lor, \land, \lor\}$, let $C$ be an $f$-circuit of $s$ gates and $n$ input variables. It is easy to show that, given a pattern $p = (p_1, p_2, \ldots, p_s) \in \{0, 1\}^s$ of $C$, we can decide in polynomial time if the pattern $p$ arises in $C$: If $p$ arises in $C$, we decide to accept, and otherwise reject. Since the satisfiability of 2CNF formula is decidable in polynomial time (Johnson 1990, p. 96), it suffices to construct a formula $\phi$ such that $\phi$ is satisfiable if and only
Clearly, \( p \) arises in \( C \), and each clause of \( \phi \) contains at most two literals. We below construct the desired formula \( \hat{\phi} \).

Let \( g_1, g_2, \ldots, g_s \) be the gates in \( C \). We initially have an empty set \( S(=\emptyset) \) of clauses, and for each \( i, 1 \leq i \leq s \), we repeat the following procedure:

1. If \( g_i \) has two inputs from the outputs of two gates, say \( g_1 \) and \( g_2 \), then check whether \( p_i = g_i(p_1, p_2) \). If \( p_i \neq g_i(p_1, p_2) \), then \( p \) never arises in \( C \), and hence we decide to reject; otherwise, we continue the procedure.

2. If \( g_i \) has an input from an output of a gate, say \( g_1 \), and the other input from an input variable, say \( x_{i_2} \), then check whether \( p_i = g_i(p_1, y) \) for each \( y \in \{0, 1\} \). If \( p_i \neq g_i(p_1, y) \) for all \( y \in \{0, 1\} \), then we decide to reject. If \( p_i = g_i(p_1, y) \) for exactly one of \( y = 0 \) and \( y = 1 \), we add a literal of \( x_{i_2} \) to \( S \) so that the following equation holds

\[
p_i = g_i(p_1, x_{i_2}).
\]

That is, if \( p_i = g_i(p_1, 1) \), then we add a positive literal of \( x_{i_2} \) to \( S \); and if \( p_i = g_i(p_1, 0) \), then we add a negative literal \( \neg x_{i_2} \) to \( S \). If \( p_i = g_i(p_1, y) \) for all \( y \in \{0, 1\} \), we do nothing and continue the procedure.

3. If both of the inputs of \( g_i \) are input variables, say \( x_{i_1} \) and \( x_{i_2} \), then we add clauses to \( S \) so that the following equation holds

\[
p_i = g_i(x_{i_1}, x_{i_2}).
\]

Since \( g_i \) computes a function of two inputs, it is sufficient to add at most two clauses, each of which contains at most two literals.

Then we construct \( \hat{\phi} \) by taking a conjunction of all the clauses in \( S \). Since we add at most two clauses to \( S \) for each \( i, 1 \leq i \leq s \), \( \phi \) is a 2CNF formula of at most \( 2s \) input variables and at most \( 2s \) clauses. Clearly, \( \phi \) is satisfiable if and only if Eqs. (7) and (8) holds for every \( i, 1 \leq i \leq s \). Thus, \( \phi \) is satisfiable if and only if \( p \) arises in \( C \).

We now show that the problems are \#P-hard. We first prove that \#\text{PAT}(\wedge) and \#\text{PAT}(\vee) are \#P-hard by a reduction from a counting problem for graphs. Let \( G = (V, E) \) be a graph with a vertex set \( V \) and an edge set \( E \). A subset \( V' \subseteq V \) is called an independent set if for any pair of \( u, v \in V' \), \( (u, v) \not\in E \). We employ the following canonical \#P-complete problem (Provan & Ball 1983).

\#\text{INDEPENDENT SET}

\textbf{Input:} A graph \( G \)

\textbf{Output:} The number of independent sets in \( G \)

\textbf{Lemma 4.} For any \( f \in \{\wedge, \vee\} \), \#\text{PAT}(f) is \#P-hard even for circuits of depth two.

\textbf{Proof.} We first give a proof for \#\text{PAT}(\wedge). Let \( G = (V, E) \) be an instance of \#\text{INDEPENDENT SET}, where \( V = \{1, 2, \ldots, n\} \) and \( E \subseteq V \times V \). For each assignment \( \alpha = (a_1, a_2, \ldots, a_n) \in \{0, 1\}^n \), we define

\[ S_\alpha = \{i \in V \mid a_i = 1\} \]

and

\[ I = \{a \in \{0, 1\}^n \mid S_\alpha \text{ is an independent set of } G\} \]

We prove the lemma by constructing in polynomial time a depth-2 \( \wedge \)-circuit \( C_G \) such that

\[ |\Gamma(C_G)| = 2^n - |I| + 1. \]

If Eq. (9) holds, we can obtain \(|I| \) by just subtracting \(|\Gamma(C_G)| \) from \(2^n + 1 \) and hence complete the proof.

The desired circuit \( C_G \) is given as follows. \( C_G \) receives \( n \) input variables \( x_1, x_2, \ldots, x_n \). In the first layer, \( C_G \) has a gate \( g_e \) for every \( e = (i_1, i_2) \in E \) that computes \( x_{i_1} \land x_{i_2} \). In the second layer, \( C_G \) has a gate \( g_{\alpha, e} \) for each pair of \( i, 1 \leq i \leq n, \) and \( e \in E \) that computes “\( x_i \) and the output of \( g_e \)” clearly, we can construct \( C_G \) in polynomial time, and \( C_G \) is a depth-2 \( \wedge \)-circuit. We note that if \( g_e \) outputs one for an assignment \( \alpha = (a_1, a_2, \ldots, a_n) \in \{0, 1\}^n \), then we have \( g_{\alpha, e}(\alpha) = a_i \) for each \( i, 1 \leq i \leq n \).

We now verify that Eq. (9) holds. Consider an arbitrary assignment \( \alpha \in I \). Clearly, every gate in the first layer of \( C_G \) outputs zero for \( \alpha \), and hence every gate in the second layer of \( C_G \) outputs zero for \( \alpha \) too. Thus, the pattern \((0, 0, \ldots, 0) \) arises in \( C_G \) for \( \alpha \). Moreover, \((0, 0, \ldots, 0) \) arises only for an input \( \alpha \in I \), since, otherwise, we have \( g_{\alpha, e}(z) = 1 \) for some \( e \in E \).

Let \( I = \{0, 1\}^n \setminus I \). We show that \( C_G \) has an unique output pattern for each \( \alpha \in I \). Consider an arbitrary pair of assignments \( \alpha = (a_1, a_2, \ldots, a_n) \in I \) and \( \beta = (b_1, b_2, \ldots, b_n) \in I \), where \( \alpha \neq \beta \). Let \( E_\alpha \) (resp., \( E_\beta \)) be a set of the edges in a subgraph of \( G \) induced by \( S_\alpha \) (resp., \( S_\beta \)). One may assume without loss of generality that \( |E_\alpha| \geq |E_\beta| \). If \( E_\alpha \neq E_\beta \), then, for any edge \( e \) such that \( e \in E_\alpha \) and \( e \not\in E_\beta \), we have \( g_{\alpha, e}(\alpha) \neq g_{\beta, e}(\beta) \), and hence the pattern for \( \alpha \) is different from the one for \( \beta \). If \( E_\alpha = E_\beta \), then \( g_{\alpha, e}(\alpha) = g_{\beta, e}(\beta) \) for every \( e \in E \), and hence \( g_{\alpha, e}(\alpha) = a_i \) and \( g_{\beta, e}(\beta) = b_i \) for each \( i, 1 \leq i \leq n \). Since \( \alpha \neq \beta \), there is an index \( i \) such that \( a_i \neq b_i \), and hence we have \( a_i = g_{\alpha, e}(\alpha) \neq g_{\beta, e}(\beta') = b_i \). Therefore, the pattern for \( \alpha \) is different from the one for \( \beta \). Consequently, we have \(|\Gamma(C_G)| \neq |I| + 1 \). Since \(|I| + |I| = 2^n \), Eq. (9) holds.

A proof for \#\text{PAT}(\vee) can be easily obtained by \( C_G \) and De Morgan’s law, that is, we can construct an \( \lor \)-circuit \( C_\alpha \) from \( C_G \) so that \(|\Gamma(C_\alpha)| = |\Gamma(C_G)| \) as follows: (1) negate every input variable of \( C_G \) by NOT-gates, (2) push the NOT-gates forward to the outputs of the gates in the second layer, and (3) remove the NOT-gates. Since each of (1), (2) and (3) preserves the number of output patterns, we complete the proof.

We can prove \#P-hardness of \#\text{PAT}(f) for every \( f \in \{\wedge, \vee, \wedge, \vee\} \) in a very similar manner to the proof of Lemma 4. Below, we give proofs for the completeness. We employ the following variant of \#\text{INDEPENDENT SET} (Provan & Ball 1983).

\#\text{BIPARTITE INDEPENDENT SET}

\textbf{Input:} A bipartite graph \( G \)

\textbf{Output:} The number of independent sets in \( G \)

\textbf{Lemma 5.} For any \( f \in \{\wedge, \vee, \wedge, \vee\} \), \#\text{PAT}(f) is \#P-hard even for circuits of depth two.

\textbf{Proof.} We give a proof only for \#\text{PAT}(\wedge), since proofs for the other cases are obtained by the hardness of \#\text{PAT}(\wedge) and De Morgan’s law.

Let \( G = (U, V, E) \) be an instance of \#\text{BIPARTITE INDEPENDENT SET}, where \( U = \{1, 2, \ldots, n\} \), \( V = \{n + 1, n + 2, \ldots, 2n\} \) and \( E \subseteq U \times V \). For each
\( a = (a_1, a_2, \ldots, a_n) \in \{0, 1\}^{2n} \), we define \( S_a = \{ i \in U | a_i = 0 \} \cup \{ i \in V | a_i = 1 \} \), and let
\[
I = \{ a \mid S_a \text{ is an independent set of } G \}.
\]

We now construct a depth-2 \( \land \)-circuit \( C_G \) such that
\[
|\Gamma(C_G)| = 2^n - |I| + 1. \tag{10}
\]

The desired circuit \( C_G \) receives \( 2n \) input variables \( x_1, x_2, \ldots, x_{2n} \). In the first layer, \( C_G \) has a gate \( g_e \) for every \( e = (i_1, i_2) \in E \) that computes \( \overline{x_{i_1}} \) and \( y_{i_2} \), where \( \overline{x_{i_1}} \) is the negation of \( x_{i_1} \). In the second layer, \( C_G \) has a gate \( g_{i,e} \) for each pair of \( i \in U \cup V \) and \( e \in E \) that computes \( \overline{x_i} \) and the output of \( g_e \).

Clearly, we can construct \( C_G \) in polynomial time, and \( C_G \) is a depth-2 circuit consisting of only \( \land \)-gates.

We can now verify that Eq. (10) holds; we can prove that \( C_G \) has the output pattern \((0, 0, \ldots, 0)\) for every \( a \in I \), and has an unique output pattern for each \( a = (a_1, a_2, \ldots, a_n) \in I \). We omit the detail, since the rest of the proof is same as the one for Lemma 4.

We lastly verify that \( \#\text{PAT}(\overline{\land}) \) and \( \#\text{PAT}(\lor) \) are \#P-hard:

**Lemma 6.** \( \#\text{PAT}(\overline{\land}) \) and \( \#\text{PAT}(\lor) \) are \#P-hard even for circuits of depth three.

**Proof.** Note that an \( \land \)-gate \( g \) can be replaced by two \( \lor \)-gates \( g' \) and \( g'' \) such that \( g' \) receives same inputs as ones of \( g \), and \( g'' \) receives two copies of the output of \( g' \); similarly, \( \lor \)-gate can be replaced by two \( \land \)-gates.

We prove the lemma by the fact above and Lemma 4 as follows. Recall that the circuit \( C_G \) given in the proof of Lemma 4 is a depth-2 \( \land \)-circuit. By replacing each \( \land \)-gate in the first layer of \( C_G \) with two \( \lor \)-gates, we obtain a depth-3 circuit whose number of patterns is same as \( C_G \). Then we can safely replace each \( \land \)-gate in the third layer with a \( \lor \)-gate, and obtain \( C'_G \). Clearly, \( C'_G \) consists of only \( \lor \)-gates, and \( |\Gamma(C'_G)| = |\Gamma(C_G)| \). Thus we complete the proof for \( \#\text{PAT}(\overline{\land}) \). We can similarly prove the hardness of \( \#\text{PAT}(\lor) \), and so omit the proof.

5 Conclusions

In this paper, we investigate computational complexity of counting output patterns of a given \( f \)-circuit, and give a complete analysis for the counting problem on \( f \in B_2 \). More formally, we prove that the problem of counting the number of the outputs patterns that arise in an \( f \)-circuit is solvable in polynomial time if \( f \in \{ \land, \lor, a_1, a_2, \overline{\land}, \overline{\lor}, \overline{a_1}, \overline{a_2} \} \); while the problem is \#P-complete even for constant-depth \( f \)-circuits if \( f \in \{ \land, \lor \}, [\land, \lor] \}

References


Busy Beaver Machines and the Observant Otter Heuristic

James Harland
School of Computer Science and Information Technology
RMIT University
GPO Box 2476
Melbourne, 3001
Australia
james.harland@rmit.edu.au

Abstract

The busy beaver problem is to find the maximum number of non-zero characters that can be printed by an n-state Turing machine of a particular type. A critical step in the solution of this problem is to determine whether or not a given n-state Turing machine halts on a blank input. Given the enormous output sizes that can be produced by some small machines, it becomes critical to have appropriate methods for dealing with the exponential behaviour of both terminating and non-terminating machines. In this paper, we investigate a heuristic which can be used to greatly accelerate execution of this class of machines. This heuristic, which we call the observant otter, is based on the detection of patterns earlier in the execution trace. We describe our implementation of this method and report various experimental results based on it, including showing how it can be used to evaluate all known ‘monster’ machines, including some whose naive execution would take around $10^{36,534}$ steps.

1 Introduction

The busy beaver is a well-known example of a non-computable function. It was introduced by Rado [18] as a simple example of such functions, and is defined in terms of a particularly simple class of Turing machines [22]. This class of machines has a single tape, infinite in both directions, which is blank on input. Rado’s original definition included only two tape symbols, 0 and 1 (or blank and non-blank, if preferred), although generalising this to an arbitrary number of symbols is straightforward. The machine is required to be deterministic, i.e. that for any state and input symbol there is exactly one transition. Each machine also includes a special state known as the halt state, from which there are no transitions. A machine is said to have $n$ states when it has one halt state and $n$ other states. The busy beaver function for $n$ is then defined as the largest number of non-zero characters that can be printed by an $n$-state machine which halts. This function is often denoted as $\Sigma(n)$; in this paper we will use the more intuitive notation of $bb(n)$. The number of non-zeroes printed by the machine is known as its productivity.

This function can be shown to grow faster than any computable function[18]. Hence it is not only non-computable, it grows incredibly quickly. Accordingly, despite well over 40 year’s worth of exponential increases in hardware capabilities in line with Moore’s famous law[15], its value has only been established with certainty for $n \leq 4$ [13]. The values for $n = 1, 2, 3$ were established by Lin and Rado [9] in the 1960s and the value for $n = 4$ by Brady in the 1970s [3]. Larger values have proved more troublesome [11, 10, 8, 20], and the lower bounds for $n = 6$ are already spectacularly large [10]. There are some interesting analyses of the current champion machines for the $n = 5$ and $n = 6$ cases [17, 14], but due to the sheer size of the numbers involved, $bb(n)$ for $n \geq 7$ may never be known.

The current state of knowledge is given in the table below. We denote by $ff(n)$ (or frantic frog) the maximum number of state transitions performed by a terminating Turing machine with $n$ states and 2 symbols.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$bb(n)$</th>
<th>$ff(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>21</td>
</tr>
<tr>
<td>4</td>
<td>13</td>
<td>107</td>
</tr>
<tr>
<td>5</td>
<td>$\geq 4098$</td>
<td>$\geq 47,176,870$</td>
</tr>
<tr>
<td>6</td>
<td>$\geq 3.51 \times 10^{18267}$</td>
<td>$\geq 7.41 \times 10^{56534}$</td>
</tr>
</tbody>
</table>

The numbers are also spectacular for machines with more than two symbols, as can be seen in the table below (where we denote the number of symbols by $m$).

<table>
<thead>
<tr>
<th>$n$</th>
<th>$m$</th>
<th>$bb(n)$</th>
<th>$ff(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>19</td>
<td>38</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>$\geq 2050$</td>
<td>$\geq 3,932,964$</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>$\geq 374,676,383$</td>
<td>$\geq 1.12 \times 10^{178}$</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>$\geq 1.7 \times 10^{952}$</td>
<td>$\geq 1.9 \times 10^{704}$</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>$\geq 1.9 \times 10^{4933}$</td>
<td>$\geq 2.4 \times 10^{6966}$</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>$\geq 3.7 \times 10^{2518}$</td>
<td>$\geq 5.2 \times 10^{13036}$</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>$\geq 1.383 \times 10^{7036}$</td>
<td>$\geq 1.025 \times 10^{4072}$</td>
</tr>
</tbody>
</table>

Copyright ©2013, Australian Computer Society, Inc. This paper appeared at the 19th Computing: Australasian Theory Symposium (CATS 2013), Adelaide, South Australia, January-February 2013. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 141, Anthony Wirth, Ed. Reproduction for academic, not-for-profit purposes permitted provided this text is included.
In order to determine what the maximum is for any given size, it is necessary to enumerate all possible Turing machines of that size, classify each of them as to whether they terminate on the blank input or not, and search all the terminating machines for the maximum. This makes it necessary to both find the final configuration for terminating machines (even though the number of steps needed might be very large), and to show that certain machines will never terminate.

One important aspect to note is that whilst the above numbers may appear large, the output patterns of the machines themselves are relatively simple. For example, the current 6-state 2-symbol champion, which prints out in the order of $10^{18628}$ characters terminates with the string below on the tape

$$1(110110)^X11$$

where $X$ is a number with 18,267 digits.

In this paper, we show how we can efficiently execute all of the machines mentioned above, and many others like them, by means of a simple heuristic that we call the observant otter.\(^1\) This analyses the execution trace of the machine, and looks for some patterns in it. Once these are identified, this pattern can be used to predict resulting configurations of the machine, which saves a significant amount of execution time. In fact, for leashed leviathans\(^2\) like the machines mentioned above, over 99% of the execution steps can be predicted in this way, which makes it possible to determine the final configurations of such machines in reasonable times.

It is also important to note that this process of predicting the machine’s behaviour based on an execution trace is not just pivotal to showing that machines terminate, but also for showing that machines do not terminate. In [5] it is discussed how proving non-termination can be done by generating hypotheses from execution traces, and then showing that particular configuration patterns will recur infinitely. The process of showing that this will recur infinitely involves the same process of predicting execution patterns as is employed in the execution of terminating machines, and in fact the observant otter generalises all the techniques used in [5]. In this way, the observant otter can be thought of as a fundamental tool for the execution of machines of this kind.

We have implemented the observant otter heuristic, and added it to a simple execution engine based on the macro machines of Marxen and Buntrock [11]. We have tested it on 99 machines, including those mentioned above and various others.\(^3\) We have found that the observant otter has been effective on all the ‘monster’ machines. We believe that this process has led to insights which will improve the design of the execution model for such machines.

It should be noted that our philosophy is to do more than just find busy beaver machines and provide evidence of their optimality. Our aim is to study a number of interesting properties of given classes of machines. In order to do so, various tools are necessary, including methods to detect both termination and non-termination, and methods to search for all machines of a given size. This means that it is crucial to generate and store data systematically, such as a list of all machines of a given size, with or without classification. This will not only allow verification of empirical results by other researchers, but also for other properties of interest to be discovered and tested. For this reason, we believe that the process of discovery requires careful documentation, and it is this documentation that is the most important outcome of this work; results such as a specific value for the busy beaver, whilst interesting per se, are simply one aspect.

This paper is organised as follows. In Section 2 we give some basic definitions and background, and in Section 3 we describe the observant otter, including showing how it works on some examples. In Section 4 we describe our implementation and show our results, and in Section 5 we discuss our interpretation of them. Finally in Section 6 we present our conclusion and some areas of further work.

2 Definitions

2.1 Turing Machines

We use the following definition of a Turing machine [22].

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
- $\Gamma$ is the tape alphabet
- $\delta$ is a partial function from $Q \times \Gamma$ to $Q \cup \{h\} \times \Gamma \times \{l, r\}$ called the transition function
- $q_0 \in Q$ is a distinguished state called the start state

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. Hence, given some notational convention for identifying the start state and halting state, a Turing machine can be characterised by the tuples which make up the definition of $\delta$. 

---

\(^1\)The alliterative nature of the term busy beaver has inspired many similar alliterative names like this one.

\(^2\)This is another term inspired by the alliterative nature of the term busy beaver. We will explain it in more detail later.

\(^3\)All taken from Heiner Marxen’s web page, http://www.dr-b-insel.de/~heiner/BB/index.html.
Many presentations of Turing machines assume that there is a single semi-infinite tape. In this paper, as is standard with busy beaver investigations, we assume that there is a single tape which is infinite in both directions.

Note also that there are no transitions for state $h$, and that as $\delta$ is a partial function, there is at most one transition for a given pair of a state and a character in the tape alphabet.

**Definition 2** We call a Turing machine $M$

- **normal** if there is exactly one tuple $(S, I, O, D, N)$ of $M$ for which $N = h$.
- **exhaustive** if $\delta$ is a total function from $Q \times \Gamma$ to $Q \cup \{h\} \times \Gamma \times \{l, r\}$, i.e. that $\forall q \in Q \forall \gamma \in \Gamma \exists q' \in Q \cup \{h\}, \gamma' \in \Gamma$ and $D \in \{l, r\}$ such that $\delta(q, \gamma) = (q', \gamma', D)$.

Note also that machines which are exhaustive but not normal are either guaranteed not to terminate (as there is no transition into the halting state and every combination of state and input symbol has a transition defined for it), or have multiple halting transitions, of which at most one can ever be used, making the other halting transitions spurious. It is interesting to note that the Wolfram prize for finding minimal universal Turing machines given in 2007 used machines which are exhaustive but contained no halting transition.

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.

As we are interested in finding the maximum number of non-zeroes that can be printed by a terminating machine, we will only consider maximising machines, defined below.

**Definition 3** Let $M$ be a Turing machine with $n$ states and $m$ symbols. $M$ is a maximising machine if for every tuple $(S, I, O, D, N)$ in $M$, whenever $N = h$, then $O$ is 1.

In a maximising machine, the final transition that is executed, i.e. the halting transition, can be guaranteed not to reduce the number of 1’s (or more generally, non-blank symbols) on the tape. This seems an entirely desirable property in a search for busy beavers.

Henceforth we will restrict our attention to machines which are normal, exhaustive and maximising. In such a machine, the (unique) halting transition $(S, I, O, D, N)$ always has $O$ as 1, $N$ as $h$, and we may arbitrarily assign $D$ to be $l$.\(^4\) We will denote the transitions of a machine in this way, so that we in general refer to the first item in the tuple $(S, I, O, D, N)$ as $S$, the second as $I$ etc.

We will find the following notion of dimension useful.

**Definition 4** Let $M$ be a Turing machine with $n$ states and $m$ symbols (where $n, m \geq 2$). Then we say $M$ has dimension $n \times m$.

Note that as we are interested in machines which are normal, exhaustive and maximising, we know that these machines will have exactly $n \times m$ transitions, one for each combination of state and tape symbol, and exactly one of these will be the halting transition.

The reason that this is of interest is that for every $n$-state $m$-symbol machine there is an $m$-state $n$-symbol machine and vice-versa. To see this, consider the mapping between transitions defined below.

**Definition 5** Let $M$ be a Turing machine with $n$ states and $m$ symbols and let $(S, I, O, D, N)$ be a transition $T$ in $M$. We define the dual transition $\text{dual}(T)$ of $T$ as follows:

- If $N = h$, then $\text{dual}(T) = (1, S, O, D, N)$
- otherwise, $\text{dual}(T) = (1, S, O, D, I)$.

We define the dual machine dual($M$) of $M$ as dual($T$)[$T \in M$].

For example, the dual of the transitions $(a, 0, 1, r, b), (b, 1, 1, l, c)$ and $(c, 0, 1, r, h)$ are $(0, a, b, r, 1), (1, b, c, l, 1)$ and $(0, c, 1, r, h)$ respectively.\(^5\)

Hence for a machine $M$ with $n$ states and $m$ symbols, dual($M$) has $m$ states and $n$ symbols. In particular, for every $n$-state 2-symbol machine $M$, there is a unique 2-state $n$-symbol machine $M'$, and vice-versa. This shows that there are as many $n$-state $m$-symbol machines as there are $m$-state $n$-symbol machines, and hence searching through all such machines has the same complexity in each case.

While we do not discuss searching through lists of machines in this paper, it suggests that a systematic search of machines that have the same dimension is simpler than doing these independently. This seems particularly appropriate for machines of dimension 10 (i.e. machines with 5 states and 2 symbols, and 2 state and 5 symbols) and 12 (6x2, 4x3, 3x4, and 2x6 states and symbols respectively).

### 2.2 Macro Machines

As is well known, a configuration of a Turing machine is the current state of execution, containing the current tape contents, the current machine state and the position of the tape head. We will use 111[b]011 to denote a configuration in which the Turing machine is in state $b$ with the string 111011 on the tape and the tape head pointing at the 0.

Macro machines [11] is a well-known technique for optimising the execution of Turing machines. Essentially, the tape is considered to consist of blocks of characters of a given fixed length $k$, and the transitions for the machine are determined according to this structure. For example, if $k = 3$, then when the machine encounters say the input string 001 in state $c$, then it will execute naively until either the tape head moves outside these

\(^{4}\)This means that each machine we consider has a *sibling* in which the direction of every transition is swapped, which will behave identically except for the direction of its movements.

\(^{5}\)It may be more intuitive to re-label the dual transitions as $(a, 0, 1, r, h), (b, 1, 2, l, b)$ and $(a, 2, 1, r, h)$. 
three specific symbols, or it will eventually repeat an earlier configuration. In the latter case, we may classify the computation as non-terminating. Otherwise, we note the state in which the machine exits from the area containing just the three given symbols, and the symbols that now occupy these places, and consider this a new ‘macro’ transition for the machine. The only additional information that is necessary is to keep track of the direction from which the tape head entered the current string, and the direction that it exited from the string. This means that for a given string, there are two transitions: one for when it enters from the left, and another for when it enters from the right. This also means that we need to keep track of not just the current (macro) symbol, but the current orientation of the tape head (left or right). Extending the notation for configurations above, we will use $111\{b\}001\{l\}110$ to denote a configuration in which the machine is in state $b$ with the string $111001110$ on the tape, and the tape head pointing to the left hand end of the $001$ string.

For the example, if the initial symbols $001$ are converted to $111$ and the tape head leaves these three symbols in the state $d$, then we may consider this a macro transition from state $b$ with input $001$ to new state $d$ with output $111$. The details are described in Marxen and Buntrock’s seminal paper [11], and there are various improvements described on Marxen’s Busy Beaver web site.

A key aspect of macro machines is that having a fixed string length makes it easy to represent repetitive sequences of strings such as $001001001001$ as $\{001\}^4$. This, together with the acceleration technique below, is what makes macro machines significantly more efficient than naive execution.

To see how this works, consider a macro machine with $k = 3$, and an execution trace in which the current state is $b$, the current string under the tape head is $\{001\}^{12}$, and the input direction is left (so that the tape head is pointing at the first 0 in the string $\{001\}^{12}$). If the output state is also $b$, the output string is $111$ and the output direction is right (so that the tape head exits the string at the right-hand end), then it is clear that a machine configuration of the form

\[ X\{b\}(001)^{12}\{l\}Y \]

will become

\[ X(111)^{12}\{b\}Y_1\{l\}Y_2 \]

where $Y_1$ is next element of $Y$ (and $Y_2$ is the rest of $Y$).

If the output state is not $b$ (i.e. it is different to the input state) or the output direction is not to the right, then we “decouple” one copy of $001$ from the longer string $\{001\}^{12}$ and proceed. In the above example, if the output string is $110$, the output state is $c$ and the output direction is left, then the configuration

\[ X\{b\}(001)^{12}\{l\}Y \]

will become

\[ X(001)^{12}\{b\}Y_3 \]

where $Y_3$ is the last element of $Y$ (and $X_4$ is the rest of $X$).

\[ X_4 \{c\}X_2\{r\}110\{001\}^{11}Y \]

where $X_2$ is the last element of $X$ (and $X_1$ is the rest of $X$).

3 The Observant Otter

Consider the Turing machine in Figure 1. This is the 5x2 champion, which terminates after 47,176,870 steps with 4,098 ones on the tape.

During the execution of this machine, the following configurations occur.

<table>
<thead>
<tr>
<th>Step</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>12393</td>
<td>001111{b}{001}^{66}100</td>
</tr>
<tr>
<td>12480</td>
<td>00111(111){b}{001}^{63}100</td>
</tr>
<tr>
<td>12657</td>
<td>00111(111){b}{001}^{60}100</td>
</tr>
</tbody>
</table>

Clearly there is a pattern here, which is that a configuration of the form

\[ 00111(111)^X\{b\}\{001\}^Y100 \]

becomes

\[ 00111(111)^X+5\{b\}\{001\}^{Y-3}100 \]

provided that $Y > 3$. Hence the role of the observant otter is to notice this pattern, and apply it. This is done by looking through the execution trace for matching ‘shapes’, and then checking to see if there is a pattern of this form. If we detect three instances of a pattern like this, then we use the three instances to predict the final configuration in this sequence. In the above example, we can see that eventually the machine will reach the configuration below.

\[ 00111(111)^{106}\{b\}\{001\}^{3}100 \]

Note that we stop with the last non-zero value of $Y$ in the pattern. This is because there are some instances in which going down to zero causes incorrect behaviour. Whilst this is a conservative choice, it does not appear to be a very costly one. It is an item of ongoing research to determine criteria for when it is ‘safe’ to go to the zero case and when it is not.

Similar patterns, with increasingly large numbers, recur through the execution trace. For example, the following three configurations occur in the 5x2 champion.
We can use this sequence to predict a configuration of

\[001(111)^{5266}(b)001\text{100}\]

This means that we add an extra execution process to the two above for macro machines. This is to search through the execution trace for patterns that match the current one, and if one is found, to calculate the resulting configuration. Otherwise, if no such pattern applies, then we proceed as in the two cases above. In some ways, the observant otter may be thought of as a natural counterpart to the process of ‘decoupling’, i.e. the second step of the macro machine execution described above. This is because this process essentially changes a string of \(S^n\) to one of the form \(SS^{n-1}\), and hence generally decrements \(n\). As in the above example for the 5x2 champion, if the process repeated in a predictable pattern, then the observant otter will act to ‘complete’ this process. Put another way, it is this process of ‘decompressing’ \(S^n\) to \(SS^{n-1}\) (which we shall refer to as the stretching stork operation) that produces potential configurations for the observant otter to detect.

In general, then, if the current configuration is

\[X_1^a \ldots X_n^a \{S\} Y^b \{D\} Z_1^k \ldots Z_m^k\]

where the \(X_i\), \(Y\) and the \(Z_k\) are strings of character of a fixed length, \(S\) is the current state and \(\{\ldots\}\) is the current direction, the observant otter searches through the execution trace of the machine for two earlier occurrences of configurations of the same ‘shape’ and for which there is a regression, i.e., two configurations of the form

\[X_1^{a_1} \ldots X_n^{a_n} \{S\} Y^{p_1} \{D\} Z_1^{k_1} \ldots Z_m^{k_m}\]

and

\[X_1^{a_2} \ldots X_n^{a_n} \{S\} Y^{s_1} \{D\} Z_1^{l_1} \ldots Z_m^{l_m}\]

such that at least one of the following three conditions holds:

- for some \(1 \leq u \leq n\) and some \(a > 0\) we have \(a_u = i_u - a\) and \(r_u = i_u - 2a\)
- for some \(a > 0\) we have \(p = j - a\) and \(s = j - 2a\)
- for some \(1 \leq v \leq m\) and some \(a > 0\) we have \(q_v = k_v - a\) and \(l_v = k_v - 2a\)

In other words, we have the same sequence of ‘macro’ characters \(X_1 \ldots X_n \{S\} Y^\{D\} Z_1 \ldots Z_m\) in all three configurations, and the index of at least one of these configurations is regressing as execution proceeds.

One issue with this approach is that searching through the history of the execution trace has quadratic complexity, and hence has the potential to significantly slow down execution. We address this problem by setting an upper limit on the number of previous steps that are stored. In the results reported below, a ‘window’ of 150 steps was sufficient for all cases tested, including those of the highest productivity. This shows that the patterns detected by the otter are remarkably local, in that it is sufficient to look at most 150 steps in the past to find patterns. Increasing this value may find more patterns, but it comes with a noticeable performance penalty. Whilst we are interested in ensuring that the computation is efficient enough to be feasible, finding a maximally efficient method is outside the scope of this paper. If performance were to become a critical issue, then standard techniques such as hashing or balanced binary trees may be appropriate for performing this search.

A further issue is to calculate the number of steps between the current configuration and the predicted one. When the difference between the three occurrences of the pattern is constant, this is straightforward. However, there are cases in which the difference increases, and so some further care is required. This is not an essentially difficult problem, and the details are not relevant to the scope of this paper, and so they are omitted.

One point that should be noted is that the observant otter heuristic is also crucial to the success of non-termination proofs. As discussed in [5], a key aspect of many non-termination proofs is to determine a pattern from the execution trace, and then show that this pattern will recur infinitely often. This is usually done by showing that a pattern is growing in size, such as observing the sequence of configurations below,

\[(11^c)^{11}\]
\[(11^c)^{21}\]
\[(11^c)^{31}\]

and inferring from them that a configuration of the form \((11^c)^{n+1}\) will eventually result in one of the form \((11^c)^{n+1}\) for any \(n \geq 1\). This process is certainly similar to the observant otter, but seems a little distinct from it (not the least because the count is always increasing here). However, the observant otter patterns often crop up when attempting to show that configurations such as \((11^c)^{11}\) will eventually result in the configuration \((11^c)^{n+1}\). For example, consider the machine in Figure 2.

An analysis of the execution trace shows that configurations of the form \((11^c)^{11}\) occur. Hence we attempt to show that this configuration

![Figure 2: Non-terminating machine](image-url)
will eventually lead to \((11)^{n+1}[c]11\). In doing so, a necessary intermediate step is to go from the configuration \(1^01^k[c]1^{m-1}\) to \(1^{n+1}1^k-2[c]1^{m+1}\), which is precisely an observant otter step. Hence, we need the observant otter to determine that the machine will eventually reach the configuration \(1^{n+1}01^j[c]1^{m+1}\) where \(l = k \div 2\) and \(j = k \mod 2\), and from there to \((11)^{n+2}[c]11\), at which point we can conclude that we have successfully shown that this machine does not terminate. A similar method has been described by Hertel [6].

Hence the observant otter seems to be fundamental to the execution of these machines, whether terminating or not.

4 Implementation and Results

We have developed an implementation of both a naive interpreter and one based on macro machines. We then added the observant otter heuristic to the macro machine implementation. This implementation is deliberately simple; there is probably a large amount of scope for improvement, particularly in performance times. Our aim with this implementation was not to break any ‘land speed’ records, so to speak, but to show that a simple approach together with an appropriate heuristic (in this case, the observant otter) is sufficient to evaluate machines with the largest known productivities to date. Our implementation is around 2,000 lines of SWI-Prolog [21].

The table in Figures 3 and 4 shows all 99 machines that we have evaluated. This list, together with all the code developed, is also available at http://www.cs.rmit.edu.au/~jah/busybeaver. We have included machines of various dimensions, including 3x3, 6x2, 3x4, 4x3 and 2x6. The results were obtained on a desktop with an Intel i7 with 3.4GHz processors and 3.2 GB of RAM running Windows XP.

We have included the time taken to execute (in seconds). This is useful to know, but the most important point to note is that these can be achieved on commodity hardware with the longest time taking around 45 minutes, and all but four machines taking less than 10 minutes each. In other words, the entire sequence of tests can be done within a few hours at most.

In the table in Figures 3 and 4, an entry of the form \((N)\) denotes a (known) number with \(N\) digits. We have adopted the convention of only explicitly writing out numbers of up to 15 digits in length. The No. column is our identifier for each machine. The Ones column is the number of non-zero characters on the tape in the final configuration. The Hops column is the number of steps needed to reach this configuration (if executed naively). The Steps column is the number of steps executed in our interpreter. The Otters column is the number of otter patterns detected during execution. The Otter % column is the percentage of the execution steps that were predicted by the observant otter heuristic.

5 Discussion

As can be seen from the table in Figures 3 and 4, for the machines of very high productivity, the observant otter has proved to be very effective, with at least 99% of the execution steps being predicted by the observant otter. It also shows some potentially surprising results, in that the machines with the largest productivities weren’t necessarily the ones which required the largest number of observant otter occurrences. For example, machine 19, the 5x2 champion, required 11 occurrences of the observant otter, but machine 21 required 74. This suggests that the number of such occurrences is a more intuitive measure of the difficulty of execution than the sheer size of the final configuration or the number of steps required to compute it.

It is also worth noting that the ratio of Hops to Steps in the above table is very high, especially for the machines of largest productivity. Clearly the Steps value increases as the productivity does, but at an exponentially lower rate than Hops.

A key decision to make in the implementation is to determine how large the ‘history window’ should be. After a little experimentation, it was found that 150 works for all the above cases, but making it smaller tended to make the performance deteriorate significantly on some of the larger machines. This shows that the otter patterns are remarkably local, in that despite the large number of steps involved, only a small fraction of the most recent ones are needed to detect patterns (or at least, the patterns detected in this way are sufficient to efficiently execute the machine). As mentioned above, it is possible to improve the efficiency of this procedure by using data structures such as a hash table rather than the simple list implementation used here.

A further potential improvement to this process is to store the otter patterns as they are found, rather than computing them ‘fresh’ each time. For example, the 11 otter occurrences in machine 19 (the 5x2 champion) are basically different instances of two particular otter patterns. However, storing and applying the correct pattern is not altogether straightforward, particularly as the number of steps in each pattern needs to be calculated differently. This is an item of future work.

A related aspect is that of finding the ‘earliest possible’ otter pattern. To return to the 5x2 champion yet again, the first otter detected by our process is the one below.

\[
\begin{array}{ll}
\text{Step} & \text{Configuration} \\
448 & 001111\{b\}\{001\}_1^{12}\{l\}_1^{100} \\
535 & 001\{111\}_1^{10}\{b\}\{001\}_1^{3}\{f\}_1^{100} \\
712 & 001\{111\}_1^{11}\{b\}\{001\}_1^{6}\{l\}_1^{100} \\
\end{array}
\]

However, there are some earlier patterns that the macro machine structure makes it difficult to detect. This is because the choice of \(k = 3\) is generally best for this machine (due to strings such as \((001)_1^{522}\) occurring in the execution trace), but this has the effect of representing a string of 18 1’s as \((111)_1^{18}\), rather than the more compact and intuitive form \(11^{18}\). In order to find the earliest possible
<table>
<thead>
<tr>
<th>No.</th>
<th>Dim.</th>
<th>Ones</th>
<th>Hops</th>
<th>Steps</th>
<th>Otters</th>
<th>Otter %</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2x4</td>
<td>84</td>
<td>6445</td>
<td>284</td>
<td>2</td>
<td>57</td>
<td>0.094000</td>
</tr>
<tr>
<td>2</td>
<td>2x4</td>
<td>90</td>
<td>7195</td>
<td>287</td>
<td>2</td>
<td>64</td>
<td>0.078000</td>
</tr>
<tr>
<td>3</td>
<td>2x4</td>
<td>90</td>
<td>7195</td>
<td>287</td>
<td>2</td>
<td>64</td>
<td>0.078000</td>
</tr>
<tr>
<td>4</td>
<td>2x4</td>
<td>2050</td>
<td>3932964</td>
<td>737</td>
<td>8</td>
<td>99</td>
<td>0.203000</td>
</tr>
<tr>
<td>5</td>
<td>2x4</td>
<td>2050</td>
<td>3932964</td>
<td>737</td>
<td>8</td>
<td>99</td>
<td>0.204000</td>
</tr>
<tr>
<td>6</td>
<td>3x3</td>
<td>31</td>
<td>2315619</td>
<td>824187</td>
<td>0</td>
<td>18.093000</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>3x3</td>
<td>5000</td>
<td>29403894</td>
<td>974</td>
<td>11</td>
<td>&gt;99</td>
<td>0.265000</td>
</tr>
<tr>
<td>8</td>
<td>3x3</td>
<td>13902</td>
<td>99881302</td>
<td>641</td>
<td>7</td>
<td>&gt;99</td>
<td>0.204000</td>
</tr>
<tr>
<td>9</td>
<td>3x3</td>
<td>9486</td>
<td>84327815</td>
<td>592</td>
<td>11</td>
<td>&gt;99</td>
<td>0.265000</td>
</tr>
<tr>
<td>10</td>
<td>3x3</td>
<td>17338</td>
<td>127529109</td>
<td>787</td>
<td>17</td>
<td>&gt;99</td>
<td>0.219000</td>
</tr>
<tr>
<td>11</td>
<td>3x3</td>
<td>12077</td>
<td>69440986</td>
<td>421</td>
<td>7</td>
<td>&gt;99</td>
<td>0.093000</td>
</tr>
<tr>
<td>12</td>
<td>5x2</td>
<td>4098</td>
<td>47176870</td>
<td>526</td>
<td>11</td>
<td>&gt;99</td>
<td>0.172000</td>
</tr>
<tr>
<td>13</td>
<td>5x2</td>
<td>4098</td>
<td>11798826</td>
<td>510</td>
<td>10</td>
<td>&gt;99</td>
<td>0.109000</td>
</tr>
<tr>
<td>14</td>
<td>5x2</td>
<td>4097</td>
<td>23554764</td>
<td>1456</td>
<td>74</td>
<td>&gt;99</td>
<td>0.593000</td>
</tr>
<tr>
<td>15</td>
<td>5x2</td>
<td>4097</td>
<td>11798796</td>
<td>498</td>
<td>10</td>
<td>&gt;99</td>
<td>0.109000</td>
</tr>
<tr>
<td>16</td>
<td>5x2</td>
<td>4096</td>
<td>11804910</td>
<td>550</td>
<td>9</td>
<td>&gt;99</td>
<td>0.125000</td>
</tr>
<tr>
<td>17</td>
<td>5x2</td>
<td>4096</td>
<td>11804896</td>
<td>544</td>
<td>9</td>
<td>&gt;99</td>
<td>0.141000</td>
</tr>
<tr>
<td>18</td>
<td>5x2</td>
<td>1915</td>
<td>2133492</td>
<td>393</td>
<td>4</td>
<td>97</td>
<td>0.078000</td>
</tr>
<tr>
<td>19</td>
<td>5x2</td>
<td>1471</td>
<td>2358064</td>
<td>931</td>
<td>41</td>
<td>98</td>
<td>0.234000</td>
</tr>
<tr>
<td>20</td>
<td>5x2</td>
<td>501</td>
<td>134467</td>
<td>290</td>
<td>6</td>
<td>90</td>
<td>0.047000</td>
</tr>
<tr>
<td>21</td>
<td>5x2</td>
<td>90604</td>
<td>8619024596</td>
<td>1561</td>
<td>9</td>
<td>&gt;99</td>
<td>0.437000</td>
</tr>
<tr>
<td>22</td>
<td>5x2</td>
<td>64665</td>
<td>4561535055</td>
<td>1283</td>
<td>10</td>
<td>&gt;99</td>
<td>0.328000</td>
</tr>
<tr>
<td>23</td>
<td>5x2</td>
<td>97104</td>
<td>7543673517</td>
<td>1362</td>
<td>17</td>
<td>&gt;99</td>
<td>0.313000</td>
</tr>
<tr>
<td>24</td>
<td>5x2</td>
<td>458357</td>
<td>(16 digits)</td>
<td>4520</td>
<td>60</td>
<td>&gt;99</td>
<td>1.063000</td>
</tr>
<tr>
<td>25</td>
<td>5x2</td>
<td>4098</td>
<td>11798826</td>
<td>510</td>
<td>10</td>
<td>&gt;99</td>
<td>0.109000</td>
</tr>
<tr>
<td>26</td>
<td>5x2</td>
<td>4097</td>
<td>23554764</td>
<td>1456</td>
<td>74</td>
<td>&gt;99</td>
<td>0.593000</td>
</tr>
<tr>
<td>27</td>
<td>5x2</td>
<td>4097</td>
<td>11798796</td>
<td>498</td>
<td>10</td>
<td>&gt;99</td>
<td>0.109000</td>
</tr>
<tr>
<td>28</td>
<td>5x2</td>
<td>4096</td>
<td>11804910</td>
<td>550</td>
<td>9</td>
<td>&gt;99</td>
<td>0.125000</td>
</tr>
<tr>
<td>29</td>
<td>5x2</td>
<td>4096</td>
<td>11804896</td>
<td>544</td>
<td>9</td>
<td>&gt;99</td>
<td>0.141000</td>
</tr>
<tr>
<td>30</td>
<td>5x2</td>
<td>1915</td>
<td>2133492</td>
<td>393</td>
<td>4</td>
<td>97</td>
<td>0.078000</td>
</tr>
<tr>
<td>31</td>
<td>5x2</td>
<td>1471</td>
<td>2358064</td>
<td>931</td>
<td>41</td>
<td>98</td>
<td>0.234000</td>
</tr>
<tr>
<td>32</td>
<td>5x2</td>
<td>501</td>
<td>134467</td>
<td>290</td>
<td>6</td>
<td>90</td>
<td>0.047000</td>
</tr>
<tr>
<td>33</td>
<td>5x2</td>
<td>90604</td>
<td>8619024596</td>
<td>1561</td>
<td>9</td>
<td>&gt;99</td>
<td>0.437000</td>
</tr>
<tr>
<td>34</td>
<td>5x2</td>
<td>64665</td>
<td>4561535055</td>
<td>1283</td>
<td>10</td>
<td>&gt;99</td>
<td>0.328000</td>
</tr>
<tr>
<td>35</td>
<td>5x2</td>
<td>97104</td>
<td>7543673517</td>
<td>1362</td>
<td>17</td>
<td>&gt;99</td>
<td>0.313000</td>
</tr>
<tr>
<td>36</td>
<td>5x2</td>
<td>458357</td>
<td>(16 digits)</td>
<td>4520</td>
<td>60</td>
<td>&gt;99</td>
<td>1.063000</td>
</tr>
<tr>
<td>37</td>
<td>5x2</td>
<td>4098</td>
<td>11798826</td>
<td>510</td>
<td>10</td>
<td>&gt;99</td>
<td>0.109000</td>
</tr>
<tr>
<td>38</td>
<td>5x2</td>
<td>4097</td>
<td>23554764</td>
<td>1456</td>
<td>74</td>
<td>&gt;99</td>
<td>0.593000</td>
</tr>
<tr>
<td>39</td>
<td>5x2</td>
<td>4097</td>
<td>11798796</td>
<td>498</td>
<td>10</td>
<td>&gt;99</td>
<td>0.109000</td>
</tr>
<tr>
<td>40</td>
<td>5x2</td>
<td>4096</td>
<td>11804910</td>
<td>550</td>
<td>9</td>
<td>&gt;99</td>
<td>0.125000</td>
</tr>
<tr>
<td>41</td>
<td>5x2</td>
<td>4096</td>
<td>11804896</td>
<td>544</td>
<td>9</td>
<td>&gt;99</td>
<td>0.141000</td>
</tr>
<tr>
<td>42</td>
<td>5x2</td>
<td>1915</td>
<td>2133492</td>
<td>393</td>
<td>4</td>
<td>97</td>
<td>0.078000</td>
</tr>
<tr>
<td>43</td>
<td>5x2</td>
<td>1471</td>
<td>2358064</td>
<td>931</td>
<td>41</td>
<td>98</td>
<td>0.234000</td>
</tr>
<tr>
<td>44</td>
<td>5x2</td>
<td>501</td>
<td>134467</td>
<td>290</td>
<td>6</td>
<td>90</td>
<td>0.047000</td>
</tr>
<tr>
<td>45</td>
<td>5x2</td>
<td>90604</td>
<td>8619024596</td>
<td>1561</td>
<td>9</td>
<td>&gt;99</td>
<td>0.437000</td>
</tr>
<tr>
<td>46</td>
<td>5x2</td>
<td>64665</td>
<td>4561535055</td>
<td>1283</td>
<td>10</td>
<td>&gt;99</td>
<td>0.328000</td>
</tr>
<tr>
<td>47</td>
<td>5x2</td>
<td>97104</td>
<td>7543673517</td>
<td>1362</td>
<td>17</td>
<td>&gt;99</td>
<td>0.313000</td>
</tr>
<tr>
<td>48</td>
<td>5x2</td>
<td>458357</td>
<td>(16 digits)</td>
<td>4520</td>
<td>60</td>
<td>&gt;99</td>
<td>1.063000</td>
</tr>
<tr>
<td>49</td>
<td>5x2</td>
<td>4098</td>
<td>11798826</td>
<td>510</td>
<td>10</td>
<td>&gt;99</td>
<td>0.109000</td>
</tr>
<tr>
<td>50</td>
<td>5x2</td>
<td>4097</td>
<td>23554764</td>
<td>1456</td>
<td>74</td>
<td>&gt;99</td>
<td>0.593000</td>
</tr>
<tr>
<td>51</td>
<td>5x2</td>
<td>4097</td>
<td>11798796</td>
<td>498</td>
<td>10</td>
<td>&gt;99</td>
<td>0.109000</td>
</tr>
<tr>
<td>52</td>
<td>5x2</td>
<td>4096</td>
<td>11804910</td>
<td>550</td>
<td>9</td>
<td>&gt;99</td>
<td>0.125000</td>
</tr>
<tr>
<td>53</td>
<td>5x2</td>
<td>4096</td>
<td>11804896</td>
<td>544</td>
<td>9</td>
<td>&gt;99</td>
<td>0.141000</td>
</tr>
<tr>
<td>54</td>
<td>5x2</td>
<td>1915</td>
<td>2133492</td>
<td>393</td>
<td>4</td>
<td>97</td>
<td>0.078000</td>
</tr>
<tr>
<td>55</td>
<td>5x2</td>
<td>1471</td>
<td>2358064</td>
<td>931</td>
<td>41</td>
<td>98</td>
<td>0.234000</td>
</tr>
<tr>
<td>56</td>
<td>5x2</td>
<td>501</td>
<td>134467</td>
<td>290</td>
<td>6</td>
<td>90</td>
<td>0.047000</td>
</tr>
</tbody>
</table>

Figure 3: Observant Otter results (part 1)
<table>
<thead>
<tr>
<th>No.</th>
<th>Dim.</th>
<th>Ones</th>
<th>Hops</th>
<th>Steps</th>
<th>Otters</th>
<th>Otter %</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>61</td>
<td>6x2</td>
<td>(20 digits)</td>
<td>(38 digits)</td>
<td>4197</td>
<td>153</td>
<td>&gt; 99</td>
<td>1.094000</td>
</tr>
<tr>
<td>62</td>
<td>6x2</td>
<td>(22 digits)</td>
<td>(43 digits)</td>
<td>33228</td>
<td>1075</td>
<td>&gt; 99</td>
<td>8.547000</td>
</tr>
<tr>
<td>63</td>
<td>6x2</td>
<td>(48 digits)</td>
<td>(96 digits)</td>
<td>20493</td>
<td>643</td>
<td>&gt; 99</td>
<td>5.312000</td>
</tr>
<tr>
<td>64</td>
<td>6x2</td>
<td>(48 digits)</td>
<td>(96 digits)</td>
<td>19203</td>
<td>641</td>
<td>&gt; 99</td>
<td>4.969000</td>
</tr>
<tr>
<td>65</td>
<td>6x2</td>
<td>(50 digits)</td>
<td>(99 digits)</td>
<td>28242</td>
<td>380</td>
<td>&gt; 99</td>
<td>6.922000</td>
</tr>
<tr>
<td>66</td>
<td>6x2</td>
<td>(61 digits)</td>
<td>(120 digits)</td>
<td>12321</td>
<td>322</td>
<td>&gt; 99</td>
<td>2.937000</td>
</tr>
<tr>
<td>67</td>
<td>6x2</td>
<td>(50 digits)</td>
<td>(100 digits)</td>
<td>29547</td>
<td>389</td>
<td>&gt; 99</td>
<td>7.016000</td>
</tr>
<tr>
<td>68</td>
<td>6x2</td>
<td>(463 digits)</td>
<td>(926 digits)</td>
<td>265475</td>
<td>12330</td>
<td>&gt; 99</td>
<td>108.500000</td>
</tr>
<tr>
<td>69</td>
<td>6x2</td>
<td>(866 digits)</td>
<td>(1731 digits)</td>
<td>288760</td>
<td>2211</td>
<td>&gt; 99</td>
<td>87.688000</td>
</tr>
<tr>
<td>70</td>
<td>6x2</td>
<td>(48 digits)</td>
<td>(96 digits)</td>
<td>20493</td>
<td>643</td>
<td>&gt; 99</td>
<td>5.312000</td>
</tr>
<tr>
<td>71</td>
<td>6x2</td>
<td>(1440 digits)</td>
<td>(2880 digits)</td>
<td>406270</td>
<td>8167</td>
<td>&gt; 99</td>
<td>128.016000</td>
</tr>
<tr>
<td>72</td>
<td>6x2</td>
<td>(6x2 digits)</td>
<td>(120 digits)</td>
<td>12321</td>
<td>322</td>
<td>&gt; 99</td>
<td>2.937000</td>
</tr>
<tr>
<td>73</td>
<td>6x2</td>
<td>(18268 digits)</td>
<td>(36535 digits)</td>
<td>6195809</td>
<td>30345</td>
<td>&gt; 99</td>
<td>2866.063000</td>
</tr>
<tr>
<td>74</td>
<td>3x4</td>
<td>(27 digits)</td>
<td>(53 digits)</td>
<td>9187</td>
<td>358</td>
<td>&gt; 99</td>
<td>3.469000</td>
</tr>
<tr>
<td>75</td>
<td>3x4</td>
<td>(141 digits)</td>
<td>(282 digits)</td>
<td>50694</td>
<td>801</td>
<td>&gt; 99</td>
<td>14.375000</td>
</tr>
<tr>
<td>76</td>
<td>3x4</td>
<td>(435 digits)</td>
<td>(869 digits)</td>
<td>103692</td>
<td>3493</td>
<td>&gt; 99</td>
<td>36.281000</td>
</tr>
<tr>
<td>77</td>
<td>3x4</td>
<td>(629 digits)</td>
<td>(120 digits)</td>
<td>168263</td>
<td>1488</td>
<td>&gt; 99</td>
<td>54.890000</td>
</tr>
<tr>
<td>78</td>
<td>3x4</td>
<td>(1302 digits)</td>
<td>(2820 digits)</td>
<td>731258</td>
<td>1367</td>
<td>&gt; 99</td>
<td>54.890000</td>
</tr>
<tr>
<td>79</td>
<td>3x4</td>
<td>(2356 digits)</td>
<td>(4712 digits)</td>
<td>1927565</td>
<td>99697</td>
<td>&gt; 99</td>
<td>108.500000</td>
</tr>
<tr>
<td>80</td>
<td>3x4</td>
<td>(4x3 digits)</td>
<td>(926 digits)</td>
<td>265475</td>
<td>12330</td>
<td>&gt; 99</td>
<td>108.500000</td>
</tr>
<tr>
<td>81</td>
<td>3x4</td>
<td>(4x3 digits)</td>
<td>(926 digits)</td>
<td>265475</td>
<td>12330</td>
<td>&gt; 99</td>
<td>108.500000</td>
</tr>
<tr>
<td>82</td>
<td>3x4</td>
<td>(4x3 digits)</td>
<td>(926 digits)</td>
<td>265475</td>
<td>12330</td>
<td>&gt; 99</td>
<td>108.500000</td>
</tr>
<tr>
<td>83</td>
<td>3x4</td>
<td>(4x3 digits)</td>
<td>(926 digits)</td>
<td>265475</td>
<td>12330</td>
<td>&gt; 99</td>
<td>108.500000</td>
</tr>
<tr>
<td>84</td>
<td>3x4</td>
<td>(4x3 digits)</td>
<td>(926 digits)</td>
<td>265475</td>
<td>12330</td>
<td>&gt; 99</td>
<td>108.500000</td>
</tr>
<tr>
<td>85</td>
<td>3x4</td>
<td>(4x3 digits)</td>
<td>(926 digits)</td>
<td>265475</td>
<td>12330</td>
<td>&gt; 99</td>
<td>108.500000</td>
</tr>
<tr>
<td>86</td>
<td>3x4</td>
<td>(4x3 digits)</td>
<td>(926 digits)</td>
<td>265475</td>
<td>12330</td>
<td>&gt; 99</td>
<td>108.500000</td>
</tr>
<tr>
<td>87</td>
<td>3x4</td>
<td>(4x3 digits)</td>
<td>(926 digits)</td>
<td>265475</td>
<td>12330</td>
<td>&gt; 99</td>
<td>108.500000</td>
</tr>
<tr>
<td>88</td>
<td>3x4</td>
<td>(4x3 digits)</td>
<td>(926 digits)</td>
<td>265475</td>
<td>12330</td>
<td>&gt; 99</td>
<td>108.500000</td>
</tr>
<tr>
<td>89</td>
<td>3x4</td>
<td>(4x3 digits)</td>
<td>(926 digits)</td>
<td>265475</td>
<td>12330</td>
<td>&gt; 99</td>
<td>108.500000</td>
</tr>
<tr>
<td>90</td>
<td>3x4</td>
<td>(4x3 digits)</td>
<td>(926 digits)</td>
<td>265475</td>
<td>12330</td>
<td>&gt; 99</td>
<td>108.500000</td>
</tr>
<tr>
<td>91</td>
<td>3x4</td>
<td>(4x3 digits)</td>
<td>(926 digits)</td>
<td>265475</td>
<td>12330</td>
<td>&gt; 99</td>
<td>108.500000</td>
</tr>
<tr>
<td>92</td>
<td>3x4</td>
<td>(4x3 digits)</td>
<td>(926 digits)</td>
<td>265475</td>
<td>12330</td>
<td>&gt; 99</td>
<td>108.500000</td>
</tr>
<tr>
<td>93</td>
<td>3x4</td>
<td>(4x3 digits)</td>
<td>(926 digits)</td>
<td>265475</td>
<td>12330</td>
<td>&gt; 99</td>
<td>108.500000</td>
</tr>
<tr>
<td>94</td>
<td>3x4</td>
<td>(4x3 digits)</td>
<td>(926 digits)</td>
<td>265475</td>
<td>12330</td>
<td>&gt; 99</td>
<td>108.500000</td>
</tr>
<tr>
<td>95</td>
<td>3x4</td>
<td>(4x3 digits)</td>
<td>(926 digits)</td>
<td>265475</td>
<td>12330</td>
<td>&gt; 99</td>
<td>108.500000</td>
</tr>
<tr>
<td>96</td>
<td>3x4</td>
<td>(4x3 digits)</td>
<td>(926 digits)</td>
<td>265475</td>
<td>12330</td>
<td>&gt; 99</td>
<td>108.500000</td>
</tr>
<tr>
<td>97</td>
<td>3x4</td>
<td>(4x3 digits)</td>
<td>(926 digits)</td>
<td>265475</td>
<td>12330</td>
<td>&gt; 99</td>
<td>108.500000</td>
</tr>
<tr>
<td>98</td>
<td>3x4</td>
<td>(4x3 digits)</td>
<td>(926 digits)</td>
<td>265475</td>
<td>12330</td>
<td>&gt; 99</td>
<td>108.500000</td>
</tr>
<tr>
<td>99</td>
<td>3x4</td>
<td>(4x3 digits)</td>
<td>(926 digits)</td>
<td>265475</td>
<td>12330</td>
<td>&gt; 99</td>
<td>108.500000</td>
</tr>
</tbody>
</table>

This finer granularity will enable ‘earlier’ otter occurrences to be detected. This is also an item of future work.

It also seems worth remarking that often the size of the otter steps predicted seems to increase exponentially as the computation goes on. For example, for the 5x2 champion, in a total of 48 million steps, there are a total of 11 otter applications, with the number of predicted steps for each one as in the table below.

<table>
<thead>
<tr>
<th>Otter number</th>
<th>Steps predicted</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>267</td>
</tr>
<tr>
<td>2</td>
<td>2,235</td>
</tr>
<tr>
<td>3</td>
<td>6,840</td>
</tr>
<tr>
<td>4</td>
<td>20,463</td>
</tr>
<tr>
<td>5</td>
<td>62,895</td>
</tr>
<tr>
<td>6</td>
<td>176,040</td>
</tr>
<tr>
<td>7</td>
<td>343,910</td>
</tr>
<tr>
<td>8</td>
<td>492,543</td>
</tr>
<tr>
<td>9</td>
<td>522,531</td>
</tr>
<tr>
<td>10</td>
<td>533,100</td>
</tr>
<tr>
<td>11</td>
<td>245,657</td>
</tr>
</tbody>
</table>

Note that the final otter step alone predicts 30,144,672 steps, or around 60% of the total number of steps. Clearly there is an exponential growth in the number of predicted steps here, yet the overall computation still terminates. This property of terminating exponential growth is the inspiration for the name leashed leviathans; there

Figure 4: Observant Otter results (part 2)
is an exponential computation happening here, but it reaches a limit at some point. This behaviour is reminiscent of the famous \(3n + 1\) sequence. For now, we note that the observant otter may be thought of as a means of coping with this behaviour.

It is also worth noting that for the machine of very high productivity, the number of Hops taken seems to be almost always the square of the number of non-zeroes in the final configuration, and that this property seems remarkably consistent, despite the very large numbers involved. There are more sophisticated analyses of busy beaver champions than we have given here (such as those by Michel [13]); here we remark that it seems intriguing to contemplate the connection between the patterns found by the observant otter and this property, particularly as there seems to be a correlation between the number of otter occurrences and adherence to this property.

As mentioned above, in the above results we always applied the otter in such a way that the predicted configuration was always 'one less' than the maximum possible. In other words, given a pattern such as \(X^n Y^m\) leading to \(X^{n+2} Y^{m-1}\), our calculation always predicted the final state as \(X^{n+2} Y^{m-1} Y\) rather than the potentially more efficient choice of \(X^{n+2m}\). Whilst this approach worked in many cases, there are some in which this approach leads to erroneous results. Determining precise criteria under which one can 'safely' use the zero case is also an item of future work. This seems particularly important for the patterns occurring in a non-termination context; when used on terminating machines, our practice is technically less efficient, but only results in a very small increase in the time taken to execute the machine.

It should also be noted that the observant otter is not a panacea; in fact there were two machines where the otter was of no use at all (numbers 6 and 37), and one where its use was significantly less significant than many others (number 74). The first two of these are remarkable for their non-adherence to the above mentioned 'square law', in that these machines produce 31 non-zeroes in 2,315,619 steps and 143 non-zeroes in 26,375,397,569,930 steps respectively. These machines do not produce configurations with large exponents (e.g. containing strings like \((001)_{123194244242}\)), and in fact do not have any patterns recognised by the observant otter. Machine 74 is a hybrid case, in that almost half the steps were predicted by the observant otter, but the other half were not. These three machines are examples of what we call wandering wastrels; these are machines that have very low productivities compared to the number of steps needed to come to the final configuration. No doubt there are other varieties of extreme machine behaviour 'out there', so that when dealing with a systematic analysis of a large number of machines, it will be necessary to have a number of different techniques in addition to the observant otter. However, we believe that the results of this paper have shown that the observant otter is a crucial component of the techniques required.

### 6 Conclusions and Further Work

We have seen how the observant otter heuristic, even when implemented in a simple manner, makes it possible to efficiently evaluate machines of very high productivity. We have also seen how this heuristic seems to be a natural extension of the implementation of macro machines, and is also appropriate for evaluating machines in general, terminating or otherwise. It still remains an open issue as to how we may determine the earliest possible application of the observant otter. As mentioned above, this is related to the design of macro machines, and to solve this problem will require a more flexible machine architecture. In particular, rather than pick some values in advance, and see which ones work best (which is the current approach), it would be better to run a (naïve) execution for say 1,000 steps, examine the trace for patterns, and to determine the most appropriate representation. In addition, rather than use a fixed block size, it would be better to adapt the representation dynamically as the machine executes. Determining the right way to do this is an item of future work. It should be noted that this adaptive compression may also be helpful for the wandering wastrel machines.

Further work is also needed in order to determine precise conditions under which the predicted configuration can be safely calculated with a value of 0 for the final value of the decrementing variable. As mentioned above, for terminating machines, there is not a great loss of efficiency in being conservative and always avoiding this case. For non-terminating machines, it may be critical to be more precise.

Another avenue of further work is to be able to store patterns, and hence be able to reuse them without having to 'rediscover' patterns. We refer to this process as the ossified ocelot heuristic, which will be the subject of a future paper. Whilst it seems an intuitively natural thing to do, a key problem is to know exactly what to store, i.e. which parts of a given configuration are strictly necessary to the pattern, and which are irrelevant. It seems also that the number of 'different' otter patterns necessary for the evaluation of the machine is an even more intuitive measure of the complexity of the machine than the number of otter applications used during evaluation.

### Acknowledgements

The author would like to thank Austin Wood, Alex Holkner, Andy Kitchen, and the referees of this paper for helpful comments and discussions related to this work.
References


## Author Index

<table>
<thead>
<tr>
<th>Author</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brass, Peter</td>
<td>3</td>
</tr>
<tr>
<td>Chan, Ho-Leung</td>
<td>11</td>
</tr>
<tr>
<td>Chan, Sze-Hang</td>
<td>11</td>
</tr>
<tr>
<td>Chi-Man</td>
<td>11</td>
</tr>
<tr>
<td>Harland, James</td>
<td>43</td>
</tr>
<tr>
<td>Karpinski, Marek</td>
<td>27</td>
</tr>
<tr>
<td>Knauer, Christian</td>
<td>3</td>
</tr>
<tr>
<td>Lam, Franky</td>
<td>21</td>
</tr>
<tr>
<td>Lam, Tak-Wah</td>
<td>11</td>
</tr>
<tr>
<td>Lee, Lap-Kei</td>
<td>11</td>
</tr>
<tr>
<td>Li, Rongbin</td>
<td>11</td>
</tr>
<tr>
<td>Liu</td>
<td>11</td>
</tr>
<tr>
<td>Morizumi, Hiroki</td>
<td>37</td>
</tr>
<tr>
<td>Schmied, Richard</td>
<td>27</td>
</tr>
<tr>
<td>Shin, Chan-Su</td>
<td>3</td>
</tr>
<tr>
<td>Smid, Michiel</td>
<td>3</td>
</tr>
<tr>
<td>Uchizawa, Kei</td>
<td>37</td>
</tr>
<tr>
<td>Vigan, Ivo</td>
<td>3</td>
</tr>
<tr>
<td>Wang, Zhenghong</td>
<td>37</td>
</tr>
<tr>
<td>Wirth, Anthony</td>
<td>iii</td>
</tr>
<tr>
<td>Wong, Raymond K.</td>
<td>21</td>
</tr>
<tr>
<td>Zhou Xiao</td>
<td>37</td>
</tr>
</tbody>
</table>
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 118 - Computing Education 2011

Volume 117 - Parallel and Distributed Computing 2011

Volume 116 - Database Technologies 2011

Volume 115 - Information Security 2011

Volume 114 - Computing Education 2011
Edited by Julián Mestre, University of Sydney, Australia. January 2012. 978-1-921770-12-0.

Volume 113 - Computer Science 2011

Volume 112 - Computer Science 2012
Edited by Mark Reynolds, The University of Western Australia, Australia and Bruce Thomas, University of Sydney, Australia. January 2012. 978-1-921770-04-3.

Volume 111 - Computer Science 2012
Edited by Michael de Raadt, Moodle Pty Ltd and Angela Carbone, Monash University, Australia. January 2012. 978-1-921770-03-6.

Volume 110 - Computer Science 2011

Volume 109 - Computer Science 2011

Volume 108 - Computer Science 2011
Edited by Josef Pieprzyk, Macquarie University, Australia and Rory T. Smith, University of South Australia, Australia. December 2012. 978-1-921770-14-2.

Volume 107 - Parallel and Distributed Computing 2011

Volume 106 - Parallel and Distributed Computing 2012
Edited by Jinjun Chen, Swinburne University of Technology, Australia and Rajiv Ranjan, University of New South Wales, Australia. January 2012. 978-1-921770-08-1.

Volume 105 - Parallel and Distributed Computing 2012

Volume 104 - Data Mining and Analytics 2012
Edited by YanchangZhao, Department of Immigration and Citizenship, Australia, Junqing Li, University of Western Australia, Australia, Peter Christen, Australian National University, Australia, and Heng Tao Shen, The University of Queensland, Australia. December 2012. 978-1-921770-14-2.