

Paolo Boldi (Ed.)

LNCS 6099

Fun with Algorithms

5th International Conference, FUN 2010
Ischia, Italy, June 2010
Proceedings



 Springer

Commenced Publication in 1973

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison

Lancaster University, UK

Takeo Kanade

Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler

University of Surrey, Guildford, UK

Jon M. Kleinberg

Cornell University, Ithaca, NY, USA

Alfred Kobsa

University of California, Irvine, CA, USA

Friedemann Mattern

ETH Zurich, Switzerland

John C. Mitchell

Stanford University, CA, USA

Moni Naor

Weizmann Institute of Science, Rehovot, Israel

Oscar Nierstrasz

University of Bern, Switzerland

C. Pandu Rangan

Indian Institute of Technology, Madras, India

Bernhard Steffen

TU Dortmund University, Germany

Madhu Sudan

Microsoft Research, Cambridge, MA, USA

Demetri Terzopoulos

University of California, Los Angeles, CA, USA

Doug Tygar

University of California, Berkeley, CA, USA

Gerhard Weikum

Max-Planck Institute of Computer Science, Saarbruecken, Germany

Paolo Boldi Luisa Gargano (Eds.)

Fun with Algorithms

5th International Conference, FUN 2010
Ischia, Italy, June 2-4, 2010
Proceedings

Volume Editors

Paolo Boldi
Università degli Studi di Milano
Dipartimento di Scienze dell'Informazione
20135 Milano, Italy
E-mail: paolo.boldi@gmail.com

Luisa Gargano
Università di Salerno
Dipartimento di Informatica ed Applicazioni
84084, Fisciano, Italy
E-mail: lg@dia.unisa.it

Library of Congress Control Number: 2010926872

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

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

ISSN 0302-9743
ISBN-10 3-642-13121-2 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-13121-9 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

springer.com

© Springer-Verlag Berlin Heidelberg 2010
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper 06/3180

Preface

FUN with Algorithms is a three-yearly conference that aims at attracting works which, besides a deep and interesting algorithmic content, also present amusing and fun aspects, are written with a keen wit, and are presented in a lively way. FUN is actually one of the main moving wheels behind most of the best scientific results, and in a sense this conference answers to the unconfessed need of having a place where we can present the most lighthearted part of our work without sacrificing precision and rigor.

The 5th International Conference on Fun with Algorithms (FUN 2010) was held at Hotel Continental Terme in Ischia (Italy), June 2–4, 2010. The island of Ischia, a worldwide famous spa, sea, and tourist resort, is the ideal venue to host an event dedicated to pleasure as well as to science.

The call for papers attracted 54 submissions from all over the world. Submitted papers were characterized by an extremely high quality and featuring a large variety of topics. After a careful and thorough reviewing process, the Program Committee selected 32 papers. The program also included three invited talks by Roberto Grossi, Prabhakar Raghavan, and Paul Spirakis. Extended versions of selected papers presented at the meeting will be published in a special issue of *Theory of Computing Systems*.

We would like to take this opportunity to thank all the authors who submitted their work to FUN 2010 and of course all the colleagues that served on the Program Committee whose comments and discussions were crucial in selecting the papers. We also wish to thank all the external reviewers (listed in the following pages) who provided further reports on the papers as well as the members of the Organizing Committee (Gennaro Cordasco and Adele Rescigno).

The *EasyChair* Conference System (<http://www.easychair.org/>) was used through all the stages (submission, review, selection, preparation of the proceedings) and greatly simplified our work: we wish to thank its creators and maintainers for their support and help. We also thank Alfred Hofmann and Anna Kramer at Springer, who provided feedback and timely answers to our questions. We are pleased to acknowledge support from Dipartimento di Informatica ed Applicazioni “R.M. Capocelli,” from Università di Salerno, and from Dipartimento di Scienze dell’Informazione (Università degli studi di Milano).

March 2010

Paolo Boldi
Luisa Gargano

Organization

Program Chairs

Paolo Boldi, Luisa Gargano

Program Committee

Ricardo Baeza-Yates, Anne Bergeron, Jean-Claude Bermond, Erik Demaine, Leah Epstein, Leszek Gasieniec, Magnus Halldorsson, Evangelos Kranakis, Fabrizio Luccio, Michael Mitzenmacher, Muthu Muthukrishnan, Gonzalo Navarro, Rasmus Pagh, David Peleg, Nadia Pisanti, Jos Rolim, Nicola Santoro, Eduardo Sany-Laber, Michael Segal, Steven Skiena, Shmuel Zaks

External Reviewers

Michael Atkinson, Valentina Ciriani, Andrew Collins, Philippe Giabbanelli, Mathieu Giraud, Ronen Gradwohl, Markus Holzer, Thore Husfeldt, David Ilcinkas, Rastislav Královič, Danny Krizanc, Dario Malchiodi, Carlo Mereghetti, Pat Morin, Nicolas Nisse, Linda Pagli, Stéphane Perennes, Roberto Radicioni, Tomasz Radzik, Adele Rescigno, Marco Rosa, Cole Smith, Henry Soldano, Krister Swenson, Ugo Vaccaro, Gerhard Woeginger, Norbert Zeh

Local Organization

Gennaro Cordasco, Luisa Gargano, Adele Rescigno

Table of Contents

Fun with Olympiad in Algorithmics (Invited Talk)	1
<i>Roberto Grossi, Alessio Orlandi, and Giuseppe Ottaviano</i>	
The FUNnest Talks That belong to FUN (Abstract) (Invited Talk)	3
<i>Prabhakar Raghavan</i>	
Fun with Games (Invited Talk)	4
<i>Paul G. Spirakis, Ioannis Chatzigiannakis, Georgios Mylonas, and Panagiota N. Panagopoulou</i>	
Do We Need a Stack to Erase a Component in a Binary Image?	16
<i>Tetsuo Asano</i>	
Kaboodle Is NP-complete, Even in a Strip	28
<i>Tetsuo Asano, Erik D. Demaine, Martin L. Demaine, and Ryuhei Uehara</i>	
A Hat Trick	37
<i>Oren Ben-Zwi and Guy Wolfowitz</i>	
Fun at a Department Store: Data Mining Meets Switching Theory	41
<i>Anna Bernasconi, Valentina Ciriani, Fabrizio Luccio, and Linda Pagli</i>	
Using Cell Phone Keyboards is (\mathcal{NP}) Hard	53
<i>Peter Boothe</i>	
Urban Hitchhiking	68
<i>Marco Bressan and Enoch Peserico</i>	
A Fun Application of Compact Data Structures to Indexing Geographic Data	77
<i>Nieves R. Brisaboa, Miguel R. Luaces, Gonzalo Navarro, and Diego Seco</i>	
On Table Arrangements, Scrabble Freaks, and Jumbled Pattern Matching	89
<i>Péter Burcsi, Ferdinando Cicalese, Gabriele Fici, and Zsuzsanna Lipták</i>	
Cryptographic and Physical Zero-Knowledge Proof: From Sudoku to Nonogram	102
<i>Yu-Feng Chien and Wing-Kai Hon</i>	

A Better Bouncer's Algorithm	113
<i>Ferdinando Cicalese, Travis Gagie, Anthony J. Macula, Martin Milanič, and Eberhard Triesch</i>	
Tradeoffs in Process Strategy Games with Application in the WDM Reconfiguration Problem	121
<i>Nathann Cohen, David Coudert, Dorian Mazauric, Napoleão Nepomuceno, and Nicolas Nisse</i>	
UNO Is Hard, Even for a Single Player	133
<i>Erik D. Demaine, Martin L. Demaine, Ryuhei Uehara, Takeaki Uno, and Yushi Uno</i>	
Leveling-Up in Heroes of Might and Magic III	145
<i>Dimitrios I. Diochnos</i>	
The Magic of a Number System	156
<i>Amr Elmasry, Claus Jensen, and Jyrki Katajainen</i>	
Bit-(Parallelism) ² : Getting to the Next Level of Parallelism	166
<i>Domenico Cantone, Simone Faro, and Emanuele Giaquinta</i>	
An Algorithmic Analysis of the Honey-Bee Game	178
<i>Rudolf Fleischer and Gerhard J. Woeginger</i>	
Mapping an Unfriendly Subway System	190
<i>Paola Flocchini, Matthew Kellett, Peter C. Mason, and Nicola Santoro</i>	
Cracking Bank PINs by Playing Mastermind	202
<i>Riccardo Focardi and Flaminia L. Luccio</i>	
Computational Complexity of Two-Dimensional Platform Games	214
<i>Michal Forišek</i>	
Christmas Gift Exchange Games	228
<i>Arpita Ghosh and Mohammad Mahdian</i>	
Return of the Boss Problem: Competing Online against a Non-adaptive Adversary	237
<i>Magnús M. Halldórsson and Hadas Shachnai</i>	
Managing Change in the Era of the iPhone	249
<i>Patrick Healy</i>	
The Computational Complexity of RACETRACK	260
<i>Markus Holzer and Pierre McKenzie</i>	
Simple Wiggling Is Hard Unless You Are a Fat Hippo	272
<i>Irina Kostitsyna and Valentin Polishchuk</i>	

The Urinal Problem	284
<i>Evangelos Kranakis and Danny Krizanc</i>	
Fighting Censorship with Algorithms	296
<i>Mohammad Mahdian</i>	
The Complexity of Flood Filling Games	307
<i>David Arthur, Raphaël Clifford, Markus Jalsenius, Ashley Montanaro, and Benjamin Sach</i>	
The Computational Complexity of the KAKURO Puzzle, Revisited	319
<i>Oliver Ruepp and Markus Holzer</i>	
Symmetric Monotone Venn Diagrams with Seven Curves	331
<i>Tao Cao, Khalegh Mamakani, and Frank Ruskey</i>	
The Feline Josephus Problem	343
<i>Frank Ruskey and Aaron Williams</i>	
Scheduling with Bully Selfish Jobs	355
<i>Tami Tamir</i>	
O(1)-Time Unsorting by Prefix-Reversals in a Boustrophedon Linked List	368
<i>Aaron Williams</i>	
Author Index	381

Fun with Olympiad in Algorithmics

Roberto Grossi, Alessio Orlandi, and Giuseppe Ottaviano

Dipartimento di Informatica, Università di Pisa
Largo Bruno Pontecorvo 3, 56127 Pisa, Italy
{grossi,aorlandi,ottaviano}@di.unipi.it

The creative approach when designing new algorithms is probably the major rewarding task for many researchers. This love for solving algorithmic problems is cultivated by some very young and talented students at high school.

The IOI (*International Olympiad in Informatics*) is a special programming contest for these students [1]. The main flavor is not just computer programming, but deep thinking in terms of algorithmics. The required skills are problem analysis, design of efficient algorithms and data structures, programming and testing [2]. The winners are among the best young computer scientists in the world. It is not so unusual that they are also winners in other International Science Olympiads, such as IMO (*International Mathematical Olympiad*). They perform extremely well when going to academia.

Teams from over 80 countries attend IOI every year. It is impressive to see how many different cultures meet at this event, many more than we can imagine at the events organized by our community. Students compete on an individual basis, with up to four students per country, and try to maximize their score during two competition days. There are three or four tasks to be solved in five hours. The typical task during a competition is a problem with input size n , where the range of possible values of n are known in advance, and a time limit s in seconds. Looking at the assigned values of n and s , the contestants can make an educated guess on the time complexity of the required algorithm. Sometimes a space limit is given, thus allowing a guess also for the space complexity.

For all the inputs, called test cases, the algorithmic solution must give a correct answer and stay within the time and space limits. Each correct answer increases the score by a fixed amount. The algorithmic problems are described using nice stories, and are non-trivial at all. Quite often, these stories are the vanilla (and masqueraded) version of research problems from the known literature. The scientific background of the teenager contestants is quite impressive for their age [3].

In this talk, we will organize a small session with selected problems, based upon our experience in training the Italian team for IOI. People are usually attracted by these quiz-style sessions, and are surprised to learn that these kids solve three or four of them in just five hours. Here, solving does not mean just providing the algorithmic ideas (the creative and funny part of the competition), but also programming and debugging is required, since the verdict is very demanding: either the program passes the test case to get the score, or it is not accepted even if it contains the greatest idea in the world but with a small glitch.

We will focus on the creative part, which is more fun for the algorithmists. Prizes will be distributed to the audience. As one can imagine, the algorithmic quality of IOI is very high and much denser than other similar competitions in Informatics. It is very rare that a contestant can solve a problem by merely applying a known algorithm; rather, she should invent a new one or make a non-trivial modification of a known one. For these reasons, IOI should be properly called Olympiad in Algorithmics, but do not let us spread this little secret with non-algorithmists :-).

References

1. International Olympiad in Informatics, <http://ioinformatics.org>
2. Skiena, S.S., Revilla, M.A.: Programming Challenges: The Programming Contest Training Manual. Springer, Heidelberg (2003)
3. Verhoeff, T., Horváth, G., Diks, K., Cormack, G.: A Proposal for an IOI Syllabus. Teaching: Mathematics and Computer Science (4/1), 193–216 (2006)

The FUNnest Talks That belong to FUN (Abstract)

Prabhakar Raghavan

Yahoo! Labs.

Abstract. This talk presents the author's personal favorites of fun talks from algorithms and complexity. Not all of these were presented at FUN, but definitely would have generated fun at FUN.

Fun with Games^{*}

Paul G. Spirakis^{1,2}, Ioannis Chatzigiannakis^{1,2}, Georgios Mylonas^{1,2},
and Panagiota N. Panagopoulou²

¹ Computer Engineering and Informatics Department, University of Patras

² Research Academic Computer Technology Institute

spirakis@cti.gr, ichatz@cti.gr, mylonasg@cti.gr, panagopp@cti.gr

Abstract. We discuss two different ways of having fun with two different kinds of games: On the one hand, we present a framework for developing multiplayer pervasive games that rely on the use of mobile sensor networks. On the other hand, we show how to exploit game theoretic concepts in order to study the graph-theoretic problem of vertex coloring.

1 Introduction

In everyday life, a *game* is a structured, competitive activity that people undertake for enjoyment. In applied mathematics however (and in particular in the branch of game theory), a game is a mathematical model used to describe situations where decision-makers with possibly conflicting interests interact. These two kinds of games seem loosely related and rather irrelevant; however, they both share the same key components: *goals*, *rules*, and *interaction*: Each player has to choose a course of actions out of a set of allowed actions in order to win, while always taking into consideration the actions taken by the opponents.

Here, we aim at exploring “funny” aspects of both kinds of games. On the one hand, we present a framework for creating, deploying and administering multiplayer pervasive games that rely on the use of ad hoc mobile sensor networks [1]. The unique feature in such games is that players interact with each other and their surrounding environment by using movement and presence as a means of performing game-related actions, utilizing sensor devices. On the other hand, we show how game-theoretic tools and concepts can be exploited in order to study the fundamental graph-theoretic problem of vertex coloring [2]. We define an imaginary game among the vertices of the graph; the analysis of the game leads to the derivation of several upper bounds on the chromatic number, as well as to a polynomial time vertex coloring algorithm that achieves all these bounds.

Developing sensor-based multiplayer pervasive games. The outstanding activity in the wireless sensor networking (WSN) research area resulted among other things in the the continuous integration of sensing devices in multiple application

^{*} Partially supported by the EU within the ICT Programme under contract IST-2008-215270 (FRONTS).

domains. Even though we are using sensors in an ever-increasing number of ways, we have only scratched the surface regarding their use in entertainment-related applications. The use of sensors such as accelerometers, e.g., in Nintendo Wii, has already been proven a major success. At the same time, there is an additional trend of detaching from traditional gaming environments, evident by the massive success of mobile platforms, like Sony's PSP, or even devices like the iPhone. Lately, *pervasive gaming* appeared as a hot new gaming genre. Some examples of games that are placed in the pervasive gaming genre are geotagging games, ubiquitous games, mixed reality games, urban games, etc. In this context, we believe that there is great potential in combining distributed sensor networks and pervasive gaming to produce exciting gaming applications.

Such games can be largely based on features like *movement*, *presence*, and *sensory input*, all provided by the use of sensor networking techniques. Players interact with each other and their surrounding environment by moving, running and gesturing as a means to perform game related actions, using sensor devices. We identify here the main issues and research challenges that arise in multiplayer pervasive games based on distributed sensor networks and provide solutions and guidelines for the most important of them, and we present Fun in Numbers (FinN, <http://finn.cti.gr>), a framework for developing pervasive applications and interactive installations for entertainment and educational purposes. Using ad hoc mobile wireless sensor network nodes as the enabling devices, FinN allows for the quick prototyping of applications that utilize input from multiple physical sources (sensors and other means of interfacing), by offering a set of programming templates and services, such as topology discovery, localization and synchronization, that hide the underlying complexity. FinN's architecture is based on a hierarchy of layers for scalability and easy customization to different scenarios (heterogeneity). A number of services are currently implemented, allowing location awareness of wireless devices in indoor environments, perform sensing tasks while on the move, coordinate basic distributed operations and offer delay-tolerant communication. We present the target application domains of FinN, along with a set of multiplayer games and interactive installations, and we describe the overall architecture of our platform.

Using game theory to study vertex coloring. One of the central optimization problems in Computer Science is the problem of *vertex coloring* of graphs: given a graph $G = (V, E)$ of n vertices, assign a color to each vertex of G so that no pair of adjacent vertices gets the same color and so that the total number of distinct colors used is minimized. The global optimum of vertex coloring (the *chromatic number*) is, in general, inapproximable in polynomial time unless a collapse of some complexity classes happens. We describe here an efficient vertex coloring algorithm that is based on *local search*: Starting with an arbitrary proper vertex coloring (e.g. the trivial proper coloring where each vertex is assigned a unique color), we do local changes, by allowing each vertex (one at a time) to move to another color class of higher cardinality, until no further local moves are possible.

We choose to illustrate this local search method via a game-theoretic analysis; we do so because of the natural correspondence of the local optima of our

proposed method to the pure Nash equilibria of a suitably defined *strategic game*. In particular, given a graph $G = (V, E)$ of n vertices and m edges, we define the *graph coloring game* $\Gamma(G)$ as a strategic game where the set of players is the set of vertices and the players share the same action set, which is a set of n colors. The payoff that a vertex v receives, given the actions chosen by all vertices, equals the total number of vertices that have chosen the same color as v , unless a neighbor of v has also chosen the same color, in which case the payoff of v is 0. We show that $\Gamma(G)$ has always pure Nash equilibria, and each pure equilibrium is a proper coloring of G . We give a polynomial time algorithm \mathcal{A} which computes a pure Nash equilibrium of $\Gamma(G)$, and show that the total number, k , of colors used in *any* pure Nash equilibrium (and thus achieved by \mathcal{A}) is $k \leq \min\{\Delta_2 + 1, \frac{n+\omega}{2}, \frac{1+\sqrt{1+8m}}{2}, n - \alpha + 1\}$, where ω, α are the clique number and the independence number of G and Δ_2 is the maximum degree that a vertex can have subject to the condition that it is adjacent to at least one vertex of equal or greater degree. (Δ_2 is no more than the maximum degree Δ of G .) Thus, in fact, we propose here a new, efficient coloring method that achieves a number of colors satisfying (together) the known general upper bounds on the chromatic number χ . Our method is also an alternative general way of proving, constructively, all these bounds.

2 Developing Games: A Platform for Sensor-Based Multiplayer Pervasive Games

We present the basic system requirements and key design goals of Fun in Numbers (FinN), a platform for developing pervasive applications and interactive installations for entertainment and educational purposes. We attempt to identify the differentiating factors of our approach from already existing ones, along with some of the respective implementation requirements. We believe that these key factors are common in both application domains that use ad hoc and mobile sensor networks for entertainment and education.

Simultaneous participation of multiple users: we envisage games and installations where groups of players participate, potentially in large numbers. The players will be in close proximity, most probably in indoor environments, and will have to engage in such applications by either interacting between themselves or with an infrastructure provided by the organizing authority. Depending on the nature of the application, players may have to cooperate or compete with each other, e.g., to reach the goals of a team-based game inside a museum, and this may be done in a real-time fashion. Regarding implementation, this assumes that there is a reliable neighborhood discovery mechanism, along with proximity detection, location-aware and context-aware providing mechanisms to the software and the players. These mechanisms are required to scale to a large number of players and to different area sizes.

Multiple types of inputs: we envisage the utilization of a plethora of inputs, the most general of which are presence, motion and other types of sensors. Such

inputs will, in the majority of cases, be provided by the mobile devices carried by the participating players. In simple words, this means that e.g., pupils or museum visitors will carry mobile devices that are able to sense their location (absolute or relative to each other and specific landmarks), their movement (both in terms motion detection and gesture recognition) and other physical measures (e.g., the device could sense if the player is in a warm/cold or light/dark place). Therefore an expandable architecture is required to cover all the different sensors that can be used on a single device and be reported to the upper layers of the system, along with mechanisms for reliable motion detection and gesture recognition. In the additional case of using cameras throughout the system, respective mechanisms for the same actions must be used.

Distributed network operation: the use of embedded sensors and ad hoc networking capabilities requires that the software executed on the mobile devices carried by players is based on lightweight mechanisms. The complex parts of the system's logic need to be implemented at the fixed infrastructure. Furthermore, depending on the final application, further functionalities may be required that rely on real-time coordination and complete knowledge of the users' whereabouts, or are executed in a disconnected part of the network. It is therefore necessary for the architecture to be distributed and to involve a certain level of modularity and heterogeneity. Delay-tolerant mechanisms can be activated to ensure the correct operation of the system and/or reliable multihop or multicasting mechanisms may be necessary to cover all possibilities of communication between players and the infrastructure.

Need for synchronization and coordination between players: in most games players are competing or cooperating in order to reach/fulfill the goals set in a specific application. Players have to directly interact with each other and the overall system in a synchronized way. Such synchronization schemes should cover updates of the state of the players and the system, and also possibly coordinate the ways that the users move and act inside the playing field. Mutual exclusion, agreement and leader election mechanisms may be used to ensure the correct operation of the system.

Non-conventional interfacing methods and use of actuators/haptics: the participants should be able to decipher both their personal and/or their team's status/score while engaging in the proposed interactive schemes, and also the system interfaces should reflect the location and context awareness inherent in such situations. The use of actuators such as lights turning on/off, opening/closing doors, haptic interfaces, etc., will enable a more immersive experience.

Pilot Multiplayer Games. In order to further demonstrate the capabilities of our system we present here three pilot games that we have implemented. The key characteristic of these games is that players engage in interactions with each other and their surrounding environment by moving and gesturing, as a

means to perform game-related actions. The player, as a physical entity, is the center of the game. The players' input is kept to a minimum (e.g., by means of performing a specific gesture) or is indirect (e.g., based on the location of the player). Similarly, the feedback of the game to the player is again minimum (e.g., win or lose) and some times sporadic (e.g., indicating that the player reached a specific location or is close to an opponent). There is minimum need for continuous visual feedback compared to most video games played today, e.g., through a display. FinN games are meant to be played in every place and at every time, with or without any fixed game "backbone" infrastructure. After the game is over, players can upload the data collected by their devices to a social networking web portal.

Moving Monk: Each player in the game is called a "monk", moving continuously amongst a predefined set of "temples". The goal for each player is to visit all of the temples as fast as possible, perform specific "prayers" in each location. A temple is defined by the coverage range of the available infrastructure and the prayers performed are specific gestures. To help monks find the temples, clues can be given regarding the exact location of a temple, but in general the players are unaware of the temples' location. The winner of the game is the first monk who gets to visit all of the temples.

Hot Potato: In this game, each device held by players randomly generates a Hot Potato, which "explodes" after a specific amount of time, eliminating the player carrying the potato. Each player can pass the potato to one of the neighboring players, by performing a specific gesture. Thus, each player tries to pass the potato of her device to the other players, so as to avoid elimination by the exploding potato. If the player tries to avoid meeting (i.e., getting outside the range of) other players, then a new potato is generated in her device with high probability. As a result, more than one potatoes may be active simultaneously in each game. When a player who already carries a potato receives an additional one, then two potatoes merge. The winner is the player who last stands alive while all other opponents have been eliminated.

Casanova: This is a two-players-only game. One of the players is randomly selected as the "Casanova", while the other one as "Bianca". The goal of Casanova is to run away from Bianca, while Bianca must not lose Casanova from her sight, running when Casanova runs and staying still when Casanova does not move. The two players are informed for who is who and the actual game starts. Casanova tries to win Bianca, by running away from her, or by staying still suddenly. This game is based on ad-hoc networks, where the need of infrastructure is not compulsory. As a result, the Casanova game is easily played anywhere and anytime.

Pilot Interactive Installations. We now present two pilot networked interactive installations that we are currently implementing using our platform. We

expect that these schemes are simple, self explaining and should challenge players (of all ages, but mostly pupils) to interact with them. Like before, we are based on two basic sensing capabilities, presence (near a point of interest or near another person), and movement detection and recognition. A notable advantage of using mobile ad hoc sensors (e.g., Sun SPOTs instead of cameras) is that each device provides a unique identification of a player, that works similarly to an RFID and can be used for user history tracking or for enhancing the overall entertainment experience.

Chromatize it!: This edutainment installation is based on the mixture of basic colors. The basic features demonstrated here are proximity between devices, player's input as well as visual output. A chromatic mass appears as soon as the player approaches the screen. By choosing among basic colors available on his device, the player colorizes the masses' minions. By doing so, he mixes colors, in an effort to match the color of the mass. The matching combination leads to an ever increasing difficulty of levels in chromatic complexity. More than one players can simultaneously participate.

Tug of war: In this highly competitive multiplayer game, players enter a 3D cube (approx. 2 x 2 x 2 m) on each side of which colors are floating. Each color defines a territory owned by a player. The aim of each player is to expand his territory as much as possible. This is achieved when the indicated gestures are performed properly and fast. Visual output as well as gesture recognition are the basic characteristics of this installation. Each player owns a chromatic territory, which he aims to expand over his opponents. By performing gestures faster than his opponents he manages to dominate. There is no limit to the amount of players.

Overall Architecture. FinN is designed and implemented targeting application scenarios where a large number of players, using wireless handheld devices with sensing capabilities, participate in various game instances and game types. These games can take place in the same or different place and time. The operation of the games may be supported by a "backbone" infrastructure that provides a number of services (e.g., localization and context awareness). The games may be coordinated by a central entity that records the games' progress. The architecture of FinN has been based upon those principles and has been implemented by a hierarchy of layers. Each layer is assigned a particular role in the game:

Guardian layer: This layer is composed by the devices used by players during the FinN games. The *Guardian* is the software component running in each player's wireless sensing device and uses the devices capabilities in terms of user interface, communication, etc. Protocols for the discovery and the communication with the "backbone" infrastructure and other Guardians are provided (echo protocol service). When another Guardian peer is discovered the player may be prompted for further action, by using the sensors and the buttons of her device.

For monitoring the evolution of the game, each game related action is represented by an *Event*. Also, Guardian peers implement services that allow them to interact even when they are disconnected from the “backbone” infrastructure for extended periods of time. In particular, when an Event occurs, the Guardian stores it to the device memory and when communication with the infrastructure is possible, then all collected Events are forwarded (delay tolerant communication service). Also, Guardians provide a subsystem, which processes the samples of the accelerometer and recognizes gestures that correspond to game-related actions.

Game Station layer: This layer implements the “backbone” infrastructure, which is important though not necessary for all the games developed. It provides localization and context awareness services and it is through this infrastructure that the data of the players are transferred to and from the higher layers of the architecture, for coordination and storing purposes. This wireless backbone is established by Station peers, with each Station controlling a specific physical area. During the initialization of each game, one Station peer becomes also the Game Engine, responsible for the coordination of the infrastructure and of the game itself. The Stations communicate with the users’ devices either through local ad-hoc networks or via personal area non-IP networks and act as gateways, essentially allowing communication between the players’ devices and the Game Engine. Multiple Stations can be attached to an Engine in order to maximize area coverage or the points of interest. During the initialization of a game, Stations communicate with the Engine and retrieve data such as the set of players, which are registered for this game instance, the associations between Avatars, player devices and POIs. Stations are also responsible for the Guardians initialization and for forwarding all data generated during the course of a game to the Engine. There is also the option of using *mobile Stations* during the course of a game. In this case, such Stations operate in a slightly different way than the stationary mode - their role is primary in the context of providing location-aware services during games, while communication with the upper layers is either suppressed (by informing lower layers to use other ways of propagating game-related events to the upper layers) or carried out in a delay-tolerant mode.

Game Engine layer: Each game instance is assigned to and also coordinated by a specific *Game Engine*, i.e., it is the local authority for each physical game site. The Engine retrieves data from higher layers and stores them locally, for the duration of a specific game. In order to avoid computational and communication overhead, data between higher layers and the Engines are synchronized periodically. Thus, the processing and storage of generated events during the game is done locally. The Engine is also a control mechanism that provides game-specific services and implements various game scenarios. Communication between the Engine and the Stations is carried out through wired and/or wireless IP-based networks. Finally the Engine features an embedded Web container for providing additional game specific information to players.

World layer: The *World* layer is the topmost layer of the hierarchy, enabling the management of multiple FinN games, physical game sites and users. This layer includes the *World Portal*, which is the central point of management in the system, providing interaction with all the different game instances operating in the real world. It is also the central storing point for all game-related data, such as player-related statistics and game history. Furthermore, it allows personalization capabilities and possible interaction with external social networking sites.

3 Exploiting Games: A Strategic Game for Efficient Vertex Coloring

Denote $G = (V, E)$ a simple, undirected graph with vertex set V and set of edges E . For a vertex $v \in V$ denote $N(v) = \{u \in V : \{u, v\} \in E\}$ the set of its neighbors, and let $\deg(v) = |N(v)|$ denote its degree. Let $\Delta(G) = \max_{v \in V} \deg(v)$ be the maximum degree of G . Let $\Delta_2(G) = \max_{u \in V} \max_{v \in N(u): d(v) \leq d(u)} \deg(v)$ be the maximum degree that a vertex v can have, subject to the condition that v is adjacent to at least one vertex of degree no less than $\deg(v)$. Clearly, $\Delta_2(G) \leq \Delta(G)$. Let $\chi(G)$ denote the chromatic number of G , i.e. the minimum number of colors needed to color the vertices of G such that no adjacent vertices get the same color (i.e., the minimum number of colors used by a *proper coloring* of G). Let $\omega(G)$ and $\alpha(G)$ denote the clique number and independence number of G , i.e. the number of vertices in a maximum clique and a maximum independent set of G .

Given a finite, simple, undirected graph $G = (V, E)$ with $|V| = n$ vertices, we define the *graph coloring game* $\Gamma(G)$ as the game in strategic form where the set of players is the set of vertices V , and the action set of each vertex is a set of n colors $X = \{x_1, \dots, x_n\}$. A *configuration* or *pure strategy profile* $\mathbf{c} = (c_v)_{v \in V} \in X^n$ is a combination of actions, one for each vertex. That is, c_v is the color chosen by vertex v . For a configuration $\mathbf{c} \in X^n$ and a color $x \in X$, we denote by $n_x(\mathbf{c})$ the number of vertices that are colored x in \mathbf{c} , i.e. $n_x(\mathbf{c}) = |\{v \in V : c_v = x\}|$. The *payoff* that vertex $v \in V$ receives in the configuration $\mathbf{c} \in X^n$ is

$$\lambda_v(\mathbf{c}) = \begin{cases} 0 & \text{if } \exists u \in N(v) : c_u = c_v \\ n_{c_v}(\mathbf{c}) & \text{else} \end{cases}.$$

A *pure Nash equilibrium* (PNE in short) is a configuration $\mathbf{c} \in X^n$ such that no vertex can increase its payoff by unilaterally deviating. Let (x, \mathbf{c}_{-v}) denote the configuration resulting from \mathbf{c} if vertex v chooses color x while all the remaining vertices preserve their colors. Then, $\mathbf{c} \in X^n$ is a pure Nash equilibrium if, for all vertices $v \in V$, $\lambda_v(x, \mathbf{c}_{-v}) \leq \lambda_v(\mathbf{c}) \quad \forall x \in X$.

A vertex $v \in V$ is *unsatisfied* in the configuration $\mathbf{c} \in X^n$ if there exists a color $x \neq c_v$ such that $\lambda_v(x, \mathbf{c}_{-v}) > \lambda_v(\mathbf{c})$; else we say that v is *satisfied*. For an unsatisfied vertex $v \in V$ in the configuration \mathbf{c} , we say that v performs a *selfish step* if v unilaterally deviates to some color $x \neq c_v$ such that $\lambda_v(x, \mathbf{c}_{-v}) > \lambda_v(\mathbf{c})$.

Existence and Tractability of Pure Nash Equilibria

Lemma 1. *Every pure Nash equilibrium \mathbf{c} of $\Gamma(G)$ is a proper coloring of G .*

Proof. Assume, by contradiction, that \mathbf{c} is not a proper coloring. Then there exists some vertex $v \in V$ such that $\lambda_v(\mathbf{c}) = 0$. Clearly, there exists some color $x \in X$ such that $c_u \neq x$ for all $u \in V$. Therefore $\lambda_v(x, \mathbf{c}_{-v}) = 1 > 0 = \lambda_v(\mathbf{c})$, which contradicts the fact that \mathbf{c} is an equilibrium. \square

Theorem 1. *For any graph coloring game $\Gamma(G)$, a pure Nash equilibrium can be computed in $O(n \cdot \alpha(G))$ selfish steps, where n is the number of vertices of G and $\alpha(G)$ is the independence number of G .*

Proof. We define the function $\Phi : P \rightarrow \mathbb{R}$, where $P \subseteq X^n$ is the set of all configurations that correspond to proper colorings of the vertices of G , as $\Phi(\mathbf{c}) = \frac{1}{2} \sum_{x \in X} n_x^2(\mathbf{c})$, for all proper colorings \mathbf{c} . Fix a *proper* coloring \mathbf{c} . Assume that vertex $v \in V$ can improve its payoff by deviating and selecting color $x \neq c_v$, and let $\mathbf{c}' = (x, \mathbf{c}_{-v})$. It can be shown that \mathbf{c}' is a proper coloring and that

$$\Phi(\mathbf{c}') - \Phi(\mathbf{c}) = n_x(\mathbf{c}) + 1 - n_{c_v}(\mathbf{c}) = \lambda_v(\mathbf{c}') - \lambda_v(\mathbf{c}) .$$

Therefore, if any vertex v performs a selfish step then the value of Φ is increased as much as the payoff of v is increased. Now, the payoff of v is increased by at least 1. So after any selfish step the value of Φ increases by at least 1. Now observe that, for all proper colorings \mathbf{c} and for all colors x , $n_x(\mathbf{c}) \leq \alpha(G)$. Therefore $\Phi(\mathbf{c}) \leq \frac{1}{2} \sum_{x \in X} (n_x(\mathbf{c}) \cdot \alpha(G)) = \frac{n \cdot \alpha(G)}{2}$. Moreover, the minimum value of Φ is $\frac{1}{2}n$. Therefore, if we allow any unsatisfied vertex (but only one each time) to perform a selfish step, then after at most $\frac{n \cdot \alpha(G) - n}{2}$ steps there will be no vertex that can improve its payoff (because Φ will have reached a local maximum, which is no more than the global maximum), so a pure Nash equilibrium will have been reached. Of course, we have to start from an initial configuration that is a proper coloring so as to ensure that \mathcal{A} will terminate in $O(n \cdot \alpha(G))$ selfish steps; this can be found easily since there is always the trivial proper coloring that assigns a different color to each vertex of G . \square

The above proof implies the following simple algorithm \mathcal{A} that computes a pure Nash equilibrium of $\Gamma(G)$ (and thus a proper coloring of G): At each step, allow one unsatisfied vertex to perform a selfish step, until all vertices are satisfied. Note that, at each step, there may be more than one unsatisfied vertices, and more than one colors that a vertex could choose in order to increase its payoff. So actually \mathcal{A} is a whole class of algorithms, since one could define a specific ordering (e.g., some fixed or some random order) of vertices and colors, and examine vertices and colors according to this order. In any case however, the algorithm is guaranteed to terminate in $O(n \cdot \alpha(G))$ selfish steps. Furthermore, each selfish step can be implemented straightforwardly in $O(n^2)$ time, since there are n vertices and n colors that each vertex can be assigned.

Bounds on the Total Number of Colors

Lemma 2. *In any pure Nash equilibrium of $\Gamma(G)$, the number k of total colors used satisfies $k \leq \Delta_2(G) + 1$ and hence $k \leq \Delta(G) + 1$.*

Proof. Consider a pure Nash equilibrium \mathbf{c} of $\Gamma(G)$, and let k be the total number of distinct colors used in \mathbf{c} . If $k = 1$ then it is easy to observe that G must be totally disconnected, i.e. $\Delta(G) = \Delta_2(G) = 0$ and therefore $k = \Delta_2(G) + 1$. Now assume $k \geq 2$. Let $x_i, x_j \in X$ be the two colors used in \mathbf{c} that are assigned to the minimum number of vertices. W.l.o.g., assume that $n_{x_i}(\mathbf{c}) \leq n_{x_j}(\mathbf{c}) \leq n_x(\mathbf{c})$ for all colors $x \notin \{x_i, x_j\}$ used in \mathbf{c} . Let v be a vertex such that $c_v = x_i$. The payoff of vertex v is $\lambda_v(\mathbf{c}) = n_{x_i}(\mathbf{c})$. Now consider any other color $x \neq x_i$ that is used in \mathbf{c} . Assume that there is no edge between vertex v and any vertex u with $c_u = x$. Then, since \mathbf{c} is a pure Nash equilibrium, it must hold that $n_{x_i}(\mathbf{c}) \geq n_x(\mathbf{c}) + 1$, a contradiction. Therefore there is an edge between vertex v and at least one vertex of every other color. Hence the degree of vertex v is at least the total number of colors used minus 1, i.e. $\deg(v) \geq k - 1$. Furthermore, let u be the vertex of color $c_u = x_j$ that v is connected to. Similar arguments as above yield that u must be connected to at least one vertex of color x , for all $x \notin \{x_i, x_j\}$ used in \mathbf{c} . Moreover, u is also connected to v . Therefore $\deg(u) \geq k - 1$. Now:

$$\begin{aligned} \Delta_2(G) &= \max_{s \in V} \max_{\substack{t \in N(s) \\ \deg(t) \leq \deg(s)}} \deg(t) \\ &\geq \max \left\{ \max_{\substack{t \in N(v) \\ \deg(t) \leq \deg(v)}} \deg(t), \max_{\substack{t \in N(u) \\ \deg(t) \leq \deg(u)}} \deg(t) \right\} \\ &\geq \min \{ \deg(u), \deg(v) \} \geq k - 1 \end{aligned}$$

and therefore $k \leq \Delta_2(G) + 1$ as needed. \square

Lemma 3. *In a pure Nash equilibrium, all vertices that are assigned unique colors form a clique.*

Proof. Consider a pure Nash equilibrium \mathbf{c} . Assume that the colors c_v and c_u chosen by vertices v and u are unique, i.e. $n_{c_v}(\mathbf{c}) = n_{c_u}(\mathbf{c}) = 1$. Then the payoff for both vertices is 1. If there is no edge between u and v then, since \mathbf{c} is an equilibrium, it must hold that $1 = \lambda_v(\mathbf{c}) \geq \lambda_v(c_u, \mathbf{c}_{-v}) = 2$, a contradiction. \square

Lemma 4. *In any pure Nash equilibrium of $\Gamma(G)$, the number k of total colors used satisfies $k \leq \frac{n + \omega(G)}{2}$.*

Proof. Consider a pure Nash equilibrium \mathbf{c} of $\Gamma(G)$. Assume there are $t \geq 0$ vertices that are each assigned a unique color. These t vertices form a clique (Lemma 3), hence $t \leq \omega(G)$. The remaining $n - t$ vertices are assigned non-unique colors, so the number of colors in \mathbf{c} is $k \leq t + \frac{n-t}{2} = \frac{n+t}{2} \leq \frac{n+\omega(G)}{2}$. \square

Lemma 5. *In any pure Nash equilibrium of $\Gamma(G)$, the number k of total colors used satisfies $k \leq \frac{1 + \sqrt{1 + 8m}}{2}$.*

Proof. Consider a pure Nash equilibrium \mathbf{c} of $\Gamma(G)$. W.l.o.g., assume that the k colors used in \mathbf{c} are x_1, \dots, x_k . Let V_i , $1 \leq i \leq k$, denote the subset of all vertices $v \in V$ such that $c_v = x_i$. W.l.o.g., assume that $|V_1| \leq |V_2| \leq \dots \leq |V_k|$. Observe that, for each vertex $v_i \in V_i$, there is an edge between v_i and some $v_j \in V_j$, for all $j > i$. If not, then v_i could improve its payoff by choosing color x_j , since $|V_j| + 1 \geq |V_i| + 1 > |V_i|$. This implies that $m \geq \sum_{i=1}^{k-1} |V_i|(k-i)$ and, since $|V_i| \geq 1$ for all $i \in \{1, \dots, k\}$, $m \geq \sum_{i=1}^{k-1} (k-i)$ or equivalently $m \geq \frac{k(k-1)}{2}$ or equivalently $k^2 - k - 2m \leq 0$, which implies $k \leq \frac{1+\sqrt{1+8m}}{2}$. \square

Theorem 2. *In any pure Nash equilibrium of $\Gamma(G)$, the number k of total colors used satisfies $k \leq n - \alpha(G) + 1$.*

Proof. Consider any pure Nash equilibrium \mathbf{c} of $\Gamma(G)$. Let t be the maximum, over all vertices, payoff in \mathbf{c} , i.e. $t = \max_{x \in X} n_x(\mathbf{c})$. Partition the set of vertices into t sets V_1, \dots, V_t so that $v \in V_i$ if and only if $\lambda_v(\mathbf{c}) = i$ (note that each vertex appears in exactly one such set, however not all sets have to be nonempty). Let k_i denote the total number of colors that appear in V_i . Clearly, $|V_i| = i \cdot k_i$ and the total number of colors used in \mathbf{c} is $k = \sum_{i=1}^t k_i$. Now consider a maximum independent set I of G . The vertices in V_1 have payoff equal to 1, therefore they are assigned unique colors, so, by Lemma 3, the vertices in V_1 form a clique. Therefore I can only contain at most one vertex among the vertices in V_1 . Our goal is to upper bound the size of I . First we prove the following:

Claim 1. If there exists some $i > 1$ such that $k_i = 1$ and I contains all the vertices in V_i , then $k \leq n - \alpha(G) + 1$.

Proof of Claim 1. Let x denote the unique color that appears in V_i . Since I contains all the vertices in V_i , then it cannot contain any vertex in $V_1 \cup \dots \cup V_{i-1}$. This is so because each vertex $v \in V_j$, $j < i$, is connected by an edge with at least one vertex of color x (otherwise v could increase its payoff by selecting x , which contradicts the equilibrium). Furthermore, each vertex in V_i has at least one neighbor of each color that appears in $V_{i+1} \cup \dots \cup V_t$. Therefore

$$|I| = \alpha(G) \leq |V_i| + \sum_{j=i+1}^t |V_j| - \sum_{j=i+1}^t k_j = n - \sum_{j=1}^{i-1} |V_j| - k + \sum_{j=1}^i k_j$$

which gives $k \leq n - \alpha(G) + \sum_{j=1}^{i-1} (k_j - |V_j|) + k_i \leq n - \alpha(G) + k_i = n - \alpha(G) + 1$. \square

So now it suffices to consider the case where, for all $i > 1$ such that $k_i = 1$, I does not contain all the vertices in V_i . So I contains at most $|V_i| - 1 = |V_i| - k_i$ vertices that belong to V_i . In order to complete the proof we need the following:

Claim 2. For all $i > 1$ with $k_i \neq 1$, I cannot contain more than $|V_i| - k_i$ vertices among the vertices in V_i .

Proof of Claim 2. This is clearly true for $k_i = 0$ (and hence $|V_i| = 0$). Now assume that $k_i \geq 2$. Observe that, for all vertices $v_i \in V_i$ there must exist an edge between v_i and a vertex of each one of the remaining $k_i - 1$ colors that

appear in V_i (otherwise, v_i could change its color and increase its payoff by 1, which contradicts the equilibrium). Fix a color x of the k_i colors that appear in V_i . If I contains all vertices of color x , then it cannot contain any vertex of any color other than x that appears in V_i . Therefore I can contain at most $i \leq (i-1)k_i = |V_i| - k_i$ vertices among the vertices in V_i . On the other hand, if I contains at most $i-1$ vertices of each color x that appears in V_i , then I contains again at most $(i-1)k_i = |V_i| - k_i$ vertices among the vertices in V_i . \square

Therefore I cannot contain more than $|V_i| - k_i$ vertices among the vertices of V_i , for all $i > 1$, plus one vertex from V_1 . Therefore:

$$|I| = \alpha(G) \leq 1 + \sum_{i=2}^t (|V_i| - k_i) = 1 + n - |V_1| - (k - |V_1|) = n - k + 1 .$$

So, in any case, $k \leq n - \alpha(G) + 1$ as needed. \square

The bounds given by Lemmata [2](#), [4](#), [5](#) and Theorem [2](#), together with the facts that any Nash equilibrium is a proper coloring (Lemma [1](#)) and that a Nash equilibrium can be computed in polynomial time (Theorem [1](#)) imply:

Corollary 1. *For any graph G , a proper coloring that uses at most $k \leq \min \left\{ \Delta_2(G) + 1, \frac{n+\omega(G)}{2}, \frac{1+\sqrt{1+8m}}{2}, n - \alpha(G) + 1 \right\}$ colors can be computed in polynomial time.*

References

1. Akribopoulos, O., Logaras, M., Vasilakis, N., Kokkinos, P., Mylonas, G., Chatzi-
giannakis, I., Spirakis, P.: Developing Multiplayer Pervasive Games and Networked
Interactive Installations using Ad hoc Mobile Sensor Nets. In: 5th International Con-
ference on Advances in Computer Entertainment Technology (ACE 2009), Athens,
Greece (2009)
2. Panagopoulou, P., Spirakis, P.: A Game Theoretic Approach for Efficient Graph
Coloring. In: Hong, S.-H., Nagamochi, H., Fukunaga, T. (eds.) ISAAC 2008. LNCS,
vol. 5369, pp. 1–15. Springer, Heidelberg (2008)

Do We Need a Stack to Erase a Component in a Binary Image?

Tetsuo Asano

Japan Advanced Institute of Science and Technology (JAIST)
Ishikawa 923-1292, Japan
t-asano@jaist.ac.jp

Abstract. Removing noises in a given binary image is one of common operations. A generalization of the operation is to erase arbitrarily specified component by reversing pixels values in the component. This paper shows that this operation is done without using any data structure like a stack or queue, or without using any extra work space in $O(n \log n)$ time for a component consisting of n pixels. This is an in-place algorithm, but the image matrix cannot be used as work space since it has a single bit for each pixel. Whenever we flip pixel value in an objective component, the component shape also changes, which causes some difficulty. An idea for our constant work space algorithm is a conversion keeping its topology.

Keywords: constant work space, binary image, component, connectivity.

1 Introduction

Consider a binary image consisting of black and white pixels. We can define connected components of white (or black) pixels. Each connected component may correspond to some object or a noise component. Erasing a component is rather easy. Starting at any pixel in the component, we iteratively include neighboring pixels of the same color into a stack while marking them. When no more extension is possible, we pop pixels from the stack and flip its pixel value. It is done in linear time in the component size. This algorithm, however, needs mark bits and a stack (or queue), whose size can be linear in the image size in the worst case. Since we have to keep locations of those pixels in the stack, the total work space can be much larger than the given binary image itself which requires $O(m)$ bits for a binary image with m pixels. We also need $O(m)$ bits for the mark bits.

In this paper we show that we can erase a given component consisting of n pixels in $O(n \log n)$ time without using any extra work space except $O(\log n)$ bits in total. The algorithm works even for a component having a number of holes in it in the same time complexity.

This is the first constant work space algorithm for erasing a component in a binary image, where erasing a component means flipping a pixel value at each pixel in the component, say from white to black. Although it is usual to assume

a read-only array for input image in other constant work space algorithms, our input image is a read/write array. But it is hard to use the image matrix as work space since it has a single bit for each pixel. Furthermore, whenever we flip pixel value in an objective component, the component shape also changes, which causes some difficulty. An idea for our constant work space algorithm is a conversion keeping its topology.

There are several related results on images, such as an in-place algorithm for rotating an image by an arbitrary angle [1] and a constant work space algorithm for scanning an image with an arbitrary angle [2] and others [9]. For in-place algorithms a number of different algorithms are reported [7].

2 Preliminary

Consider a binary image G which consists of n white (value 1) and black (value 0) pixels. When two pixels of the same color are adjacent horizontally or vertically, we say they are 4-connected [10]. Moreover, if there is a pixel sequence of the same color interconnecting two pixels p and q and every two consecutive pixels in the sequence are 4-connected, then we also say that they are 4-connected. We can define 8-connectivity in a similar fashion. In the 4-connectivity we take only four among eight immediate neighbors (pixels in the 3×3 -neighborhood) of a pixel. In the 8-connectivity we take all of those eight immediate neighbors as 8-connected neighbors.

A 4-connected (resp., 8-connected) component is a maximal set of pixels of the same color any two of which are 4-connected (resp., 8-connected). Hereafter, it is referred to as a component in short if there is no confusion. Following the tradition we assume that white components are defined by 4-connectivity while black ones by 8-connectivity. A binary image may contain many white components. Some of them may have holes, which are black components. Even holes may contain white components, called islands, and islands may contain islands' holes, etc.

In this paper a pixel is represented by a square. The four sides of the square are referred to as edges. An edge is called a boundary edge if it lies between two pixels of different colors. We orient each boundary edge so that a white pixel always lies to its left. Thus, external boundaries are oriented in a counter-clockwise manner while internal boundaries are clockwise oriented.

The leftmost vertical boundary edge on a boundary is defined as a **canonical edge** of the boundary. If there are two or more such edges then we take the lowest one. The definition guarantees that each boundary, internal or external, has a unique canonical edge. It is also easily seen that the canonical edge of an external boundary is always downward (directed to the South) and that of an internal one upward (directed to the North). So, when we find a canonical edge, it is also easy to determine whether the boundary containing it is external or not. It suffices to check the direction of the canonical edge.

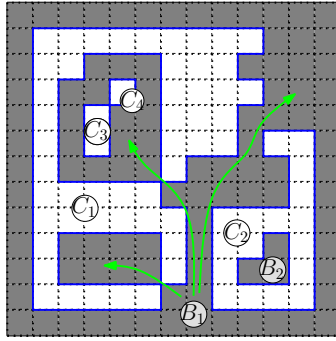


Fig. 1. (White) components C_1 and C_2 , (black) holes B_1 and B_2 , and (white) islands C_3 and C_4 in a binary image. There are four (white) components consisting of white pixels. Note that C_3 and C_4 are not 4-connected. Holes consist of black pixels. Since 8-connectivity is used for (black) pixels, this image contains only two black components.

3 Erasing a Connected Component in Constant Work Space

One of the most fundamental problems in computer vision or pattern recognition is, given a query pixel p in a binary image, to enumerate all pixels belonging to a component to which the pixel p belongs. The problem is also considered for an intensity image. Suppose we know a local rule (or a function using local information around a pixel in question) on how to partition a given intensity image into homogeneous regions. Then, the problem of extracting a region to which a query pixel belongs is just the same as above for a binary image.

The problem is easily solved using a stack or queue. Starting from a query pixel q , we expand a search space just as wave is propagated from q . Whenever we find a pixel of the same color reachable from q which has not been checked yet, we put it into the data structure and check its neighborhood to look for unvisited pixels of the same color. This simple algorithm works quite well. In fact, it runs in time linear in the number of pixels of the component (or component size). Unfortunately, it is known that the size of the data structure is linear in the size of the component in the worst case [5]. This storage size is sometimes too expensive. We could also use depth-first algorithm with mark bits over the image. In this case the total storage size is reduced to $O(n)$ bits for an image of n pixels, but we also need storage for recursive calls of the depth-first search.

A question we address in this paper is whether we can solve the problem in a more space efficient manner. That is, can we design an algorithm for erasing a component without using any extra array? An input binary image is given using an array consisting of n bits in total. This is an ordinary bit array. We are allowed to modify their values, but it is hard to use the array to store some useful information to be used in the algorithm since we have only one bit for each pixel.

In the problem above we are requested to enumerate all pixels belonging to the same component as a query pixel. Whenever we find a pixel to be output, we output its coordinates and to prevent duplicate outputs we flip the pixel value to 0. This corresponds to erasing a component containing a query pixel. Thus, our problem is restated as follows.

Problem: Let G be a binary image. Given an arbitrary pixel q in G , erase the component containing the query pixel q . Here, by erasing a component we mean flipping a color of each pixel in the component.

How fast can we erase a connected component? This is a problem we address in this paper.

An algorithm to be presented consists of the following four steps. At the first step a query pixel is specified. Assuming it is a white pixel, we compute the canonical edge e_s of the component containing the query pixel.

Then, at the second step, we follow the external boundary of the component starting from the canonical edge. During the traverse we also try to find a canonical edge of a hole by extending a horizontal ray to the right at each downward edge. Recall that no extra array is available. When we extend the ray to find a boundary edge e , we have to determine whether the edge e is on a hole or not. The decision is done by finding the canonical edge on the boundary to which e belongs. It is on the external boundary if it is the canonical edge e_s found at the first step. Otherwise, it is on a hole. We can perform this test using only constant work space. It takes quadratic time if we just follow the boundaries, but the running time is shortened to $O(n \log n)$ time by using bidirectional search.

Once we find any hole, we traverse it while flipping white pixels on the way and finally returning to the original edge from the canonical edge by walking to the left until we touch any boundary edge.

The above flipping operations remove all the holes and a single connected component is left. At the third step we traverse the boundary again and slim the component into a tree that is one-pixel wide by erasing all possible safe pixels in the component. Here, a pixel p is safe if and only if removal of p (flipping the pixel value of p) does not separate any component within the 3×3 neighborhood around p . Safe pixel check is done in constant time.

The final step is to erase all the pixels in the thinned component. It is rather easier than others.

Now, we will describe the four step in more detail.

Step 1: Locating a given pixel

Given a query white pixel p , we want to locate it in a given binary image. In other words, we want to find a (connected) component of white pixels, which is equivalently to find a canonical edge of the component. For the purpose we first traverse the image horizontally to the left until we encounter a black pixel. The eastern edge e of the pixel is a candidate of the canonical edge. To verify it we follow the boundary starting from the edge whether we encounter any other vertical edge that is lexicographically smaller than e . Here, a vertical edge e is lexicographically smaller than another vertical edge e' if e lies to the left of e'

(more precisely, the x -coordinate of e is smaller than that of e') or both of them lie on the same vertical line but e is below e' on the line. If there is no smaller edge than e then it is certainly the canonical edge of the external boundary we seek. Otherwise, starting from the pixel just to the left of e we perform the same procedure again. Figure 2 illustrates how the canonical edge is found.

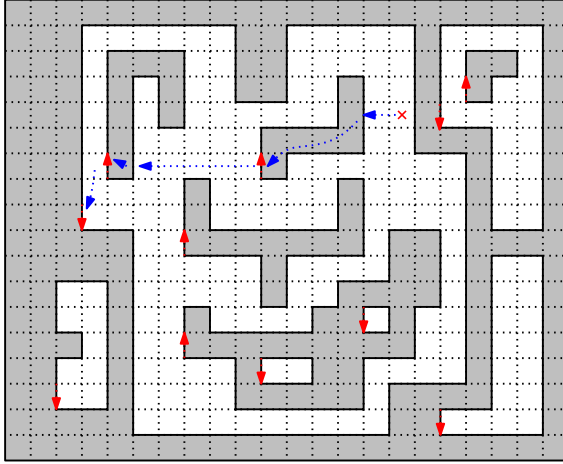


Fig. 2. Finding a canonical edge of a component containing a query point p by following boundaries at the first step of the algorithm

Step 1: Locating a given pixel

Let p be a given white pixel and f be the western edge of the pixel p .

```

do{
    e = ScanLeft(f).
    f = CanonicalEdge(e).
} while(f is an internal edge)
return f.
ScanLeft(e){ // move to the left until we encounter an edge between 0 and 1
    do{
        e = vertical edge just to the left of e.
    } while(e is an edge between 0 and 1)
    return e
CanonicalEdge(f){ // edge f is canonical if no edge on the same boundary
is smaller than f.
    e* = e_s = f.
    do{
        e = nextEdge(e).
        if e < e* then e* = e.
    } while(e ≠ e_s)
    return e*

```

```

nextEdge( $e$ ){
    return the next (uniquely determined) boundary edge of  $e$ .
}

```

Lemma 1. *The algorithm given above finds the canonical edge of the external boundary of a component which contains a query pixel p , in time linear in the size (the number of pixels) of the component.*

Proof. Since the canonical edge is defined to be the leftmost vertical boundary edge, every time when we apply the function `LeftScan()` we move to the left until we reach some boundary. If it happens to be an internal boundary, we follow it to find its canonical edge and then apply `LeftScan()` again from there. Due to the same reason we further move to the left. Thus, eventually we must reach the external boundary. Once we reach it, then it suffices to follow the boundary. Thus, the total running time is linear in the size of the component.

Step 2: Removing holes

At the first step we obtain the canonical edge e_s of the external boundary. At the next step we remove all the holes by merging them to others or to the external boundary. For the purpose we traverse the boundary again. This part is just the same as the algorithm for reporting components with their sizes in our previous paper [4]. That is, we traverse the boundaries starting from the canonical edge of the external boundary.

At each downward edge e , we walk to the right until we encounter a boundary edge f . If we find f a canonical edge after following the boundary, then we move to the hole and continue the traverse again from f . Otherwise, we go back to the edge e and continue the traverse.

At each upward edge e , we check whether it is a canonical edge of a hole by following the boundary. If it is the case, we walk to the left from e until we hit some boundary corner. There are two cases to consider. If we touch just one corner, then we erase the white pixels visited during the walk from e , which merges the hole into the boundary of the corner. If we touch two corners, we erase the white pixels visited during the walk except the last one and also erase the pixel just below the last pixel.

After erasing those pixels we still keep walking to the left until we reach a boundary edge and then start traversing the boundary again from there. Repeating this process until we come back to the starting canonical edge, the second step is done. Figure 3 illustrates behavior of the algorithm.

Step 2: Removing holes

```

 $e = e_s$ : // the canonical edge of the external boundary
do{
     $e = \text{nextEdge}(e)$ .
    if  $e$  is downward then {
         $f = \text{ScanRight}(e)$ .
        Check whether  $f$  is canonical or not.
    }
}

```

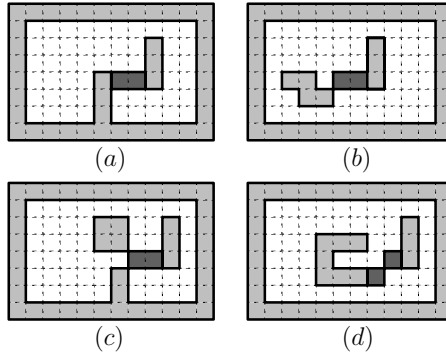


Fig. 3. Step 2: Removing holes. We traverse the boundaries. At each downward edge we check whether its right corresponding edge is canonical or not. If so, we move to the edge. At each canonical edge we walk to the left while erasing pixels until we touch some other boundary. (a) touching external boundary, (b) touching a hole, (c) touching two corners, and (d) touching two corners of the same component.

```

if  $f$  is a canonical edge of a hole then  $e = f$ .
} else if  $e$  is upward and a canonical edge of a hole then{
    traverse to the left from  $e$  until we touch some boundary corner.
    Let  $p_1, \dots, p_k$  be those pixels.
    If the last pixel  $p_k$  touches one corner (or one edge) then
        erase all those pixels.
    If it touches two corners then{
        erase them except the last one  $p_k$  and
        erase the pixel just below  $p_k$ .
    }
    keep traversing to the left until it encounters a vertical boundary edge  $f$ .
 $e = f$ .
}
} while( $e \neq e_s$ )

```

Lemma 2. *The algorithm given above transforms a component with holes into one without any hole in $O(n \log n)$ time, where n is the number of pixels in the component.*

Proof. In the algorithm we traverse the boundaries starting from the canonical edge of the external boundary. At each downward edge e we perform `ScanRight()` until we encounter a vertical boundary edge f and check whether f is a canonical edge of an internal boundary or not. The test is done by bidirectional search. Since the test is done at most twice for each vertical edge, the total time we need is bounded by $O(n \log n)$ (the proof is similar to the one in [3]).

Once we find a canonical edge of an internal boundary, we move to the boundary and continue the traverse. Then, eventually we come back to the canonical edge f again. In the algorithm we check every upward edge whether it is a

canonical edge. If we find such an upward canonical edge f then we walk to the left until we touch some boundary corner. Let p_1, \dots, p_k be a sequence of pixels visited in this walk. If the last pixel p_k touches a single corner, then we can safely erase all these pixels without touching any other boundary. After erasing them the internal boundary which used to contain f is merged into the boundary of the corner. Thus, one hole is removed. If the last pixel p_k touches two corners from different sides, then erasing p_k may cause a trouble. Refer to Figure 3(d). If the two corners belong to the same boundary, then erasing p_k creates a new hole. So, to avoid the situation, we erase p_1, \dots, p_{k-1} and the pixel p' just below p_k . The pixel p' must be a white pixel since otherwise we have touched it before reaching p_k . Since p_k touches two corners from both sides (from above and below), the row of p_k is not the bottom row. Recall that holes are treated using 8-connectivity. So, the boundary is merged into that of the lower corner without creating a new hole.

Step 3: Thinning a component

Now, we can assume that a connected component forms a simple polygon (without any hole). What we do next is to erase fat parts so that a one pixel wide skinny pattern is obtained. Again we traverse the boundary. For each downward edge e we traverse horizontally to the right until we encounter another boundary edge f . During the traverse we erase every **safe** pixel. This process is illustrated in Figure 4.

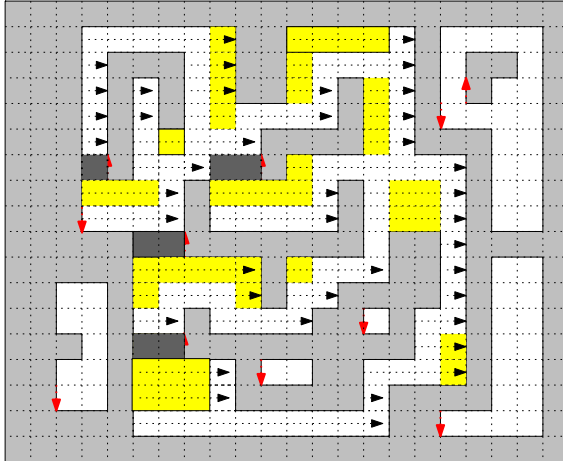


Fig. 4. Extension from the left wall while removing every safe pixel. Colored pixels have been flipped into 0.

Step 3: Thinning a component

e_s = canonical edge of a component, and let $e = e_s$.

do{

$e = \text{nextEdge}(e)$.


```

if  $e$  is downward edge then {
   $p$  = the eastern pixel of  $e$ .
  while( $p$  is a safe pixel of value 1){
    Erase the pixel  $p$  and  $p$  = the right pixel of  $p$ .}
   $e$  = the western edge of the pixel  $p$ .
  while(the pixel value of  $p$  is 1){
    if( $p$  is a safe pixel) then Erase the pixel  $p$ .
     $p$  = the eastern pixel of  $p$ .}
  }
} while( $e \neq e_s$ )

```

Lemma 3. *Implementing the algorithm above results in a slimmed component in which every remaining pixel in the component touches the external boundary.*

Proof. The algorithm scans pixels starting from each downward boundary edge until it reaches the external boundary. Thus, every pixel in the component must be examined. A pixel is erased as far as it is safe. Suppose a pixel p in the component which does not touch any boundary remains in the resulting image. Then, if we walked from the pixel p to the left then we would encounter a boundary edge, which must be downward. The downward edge may be created by a boundary edge further to the left, but there is no reason that the pixel is left without being erased, a contradiction.

Step 4: Erasing the skinny component

Applying the operation we obtain a skinny pattern such that every pixel touches the external boundary as shown in Figure 5. Now, it is not so hard to erase all the pixel in the final pattern. We traverse the boundary again. Whenever the pixel associated with the current edge is a safe pixel, we erase it, and otherwise we leave it as it is. In practice we must be more careful not to miss any pixel. For that we have a procedure to find the next pixel to be handled. The detail is found in the following pseudocode.

Step 4: Erasing the thinned component

e_s = canonical edge of the component.

p = Pixel(e).

$e = e_s$.

```

do{
  do{
     $e$  = nextEdge( $e$ ).
  } while(Pixel( $e$ ) =  $p$ )
   $q$  =  $p$ .
   $p$  = nextPixel( $p$ ).
  if  $p$  is nil then  $p$  = Pixel( $e$ ).
  erase the pixel  $q$ .
} while( $p$  has a neighbor)

```

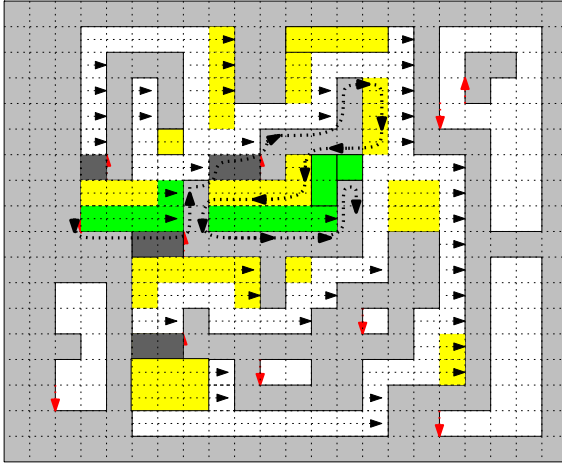


Fig. 5. Erasing the skinny component by removing safe pixels (colored pixels) in order

```

nextPixel( $p$ ){
  if only one pixel  $q$  is 4-adjacent to  $p$  then return  $q$ .
  else return nil.
}

```

Lemma 4. *The algorithm above erases all the pixels in a given component in linear time in the size of the component.*

Proof. In the algorithm we start traversing the boundary from its canonical edge, the leftmost vertical edge. Unless the pixel associated with the edge is a branching pixel (to above and to the right), the pixel is safe and it is erased. If it is a branching point, then we traverse the boundary. It never happens that we reach the starting pixel again without erasing any pixel. The reason is as follows: We know that the component forms a simple polygon consisting of white pixels such that every such pixel touches the boundary. If we define a graph representing adjacency of those white pixels in the component, it must be a tree. So, if we traverse the boundary of the polygon from some edge and come back to the same pixel again, then we must pass some leaf node in the graph. The white pixel corresponding to the leaf node is safe and thus it must have been erased in the algorithm. This means that when we start traversing the boundary, we must have erased at least one pixel before coming back to the same pixel again. Therefore, every pixel must be erased in the algorithm.

Combining the four lemmas above, we have the following theorem.

Theorem 1. *Given a binary image B with n pixels in total and a pixel p , a connected component containing p can be erased in $O(n \log n)$ time using constant work space by flipping pixel values of those pixels in the component.*

Unfortunately we cannot erase a component in linear time since we have to traverse boundaries to find canonical edges of internal boundaries.

4 Some Applications of the Algorithm

4.1 Removing Small Connected Components as Noise

One of basic tasks in image processing on binary images is to remove noise. If we define a noise to be a small component, with size bounded by some small number, we can remove all such noise components by applying our algorithm using only constant work space.

In the first step we scan the entire image and finds every component. Whenever we find a canonical edge, we scan the pixels in the component to compute the size of the component. If it is small, then we apply our algorithm to erase the component. It runs in $O(n \log n)$ time for a binary image with n pixels.

Suppose T is a specified size for small component. If $O(T)$ work space is available, we can do better. We scan the entire image to find a white pixel p such that its left and lower pixels are both black. We grow a white region reachable from p using a queue of size T . If the component is small enough, we can include all the pixels in the component into the queue, and thus it is easy to erase them. Otherwise, the queue will be overflowed. Then, we conclude that the component is a large component, not a noise. Since an enqueue operation takes $O(\log T)$ time to avoid duplication, the algorithm takes $O(n \log T)$ time.

4.2 Region Segmentation

One of the most fundamental tasks for pattern recognition for color images is region segmentation, which partitions a given color image into meaningful regions. A number of algorithms have been proposed so far. Here we simplify the problem. That is, we assume that there is a simple rule for determining whether any two adjacent pixels belong to the same region or not.

In this particular situation we can solve the following problem:

Region Clipping: Given a color image G and an arbitrary pixel p , report all the pixels belonging to a region containing p using a given local rule for determining similarity of pixels.

It is rather easy to design such an algorithm if some sufficient amount of work space is available. What about if only constant work space is available? Well, we can apply our algorithm by assuming a binary image implicitly defined using the given local rule. To clip a region we don't need to convert the whole image into a binary image, but it suffices to convert the region and its adjacent areas into binary data.

Using our algorithm we can collect all regions with some useful information such as areas and average pixel values, etc., as well in $O(n \log n)$ time. If every region is small, then our algorithm runs in almost linear time.

5 Concluding Remarks

This paper has presented an in-place algorithm for erasing an arbitrarily specified component without using mark bits or extra array. The algorithm runs in

$O(n \log n)$ time when the component to be erased consists of n pixels. Since the output is written on an input array and hence the input array allows write as well as read, which is a difference from other constant work space (or log-space) algorithms assuming read-only input arrays. If we are interested only in enumerating all pixels in a component without changing pixel values, it is easier in some sense. An efficient $O(n \log n)$ -time algorithm is known in our unpublished paper [4]. A basic idea for traversing component boundaries is similar to that of traversing planar subdivision in the literature [6,8]. The algorithmic techniques developed here would be useful for other purposes in computer vision.

Acknowledgment

The author would like to express his sincere thanks to Sergey Bereg and David Kirkpatrick for their stimulating discussions and basic ideas. The part of this research was partially supported by the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Scientific Research on Priority Areas and Scientific Research (B).

References

1. Asano, T., Bitou, S., Motoki, M., Usui, N.: In-Place Algorithm for Image Rotation. In: Tokuyama, T. (ed.) ISAAC 2007. LNCS, vol. 4835, pp. 704–715. Springer, Heidelberg (2007)
2. Asano, T.: Constant-Working-Space Image Scan with a Given Angle. In: Proc. 24th European Workshop on Computational Geometry, Nancy, France, pp. 165–168 (2008)
3. Asano, T., Bereg, S., Kirkpatrick, D.: Finding Nearest Larger Neighbors: A Case Study in Algorithm Design and Analysis. In: Albers, S., Alt, H., Näher, S. (eds.) Festschrift Mehlhorn. LNCS, vol. 5760, pp. 249–260. Springer, Heidelberg (2009)
4. Asano, T., Bereg, S., Buzer, L., Kirkpatrick, D.: Binary Image Processing with Limited Storage (unpublished paper)
5. Asano, T., Tanaka, H.: Constant-Working Space Algorithm for Connected Components Labeling. Technical Report, Special Interest Group on Computation, IEICE of Japan (2008)
6. Bose, P., Morin, P.: An improved algorithm for subdivision traversal without extra storage. *International Journal of Computational Geometry and Applications* 12(4), 297–308 (2002)
7. Brönnimann, H., Chen, E.Y., Chan, T.M.: Towards in-place geometric algorithms and data structures. In: Proc. 20th Annual ACM Symposium on Computational Geometry, pp. 239–246 (2004)
8. de Berg, M., van Kreveld, M., van Oostrum, R., Overmars, M.: Simple traversal of a subdivision without extra storage. *International Journal of Geographical Information Science* 11(4), 359–373 (1997)
9. Malgouyres, R., Moreb, M.: On the computational complexity of reachability in 2D binary images and some basic problems of 2D digital topology. *Theoretical Computer Science* 283, 67–108 (2002)
10. Rosenfeld, A.: Connectivity in Digital Pictures. *Journal of ACM* 17(3), 146–160 (1970)

Kaboodle Is NP-complete, Even in a Strip

Tetsuo Asano¹, Erik D. Demaine², Martin L. Demaine², and Ryuhei Uehara¹

¹ Japan Advanced Institute of Science and Technology (JAIST)

Ishikawa 923-1292, Japan

{t-asano, uehara}@jaist.ac.jp

² Computer Science and Artificial Intelligence Lab, Massachusetts Institute of Technology

(MIT), Cambridge, MA 02139, USA

{edemaine, mdemaine}@mit.edu

Abstract. Kaboodle is a puzzle consisting of several square cards, each annotated with colored paths and dots drawn on both sides and holes drilled. The goal is to join two colored dots with paths of the same color (and fill all holes) by stacking the cards suitably. The freedoms here are to reflect, rotate, and order the cards arbitrarily, so it is not surprising that the problem is NP-complete (as we show). More surprising is that any one of these freedoms—reflection, rotation, and order—is alone enough to make the puzzle NP-complete. Furthermore, we show NP-completeness of a particularly constrained form of Kaboodle related to 1D paper folding. Specifically, we suppose that the cards are glued together into a strip, where each glued edge has a specified folding direction (mountain or valley). This variation removes the ability to rotate and reflect cards, and restricts the order to be a valid folded state of a given 1D mountain-valley pattern.

Keywords: Kaboodle, Transposer, silhouette, puzzles, origami.

1 Introduction

Kaboodle: The Labyrinth Puzzle is a puzzle created and developed in 2007 by Albatross Games Ltd., London [1]. This “multi-layer labyrinth” consists of four square cards; see Fig. 1. (In fact, each card is octagonal, but the pattern on it is a square.) Each card has holes drilled in different locations, and various colored paths and dots drawn on both sides. The goal is to arrange the cards—by rotation, reflection, and stacking in an arbitrary order—to create a continuous monochromatic path between the corner dots of the same color that is visible on one side of the stack. The goal of this paper is to understand what makes this puzzle NP-complete, when generalized to n cards instead of four.

Kaboodle is an example of a broader class of puzzles in which patterned pieces with holes must be arranged to achieve some goal, such as monochromatic sides. For example, Albatross Games Ltd. places Kaboodle in a series of puzzles called *Transposers* [2] which all have this style. See [4] for descriptions, and [10] for the relevant patent. Our NP-hardness proofs for Kaboodle immediately imply NP-completeness for this general family of puzzles, though there are likely other special cases of interest.

¹ <http://www.transposer.co.uk/KABpage1.htm>

² <http://www.transposer.co.uk/>



Fig. 1. The four Kaboozle cards and one of the ten solutions

An earlier form of this type of puzzle is a *silhouette puzzle*, where pieces are regions with holes (no pattern beyond opaque/transparent) and the goal is to make a target shape. Perhaps the first silhouette puzzle, and certainly the best known, is the “Question du Lapin” or “Rabbit Silhouette Puzzle”, first produced in Paris around 1900 [7, p. 35]. Fig. 2 shows the puzzle: given the five cards on the left, stack them with the right orientations to obtain one of two different rabbit silhouettes. The puzzle can be played online [3].

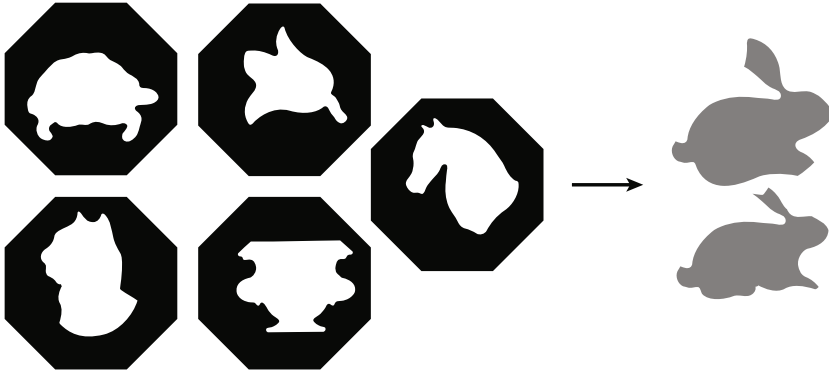


Fig. 2. The classic silhouette puzzle “Question du Lapin”

The freedoms in a silhouette puzzle are reflection and rotation of the cards; the card stacking order has no effect on the silhouette. (In fact, both rabbits can be obtained without reflecting the cards in Fig. 2, so that puzzle only needs rotation.) Are these freedoms enough for NP-completeness? We show that indeed silhouette puzzles are NP-complete, even allowing just rotation or just vertical reflection of the pieces. Furthermore, we show that Kaboozle is NP-complete under the same restriction of just rotation or just vertical reflection.

³ <http://www.puzzles.com/PuzzlePlayground/Silhouettes/Silhouettes.htm>

But is reflection or rotation necessary for Kaboozle to be NP-complete? We show that Kaboozle is NP-complete even when the cards can only be stacked in a desired order, without rotation or reflection. We also show that Kaboozle is NP-complete when restricted to a restricted class of orderings that arise from paper folding, as described below.

Our folding variation of Kaboozle is inspired by a 1907 patent [5] commercialized as the (politically incorrect) “Pick the Pickaninnies” puzzle [8]. This puzzle consists of a single piece, shown on the left of Fig. 2, with holes, images (stars), and crease lines. The goal is to fold along the crease lines to make an array of stars, as shown on the right. This type of puzzle severely limits the valid stacking orders of the parts, while also effectively forbidding rotation and reflection of the parts.

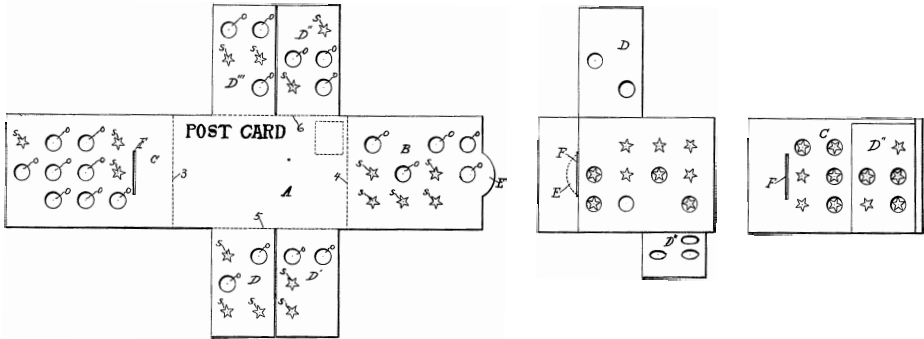


Fig. 3. Puzzle commercialized as “Pick the Pickaninnies”. Figure from [5].

We consider a simple general puzzle along these lines, by restricting a generalized Kaboozle puzzle. Namely, we glue all the cards in the Kaboozle puzzle into a strip, and specify the folding direction (mountain or valley) on each glued edge (crease). Now the only freedom is folding the 1D strip of paper down to a unit size, respecting the folding directions. This freedom is a weak form of the ordering of the cards; rotation and reflection are effectively forbidden.

This idea also comes from problems in computational origami. In polynomial time, we can determine whether a mountain-valley pattern on a 1D strip of paper can be folded flat, when the distances between creases are not all the same [1]. A recent notion is *folding complexity*, the minimum number of simple folds required to construct a unit-spaced mountain-valley pattern (string) [2]. For example, n pleats alternating mountain and valley can be folded in a polylogarithmic number of simple folds and unfolds. On the contrary, the number of different ways to fold a uniform mountain-valley pattern of length n down to unit length is not well-investigated. The number of foldings of a paper strip of length n to unit length has been computed by enumeration, and it seems to be exponentially large; the curve fits to $\Theta(3.3^n)$ [6, A000136]. However, as far as the authors know, the details are not investigated, and it was not known whether this function is polynomial or exponential. Recently, the last author showed theoretical lower and upper bounds of this function: it is $\Omega(3.07^n)$ and $O(4^n)$ [9]. These results imply that

a given random mountain-valley pattern of length n has $\Theta(1.65^n)$ foldings on average, which is bounded between $\Omega(1.53^n)$ and $O(2^n)$.

Intuitively, the folding version of the Kaboozle puzzle seems easy. Perhaps we could apply the standard dynamic programming technique from one side of the strip? But this intuition is not correct. Essentially, the problem requires folding a 1D strip of paper, but the strip has labels which place constraints on the folding. Despite the situation being quite restrictive, we prove the problem is still NP-complete.

Therefore we conclude that the generalized Kaboozle problem is NP-complete even if we allow only one of ordering, rotation, or reflection of the cards, and in the ordering case, even if the ordering comes from a 1D strip folding.

2 Preliminaries

We generalize the number of the Kaboozle cards to $n + 1$. Each *card* is square, with some fragments of a path drawn on both sides, and some holes drilled into it. We will use just one color of path we have to join. The (potential) endpoints of a path are distinguishable from the other fragments. To simplify, we assume that the cards are numbered $0, 1, 2, \dots, n$.

A *strip* of the cards can be constructed as follows: for each $0 \leq i \leq n - 1$, the right side of the card i is glued to the left side of the card $i + 1$, and that side is called the $(i + 1)$ st *crease*. Each crease has a *label* “M” or “V” which means that the strip must be mountain folded or valley folded at the crease. (We define one side of the strip as the *top side*, and creases are mountain or valley folded with respect to this side.) We assume that the label of the first crease is “M” without loss of generality, or otherwise specified. For a strip of the cards, a *folded state* is a flat folding of unit length (where the unit is the width of a card) such that each crease is consistent with its label. (A folded state always exists for any string of labels [9].)

The main problem in this paper is the following:

Input: A strip of $n + 1$ Kaboozle cards, each with a label of length m .

Question: Determine whether the strip has a folded state that is consistent with the labels, and exactly one connected path is drawn on a surface of the folded state.

We begin with an observation for folding a unit pattern:

Observation 1 *A strip of $n + 1$ cards with n creases has a unique folded state if and only if the crease pattern is a pleat, i.e., “MVMV $\cdot\cdot$ MV” or “MVMV $\cdot\cdot$ MVM”.*

Proof. Suppose that a mountain-valley pattern has a unique folded state. Without loss of generality, we assume that the first crease is a mountain. If the second crease is also a mountain, we have two folded states of the cards 1, 2, and 3: 2, 1, 3 and 2, 3, 1. Hence the second crease must be valley. We can repeat the argument for each crease, and obtain the pleat pattern. \square

Using the pleats, we introduce a useful folding pattern for NP-completeness, namely, the *shuffle pattern* of length i : “(MV) $^{i-1}$ MM(VM) $^{i-1}$ ”⁴ By Observation 1, the left and

⁴ Here we use the standard notation x^k for string repetition. For example, “(MV) 3 MM(VM) 3 ” = “MVMVMVMVMVMVM”.

right pleats are folded uniquely and independently. However, these pleats can be combined in any order to fold to unit length. Thus we have $\binom{2^i}{i}$ distinct foldings of the shuffle pattern of length i . We note that the center card of the shuffle pattern of length i , the card $i + 1$ in our notation, always appears on one side of any folded state. We call this side the *top* of the shuffle pattern, and card $i + 1$ the *top card* (although it may come to the “bottom” in a natural folding).

3 NP-completeness of Generalized Kaboozle

It is easy to see that all the problems in this paper are in NP. Hence we concentrate on the proofs of NP-hardness. Our reduction is from the *1-in-3 SAT* problem:

Input: A conjunctive normal form (CNF) Boolean formula $F(x_1, \dots, x_n) = c_1 \wedge c_2 \wedge \dots \wedge c_m$, where each clause $c_i = \ell_1^i \vee \ell_2^i \vee \ell_3^i$ has three literals $\ell_j^i \in \{x_1, \dots, x_n, \bar{x}_1, \dots, \bar{x}_n\}$.

Question: Determine whether F has a truth assignment such that each clause contains exactly one true literal.

This problem is a well-known NP-complete variant of 3-satisfiability [3, LO4].

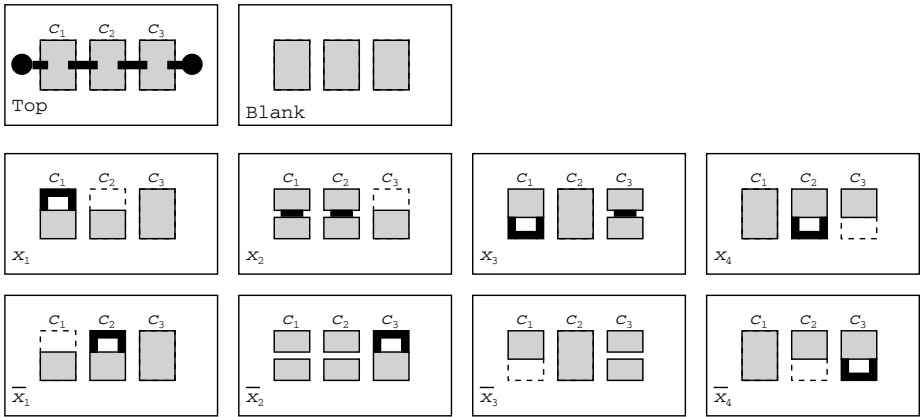


Fig. 4. Example of the reduction for $F(x_1, x_2, x_3, x_4) = (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee x_4) \wedge (\bar{x}_2 \vee x_3 \vee \bar{x}_4)$

For a given CNF formula $F(x_1, \dots, x_n)$ with n variable and m clauses, we use $4n + 1$ Kaboozle cards as follows. Fig. 4 shows an example of the reduction for $F(x_1, x_2, x_3) = (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee x_4) \wedge (\bar{x}_2 \vee x_3 \vee \bar{x}_4)$. Each gray area is a hole in the card, each black line is a fragment of the unique path, and the black circles are the endpoints of the unique path.

Top card: One *top* card is placed at the top of the shuffle pattern, and it represents m clauses. On the top card, two endpoints of the unique path are drawn, and each clause is represented by a hole in the card. Each hole has two dimples corresponding to the borders of the path and that will be extended to one of three possible directions by the variable cards described below.

Variable card: We use $2n$ variable cards. Here, the index i with $1 \leq i \leq n$ is used to represent the i th variable, and the index j with $1 \leq j \leq m$ is used to represent the j th clause. Each card represents either x_i or \bar{x}_i . We make m gadgets on the card for the variable x_i as follows.

If neither x_i nor \bar{x}_i appear in clause c_j , the card x_i has a hole at that place. Hence this card has no influence at that place of clause c_j .

If x_i appears in clause c_j , the card x_i has a part of the path at that place. According to the position (first, second, or third literal) in the clause, the path is depicted at top, center, or bottom, respectively, as shown in Fig. 4.

If \bar{x}_i appears in clause c_j , the card x_i has a *cover area* of the path at that place. This white area covers the corresponding path drawn on the variable card corresponding to \bar{x}_i , as shown in Fig. 4.

Each variable card \bar{x}_i is symmetric to the variable card x_i , and hence omitted.

Blank card: We use $2n$ blank cards depicted in Fig. 4. They will be used to join variable cards and the top card. They have no influence on the appearance of the variable cards.

We first show that generalized Kaboodle is NP-complete, without requiring a strip folding:

Theorem 2. *Generalized Kaboodle is NP-complete, even forbidding reflection and rotation.*

Proof. We use the top card and $2n$ variable cards. Make the cards asymmetric, e.g., by shifting the gadgets on each card a little, to forbid reflecting or rotating the cards (if that is allowed). Clearly, the reduction can be done in a polynomial time.

Because of the pictures of the endpoints of the unique path, the top card must be on top. It is not difficult to see that card x_i has no influence on cards x_j and \bar{x}_j if $i \neq j$. Hence it is sufficient to consider the ordering between each pair x_i and \bar{x}_i for $i = 1, 2, \dots, n$.

When $F(x_1, \dots, x_n)$ has a solution, i.e., each clause c_j contains exactly one true literal ℓ_i^j , the card corresponding to the literal activates one of three parts on the card that joins the two endpoints of the parts of path incident to the hole representing c_j in the top card. For example, consider the (wrong) assignment $x_1 = 0, x_2 = 1, x_3 = 0$, and $x_4 = 1$ for $F(x_1, x_2, x_3, x_4)$ from Fig. 4, as shown in Fig. 5. Then we put the card \bar{x}_1 over the card x_1 , the card x_2 over the card \bar{x}_2 , and so on. Then, the card \bar{x}_1 covers the parts of the path on the card x_1 , the card x_2 covers the parts of the path on the card \bar{x}_2 , and so on. Any two cards corresponding to different variables can be stacked in any order. For example, we can arrange “top”, $\bar{x}_1, x_1, x_2, \bar{x}_2$; “top”, $\bar{x}_1, x_2, \bar{x}_2, x_1$; or “top”, $\bar{x}_1, x_2, x_1, \bar{x}_2$; and so on. For this assignment, the clause $c_1 = (x_1 \vee x_2 \vee x_3)$ satisfies the condition of the 1-in-3 3SAT because only x_2 is true. Hence the hole corresponding to c_1 in the top card is filled and the path is joined properly. On the other hand, all literals are true in the clause c_2 , and no literal is true in the clause c_3 . Hence the hole corresponding to c_2 produces loops and the path is disconnected at the hole corresponding to c_3 .

Therefore, the two endpoints of the path on the top card are joined by one simple path if and only if each c_j contains exactly one true literal. \square

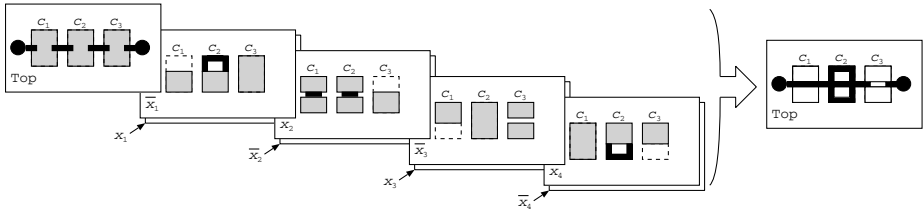


Fig. 5. For $F(x_1, x_2, x_3) = (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee x_4) \wedge (\bar{x}_2 \vee x_3 \vee \bar{x}_4)$, a wrong ordering of the cards that corresponds to a wrong assignment $x_1 = 0, x_2 = 1, x_3 = 0$, and $x_4 = 1$. For this assignment, the first clause c_1 contains one true literal, the second clause c_2 contains three true literals, and the third clause c_3 contains no true literal.

We now turn to the main theorem.

Theorem 3. *Generalized Kaboozle is NP-complete even in a strip with fixed mountain-valley pattern.*

Proof. We use the top card, $2n$ variable cards, and $2n$ blank cards. We join these cards into a strip as “ x_n -b- x_{n-1} -b- \dots -b- x_2 -b- x_1 -b-top-b- \bar{x}_1 -b- \bar{x}_2 -b- \dots -b- \bar{x}_{n-1} -b- \bar{x}_n ”, where “b” means a blank card. Fig. 6 shows the example from Fig. 4. We glue the blank cards upside down, which will be reflected by folding to unit length. The mountain-valley pattern is the shuffle pattern of length n ; that is, the creases on either side of the top card are mountain, and from there, the other creases are defined to form two pleats of length n .

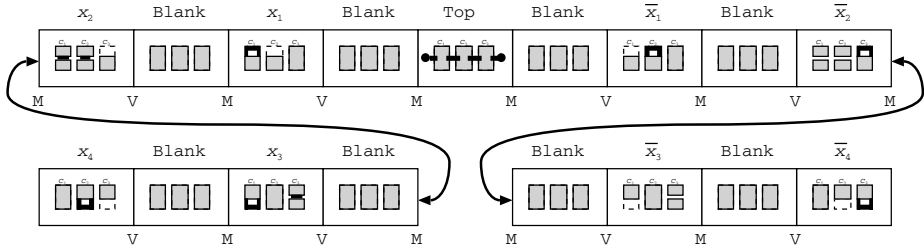


Fig. 6. The cards joined in a strip

Now, the left pleat of the top card makes the sequence of x_i s, and the right pleat makes the sequence of \bar{x}_i s. For each pair of x_i and \bar{x}_i , we can choose the ordering between the corresponding cards with an appropriate shuffling. This means that we can assign true or false to this variable. Moreover, thanks to the blank cards between the variable cards, we can arrange the ordering of the cards x_i and \bar{x}_i independently for each i . Hence, by Theorem 2 and the property of the shuffle pattern, the constructed Kaboozle strip with fixed mountain-valley pattern has a solution if and only if the 1-in-3 3SAT has a solution. \square

Carefully checking the proof of the main theorem, we can also let the mountain-valley pattern be free:

Corollary 1. *Generalized Kaboozle is NP-complete even in the strip form and allowing any mountain-valley pattern.*

Proof. We use the same strip in the proof of Theorem 3. Even if the mountain-valley pattern is not specified, the top card should be on top; otherwise, the endpoints of the path disappear. Hence both creases bordering the top card are mountains. If the 1-in-3 3SAT instance has a solution, the constructed Kaboozle puzzle has a solution by the folding in the proof of Theorem 3. On the other hand, if the Kaboozle puzzle has a solution, we can extract the ordering between x_i and \bar{x}_i for each i with $1 \leq i \leq n$ from the folded state. From these orderings, we can construct the solution to the 1-in-3 3SAT instance. \square

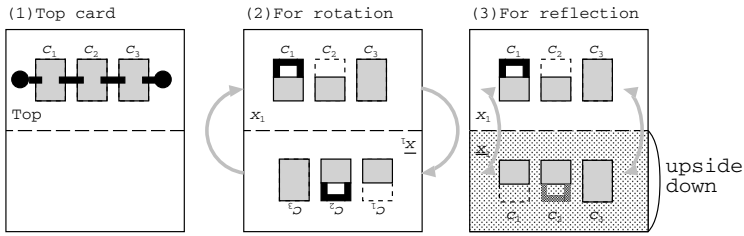


Fig. 7. Gadgets for rotation and reflection

By combining gadgets, we can show that generalized Kaboozle is also NP-complete if we allow only either rotation or reflection. Note that we can rotate a card 180° by the combination of a horizontal reflection and a vertical reflection. To forbid this kind of cheating with cards, we restrict reflection to be vertical.

Theorem 4. *Generalized Kaboozle is NP-complete even if the card ordering is fixed (or free), and (1) only 180° rotation of the cards is allowed, or (2) only vertical reflection of the cards is allowed.*

Proof. As in the proof of Theorem 2, we prepare the top card and $2n$ variable cards. Now, the top card is enlarged to twice of the original cards; see Fig. 7(1).

Rotation: For each variable x_i , two variable cards x_i and \bar{x}_i are glued so that 180° rotation exchanges them; see Fig. 7(2).

Vertical reflection: For each variable x_i , two variable cards x_i and \bar{x}_i are glued so that a vertical reflection exchanges them; see Fig. 7(3).

Then it is easy to see that the ordering of the cards has no influence, except the top card which should be the top, and the resultant Kaboozle has a solution if and only if the 1-in-3 3SAT instance has a satisfying truth assignment. \square

Along similar lines, we can show that silhouette puzzles are NP-complete:

Theorem 5. *Silhouette puzzles are NP-complete even if (1) only 180° rotation of the cards is allowed, or (2) only vertical reflection of the cards is allowed.*

Proof. We reduce from regular (not 1-in-3) SAT, mimicking the gadgets in Fig. 7. The top card has one hole per clause, all in the top half of the card. Each variable card reserves the top and bottom halves for the true and false literals; each side has a solid patch for each clause the literal satisfies, and a hole for all other clauses. As in Fig. 7 the top and bottom sides are rotations or vertical reflections of each other according to the variation. A rectangular silhouette is possible if and only if the formula is satisfiable. \square

Acknowledgement

The authors thank Yoshio Okamoto for helpful discussions.

References

1. Arkin, E.M., Bender, M.A., Demaine, E.D., Demaine, M.L., Mitchell, J.S.B., Sethia, S., Skiena, S.S.: When can you fold a map? *Computational Geometry: Theory and Applications* 29(1), 23–46 (2004)
2. Cardinal, J., Demaine, E.D., Demaine, M.L., Imahori, S., Langerman, S., Uehara, R.: Algorithmic folding complexity. In: Dong, Y., Du, D.-Z., Ibarra, O. (eds.) *ISAAC 2009*. LNCS, vol. 5878, pp. 452–461. Springer, Heidelberg (2009)
3. Garey, M.R., Johnson, D.S.: *Computers and Intractability — A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York (1979)
4. Scherphuis, J.: Jaap’s Puzzle Page: Transposer / Trixy / Stained (2009), <http://www.jaapsch.net/puzzles/trixy.htm>
5. Lehman, F.H.: Puzzle. U.S. Patent 856,196 (June 1907)
6. Sloane, N.J.A.: The On-Line Encyclopedia of Integer Sequences (2010), <http://www.research.att.com/~njas/sequences>
7. Slocum, J., Botermans, J.: *Puzzles Old and New: How to Make and Solve Them*. University of Washington Press (1988)
8. Stegmann, R.: Rob’s Puzzle Page: Folding — Paper/Card (2010), <http://home.comcast.net/~stegmann/allother.htm#fold-paper>
9. Uehara, R.: Stretch minimization problem of a strip paper. In: *5th International Conference on Origami in Science, Mathematics and Education (5OSME)*, Singapore (2010)
10. Chaim Raphael Weinreb. Puzzle. International Patent WO 99/15248 (and GB2345645, EP1021227, US5281986) (April 1999), <http://www.jaapsch.net/puzzles/patents/wo9915248.pdf>

A Hat Trick

Oren Ben-Zwi and Guy Wolfovitz

Department of Computer Science, Haifa University, Haifa, Israel
nbenzv03@cs.haifa.ac.il,
gwolfovi@cs.haifa.ac.il

Abstract. Consider the following game. There are n players, each wearing a hat colored red or blue. Each player does not see the color of her own hat but does see the colors of all other hats. Simultaneously, each player has to guess the color of her own hat, without communicating with the other players. The players are allowed to meet beforehand, hats-off, in order to coordinate a strategy. We give an explicit polynomial time deterministic strategy which guarantees that the number of correct guesses is at least $\max\{n_r, n_b\} - O(n^{1/2})$, where n_r is the number of players with a red hat and $n_b = n - n_r$. This answers a question of Feige.

1 Introduction

A group of n players is gathered, n_r of which wear a red hat and $n_b = n - n_r$ of which wear a blue hat. Every player in the group can see the colors of the hats of the other players, but cannot see and does not know the color of her own hat, a color which has been picked by an adversary. No form of communication is allowed between the players. At the mark of an unseen force, each player simultaneously guesses the color of her hat. The objective of the players as a group is to make the total number of correct guesses as large as possible. In order to achieve this goal, the players are allowed to meet beforehand, hats-off, and agree upon some strategy. Our main result follows.

Theorem 1. *There exists an explicit polynomial time deterministic strategy which guarantees at least $\max\{n_r, n_b\} - O(n^{1/2})$ correct guesses.*

Let us give a few remarks. First, our main result is optimal, in the sense that any deterministic strategy can guarantee only $\max\{n_r, n_b\} - \Omega(n^{1/2})$ correct guesses in the worst case; this was proved by Feige [3] and Doerr [2]. Second, our main result improves a result of Doerr [2] who gave an explicit polynomial time deterministic strategy which guarantees at least $\max\{n_r, n_b\} - O(n^{2/3})$ correct guesses, and a result of Feige [3] who gave a non-explicit deterministic strategy which guarantees at least $\max\{n_r, n_b\} - O(n^{1/2})$ correct guesses. Feige further asked whether there exists an explicit polynomial time deterministic strategy which guarantees this last bound, and our main result answers this question affirmatively. Lastly it should be noted that Winkler [4], who brought the problem to light, gave a simple explicit polynomial time deterministic strategy which guarantees $\lfloor n/2 \rfloor$ correct guesses.

The proof of Theorem [1](#) has two parts. First, we design an explicit polynomial time randomized strategy for the players, a strategy which guarantees that under any hat assignment, the expected number of correct guesses is $\max\{n_r, n_b\} - O(n^{1/2})$. We then derandomize this strategy by giving an explicit polynomial time deterministic strategy that always achieves, up to an $O(n^{1/2})$ additive factor, the expected number of correct guesses of the randomized strategy. We note that the derandomization is based on a generalization of a technique due to Aggarwal et al. [\[2\]](#).

2 Randomized Strategy

Let the players agree in advance on some ordering so that the i th player is well defined and known to all. Under a given hat assignment, let $\chi_r(i)$ be the number of red hats that the i th player sees. Analogously, let $\chi_b(i)$ be the number of blue hats that the i th player sees. Say that a player is red (respectively blue) if she wears a red (respectively blue) hat.

Our strategy is a collection of randomized strategies, one for each player. We describe the strategy of the i th player, Paula. First Paula computes two integers $a(i)$ and $b(i)$, and sets $p(i) = a(i)/b(i)$. If $|\chi_r(i) - \chi_b(i)| \leq 1$, then Paula takes $a(i) = 1$ and $b(i) = 2$, so that $p(i) = 1/2$. Otherwise, $|\chi_r(i) - \chi_b(i)| \geq 2$ and so we have either $\chi_r(i) = n/2 + c$ for some $c > 0$ or $\chi_b(i) = n/2 + c$ for some $c > 0$ (but not both). In the former case Paula takes $a(i) = \min\{\lfloor n^{1/2} \rfloor, \lceil c \rceil\}$ and $b(i) = \lfloor n^{1/2} \rfloor$, so that $p(i) = \min\{1, \lceil c \rceil / \lfloor n^{1/2} \rfloor\}$ and in the latter case she takes $a(i) = \lfloor n^{1/2} \rfloor - \min\{\lfloor n^{1/2} \rfloor, \lceil c \rceil\}$ and $b(i) = \lfloor n^{1/2} \rfloor$, so that $p(i) = 1 - \min\{1, \lceil c \rceil / \lfloor n^{1/2} \rfloor\}$. Note that $a(i)$, $b(i)$ and $p(i)$ can be computed in polynomial time. Having $p(i)$ at hand, Paula draws a uniformly random real p in the unit interval, guesses red if $p \leq p(i)$ and blue otherwise.

Lemma 1. *If each player follows the above strategy then the expected number of correct guesses is at least $\max\{n_r, n_b\} - O(n^{1/2})$.*

Proof. We shall assume throughout the proof that $n_r \geq n_b$; the argument for the other case is symmetric. We consider the following cases.

- $n_r = n_b$. In that case, every player guesses correctly with probability $1/2$. Thus the expected number of correct guesses is $\max\{n_r, n_b\}$.
- $n_r \in \{n_b + 1, n_b + 2\}$. In that case, every red player guesses red with probability $1/2$ and every blue player guesses blue with probability $1 - O(n^{-1/2})$. Thus, the expected number of correct guesses is $n_r/2 + n_b(1 - O(n^{-1/2}))$, which is clearly at least $\max\{n_r, n_b\} - O(n^{1/2})$.
- $n_r \geq n_b + 3$. Let $x > 1$ satisfy $n_r = n/2 + x$, so that $n_b = n/2 - x$. First assume that $\lceil x \rceil \leq \lfloor n^{1/2} \rfloor$. In that case, every red player guesses red with probability $\lceil x - 1 \rceil / \lfloor n^{1/2} \rfloor = (\lceil x \rceil - 1) / \lfloor n^{1/2} \rfloor$, and every blue player guesses blue with probability $1 - \lceil x \rceil / \lfloor n^{1/2} \rfloor$. Therefore, the expected number of correct guesses is

$$\begin{aligned}
 & (n/2 + x)(\lceil x \rceil - 1)/\lfloor n^{1/2} \rfloor + (n/2 - x)(1 - \lceil x \rceil/\lfloor n^{1/2} \rfloor) = \\
 & (n/2 + x)\lceil x \rceil/\lfloor n^{1/2} \rfloor - (n/2 + x)/\lfloor n^{1/2} \rfloor + (n/2 - x)(1 - \lceil x \rceil/\lfloor n^{1/2} \rfloor) \geq \\
 & (n/2 - x)\lceil x \rceil/\lfloor n^{1/2} \rfloor - (n/2 + x)/\lfloor n^{1/2} \rfloor + (n/2 - x)(1 - \lceil x \rceil/\lfloor n^{1/2} \rfloor) \geq \\
 & \qquad \qquad \qquad (n/2 - x) - (n/2 + x)/\lfloor n^{1/2} \rfloor \geq \\
 & \qquad \qquad \qquad n/2 - 4n^{1/2},
 \end{aligned}$$

which is at least $\max\{n_r, n_b\} - O(n^{1/2})$, since $\max\{n_r, n_b\} \leq n/2 + O(n^{1/2})$. Next assume that $\lceil x \rceil > \lfloor n^{1/2} \rfloor$. In that case, every red player guesses her hat correctly with probability 1 and so the expected number of correct guesses is at least $n_r \geq \max\{n_r, n_b\} - O(n^{1/2})$. \square

3 Derandomization

The randomized strategy we gave above has two phases. In the first phase the i th player computes in deterministic polynomial time some number $p(i)$ in the unit interval. Moreover, for some p_r and p_b that depend each only on the number of red hats and the number of blue hats, we have $p(i) = p_r$ if the i th player is red and $p(i) = p_b$ if the i th player is blue. Given the first phase, the second phase guarantees that the expected number of correct guesses is $p_r n_r + (1 - p_b)n_b$, which was shown to be at least $\max\{n_r, n_b\} - O(n^{1/2})$. What we show in this section is that given that for all $1 \leq i \leq n$, the i th player has determined $p(i)$, we can replace the second phase of the randomized strategy by an explicit polynomial time deterministic strategy that guarantees that at least $p_r n_r - O(n^{1/2})$ red players make a correct guess and at least $(1 - p_b)n_b - O(n^{1/2})$ blue players make a correct guess. This will imply Theorem [II](#).

Suppose that for all $1 \leq i \leq n$, the i th player has determined $a(i), b(i)$ and $p(i)$. The following is the strategy that the i th player follows in order to determine her guess.

1. Let $X(i) = \sum_j j$, where the sum ranges over all $j \neq i$ such that the j th player is red.
2. Let $Y(i) = \sum_j 1$, where the sum ranges over all $j < i$ such that the j th player is red.
3. Let $Z(i) = i + X(i) + (b(i) - 1)Y(i) \pmod{b(i)}$.
4. Guess red if $Z(i) < a(i)$, blue otherwise.

Note that the above deterministic strategy can be implemented so that its running time is polynomial in n . This fact together with the next lemma proves Theorem [II](#).

Lemma 2. *Suppose that for all $1 \leq i \leq n$, the i th player has computed $a(i), b(i)$ and $p(i)$. If each player follows the above strategy, then the number of red players that make a correct guess is at least $p_r n_r - O(n^{1/2})$ and the number of blue players that make a correct guess is at least $(1 - p_b)n_b - O(n^{1/2})$.*

Proof. In what follows we make use of the following facts, which follow from the definition of $a(i)$ and $b(i)$ in the previous section. If the i th player and the j th player both have a hat of the same color, then $a(i) = a(j)$ and $b(i) = b(j)$. Furthermore, for all $1 \leq i \leq n$, $1 \leq b(i) \leq 2n^{1/2}$.

Let us first consider the red players. If $0 \leq n_r \leq 1$ then at least $n_r - 1 \geq p_r n_r - O(n^{1/2})$ red players guess red. Assume $n_r \geq 2$. Let $1 \leq i < j \leq n$ be two indices of players so that the i th player's hat and the j th player's hat are both red and furthermore, for all $i < k < j$ we have that the k th player's hat is blue. Let $a(i) = a(j) = a$ and $b(i) = b(j) = b$ so that $p_r = a/b$. We have $i + X(i) = j + X(j)$ and $Y(j) - Y(i) = 1$. Thus $Z(j) - Z(i) = b - 1 \pmod{b}$. This implies that out of each b consecutive red players, a guess red. Thus, since $b \leq 2n^{1/2}$, at least $p_r n_r - O(n^{1/2})$ red players guess red.

Next consider the blue players. If $0 \leq n_b \leq 1$ then at least $n_b - 1 \geq (1 - p_b)n_b - O(n^{1/2})$ blue players guess blue. Assume $n_b \geq 2$. Let $1 \leq i < j \leq n$ be two indices of players so that the i th player's hat and the j th player's hat are both blue and furthermore, for all $i < k < j$ we have that the k th player's hat is red. Let $a(i) = a(j) = a$ and $b(i) = b(j) = b$ so that $p_b = a/b$. We have $X(i) = X(j)$ and $Y(j) - Y(i) = j - i - 1$. Thus $Z(j) - Z(i) = j - i + (b - 1)(j - i - 1) \pmod{b} = b(j - i) - b + 1 \pmod{b} \equiv 1 \pmod{b}$. This implies that out of each b consecutive blue players, $b - a$ guess blue. Thus, since $b \leq 2n^{1/2}$, at least $(1 - p_b)n_b - O(n^{1/2})$ blue players guess blue. \square

References

1. Aggarwal, G., Fiat, A., Goldberg, A.V., Hartline, J.D., Immorlica, N., Sudan, M.: Derandomization of auctions. In: Proceedings of the 37th Annual ACM Symposium on Theory of Computing, pp. 619–625. ACM, New York (2005)
2. Doerr, B.: Integral approximation. Habilitationsschrift. Christian-Albrechts-Universität zu Kiel (2005)
3. Feige, U.: You can leave your hat on (if you guess its color). Technical Report MCS04-03, Computer Science and Applied Mathematics, The Weizmann Institute of Science (2004)
4. Winkler, P.: Games people don't play. In: Wolfe, D., Rodgers, T. (eds.) Puzzlers' tribute: a feast for the mind, A.K. Peters Ltd., Wellesley (2002)

Fun at a Department Store: Data Mining Meets Switching Theory

Anna Bernasconi¹, Valentina Ciriani², Fabrizio Luccio¹, and Linda Pagli¹


¹ Dipartimento di Informatica, Università di Pisa, Italy
{annab,luccio,pagli}@di.unipi.it

² Dipartimento di Tecnologie dell'Informazione,
Università degli Studi di Milano, Italy
valentina.ciriani@unimi.it

Abstract. In this paper we introduce new algebraic forms, SOP^+ and $DSOP^+$, to represent functions $f : \{0, 1\}^n \rightarrow \mathbb{N}$, based on arithmetic sums of products. These expressions are a direct generalization of the classical SOP and DSOP forms. We propose optimal and heuristic algorithms for minimal SOP^+ and $DSOP^+$ synthesis. We then show how the $DSOP^+$ form can be exploited for Data Mining applications. In particular we propose a new compact representation for the database of transactions to be used by the LCM algorithms for mining frequent closed itemsets.

Keywords: SOP, Implicants, Data Mining, Frequent Itemsets, Blulife.

Consider a department store with a very good body care department. Among many products, the following are on demand:

a - *Algesiv*: adhesive for dental plates, **b** - *Blulife*: spray for breath with a floral scent, **c** - *Crinagen*: lotion against hair loss, **d** - *Deocontrol*: drops for feet odor control, **e** - *Earbeauty*: spoon for taking out earwax, **f** - *Fleastop*: powder against flea invasion, **g** - *Gluttonase*: tablets for stomachache, **h** - *Haltmuc*: tampon for nasal mucus, **i** - *Itchand*: plastic hand for back scratching, **j** - *Johnheaven*: toilet deodorant 

Now, take a look at the customers' baskets (called *transactions* in the following). No wonder that most customers buying *b* also buy *g* and occasionally *j*; or that the ones buying *f* quite often buy *i* as well. It is also understandable that several people tend to buy *b*, *d*, and *e* together, and is not uncommon to find *a* and *c* in the same basket, where, on the other hand, *e* seldom appears. But seems to be a mystery why no-one buying *h* also buys *c*.

Studying associations among the items occurring jointly in a set of transactions is of paramount importance for conducting business of many kinds. Data mining is the main area in which such studies have been developed, where knowledge of the phenomena involved and related computational methods have

¹ Some of these products are actually on the market, some are due to the authors' fantasy. Just guess.

reached maturity through a wealth of significant publications, e.g. see [10,75], or the comprehensive bibliography of [3]. A point that has probably to be examined in more depth is the joint study of the items present *and absent* in a transaction, as brought to the general attention in [8].

In this work we propose an innovative way at describing and processing transactions, whose origin comes from the theory of digital circuits. The two *products*: $\bar{a}b\bar{c}\bar{d}\bar{e}\bar{f}g\bar{h}\bar{i}\bar{j}$ and $\bar{a}b\bar{c}\bar{d}\bar{e}\bar{f}g\bar{h}\bar{i}j$ will indicate the two transactions consisting of items b, g and b, g, j , respectively. The product (same as before, with a missing variable j) $\bar{a}b\bar{c}\bar{d}\bar{e}\bar{f}g\bar{h}\bar{i}$ is obtained as the *sum* of the two products above and accounts for the pair. I.e., products of variables will represent transactions or sets of transactions, where the absence of items, corresponding to *complemented* variables, appears as a byproduct of the notation. Two points, however, have to be clarified immediately. As the possible items are generally many more than the ones contained in a transaction, an efficient notation to avoid long chains of complemented variables in a product must be adopted. Second, the same transaction, or set of transactions, may appear more than once. In this case an integer coefficient will be appended to the corresponding product, thus requiring an extension of switching theory from Boolean to integer algebra. This will also lead to the introduction of a new problem in circuit design, and of a computational method for its solution.

1 Functions $\{0, 1\}^n \rightarrow \mathbb{N}$ and SOP⁺ Forms

Consider a space $\{0, 1\}^n$ on n variables x_1, \dots, x_n that take the values $0, 1 \in \mathbb{N}$ (note that we are not dealing with Boolean variables). A *point* $x \in \{0, 1\}^n$ is represented by an assignment of values to x_1, \dots, x_n . For simplifying the notation, the variables will be possibly indicated as a, b, c, \dots . We shall study functions $f : \{0, 1\}^n \rightarrow \mathbb{N}$, i.e., for each point x the value of $f(x)$ is an arbitrary natural number.

As in Boolean algebra, a *literal* is a variable in *direct* or *complemented* form, with obvious meaning. A *product* (of literals) p is an elementary function. Although we use multiplication instead of AND, the value of p is formally as in Boolean algebra since $0 \cdot a = 0$, $1 \cdot a = a$. We then have $p(x) = 0$ or $p(x) = 1$, depending on x . If a product p contains k literals, the points $x \in \{0, 1\}^n$ such that $p(x) = 1$ form a subspace (or *cube*) $\{0, 1\}^{n-k}$.

For $k \in \mathbb{Z}^+$, let $kp = p + p + \dots + p$ (k terms, with $+$ denoting addition). We have $kp(x) = 0$ for $p(x) = 0$, $kp(x) = k$ for $p(x) = 1$. The expression $k_1p_1 + \dots + k_r p_r$, $r \geq 1$, is called a *SOP⁺ form*, and k_i is called the *multiplicity* of p_i . For a given SOP⁺ form and a given $x \in \{0, 1\}^n$, let (A, B) be the partition of $\{1, \dots, r\}$ such that $p_i(x) = 0$ for $i \in A$ and $p_i(x) = 1$ for $i \in B$. The value (or *weight*) of the form in x is then $k_1p_1(x) + \dots + k_r p_r(x) = \sum_{i \in B} k_i$.

Definition 1. Given a function f , a SOP⁺ form for f (shortly, SOP⁺ for f), denoted by $S^+f = k_1p_1 + \dots + k_r p_r$, is such that: $S^+f(x) = 0$ for all x such that $f(x) = 0$; $S^+f(x) \geq f(x)$ for all x such that $f(x) > 0$.

Note that if f has values only in $\{0, 1\}$, a SOP^+ form for f with all products with multiplicity 1 is formally identical to a SOP form in Boolean algebra. We now state:

- A *minimal SOP⁺* for f is one with minimum weight $\sum_{i=1}^r k_i$.
- A *strictly minimal SOP⁺* for f is a minimal SOP^+ where the total number of literals in p_1, \dots, p_r is minimum.

We now borrow some concepts and terminology from switching theory. Given a function f , a product p whose non-zero points are also non-zero points in f is called an *implicant* of f . Note that this condition is independent of the relative values of f and p wherever both are non-zero. All the products in a SOP^+ form for f are implicants of f .

A function f of up to six variables can be represented by an obvious extension of a Karnaugh map. Implicants are represented in the map as in switching theory. As a working example, a function f of four variables a, b, c, d is represented in Figure 1(A), together with three of its implicants.

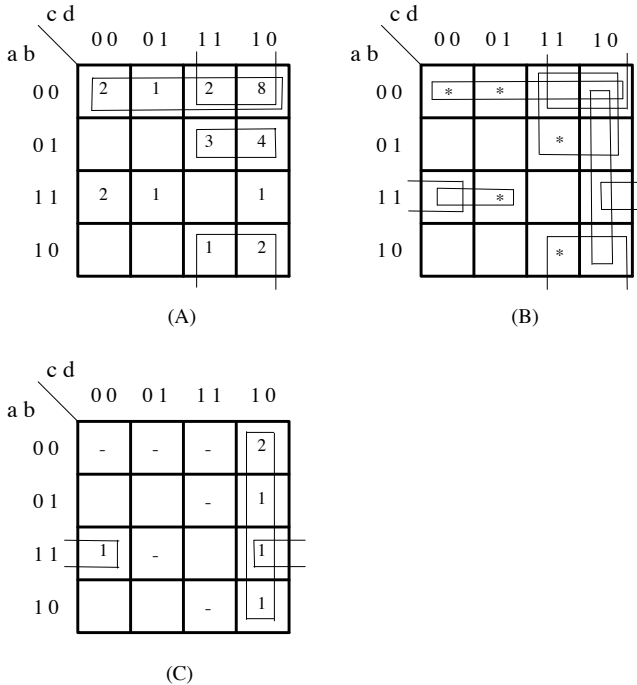


Fig. 1. (A) A function f and three of its implicants $\overline{a}\overline{b}$, $\overline{b}c$, $\overline{a}bc$. The third one is not prime. (B) All the prime implicants of f . Four of them are also essential, due to the starred points. The terms $2\overline{a}\overline{b}$, $3\overline{a}c$, $\overline{b}c$, $ab\overline{c}$ are inserted in a minimal SOP^+ . (C) The new values of the points of f . Some of the original prime implicants, now containing don't cares, may be removed. In the present case only two implicants remain, and a minimal SOP^+ is completed with the inclusion of $2c\overline{d}$, $ab\overline{d}$. In conclusion we have: $S^+ f = 2\overline{a}\overline{b} + 3\overline{a}c + \overline{b}c + ab\overline{c} + 2c\overline{d} + ab\overline{d}$.

2 Minimal SOP⁺ Synthesis

As in switching theory, a *prime implicant* p of a function f is such that no other implicant of f “covers” p . A prime implicant p is *essential* if it contains at least one *essential point*, i.e., a non-zero point of f covered by p and by no other prime implicant of f . In our working example, all the prime implicants of f are shown in Figure 1(B). Four of them are essential, due to the essential points marked with stars. A minimal (or a reasonably low weight) SOP⁺ is found with the following procedure.

Procedure 1

1. Start with an empty S^+f .
2. Determine the set P of all the prime implicants of f , as in switching theory. I.e., for this step all the non-zero points of f can be set to 1.
3. Consider the subset E of all essential prime implicants (if any). For each $e \in E$, consider the essential point where f has maximal value k . Assign (temporary) multiplicity k to e and include ke in S^+f .

In Figure 1(B), for example, implicant $\bar{a}\bar{b}$ covers two essential points where f has values 2 and 1. The term $2\bar{a}\bar{b}$ is then inserted into S^+f . Note that the multiplicity 2 of $\bar{a}\bar{b}$ may be increased in later steps.

4. For all the points covered by each essential implicant e , decrease the value of f of an amount equal to the multiplicity assigned to e in S^+f . If a point reaches a value ≤ 0 , mark it as a don't care condition.

In the example of Figure 1, the points covered by implicant $\bar{a}\bar{b}$ assume the values 0, -1, -4, 2 left to right (the last two points are covered by $2\bar{a}\bar{b}$, $3\bar{a}c$, and $\bar{a}c$). From now on, the first three points will be treated as don't cares.

5. The resulting points must be covered as it is done for an incompletely specified function. For this purpose only the points with non-zero values must be covered. Remove from consideration each original prime implicant p covering non-zero points all covered by another prime implicant p' . Then iterate Steps 3 to 5.

In Figure 1(C), implicants $\bar{a}\bar{b}$, $\bar{a}c$, $\bar{b}c$, $ab\bar{c}$ have been removed, because their non-zero points are also covered by $c\bar{d}$ or $ab\bar{d}$. These two implicants become essential and are selected, the first one with multiplicity 2. In this simple case the function has been completely covered and we have: $S^+f = 2\bar{a}\bar{b} + 3\bar{a}c + \bar{b}c + ab\bar{c} + 2c\bar{d} + ab\bar{d}$. The total weight of the form is 10.

6. The prime implicants remaining at this point (if any) are all non essential. Note that some of them may cover don't care points. The synthesis then proceeds in two possible ways. One, heuristic, gives rise to a reasonable but possibly non minimal solution. The other, enumerative in nature, yields a minimal form at a cost of a possibly exponential number of steps. Let Q be the current set of prime implicants.

- (a) Heuristic. Select a prime implicant $p \in Q$ where the smallest value s of the covered non-don't care points is maximal over all the other implicants in Q . Treat p as if it were essential, assigning to it a multiplicity s . Insert sp in S^+f and restart from Step 4.

- (b) Enumerative. Select a prime implicant $p \in Q$ where the greatest value g of the covered non-don't care points is minimal over all the other implicants in Q . Take the following $g + 1$ alternatives, and for each one of them restart from Step 4. Alternative 0: eliminate p from Q . Alternative i , with $i = 1, 2, \dots, g$: treat p as it were essential, assigning to it a multiplicity i and inserting ip in $S^+ f$.

For finding a strictly minimal SOP^+ , only two minor corrections are needed in Procedure 1. Namely, in Step 5 implicant p is removed only if the number of literals in p is greater than or equal to the number of literals in p' . In Step 6(a), if the same value s is found in several implicants, select one of them with minimum number of literals. For the sample function of Figure 1 nothing changes in the selection of implicants.

Theorem 1. (Correctness). *For any function f , Procedure 1 generates a SOP^+ form (heuristic version, Step 6(a)), or a minimal SOP^+ form (enumerative version, Step 6(b)), or a strictly minimal SOP^+ form (with Step 5 corrected as indicated).*

The time needed by the procedure is studied as in standard switching synthesis. Both the heuristic and the enumerative versions may be exponential in the number n of variables, because the space where f is defined has 2^n points, and possibly all such points must be examined. However, the number N of prime implicants of f may be exceedingly large, and the time is usually evaluated also as a function of N . An immediate extension of a well known result in switching theory indicates that the minimal SOP^+ synthesis is an NP-hard problem. In fact, from an elementary analysis of Procedure 1 we easily have:

Theorem 2. (Complexity). *The time needed by Procedure 1, in the worst case, is polynomial in N (heuristic version), and exponential in N (enumerative version).*

3 Disjoint SOP^+ Forms

We are now interested in covering the points of f “exactly”. That is, we look for a sum of implicants whose weight, in any point x , is equal to $f(x)$. Borrowing a term from switching theory, we shall speak of a *disjoint* form.

Definition 2. *A disjoint SOP^+ form for a function f (shortly, $DSOP^+$ for f), denoted by $DS^+ f = k_1 p_1 + \dots + k_r p_r$, is such that: $DS^+ f(x) = f(x)$ for all x .*

If f has values only in $\{0, 1\}$, a $DSOP^+$ form for f is formally identical to a $DSOP$ form in Boolean algebra [2]. As for the SOP^+ forms, a *minimal $DSOP^+$ for f* is one with minimum weight $\sum_{i=1}^r k_i$. And a *strictly minimal $DSOP^+$ for f* is a minimal $DSOP^+$ where the total number of literals in p_1, \dots, p_r is minimum. However we shall consider only $DSOP^+$ forms with a reasonably low weight, due to the high time needed for determining the minimal forms. We then propose the following heuristic procedure:

Procedure 2

1. Start with an empty $DS^+ f$.

2. Determine a heuristic SOP⁺ form with Procedure 1. Let P be the set of all the implicants contained in this form.

For the function of Figure 1 take the SOP⁺ form already found, where $P = \{\bar{a}\bar{b}, \bar{a}c, \bar{b}c, ab\bar{c}, c\bar{d}, ab\bar{d}\}$.

3. Let $P = \{p_1, \dots, p_r\}$. For each p_i consider the point (or the points) covered by p_i where f has minimal value k_i . Then select one implicant $p \in P$ which maximizes such a value, say k (i.e., k is maximal among all k_i). If more implicants have the same value k , select p with minimal number of literals. Assign multiplicity k to p , include kp in DS^+f , and decrease by k the value of f in all the points covered by p (clearly at least one point in p will get value 0). Remove, from P , implicant p and all the implicants covering a point with value 0.

For the function of Figure 1 we have $k = 2$ in $\bar{a}c$, and $k = 1$ in all the other implicants of P , hence $2\bar{a}c$ is included in DS^+f . Then the value of f is decreased by 2 in the points covered by $\bar{a}c$, and the situation of Figure 2(A) results. The occurrence of value 0 in point 0011 causes the elimination of $\bar{a}c, \bar{a}\bar{b}, \bar{b}c$ from P and we have $P = \{ab\bar{c}, c\bar{d}, ab\bar{d}\}$.

4. Repeat Step 3 until there are prime implicants in P . In our example we select $c\bar{d}$ with multiplicity 1, remove $ab\bar{d}$ from P , and select $ab\bar{c}$ with multiplicity 1, to end up with P empty. The new values for the points of f are shown in Figure 2(B).
5. If there are still points to cover (i.e., points with a non-zero value), restart for them from Step 2.

In our example the new set $P = \{\bar{a}\bar{b}\bar{c}, \bar{a}\bar{b}\bar{d}, \bar{a}bc, \bar{a}c\bar{d}, \bar{b}c\bar{d}, \bar{a}bc, ab\bar{c}\bar{d}\}$ of prime implicants is indicated in Figure 1(B). $\bar{a}\bar{b}\bar{c}, \bar{a}bc, \bar{a}bc$ and $ab\bar{c}\bar{d}$ are selected with multiplicity 1. Then P is newly evaluated in Step 2, yielding the only implicant $\bar{a}\bar{b}c\bar{d}$ that is selected with multiplicity 5. We have the final form: $DS^+f = 2\bar{a}c + ab\bar{c} + c\bar{d} + \bar{a}\bar{b}\bar{c} + \bar{a}bc + \bar{a}bc + ab\bar{c}\bar{d} + 5\bar{a}\bar{b}c\bar{d}$, with total weight 13.

4 DSOP⁺ Forms and Data Mining

In this section we show as DSOP⁺ forms can find an interesting application to data mining, and in particular to one of its most important problems: the mining of (maximal, closed) frequent itemsets.

Let \mathcal{I} be a set of *items*, and \mathcal{D} a database of *transactions*, where each transaction $t \in \mathcal{D}$ is a subset of \mathcal{I} . The total size of the database \mathcal{D} is denoted by $\|\mathcal{D}\|$ and defined as $\|\mathcal{D}\| = \sum_{t \in \mathcal{D}} |t|$, where $|t|$ denotes the cardinality of t , i.e., the number of items in the transaction t . A subset I of \mathcal{I} is called an *itemset*. Given an itemset I , $\mathcal{D}(I)$ denotes the set of transactions including I , i.e., $\mathcal{D}(I) = \{t \in \mathcal{D} \mid I \subseteq t\}$. The cardinality of $\mathcal{D}(I)$ is called the *support* of I w.r.t. \mathcal{D} , and it is denoted as $\text{supp}_{\mathcal{D}}(I)$. For a given support threshold σ , an itemset I is *frequent* if $\text{supp}_{\mathcal{D}}(I) \geq \sigma$. If a frequent itemset I is included in no other frequent itemset J , I is called *maximal*. An itemset I is called *closed* if there exists no itemset J , with $I \subset J$, such that $\mathcal{D}(I) = \mathcal{D}(J)$. Given a set $\mathcal{S} \subseteq \mathcal{D}$ of transactions, let $\mathcal{I}(\mathcal{S})$ be the set of items common to all transactions in \mathcal{S} , i.e.,

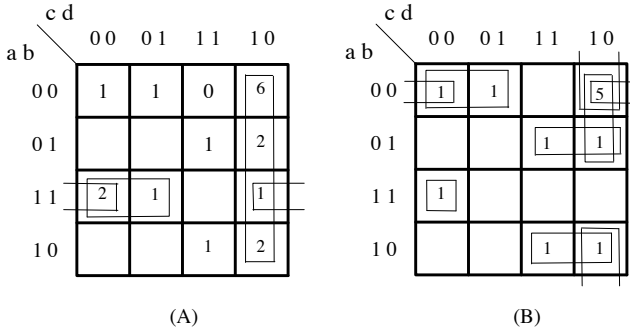


Fig. 2. (A) The function of Figure 1 after the essential prime implicant $\bar{a}c$ has been selected with multiplicity 2. Only three of the original prime implicants remain in P . (B) The new values of the points of f after $ab\bar{c}$ and $c\bar{d}$ have been selected with multiplicity 1, and the new prime implicants after Procedure 1 has been called again. In conclusion we have: $DS^+ f = 2\bar{a}c + ab\bar{c} + c\bar{d} + \bar{a}\bar{b}\bar{c} + \bar{a}bc + a\bar{b}c\bar{d} + 5\bar{a}b\bar{c}\bar{d}$.

$\mathcal{I}(S) = \bigcap_{t \in S} t$. The *closure* of an itemset I , denoted by $clo(I)$, is defined as the set of items common to all transactions in $\mathcal{D}(I)$: $clo(I) = \mathcal{I}(\mathcal{D}(I)) = \bigcap_{t \in \mathcal{D}(I)} t$. Observe that $clo(I)$ is the unique smallest closed itemset including I . Moreover, an itemset I is closed if and only if $clo(I) = I$.

We can now formally state the problem of mining frequent itemsets as follows: *Given a database of transactions \mathcal{D} and an arbitrary integer support threshold σ , enumerate all frequent itemsets in \mathcal{D} .* Depending on the threshold σ , the number of frequent itemsets can be exponential in the database size $\|\mathcal{D}\|$, therefore some variants of this classical problem have been studied in order to reduce the output size, such as the mining of *frequent closed* itemsets, or the mining of *frequent maximal* itemsets. The first variant of the problem is of particular interest since all frequent itemsets and their supports can be computed from the set of frequent closed itemsets and their supports, so that there is no loss of information if mining only closed itemsets. However, the number of frequent closed itemsets, which is often much smaller than the number of frequent itemsets, can still be exponential in the database size [14]. Therefore, a new variant of the classical problem of mining frequent closed itemsets has been introduced and studied: the mining of the top- K frequent closed itemsets, where the threshold σ is chosen as the maximum value yielding at least K frequent closed itemsets [7, 13].

Many algorithms have been proposed for enumerating all frequent (closed, maximal) itemsets. Among these algorithms, *Linear time Closed itemset Miner (LCM)* [9, 10, 11, 12] is one of the most efficient; indeed, its time complexity is theoretically bounded by a linear function in the number of linear closed itemsets, while other existing algorithms are not.

Our idea is to represent the database of transactions as a DSOP⁺ form. As shown in the following, it is indeed possible to derive a DSOP⁺ form that represents the database \mathcal{D} without loss of information. A fundamental advantage of this representation is that the DSOP⁺ form is in general more compact in size

than \mathcal{D} . Moreover, the concepts of frequent/closed/maximal itemsets, and the classical operations on \mathcal{D} , such as computing the closure of an itemset or counting its frequency, can be easily reformulated in this setting. Thus, all algorithms, included LCM, can be reformulated and applied in this new framework.

Finally, another advantage of our new formulation is the possibility of investigating and mining not only sets of items that are often present in the transactions of \mathcal{D} , but also items that are often absent. The importance of considering also absent items has been first outlined in [8]. In their seminal paper they developed a more general setting where standard association rules become just one type of the possible recurring patterns that can be identified from a data set. In particular, correlations can be discovered, and the presence or absence of items forms a basis for generating rules.

4.1 Database Representation

First of all, we observe that a database \mathcal{D} of transactions can be represented, without loss of information, as an integer valued function $f_{\mathcal{D}}$ depending on n variables x_1, \dots, x_n that take the values 0 or 1: $f_{\mathcal{D}} : \{0, 1\}^n \rightarrow \mathbb{N}$. In fact, each transaction $t \in \mathcal{D}$ can be represented by its n -bit characteristic vector $x_1 \dots x_n$, where, for all $1 \leq i \leq n$, $x_i = 1$ if the item $i \in \mathcal{I}$ is included in the transaction t , and $x_i = 0$, otherwise. Thus, for all $(x_1, \dots, x_n) \in \{0, 1\}^n$, $f_{\mathcal{D}}(x_1, \dots, x_n)$ is defined as the number of transactions $t \in \mathcal{D}$ whose characteristic vector is $x_1 \dots x_n$.

We can now use for $f_{\mathcal{D}}$ terms and concepts of switching theory, and in particular we can apply the heuristic procedure described in Section 3 to derive $DS^+ f_{\mathcal{D}}$, i.e., a minimal DSOP⁺ form representing $f_{\mathcal{D}}$. Observe that there is no information loss representing \mathcal{D} as a disjoint SOP⁺ form:

Proposition 1. *$DS^+ f_{\mathcal{D}}$ represents exactly the database \mathcal{D} of transactions over the set of items \mathcal{I} .*

Observe that each implicant p in $DS^+ f_{\mathcal{D}}$ represents a subset of transactions of \mathcal{D} . In fact, p possibly covers more points of $f_{\mathcal{D}}$. Precisely, if p contains ℓ literals, then p represents a cube of dimension $n - \ell$ and covers $2^{n-\ell}$ points.

Let us consider our working example of the function f , depending on four variables a, b, c, d , represented in Figure 1. We can view f as the function representing a database \mathcal{D} of transactions on the set of items $\mathcal{I} = \{a, b, c, d\}$. The transactions in \mathcal{D} are the empty transaction \emptyset , occurring twice, the transactions $\{d\}$, $\{a, b, c\}$, $\{a, b, d\}$, and $\{a, c, d\}$, each occurring once, the transactions $\{a, b\}$, $\{a, c\}$, and $\{c, d\}$ occurring twice, $\{b, c, d\}$ occurring three times, the transaction $\{b, c\}$ occurring four times, and $\{c\}$ occurring 8 times. Thus, \mathcal{D} can be represented by the DSOP⁺ computed for f : $DS^+ f = 2\bar{a}c + ab\bar{c} + c\bar{d} + \bar{a}\bar{b}\bar{c} + \bar{a}bc + a\bar{b}c + ab\bar{c}\bar{d} + 5\bar{a}\bar{b}c\bar{d}$. For instance, the implicant $p = c\bar{d}$ represents the cube of dimension 2 in $\{0, 1\}^4$ given by the four points 0010, 0110, 1010, 1110. Thus, p represents the four transactions $\{c\}$, $\{b, c\}$, $\{a, c\}$, $\{a, b, c\}$. The literals in p represent items always present or always absent in these transactions: *Crinagen* (c) is always present, while *Deocontrol* (d) is always absent. *Algesiv* and *Bluelife*, i.e., a and b , appear in all possible combinations, and are called *don't care items*.

Let us now evaluate the size of the DSOP⁺ representation of a database \mathcal{D} , i.e., the size of $DS^+ f_{\mathcal{D}}$.

Definition 3. *The total size of a $DS^+ f_{\mathcal{D}} = \sum_{i=1}^r k_i p_i$, where k_i denotes the multiplicity of the implicant p_i , is*

$$\|DS^+ f_{\mathcal{D}}\| = \sum_{i=1}^r k_i |p_i|,$$

where $|p_i|$ is the number of literals in p_i .

For instance, the size of $DS^+ f = 2\bar{a}c + ab\bar{c} + c\bar{d} + \bar{a}\bar{b}\bar{c} + \bar{a}bc + a\bar{b}c + ab\bar{c}\bar{d} + 5\bar{a}\bar{b}c\bar{d}$ is 42, while the traditional representation of the database \mathcal{D} has size 47.

In this new framework, we explicitly represent the items that are always absent in a cube of transactions, while the traditional representation lists only the present items. The explicit representation of absent items could be not advantageous. For example, if the transactions in \mathcal{D} are very sparse, as in many practical data mining problems, and thus the majority of implicants in $DS^+ f_{\mathcal{D}}$ have small dimension, the proposed representation could result much larger than $\|\mathcal{D}\|$.

In order to overcome this problem, we propose an alternative representation of the implicants, where only present and don't care items are explicitly represented. We denote with \tilde{x} an item x that is a don't care in an implicant. For instance, the implicant $c\bar{d}$ is now represented as $\tilde{a}\tilde{b}c$. The new representation (denoted as $DS^+ \tilde{f}_{\mathcal{D}}$) uses don't care literals instead of negative ones. The size of an implicant p is now given by its number of positive and don't care literals.

The representation of our running example now becomes $DS^+ \tilde{f} = 2\tilde{b}c\tilde{d} + ab\tilde{d} + \tilde{a}\tilde{b}c + \tilde{d} + bc\tilde{d} + ac\tilde{d} + ab + 5c$, and its size reduces to 26.

The size of this new representation is always less or equal to the size of the traditional database representation, as shown in the following proposition.

Proposition 2. $\|DS^+ \tilde{f}_{\mathcal{D}}\| \leq \|\mathcal{D}\|$.

Proof. First suppose that each implicant in $DS^+ \tilde{f}_{\mathcal{D}}$ represents a single transaction (i.e., no merge occurred during the synthesis). In this case the size of our representation is equal to the size of \mathcal{D} , since each implicant is described using only the positive literals, as in the traditional notation.

Suppose now that at least a merge occurred during the synthesis, producing a cube of dimension d . This means that 2^d transactions are now represented by just one implicant. This implicant is described by d don't care literals, and $t \leq n - d$ positive literals. The transactions composing this cube have the following structure: they all contain the t positive literals representing items always present, together with all possible combinations of the d don't care literals representing don't care items. Recall that the negative literals are not represented. The size of these 2^d transactions is equal to $2^d t + 2^{d-1} d$, as it can be easily verified by induction. The size of the merged implicant ($d + t$) is therefore always less or equal to this value for $d \geq 0$ and $t \geq 0$.

Note that this new notation can be directly used in the synthesis heuristics.

4.2 LCM over DSOP⁺

Data mining algorithms usually store and maintain the input database \mathcal{D} using *FP-trees*, which are a particular version of a trie [4]. Other algorithms use even more sophisticated graph based data structures, e.g., *Zero-Suppressed BDDs* [6], while the first implementations of LCM algorithms [9,10,11] simply use arrays.

In our new approach, we represent cubes of transactions, instead of simple transactions. However, the description of a cube can be maintained and stored exactly as a single transaction. Therefore, the previous data structures can be used also for $DS^+ \tilde{f}_{\mathcal{D}}$, where don't care literals can be handled as new literals. Since our representation is smaller than \mathcal{D} , the applied data structures can be even more compact.

We now briefly show how to exploit $DS^+ \tilde{f}_{\mathcal{D}}$ in the classical LCM algorithm for the mining of frequent closed itemsets. The LCM algorithm is based on the *prefix preserving extension* property, which guarantees that any closed itemset is generated by the extension of exactly one other closed itemset. Thus, all frequent closed itemsets can be enumerated in a depth-first manner.

The basic operations involving itemsets are the frequency computation and the closure of a single itemset. These operations can be easily reformulated for the $DS^+ \tilde{f}_{\mathcal{D}}$ representation of the database. As shown in the following, each operation will manipulate implicitly the transactions in an implicant, without explicitly decomposing it.

Recall that we are interested in mining present and absent items, therefore our itemsets are composed of items represented with a positive or a complemented literal. For instance, the itemset $\{a, b, \bar{c}\}$ means that a and b are present, while c is absent.

For both operations we must select the products of $DS^+ \tilde{f}_{\mathcal{D}}$ representing at least one transaction containing a given itemset. These products can be characterized as follows.

Property 1. Let p be a product in $DS^+ \tilde{f}_{\mathcal{D}}$. At least one transaction represented by p contains the items of an itemset t if and only if:

1. the items represented in t by a positive literal occur in p in positive or don't care form;
2. the items represented in t by a negative literal are absent from p or occur in don't care form.

Let us now consider the frequency computation. Let t be an itemset and p a product in $DS^+ \tilde{f}_{\mathcal{D}}$, with multiplicity k . Each p satisfying Property 1 contributes to the frequency of t for a value that depends on the don't cares in p , as shown in the following property.

Property 2 (Frequency). Let d be the dimension of the cube represented by a product p , i.e., the number of don't care literals, and let g be the number of positive or negative literals in the itemset t appearing as don't care literals in p . The number of transactions in p containing t is equal to $k \cdot 2^{d-g}$.

Thus, the total frequency of the itemset t is given by the sum of the frequencies computed for all products in $DS^+ \tilde{f}_D$ satisfying Property [11](#).

For example, let us compute the frequency of the itemset $t = \{\bar{b}, c\}$ in $DS^+ \tilde{f} = 2\tilde{b}\tilde{c}\tilde{d} + \tilde{a}\tilde{b}\tilde{c} + \tilde{d} + \tilde{b}\tilde{c}\tilde{d} + \tilde{a}\tilde{c}\tilde{d} + \tilde{a}\tilde{b} + 5c$. The products contributing to the frequency of t are $2\tilde{b}\tilde{c}\tilde{d}$, $\tilde{a}\tilde{b}\tilde{c}$, $\tilde{a}\tilde{c}\tilde{d}$, and $5c$. The overall frequency of t is then $2 \cdot 2^{2-1} + 2^{2-1} + 2^{1-0} + 5 \cdot 2^{0-0} = 13$.

In order to compute the closure of an itemset t we must select the transactions containing it, and compute their intersection, as described in the following property.

Property 3 (Closure). Let t be an itemset, and p be a product of $DS^+ \tilde{f}_D$ satisfying Property [11](#). The intersection $I_t(p)$ of the subset of transactions of p containing t is given by the union of the following literals:

1. the positive literals of p ;
2. the negative literals of p , that is the literals not appearing in p ;
3. the literals representing the items in t .

The closure of t is given by the intersection of the sets $I_t(p)$ computed for all p satisfying Property [11](#).

Observe that the don't care literals cannot be contained in an intersection, as they represent items appearing in all possible combinations, that therefore cannot be always present or always absent.

For example, let us compute the closure of $t = \{\bar{b}, c\}$ in $DS^+ \tilde{f} = 2\tilde{b}\tilde{c}\tilde{d} + \tilde{a}\tilde{b}\tilde{c} + \tilde{d} + \tilde{b}\tilde{c}\tilde{d} + \tilde{a}\tilde{c}\tilde{d} + \tilde{a}\tilde{b} + 5c$. The products $\tilde{b}\tilde{c}\tilde{d}$, $\tilde{a}\tilde{b}\tilde{c}$, and c contribute to the closure through the same subset $\{\bar{b}, c\}$; while the product $\tilde{a}\tilde{c}\tilde{d}$ contributes with $\{a, \bar{b}, c\}$. Taking the intersection, we get $clo(t) = \{\bar{b}, c\}$, which means that t is closed. Let us now compute the closure of $t = \{\bar{a}, b\}$. The products $\tilde{b}\tilde{c}\tilde{d}$, $\tilde{a}\tilde{b}\tilde{c}$, $\tilde{b}\tilde{c}\tilde{d}$ and c all contribute to the closure through the same subset $\{\bar{a}, b, c\}$; while the transactions in the other products do not contain t . Thus, $clo(t) = \{\bar{a}, b, c\}$.

The efficiency of the LCM algorithm has been improved with several optimizations, as shown in [\[11\]\[12\]](#). Again, these new versions of the algorithm can exploit the proposed database representation.

5 Conclusion

Our future work includes an experimental evaluation of the proposed approach. In particular we intend to show the gain in time and size of the LCM algorithm when its input is represented as a disjoint SOP⁺ form. Note that this compact form can be derived in polynomial time using heuristics, and it can then be represented by the same data structures used by LCM (e.g., FP-tee, bitmap, ZDDs). Recall that instead of representing single transactions, we now represent subsets of transactions with a single object (product); moreover all transactions in a product can be manipulated implicitly without decomposing the product. It would be also very interesting to analyze the approximation properties of this new synthesis problem.

Logic synthesis algorithms have good performances when the number of variables is limited. In Data Mining instead the number of items is usually quite high. Hence, we propose to afford the problem by grouping the items hierarchically, e.g., we could use variables to indicate single departments of a store (meat, vegetables, body care, etc.). In case of interesting associations among departments, the study can be refined inside them.

References

1. Agrawal, A., Mannila, H., Srikant, S., Toivonen, H., Verkamo, A.I.: Fast Discovery of Association Rules. In: *Advances in Knowledge Discovery and Data Mining*, pp. 307–328. MIT Press, Cambridge (1998)
2. Bernasconi, A., Ciriani, V., Luccio, F., Pagli, L.: A New Heuristic for DSOP Minimization. In: *Proc. 8th International Workshop on Boolean Problems* (2008)
3. Han, J., Kamber, M.: *Data Mining: Concept and Techniques*, 2nd edn. Bibliographic Notes for Chapter 5. Morgan Kaufman Publ., San Francisco (2006)
4. Han, J., Pei, J., Yin, Y.: Mining Frequent Patterns without Candidate Generation. In: *SIGMOD 2000*, pp. 1–12 (2000)
5. Kirsch, A., Mitzenmacher, M., Pucci, G., Pietracaprina, A., Upfal, E., Vandin, F.: An Efficient Rigorous Approach for Identifying Statistically Significant Frequent Itemsets. In: *Proc. 27th ACM Symposium on Principles of Database Systems (PODS)*, pp. 117–126 (2009)
6. Minato, S., Arimura, H.: Frequent Closed Item Set Mining Based on Zero-suppressed BDDs. *Information and Media Technologies* 2(1), 309–316 (2007)
7. Pietracaprina, A., Vandin, F.: Efficient Incremental Mining of Top-K Frequent Closed Itemsets. In: *Corruble, V., Takeda, M., Suzuki, E. (eds.) DS 2007. LNCS (LNAI)*, vol. 4755, pp. 275–280. Springer, Heidelberg (2007)
8. Silverstein, C., Brin, S., Motwani, R.: Beyond Market Baskets: Generalizing Association Rules to Dependence Rules. *J. Data Mining and Knowledge Discovery* 2(1), 36–68 (1998)
9. Uno, T., Asai, T., Uchida, Y., Arimura, H.: LCM: An Efficient Algorithm for Enumerating Frequent Closed Item Sets. In: *Proc. IEEE ICDM 2003 Workshop FIMI 2003* (2003)
10. Uno, T., Asai, T., Uchida, Y., Arimura, H.: An Efficient Algorithm for Enumerating Closed Patterns in Transaction Databases. In: *Suzuki, E., Arikawa, S. (eds.) DS 2004. LNCS (LNAI)*, vol. 3245, pp. 16–31. Springer, Heidelberg (2004)
11. Uno, T., Kiyomi, M., Arimura, H.: LCM ver. 2: Efficient Mining Algorithms for Frequent/Closed/Maximal Itemsets. In: *Proc. IEEE ICDM 2004 Workshop FIMI 2004* (2004)
12. Uno, T., Kiyomi, M., Arimura, H.: LCM ver.3: Collaboration of Array, Bitmap and Prefix Tree for Frequent Itemset Mining. In: *Proc. of the 1st International Workshop on Open Source Data Mining, OSDM 2005*, pp. 77–86 (2005)
13. Wang, J., Han, J., Lu, Y., Tzvetkov, P.: TFP: An Efficient Algorithm for Mining Top-K Frequent Closed Itemsets. *IEEE Transactions on Knowledge and Data Engineering* 17, 652–664 (2005)
14. Yang, G.: The Complexity of Mining Maximal Frequent Itemsets and Maximal Frequent Patterns. In: *Proc. of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 344–353 (2004)

Using Cell Phone Keyboards Is (\mathcal{NP}) Hard

Peter Boothe

Manhattan College

`peter.boothe@manhattan.edu`

Abstract. Sending text messages on cell phones which only contain the keys 0 through 9 and # and * can be a painful experience. We consider the problem of designing an optimal mapping of numbers to sets of letters to act as an alternative to the standard $\{2 \rightarrow \{abc\}, 3 \rightarrow \{def\} \dots\}$. Our overall goal is to minimize the expected number of buttons that must be pressed to enter a message in English. Some variations of the problem are efficiently solvable, either by being small instances or by being in P , but the most realistic version of the problem is \mathcal{NP} hard. To prove \mathcal{NP} -completeness, we describe a new graph coloring problem UNIQUEPATHCOLORING. We also provide numerical results for the English language on a standard corpus which describe several mappings that improve upon the standard one. With luck, one of these new mappings will achieve success similar to that of the Dvorak layout for computer keyboards.

Typing on a keyboard which has fewer keys than there are letters in the alphabet can be a painful task. There are a plethora of input schemes which attempt to make this task easier (e.g. [6,5,2] and many more), but the one thing they have in common is that all of these input methods use the standard mapping of numeric keys to alphabetic numbers of $\{2 \rightarrow \{abc\}, 3 \rightarrow \{def\}, 4 \rightarrow \{ghi\}, 5 \rightarrow \{jkl\}, 6 \rightarrow \{mno\}, 7 \rightarrow \{pqrs\}, 8 \rightarrow \{tuv\}, 9 \rightarrow \{wxyz\}\}$. We will consider schemes of rearranging the numbers on the keys to make messages easier to type. Most variants of this problem turn out to be NP-hard, unfortunately.

Before we get any farther, let us sketch the basic problem that we will keep revising and revisiting.

Problem 1 (MINIMUMKEYSTROKES)

INSTANCE A set of letters corresponding to an alphabet A ($|A| = n$), a number of keys k , an input method IN , and a set of tuples of words and frequencies W . The frequencies in W are integers, and the words are made up of solely of elements of A . We will treat IN as a function which, given a partition of A and a word w , returns how many keystrokes are required to type w .

QUESTION. What is the best partition of A into k sets, such that the total number of keystrokes to type every word in W its associated frequency times is minimized? Equivalently, what is the partition of P of A ($|P| = k$) that will minimize

$$\sum_{(w,f) \in W} f * \text{IN}(w, P)$$



Fig. 1. A cell phone keyboard. Each key is mapped to a letter sequence: $2 \rightarrow (abc)$, $3 \rightarrow (def)$, $4 \rightarrow (ghi)$, $5 \rightarrow (jkl)$, $6 \rightarrow (mno)$, $7 \rightarrow (pqrs)$, $8 \rightarrow (tuv)$, $9 \rightarrow (wxyz)$.

We will consider two different real-world schemes for IN (basic typing and T9), and for each variant or sub-variant that is proven \mathcal{NP} -hard, we consider the restriction on P that requires that we keep the alphabet in alphabetical order. In all cases where it is computationally feasible we provide results for the case of the English language, on 8-key keyboards, using the British National Corpus [1] (BNC). One feature of note is that no matter what scheme we use, the problem is trivial if the number of keys is not smaller than the number of letters in the alphabet ($k \geq |A|$). Using tiny keyboards is only (computationally) difficult when the keyboards do not have enough keys.

1 Setting a Baseline with the Easiest Problem

The baseline we compare against is the most painful method of text entry. In this method, to type an ‘a’, the user of the cellphone types the ‘2’ key; to type a ‘b’ they type ‘22’, to type ‘c’ they type ‘222’, to type a ‘d’ they type ‘3’, to type an ‘e’ they type ‘33’, and so on. To type a word with multiple letters that use the same key, such as ‘accept’, the user must pause between keystrokes. Thus, the full key entry sequence for ‘accept’, using ‘.’ to indicate a pause, is ‘2.22.223378’.

We will neglect the pauses and concern ourselves solely with the number of keystrokes required. This layout and input method imply that ‘a’ will always require one button press, ‘b’ will always require two, and so on. Thus, to get a baseline of how many keystrokes are required to enter the entirety of our corpus of words W , we count the total number of occurrences of each letter, and multiply that number of occurrences by the number of button pushes required

by that letter. Running this on the British National Corpus using the standard cell phone keyboard layout, we find that entering the entirety of the 100,106,029 word occurrences in the corpus would require 948,049,046 button presses.

If we examine a cell phone keyboard (Fig. 11) then we observe some terrible design choices. The frequently-occurring letters 'r' and 's' require more keystrokes than 'q'! Surely, a better designed keyboard can do better than this. If we just reversed the order of the letters on the 7 key, from 'pqrs' to 'srqp', the number of button presses required would drop to 865,118,331 — a savings of more than 8.7%. If we assume that every button press takes an equal amount of time, then this corresponds to the users spending 8.7% less time entering their text messages. In an effort to discover how great these savings can be, we define the problem BASICCELLPHONETYPING based on the template problem on page 54.

Problem 2 (BASICCELLPHONETYPING)

INSTANCE *A set of letters corresponding to an alphabet A ($|A| = n$), a number of keys k , and a set of tuples of words and frequencies W . The frequencies in W are integers, and the words are made up of solely of elements of A .*

QUESTION. *What is the best partition P of A into k sequences, such that the total number of button presses to type every word in W its associated frequency times is minimized? The number of button presses is equal to the order of the letter in the sequence assigned to a given key. For example, in the key $2 \rightarrow \{abc\}$, a requires one keystroke and c requires 3. Equivalently, if we denote the number of button presses required to type letter a with partition P as $\text{IN}_P(a)$, then we want to find the P that minimizes*

$$\sum_{(w,f) \in W} \sum_{c \in w} \text{IN}_P(c) * f$$

To solve BASICCELLPHONETYPING we construct a provably optimal greedy algorithm which works in time $O(|W| + |A| \log |A|)$. Our algorithm is detailed in Fig. 12, and involves creating a histogram of the letters, and then assigning letters to keys round-robin style in the order from most-popular to least-popular. To prove optimality, we invoke an exchange argument.

Theorem 3. *The greedy algorithm finds an optimal solution to BASICCELLPHONETYPING.*

Proof. We will assume, for simplicity of proof, that all letters occur a different number of times in the corpus. We begin by comparing two assignments of letters to keys by, for each assignment, constructing a sequence of sets, or spectrum, S . The set S_1 (layer 1) is the set of all letters which require a single button press (that is, they are the first letter in their sequence on their assigned key), S_2 (layer 2) is the set of all letters which require two button presses, and so on. If two assignments have equal spectra, then one assignment may be transformed into the other assignment simply by swapping letter between keys without increasing


```

GreedyBasicCellPhoneTyping ( $A, k, W$ )
//  $A$  is the set of letters
//  $k$  is the number of keys
//  $W$  is a set of pairs of words and their corresponding frequencies
lettercount  $\leftarrow$  new_map()
for  $c \in A$ 
    lettercount[ $c$ ]  $\leftarrow$  0
for (word, frequency)  $\in W$ 
    for  $c \in$  word
        lettercount[ $c$ ]  $\leftarrow$  lettercount[ $c$ ]+frequency
ordered  $\leftarrow$  [(lettercount[ $c$ ],  $c$ ) for  $c \in$  lettercount]
sort(ordered)
keys  $\leftarrow$  a array of length  $k$  where each element is an empty list
for  $i \leftarrow 0 \dots \text{length}(\text{ordered})$ 
    (count, char)  $\leftarrow$  ordered[ $i$ ]
    append(char, keys[ $i \bmod k$ ])
return keys

```

Fig. 2. The greedy algorithm for Prob. 2

or decreasing the total number of button pushes required to enter the corpus of words. Two assignments are isomorphic iff their spectra are equal.

Now assume for the sake of contradiction that we have assignment D , with spectrum T , that is not isomorphic to the greedy solution G with spectrum S . In a spectrum resulting from the greedy algorithm, all letters at layer i occur strictly more frequently than the letters at layer j , $j > i$. Also, in S , for all layers i except for the last layer, $|S_i| = k$. Because T and S are not isomorphic, in T there must be at least one layer i such that $T_i \neq S_i$. Let T_i be the first layer for which that is true.

There are two cases for layer T_i . In the first case, $|T_i| < k$, and we may remove any element from a later layer place it in layer T_i and create a strictly better layout. In the second case, there is some letter a such that $a \in T_i$ and $a \notin S_i$. We choose the least-frequent such a . Because the greedy layout algorithm creates layers in order of frequency, we know that the frequency of a is less than that of some letter b in layer T_j , $j > i$. Therefore, by creating a new layout with a and b swapped between T_i and T_j , we have strictly decreased the number of button presses. Therefore, in both cases, T was not an optimal layout, and we have reached our contradiction. \square

For the BNC, the optimal layout has the spectrum

$$\{\{etaoinsr\}, \{hldcum.fp\}, \{gwybvkbxj\}, \{qz\}\}$$

and any layout with that spectrum requires 638,276,114 button presses to entire the entire BNC instead of our original requirements of 948,049,046. This represents a savings of 32.67% over our original layout, but only for the users who type using this most basic of input methods. Unfortunately, this is the group of users who are likely the least adaptable to change, as almost all “digital natives” use

predictive methods to input text messages¹. To place the least burden of “newness” on these users while still decreasing the number of button presses, we now consider schemes where the only alteration of the keyboard is the rearrangement of the sequence of letters for a given key.

By an argument symmetric to the proof of Theorem 3, we find that the optimal layout in this new scheme is to place the letters of a given key in sorted order, according to frequency. Thus, the keyboard layout changes to $2 \rightarrow [acb]$, $3 \rightarrow [edf]$, $4 \rightarrow [ihg]$, $5 \rightarrow [lkj]$, $6 \rightarrow [onm]$, $7 \rightarrow [srpq]$, $8 \rightarrow [tuv]$, $9 \rightarrow [wxyz]$ and requires 678,547,463 button presses to enter the whole corpus. This layout represents a 28.43% savings, and it has the added benefit that it does not change which key is mapped to which set of letters. This makes it **legacy preserving**, as creating a keyboard with this layout will not invalidate such advertising gems as 1-800-FLOWERS. Thus, we can speed up users by 28.43% (neglecting inter-letter pauses) and not undermine more than half a century of advertising. This layout represents perhaps the most plausible layout yet². After setting up this baseline for the easy problem, we turn our attention to optimizing cellphone keyboards for predictive input methods.

2 The T9 Input Method

In an effort to minimize the pain of cellphone keyboard typing, cellular telephone manufacturers have created the T9 input method, which attempts to guess which letter (of the three or four possible) is intended when a user hits a single key. Enhancements of this input method also provide speculative lookahead to report, at any given time, what word it is that the user is most likely trying to enter and provide a completion. Unfortunately, because there are fewer keys on the cellphone keyboard than there are letters in the English alphabet, there are words which have different spellings but the same input sequence. As an example, in the traditional mapping of keys to characters both ‘me’ and ‘of’ have the input sequence ‘63’. Two words with the same input sequence force the user to press a third button to cycle through the possibilities in order from most likely (‘of’) to least likely (‘me’). Even worse: ‘home’, ‘good’, ‘gone’, ‘hood’, ‘hoof’, ‘hone’, and ‘goof’ all have the input sequence ‘4663’. If the * key is used as the “next match” button, then the user will have to type in ‘4663*****’ to type in the word ‘goof’, for a total of 10 button presses — exactly the same number of button presses required using the default layout and the basic input method, and three more button presses than is required when using the basic input method with an optimized layout.

¹ I have no statistics on this except for an informal survey of one of my classes. In that class, every student who used SMS either had a cellphone with a 26+ key alphabetic keyboard or used a predictive method.

² The described layout is the only legacy preserving layout in this paper, and therefore should be considered the most practical suggestion. It also has the added benefit of not angering any of the organizations behind the numbers 1-800-FLOWERS, 1-800-THRIFTY, and 1-800-MARINES.

When two words have the same input sequence (neglecting the *'s at the end) then these words are **t9onyms**. When two or more words are t9onyms, then the less-popular words require extra key presses, raising the expected number of key presses to type in our corpus. To type a given word, one must press one button for every character in the word, followed by pressing the * key as many times as there are t9onyms which are more likely than the desired word. Because typing on a cellphone keyboard is already an unpleasant experience, we would like to minimize the expected number of keystrokes. The number of keystrokes a word requires is equal to the number of letters in the word, plus the number of t9onyms which are more frequent than the word. Formally, we extend Prob. **11** and define the MINIMUMT9KEYSTROKES problem as:

Problem 4 (MINIMUMT9KEYSTROKES)

INSTANCE *An alphabet A , a set of words and their associated frequencies W , and a number of keys k ($|A| > k$).*

QUESTION. *What is the partition P of A into k sets which minimizes the total button presses required to enter the entire corpus using the T9 input method?*

and the corresponding decision problem is

Problem 5

INSTANCE *A set A , a set W of sequences of elements of A , a mapping f from sequences in W to the integers, and a number t .*

QUESTION. *Is there a partition P of A into k sets such that, if we denote $P(w)$ as the sequence of partitions $[p_i | w_i \in p_i, p_i \in P]$,*

$$\sum_{w \in W} (\text{len}(w) + \text{order}(w, P, W, f)) * f(w) \leq t$$

where $\text{len}(w)$ is the length of the sequence w and $\text{order}(w, P, W, f)$ is the size of $\{s \in W | P(s) = P(w) \wedge f(s) \leq f(w) \wedge s < w\}$, which is the set of sequences which map to the same sequence of partitions, but are not more popular, and are lexicographically smaller than w .

This problem is in \mathcal{NP} , as any assignment may be verified to require fewer than t button pushes in time proportional to the total length of all the words in W . To prove completeness, however, we first prove the \mathcal{NP} -completeness of an intermediate problem: UNIQUEPATHCOLORING.

Problem 6 (UNIQUEPATHCOLORING)

INSTANCE *A graph $G = (V, E)$, a set of paths P , and a parameter k . A path p is a sequence of vertices in which adjacent vertices in p are also adjacent in the edge set E .*

QUESTION. *Is there a k -coloring of G such that every path $p \in P$ has a unique coloring? If we consider the coloring function $\chi(v)$ to map vertices to colors,*

then we can extend this notation by having $\chi(p)$ map a path $p = [v_1, v_2, \dots]$ to the sequence $\chi(p) = [\chi(v_1), \chi(v_2), \dots]$. Is there a coloring χ of V such that

$$|\{\chi(p) \mid p \in P\}| = |P| ?$$

Theorem 7. UNIQUEPATHCOLORING is \mathcal{NP} -complete.

Proof. To prove that UNIQUEPATHCOLORING is \mathcal{NP} -complete, we begin by noting that any coloring of V may be verified, in polynomial time, to map each path to a unique sequence of colors. Therefore, the problem is in \mathcal{NP} . To prove completeness, we reduce from GRAPHCOLORING [3].

An instance of GRAPHCOLORING consists of a graph $G = (V, E)$ and a parameter k . We then ask the question of whether there is a k -coloring χ of the vertices of the graph such that $\forall (u, v) \in E, \chi(u) \neq \chi(v)$. We transform an instance of GRAPHCOLORING into an instance of UNIQUEPATHCOLORING in the following manner:

Given $G = (V, E)$ and k from GRAPHCOLORING, we create

$$G' = (V \cup \{0, 1\}, E \cup \{(0, 1), (1, 0), (0, 0), (1, 1)\} \cup \{(v, 0) \mid v \in V\})$$

We then uniquely number each edge in E with the numbers $1 \dots |E|$. For each edge $e = (u, v)$ numbered i , we create the path set

$$p_i = \{[v, 0, b_1(i), b_2(i), b_3(i), \dots, b_{\lceil \log_2 i \rceil}], \\ [u, 0, b_1(i), b_2(i), b_3(i), \dots, b_{\lceil \log_2 i \rceil}]\}$$

where $b_1(i)$ is the first digit of i in binary, $b_2(i)$ is the second digit of i in binary, and so on. Now we create our path set

$$P = \{[0, 1], [1, 0]\} \cup \bigcup_{i=1}^{|E|} p_i .$$

We then ask the question of whether G' and P can be unique-path colored using only k colors.

If there is a k -coloring χ of G , then we use that coloring to generate a k -unique-path coloring of G' and P in the following manner: First, assign vertices 0 and 1 different colors from the range of χ . Next, assign each vertex to the color it received in the k -coloring of G' . Now, every element of our path set corresponds to a unique color sequence. $[0, 1]$ and $[1, 0]$ are the only sequences of length two, and because we assigned these two vertices different colors, $\chi([0, 1]) \neq \chi([1, 0])$. If path x and path y are not from the same p_i , then they have a different sequence of zeroes and ones. Therefore, because 0 and 1 are assigned different colors, the only possible path that a given path in some p_i might be confused with is the other path in that p_i . But each p_i corresponds to an edge in E , and we know for all edges in $(u, v) \in E$ that $\chi(u) \neq \chi(v)$. Therefore, both paths within a given

p_i also have a distinct color sequence. Therefore, given a k -coloring of G , we can generate a k -unique-path coloring of G' and P .

Now we prove that given a k -unique-path coloring of G' and P we can create a k -coloring of G . Given a k -unique path coloring, we are guaranteed that no path in P has the exact same coloring as any other path in P . This means that, for all i , the two paths in p_i are distinct. The only difference between the two paths in p_i is in the first vertex of the path. Therefore, for every p_i , the vertices of corresponding edge in must be given distinct colors, and this is true for all i and $e \in E$. Therefore, for all $(u, v) \in E$, the color assigned to u is distinct from the color assigned to v . Therefore, from a k -unique-path coloring of G' and P , we have a k -coloring of $G = (V, E)$. \square

This proof implies not just that UNIQUEPATHCOLORING is \mathcal{NP} -complete, but that it is \mathcal{NP} -complete even when we restrict ourself to just 3 colors, because the number of colors in the UNIQUEPATHCOLORING is exactly equal to the number of colors in the instance of GRAPHCOLORING, and 3-coloring a graph is an \mathcal{NP} -complete problem [4].

Now that we have proven UNIQUEPATHCOLORING to be \mathcal{NP} -complete, we immediately use it in a reduction.

Theorem 8. MINIMUMT9KEYSTROKES (as specified in Problem 5) is \mathcal{NP} -complete.

Proof. As we noted on page 58, MINIMUMT9KEYSTROKES is in \mathcal{NP} . To prove completeness, we reduce from UNIQUEPATHCOLORING. An instance of UNIQUEPATHCOLORING consists of a graph $G = (V, E)$, a set of paths P , and a parameter k . We then ask the question of whether there is a vertex coloring of G using k colors where each path in P ends up with a unique sequence of colors.

We transform an instance of UNIQUEPATHCOLORING into an instance of MINIMUMT9KEYSTROKES in the following way: We make A be the set of vertices V , and we make W be the set of paths P , and we assign each of these new ‘words’ a frequency of 1. We then ask the question of whether there exists a partition of A into k sets such that the total number of button presses required is no greater than $t = \sum_{w \in W} \text{len}(w)$.

Now we prove that if there is a UNIQUEPATHCOLORING using k colors, then there exists a solution to the corresponding instance of MINIMUMT9KEYSTROKES. In particular, a k -unique-path coloring of G give us a partition of the vertices V into k sets. We map each set to a single key in our solution to MINIMUMT9KEYSTROKES. Because each path has a unique coloring, each word in the corresponding MINIMUMT9KEYSTROKES has a unique pattern of button presses. Therefore, the number of more popular words with the same keystrokes as a given word is exactly zero. Thus, because each word in the corpus has a frequency of one, to type in the entire corpus only requires exactly as many keystrokes as there are words in the corpus, and therefore, the k unique path coloring of V corresponds exactly to an assignment of letters to keys such that the total number of keystrokes is exactly $\sum_{w \in W} \text{len}(w)$, which is exactly t .

Next, we prove that a solution to the `MINIMUMT9KEYSTROKES` problem instance implies a solution to the corresponding `UNIQUEPATHCOLORING` problem. If there exists a partition of A into k partitions such that

$$\sum_{w \in W} (\text{len}(w) + \text{order}(w, P, W, f)) * f(w) \leq t = \sum_{w \in W} \text{len}(w)$$

then, because $f(w) = 1, \forall w \in W$, it must be true that $\text{order}(w, P, W, f) = 0, \forall w \in W$. Therefore, every word $w \in W$ has a unique sequence of button presses if we partition A into k partitions according to P . To generate our k -unique-path coloring of G , we color each set in P a single unique color. Because each word in w has a unique sequence of partitions, each path has a unique coloring, and therefore we can color the corresponding instance of G using k colors. Thus, a partition of V into k partitions which allows the corpus to be entered in less than t keystrokes implies a k -unique path coloring of G . Therefore, because `UNIQUEPATHCOLORING` can be reduced to `MINIMUMT9KEYSTROKES` and is in \mathcal{NP} , `MINIMUMT9KEYSTROKES` is \mathcal{NP} -complete. \square

This proof of \mathcal{NP} -completeness is of a relatively strong form: just like `UNIQUEPATHCOLORING`, our problem remains \mathcal{NP} -complete even if we restrict ourselves to $k = 3$ (only 3 buttons on the cell phone keyboard).

Because our problem is \mathcal{NP} -complete, we will not try to find a general solution. In our specific instance, with 26 letters and an eight key keyboard (1 and 0 are reserved for other uses), we must choose the best layout from among

$$\binom{26}{8} \binom{18}{8} \binom{10}{8} = 3,076,291,325,250$$

different possibilities. Given that, currently, it takes around a tenth of a second to count the button presses required to enter the BNC, an exhaustive search will take 307,629,132,525 seconds, or 9.75 millennia. Therefore, we have two remaining avenues of attack: find the best answer possible using stochastic methods, or solving a simpler problem. Using stochastic methods (a simple genetic algorithm), the best layout we found was

$$\{\{eb\}, \{lcd\}, \{sh\}, \{zmx\}, \{fayg\}, \{toj\}, \{unpv\}, \{irkwq\}\}$$

which required 441,612,049 button presses to enter the entire corpus. Our genetic algorithm was quite simple: From an initial gene pool of random layouts, we discarded all but the best layouts (the survivors). Then, we added to the gene pool random mutations of the survivors to fill out the gene pool back to its original size, and repeated the process of selection and mutation for some time. As compared to the requirements of the default layout (443,374,079), this represents a savings of less than 1%, and is thus not a very attractive alternative to existing layouts. This low amount of improvement, while not good news for our proposed reordering, is excellent news for all current cell phone users: the default cell phone keyboard layout seems relatively efficient for T9 text entry in English!

2.1 Alphabetic Order

Our arbitrary remappings of the keyboard may be quickly deemed unusable, simply because it is almost impossible to tell where a given key is located without memorizing the entire new layout. Therefore, we define a refinement of the keyboard layout problem, in which the characters are required to be kept in alphabetical order as a sequence, and our only choices are about how to divide the subsequence into key assignments. The default layout can be described as the partition $abc|def|ghi|jkl|mno|pqrs|tuv|wxyz$. Is this partition optimal? To decide this, we define the problem `MINIMUMKEYSTROKEPARTITION` as:

Problem 9 (`MINIMUMKEYSTROKEPARTITION`)

INSTANCE *A sequence of letters $A = \{a_1, a_2, \dots\}$, a set of words and their associated probabilities W , and a set of keys $K = \{2, 3, 4, \dots\}$ ($|A| > |K|$).*

QUESTION. *What is the mapping f of $A \rightarrow K$ which minimizes the expected number of characters to type a word from W , and where $f(a_1) = 2$ and if $f(a_i) = k$, then either $f(a_{i+1}) = k$ or $f(a_{i+1}) = k + 1$?*

The number of partitions of a sequence of size n into k subsequences is equal to $\binom{n-1}{k-1}$, and in the particular case of the 26 letter alphabet and the eight keys available on a cell phone keyboard, we find that there are $\binom{26-1}{8-1} = 480,700$ possible sequences. This means that, in the case of our corpus which requires tenths of a second to test against a proposed solution, we can test every possible sequence in just 480,700 seconds. Therefore, our final result comes from simply generating and testing every single partition of the alphabet into 8 sets. We found that the best partition was

$$\{\{ab\}, \{cde\}, \{fghij\}, \{klm\}, \{nop\}, \{qrs\}, \{t\}, \{vwxyz\}\}$$

which required 442,717,436 button presses, yet again for a savings of less than 1% over the default layout.

3 Conclusion

We have developed three problems based on the idea of making cell phone keyboard more efficient for typing text messages. Different text entry methods yield problems with very different levels of solvability. If we restrict ourselves to the basic text entry method, then a greedy algorithm will work for finding the optimal layout. Even better, the greedy algorithm works whether we want to restrict ourselves to rearranging the order of letters on a single key, or to rearranging all letters on all keys. On the other hand, if we try to optimize against the more complex T9 input method, then we find that the problem is \mathcal{NP} -complete. After proving completeness, we gave numerical results which indicate that it will be difficult (or perhaps impossible) to significantly improve on the default layout of letters. This stands in sharp contrast to the simpler input method, where we

were able to improve by more than 27% just by reordering the letters on the keys, but never moving a letter from one key to another.

We have also discovered a new \mathcal{NP} -complete problem UNIQUEPATHCOLORING, which may be of use in other contexts. In particular, it is rare in the literature to find an \mathcal{NP} -complete problem that contains both partitions and sequences, which is why we had to invent a new one here. The hope, as always, is that this problem will prove useful to others.

We left as future work any exploration of the issue of needing to pause between letters in the basic typing method, and, while there do exist even more sophisticated input methods[6], we have left them unmentioned and unexamined. Optimizing a keyboard for the basic text input method was relatively straightforward, but, for the T9 input method, optimally using a cell phone keyboard is \mathcal{NP} hard.

References

1. British National Corpus Consortium. British National Corpus (2000)
2. Evans, F., Skiena, S., Varshney, A.: Resolving ambiguity in overloaded keyboards (2004), <http://cs.smith.edu/~orourke/GodFest/>
3. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman, New York (1979)
4. Garey, M.R., Johnson, D.S., Stockmeyer, L.: Some simplified NP-complete problems. In: STOC 1974: Proceedings of the sixth annual ACM symposium on theory of computing, pp. 47–63. ACM, New York (1974)
5. Gong, J., Tarasewich, P., MacKenzie, I.S.: Improved word list ordering for text entry on ambiguous keypads. In: NordiCHI 2008: Proceedings of the 5th Nordic conference on human-computer interaction, pp. 152–161. ACM, New York (2008)
6. Nesbat, S.B.: A system for fast, full-text entry for small electronic devices. In: ICMI 2003: Proceedings of the 5th international conference on multimodal interfaces, pp. 4–11. ACM, New York (2003)

A A Genetic Algorithm for Discovering Efficient T9 Layouts

```

#include <algorithm>
#include <iostream>
#include <fstream>
#include <map>
#include <string>
#include <vector>
#include <sys/time.h>

#define POPULATION 32
#define GENERATIONS 20

using namespace std;

typedef pair<int, string> pis;
vector<pis> words;
long long total;

bool piscompare(pis a, pis b) { return a.first > b.first; }

long long quality(vector<string> &partition)
{
    map<char, string> rmap;
    int i = 0;
    string buttons = "23456789";
    for (vector<string>::iterator s = partition.begin();
         s != partition.end();
         ++s, ++i) {
        for (string::iterator c = s->begin(); c != s->end(); ++c) {
            rmap[*c] = buttons[i];
        }
    }

    map<string, vector<pis> > typing;
    for (vector<pis>::iterator fw = words.begin();
         fw != words.end();
         ++fw) {
        int freq = fw->first;
        string word = fw->second;
        string presses = "";
        for (string::iterator c = word.begin(); c != word.end(); ++c) {
            presses += rmap[*c];
        }

        if (typing.count(presses) == 0)
            typing[presses] = vector<pis>();
        typing[presses].push_back(*fw);
    }

    for (map<string, vector<pis> >::iterator w = typing.begin();
         w != typing.end();
         ++w) {
        sort(w->second.begin(), w->second.end(), piscompare);
    }

    long long totalpresses = 0;
    for (map<string, vector<pis> >::iterator w = typing.begin();
         w != typing.end();
         ++w) {
        int wordsize = w->first.size();
        for (int i = 0; i < w->second.size(); i++) {
            totalpresses += (wordsize + i) * w->second[i].first;
        }
    }
    return totalpresses;
}

void printpart(vector<string> &p)
{
    for (vector<string>::iterator i = p.begin();
         i != p.end();
         ++i)
        cout << *i << " ";
    cout << endl;
}

vector<string> randpartition(string in)
{
    vector<string> p;
    for (int i = 0; i < 8; i++)
        p.push_back("");

    while (in.size() > 0) {
        int pos = random() % in.size();
        int key = random() % 8;
        p[key].append(1, in[pos]);
        in.erase(pos, 1);
    }
}

```

```

    }
    return p;
}

typedef pair<long long, vector<string> > pllvs;
bool lvscompare(pllvs a, pllvs b) { return a.first < b.first; }

vector<vector<string> > cull(vector<vector<string> > &pop)
{
    vector<pllvs> fitness;
    for (vector<vector<string> >::iterator i = pop.begin();
         i != pop.end();
         ++i) {
        fitness.push_back(pllvs(quality(*i), *i));
    }

    sort(fitness.begin(), fitness.end(), lvscompare);

    vector<vector<string> > newpop;
    for (int i = 0;
         i < fitness.size() / 4;
         ++i) {
        cout << fitness[i].first << " ";
        printpart(fitness[i].second);
        newpop.push_back(fitness[i].second);
    }

    return newpop;
}

vector<string> mutate(vector<string> in)
{
    for (int count = 0; count < 2; count++) {
        int i;
        do { i = random() % in.size(); } while (in[i].size() == 0);

        int c = random() % in[i].size();
        in[random() % in.size()].append(1, in[i][c]);
        in[i].erase(c, 1);
    }

    printpart(in);
    return in;
}

int main()
{
    ifstream win("words/wordlist");
    win >> total;

    string word;
    win >> word;

    int freq;
    while (win >> freq) {
        win >> word;
        words.push_back(pis(freq, word));
    }

    // Seed it
    timeval t1;
    gettimeofday(&t1, NULL);
    srandom(t1.tv_usec * t1.tv_sec);

    string alphabet = "abcdefghijklmnopqrstuvwxyz";
    vector<vector<string> > population;
    for (int i=0; i < POPULATION; i++)
        population.push_back(randompartition(alphabet));

    for (int i=0; i < GENERATIONS; i++) {
        population = cull(population);
        while (population.size() < POPULATION) {
            population.push_back(mutate(population[random() % population.size()]));
        }
    }

    return 0;
}

```

B Code for Analyzing All Alphabetic-Order T9 Layouts

```

#include <algorithm>
#include <iostream>
#include <fstream>
#include <map>
#include <string>

```

```

#include <vector>

using namespace std;

typedef pair<int, string> pis;
vector<pis> words;
long long total;

bool piscompare(pis a, pis b) { return a.first > b.first; }

long long quality(vector<string> &partition)
{
    map<char, string> rmap;
    int i = 0;
    string buttons = "23456789";
    for (vector<string>::iterator s = partition.begin();
         s != partition.end();
         ++s, ++i) {
        for (string::iterator c = s->begin(); c != s->end(); ++c) {
            rmap[*c] = buttons[i];
        }
    }

    map<string, vector<pis> > typing;
    for (vector<pis>::iterator fw = words.begin();
         fw != words.end();
         ++fw) {
        int freq = fw->first;
        string word = fw->second;
        string presses = "";
        for (string::iterator c = word.begin(); c != word.end(); ++c) {
            presses += rmap[*c];
        }

        if (typing.count(presses) == 0)
            typing[presses] = vector<pis>();
        typing[presses].push_back(*fw);
    }

    for (map<string, vector<pis> >::iterator w = typing.begin();
         w != typing.end();
         ++w) {
        sort(w->second.begin(), w->second.end(), piscompare);
    }

    long long totalpresses = 0;
    for (map<string, vector<pis> >::iterator w = typing.begin();
         w != typing.end();
         ++w) {
        int wordsize = w->first.size();
        for (int i = 0; i < w->second.size(); i++) {
            totalpresses += (wordsize + 1) * w->second[i].first;
        }
    }
    return totalpresses;
}

int start = 0;
int end = -1;
void genallpartitions(string &s, int count, int index,
                    vector<string> &part)
{
    if (count == 0) {
        if (--start > 0) {
            --end;
        } else {
            // Analyze it
            cout << quality(part) << " ";
            for (vector<string>::iterator i = part.begin();
                 i != part.end();
                 ++i)
                cout << *i << " ";
            cout << endl;
            if (--end == 0) exit(0);
        }
    } else if (count == 1) {
        part.push_back(s.substr(index, s.size()-index));
        genallpartitions(s, count-1, s.size(), part);
        part.pop_back();
    } else {
        for (int size = 1; size < s.size() - index - count + 2; size++) {
            part.push_back(s.substr(index, size));
            genallpartitions(s, count-1, index+size, part);
            part.pop_back();
        }
    }
}

int main(int argc, char** argv)
{

```

```
if (argc == 3) {
    start = atoi(argv[1]);
    end = atoi(argv[2]);
}

ifstream win("words/wordlist");
win >> total;

string word;
win >> word;

int freq;
while (win >> freq) {
    win >> word;
    words.push_back(pis(freq,word));
}

string alphabet = "abcdefghijklmnopqrstuvwxy";
vector<string> partition;
genallpartitions(alphabet, 8, 0, partition);
return 0;
}
```

Urban Hitchhiking

Marco Bressan and Enoch Peserico

Dip. Ing. Informazione, Univ. Padova, Italy
{bressanm, enoch}@dei.unipd.it

Abstract. You are an urban hitchhiker. All drivers are willing to give you a ride, as long as they do not have to alter their trajectories to accommodate your needs. How (and how quickly) can you get to your destination? We analyze two scenarios, depending on whether hitchhikers have a global picture of who is going where through some information infrastructure, or only a local picture - i.e. they can only ask cars passing by where they are going.

1 Introduction

You are an urban hitchhiker. All drivers are willing to give you a ride, as long as they do not have to alter their trajectories to accommodate your needs. How (and how quickly) can you get to your destination? This is a fundamental question in the area of *dynamic ride-sharing*, which is attracting ever more attention due to soaring oil prices and increasing pollution concerns [1,2,3,4,5,8,9]. The answer (not surprisingly) depends mainly on whether you have a global picture of who is going where (most likely through some information infrastructure), or only a local picture - i.e. you can only ask cars passing by where they are going. We examine the two cases respectively in Section 4 and Section 3 after presenting, in Section 2, our city traffic model. Section 5 concludes the paper with a brief analysis of the significance of our results and of the many problems this preliminary work leaves open.

2 A Simple City Traffic Model

There is a vast number of traffic models in the literature (see [6] for a review); however, most of them focus on congestion control and are too complex to be solved analytically at the city level. Our model is much simpler, although it is still sufficiently rich to produce non-trivial results and, we feel, to capture the essence of the problem.

We model the city and its periphery as a circle of radius R (see Figure 2), where the unit of space is the *contact distance* - the maximum distance at which a hitchhiker can signal a car “passing by” to stop and query it for information (indicatively ≈ 30 meters, the range of class 2 Bluetooth devices present on most of today’s mobile phones and perhaps half a city block). The contact distance is also the minimum distance for which we assume that a hitchhiker may seek a ride

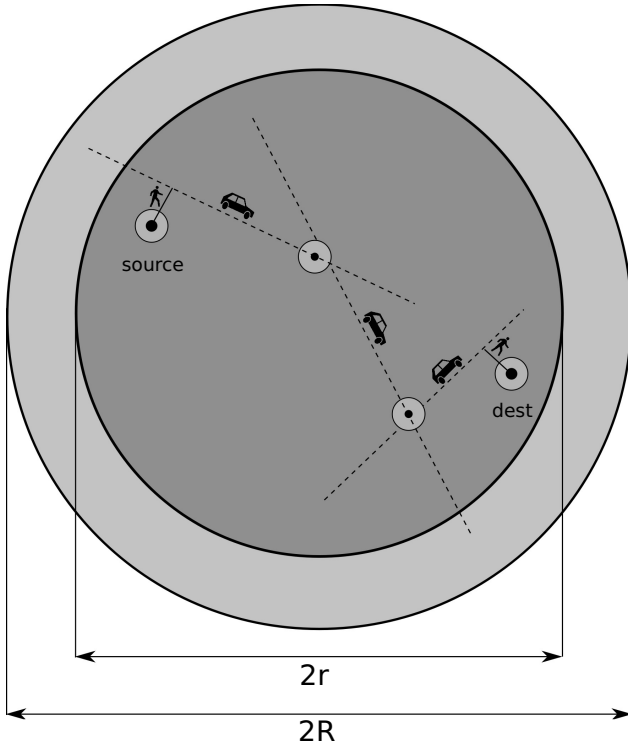


Fig. 1. The city (radius r) containing source and destination, and the city plus the periphery (radius $R > r$). Cars travel in a straight trajectory (dashed line) from a random source to a random destination, and the hitchhiker may hop into them if they pass within unit distance of him. The hitchhiker may also walk (solid line) towards the destination or towards a good ride.

instead of simply resorting to walking. We set the time scale so that traveling 1 unit of space takes a car 1 unit of time, and a pedestrian P units.

The city itself is a smaller, concentric circle of radius r . We only consider hitchhiking within the city. The periphery - the annulus around the city proper - simply serves as an abstraction for the sources and sinks of traffic outside of the city. Cars enter the traffic system of the city and its periphery at points chosen uniformly at random within the (larger) circle of radius R according to a Poisson process with intensity of $\frac{1}{G}$ cars per unit of area per unit of time; each car then heads in a straight line towards a destination also chosen uniformly at random. It is easy to verify that the process describing arrivals/departures of cars *in transit* through any unit circle is poissonian, with an intensity $\Theta(\frac{R}{G}) = \Theta(\rho)$ that is within a factor $1 - \frac{r}{R}$ for *any such area* within the city. The rest of the paper assumes $1 - \frac{r}{R} = \Theta(1)$, which is equivalent to the (realistic) assumption that a constant fraction of the city's traffic has a source or a destination outside the city proper.

The hitchhiker wants to move, within the city, from an arbitrary source to an arbitrary destination at distance $d \geq 1$ from the source. While a private car allows the hitchhiker to cover this distance in time d with only one leg, hitchhiking requires, in general, more time (and perhaps more legs). In the rest of the paper, we will often state results in terms of *effective speed*, i.e. the average speed of the hitchhiker from the source to the destination.

3 Hitchhiking with Local Information Only

This section examines the case of a hitchhiker equipped only with local knowledge - i.e. who can only query nearby cars (those within distance 1) about their destination. A strictly optimal strategy is beyond the scope of this work; we can, however, provide a simple strategy that is optimal in terms of effective speed (and among effective speed-optimal strategies, in terms of legs) within a small constant factor.

Let a ride be *good* if it can take the hitchhiker at least twice as close to the destination as he currently is; and let it be *better* than another ride if it can take the hitchhiker closer to the destination. Consider the following three simple hitchhiking rules:

1. Initially, walk towards the destination taking the first good ride encountered.
2. When encountering a new potential ride, take it if and only if a) it is better than any ride seen so far and b) the current ride, if any, is no longer good.
3. As soon as the current ride is no longer making progress towards the destination dismount and start walking towards the destination.

We can prove that:

Theorem 1. *If the hitchhiker's current distance from the destination is $d > \frac{1}{\rho}$, he will reach within distance $\frac{1}{\rho}$ of the destination in $O(d)$ time (only $O(\frac{1}{\rho})$ of which spent walking) and $O(1)$ legs. If the hitchhiker's current distance is d with $\frac{1}{P\rho} < d \leq \frac{1}{\rho}$, he will halve his current distance in expected $O(\frac{1}{\rho})$ time and $O(1)$ legs, and will reach within distance $\frac{1}{P\rho}$ of the destination in $O(\frac{\log(P)}{\rho})$ time and $O(\log(P))$ legs. If the hitchhiker's current distance is $d \leq \frac{1}{P\rho}$, he will reach the destination in time $O(dP)$.*

Proof. First of all note that, if the hitchhiker is not moving, the probability density (in time) of witnessing a previously unseen car pass by that will take him within distance δ of the destination is within a constant factor (see Section 2) of a function $p(\delta/d)$ that depends solely on the ratio between δ and the current distance d to the destination: $p = \Theta(\rho \cdot \frac{\delta}{d})$, where $\frac{\delta}{d}$ is (within a small constant factor) the ratio between the length of the circumference of radius d centered on the current position and passing through the destination and the length δ of its arc centered on the destination. The expected time to witness a ride which carries the hitchhiker within distance δ of the destination is then $1/p = O(\frac{1}{\rho} \cdot \frac{d}{\delta})$.

The same holds, within a small constant factor, when the hitchhiker is walking, since the speed of cars relative to him will be between $1 + 1/P$ (for cars with the same trajectory but opposite direction) and $1 - 1/P$ (for cars with the same trajectory and the same direction).

This is no longer true while riding, since the probability density of witnessing a ride depends on the angle its trajectory makes with the trajectory of the current ride. In particular, the probability density of witnessing a ride with a trajectory at an angle $\alpha < \frac{\pi}{2}$ from that of the hitchhiker is $\rho \sin(\alpha)$. Let x be the current distance between the hitchhiker and the point of the ride closest to the destination, d' be the distance of this point from the destination. The probability density of witnessing a ride that would take the hitchhiker within d'' of the destination depends on the sine of the angle formed by the current trajectory and the direction of destination (which for $x > 2d'$ can be approximated by d'/x) and the angle formed by the circle of radius d'' (which for $x > 2d'$ can be approximated by d''/x). Therefore, the hitchhiker witnesses approximately $\rho \cdot \frac{d''}{x} \cdot \frac{d'}{x}$ rides per time unit that would carry him within distance d'' of the destination. While the current ride is still good (i.e. for $d \leq x \leq 2d'$), the total number of witnessed rides that would carry within d'' of the destination is $\int_{d'}^d \rho \cdot \frac{d''}{x} \cdot \frac{d'}{x} dx = \Theta(\rho d'')$; and thus the best witnessed ride carries the hitchhiker within an expected distance $\Theta(1/\rho)$. Once the ride is no longer good, the hitchhiker (now at distance $\Theta(d')$ from the destination) has to wait or walk $\Theta(\frac{1}{\rho})$ time units in expectation to witness a ride better than any ride seen so far. Therefore, in expected $O(d)$ time and $O(1)$ legs, the hitchhiker reaches within distance $O(1/\rho)$ and, with an additional constant number of good legs (as we will prove now), he reaches within distance $1/\rho$ from the destination.

Once within distance $d = O(1/\rho)$ from the destination, the probability density of witnessing a good ride is $\Theta(\rho)$. Then the expected time to witness a good ride is $O(1/\rho)$, while the ride time is $\Theta(d) = O(1/\rho)$, and therefore wait time dominates travel time. This means that, while riding, the hitchhiker witnesses $O(1)$ good rides, and thus the best ride seen so far would carry him, in expectation, within a distance from the destination which is a constant factor smaller than the current distance. Therefore, the hitchhiker has to wait $O(1/\rho)$ time units in expectation to hop into the next ride, which halves its distance from the destination in $O(1/\rho)$ expected time and $O(1)$ expected legs. The hitchhiker can then reach within distance $\frac{1}{P\rho}$ of the destination in $\Theta(\log(P))$ legs and $O(\log(P)/\rho)$ time.

Once within distance $\frac{1}{P\rho}$ of the destination, walking to it requires time $1/\rho$, within a constant of that required to witness a ride.

In a nutshell, Theorem [1](#) says that a hitchhiker with only local information can expect to make progress towards the destination at a speed within a constant factor of that achievable with a private car, up to a distance $\frac{1}{\rho}$ from the destination - approximately the expected distance a car can travel before passing by another car. Within that range, the hitchhiker's effective speed drops proportionally with the distance to the destination, until at range $\frac{1}{P\rho}$ it reaches within a constant factor of walking speed.

It is natural to ask whether a hitchhiker aware of all car traffic - rather than only of that passing nearby - can reduce his travel time and/or the number of legs he will need. We prove this is indeed the case in Section 4.

4 Hitchhiking with Global Information

This section examines strategies for a hitchhiker armed with global knowledge of the position and destination of all cars that are currently moving. This could be the case, for example, when a central authority continuously gathers all the traffic information and provides directions to hitchhikers. It turns out that (unlike the case of local information) with global information two-leg trips are always asymptotically optimal, and indeed even one-leg trips are asymptotically optimal except for a relatively narrow interval of source-destination distances.

4.1 One-Leg Trips

In these scenario, the hitchhiker attempts to reach his destination using at most a single ride. His strategy is extremely simple. Armed with global knowledge of car traffic, he can compute for each car the minimum time required to hop onto it (by walking and/or waiting), ride on it, and finally walk to the destination. He then chooses the car yielding the shortest time to the destination - unless, of course, simply walking directly to the destination is faster. We can prove the following:

Theorem 2. *For $d < \frac{1}{\sqrt{P\rho}}$ the hitchhiker moves at the pedestrian's speed $\frac{1}{P}$. For $\frac{1}{\sqrt{P\rho}} \leq d < \frac{P}{\sqrt{\rho}}$, the hitchhiker moves at an effective speed $\Theta(\frac{\sqrt[3]{d^2}}{\sqrt[3]{d^2 + \sqrt[3]{P^2/\rho}}})$. For $d \geq \frac{P}{\sqrt{\rho}}$, the hitchhiker moves at an effective speed $\Theta(1)$.*

Proof. Assume the hitchhiker is willing to spend t time units walking or waiting around the source before hopping into a ride, and t time units walking to the destination after hopping off the ride. He can walk a distance $x \leq \frac{t}{P}$ from a in every direction, and then wait $t - x$ time units; thus he witnesses $\frac{t^2\rho}{P}$ distinct rides in expectation, and the fraction of rides that carry the hitchhiker to within distance t/P from the destination, ensuring him a walk time no greater than t , is proportional to $\frac{t/P}{d}$ in expectation. Therefore the expected number of such rides is $\frac{t^2\rho}{P} \frac{t/P}{d} = \frac{t^3\rho}{P^2d}$; when this quantity is 1, which holds for $t = \sqrt[3]{\frac{P^2d}{\rho}}$, we have at least a constant probability of witnessing a ride. This gives us the *critical time* $t_c = \sqrt[3]{\frac{P^2d}{\rho}}$, i.e. the expected time to witness a useful ride.

We can now find the values of d which correspond to “phase transitions” in the effective speed. For d small enough, the critical time t_c dominates the time dP required to walk to the destination: $\sqrt[3]{\frac{P^2d}{\rho}} \geq dP$, which gives $d \leq \frac{1}{\sqrt{P\rho}}$. Below this distance, the hitchhiker can at best move at speed $\frac{1}{P}$ by walking.

For d large enough, the traveling time dominates the waiting time: $d \geq \sqrt[3]{\frac{P^2 d}{\rho}}$, which gives $d \geq \frac{P}{\sqrt{\rho}}$. Above this distance, the hitchhiker moves at an effective speed $\Theta(1)$.

Between the two thresholds, i.e. for $\frac{1}{\sqrt{P\rho}} \leq d < \frac{P}{\sqrt{\rho}}$, the effective speed grows as $\Theta\left(\frac{d}{d+t_c}\right) = \Theta\left(\frac{d}{d+\sqrt[3]{\frac{P^2 d}{\rho}}}\right) = \Theta\left(\frac{\sqrt[3]{d^2}}{\sqrt[3]{d^2} + \sqrt[3]{\frac{P^2}{\rho}}}\right)$.

According to Theorem 2, as the distance between source and destination increases, the hitchhiker's expected speed grows from walking speed to within a constant factor of the unit speed achievable with a private car. In fact, we can prove a stronger result: as the distance increases, the hitchhiker's effective speed tends to 1 with probability that also tends to 1:

Theorem 3. *The hitchhiker moves at an effective speed $\geq 1 - k\sqrt[3]{\frac{P^2}{d^2\rho}}$ with probability $\geq 1 - e^{-k^3}$.*

Proof. The total time spent by the hitchhiker is the sum of the traveling time (which is approximately d) and the walking/waiting time (which is approximately t). When the former dominates the latter, the effective speed is approximately $\frac{d}{d+t} = 1 - \frac{t}{d}$ which, for $t = kt_c$, becomes $1 - \frac{k\sqrt[3]{\frac{P^2 d}{\rho}}}{d} = 1 - k\sqrt[3]{\frac{P^2}{d^2\rho}}$. According to the proof of the previous theorem, waiting a time t gives a probability of witnessing a ride of at least $1 - e^{-\frac{t^3\rho}{P^2 d}} = 1 - e^{-k^3\frac{t^3\rho}{P^2 d}} = 1 - e^{-k^3}$.

4.2 Two-Leg Trips

In this scenario, the hitchhiker can employ up to two rides to reach the destination. Again, the strategy is very simple. Armed with global knowledge of car traffic, the hitchhiker computes for every pair of cars the shortest total time required to reach the first, ride on it, walk to/wait for the second car, ride on it and finally walk to the destination. It turns out that this does not reduce - compared to the single ride case - the critical distance $\frac{1}{\sqrt{P\rho}}$ at which the hitchhiker is better off walking. However, at distances greater than that, two-leg trips yield better effective speeds than one-leg trips. More specifically, we show that:

Theorem 4. *For $d < \frac{1}{\sqrt{P\rho}}$ the hitchhiker moves at the pedestrian's speed $\frac{1}{P}$. For $\frac{1}{\sqrt{P\rho}} \leq d < \frac{\sqrt{P}}{\sqrt{\rho}}$, the hitchhiker moves at an effective speed $\Theta\left(\frac{d}{d+\sqrt{P/\rho}}\right)$. For $d \geq \frac{\sqrt{P}}{\sqrt{\rho}}$, the hitchhiker moves at an effective speed $\Theta(1)$.*

Proof. Assume the hitchhiker is willing to spend t time units walking or waiting around the source before hopping onto a car, t time units waiting for the second ride once he leaves the first, and t time units walking to the destination after dismounting from the second ride. As in the case of Theorem 2, the hitchhiker witnesses $\frac{t^2\rho}{P}$ distinct rides around the source in expectation, and a constant

fraction of these carry him within a constant angle from the source-destination axis. Therefore the expected number of such rides is $\frac{t^2\rho}{P}$; when this quantity is 1, which holds for $t = \sqrt{P/\rho}$, we have at least a constant probability of witnessing a ride. This gives us the *critical time* $t_c = \sqrt{P/\rho}$, i.e. the expected time to witness a useful ride. Note that this holds also for a ride which travels towards the destination, and thus with constant probability there will be two rides which spatially intersect and give a feasible 2-leg solution.

We can now find the values of d which correspond to “phase transitions” in the effective speed. For d small enough, the critical time t_c dominates the time dP required to walk to the destination: $\sqrt{\frac{P}{\rho}} \geq dP$, which gives $d \leq \frac{1}{\sqrt{P\rho}}$. Below this distance, the hitchhiker can at best move at speed $\frac{1}{P}$ by walking. For d large enough, the traveling time dominates the waiting time: $d \geq \sqrt{\frac{P}{\rho}}$, which gives $d \geq \sqrt{P/\rho}$. Above this distance, the hitchhiker moves at an effective speed $\Theta(1)$. Between the two thresholds, i.e. for $\frac{1}{\sqrt{P\rho}} \leq d < \sqrt{P/\rho}$, the effective speed grows as $\Theta(\frac{d}{d+t_c}) = \Theta(\frac{d}{d+\sqrt{P/\rho}})$.

Again, we can easily prove that, as the travel distance grows beyond the minimum distance yielding effective speed $\Theta(1)$, the effective speed of the hitchhiker converges to 1 with probability that also converges to 1 - and this convergence is slightly faster than in the case of single rides:

Theorem 5. *The hitchhiker moves at an effective speed $\geq 1 - k\sqrt[3]{\frac{P}{d^2\rho}}$ with probability $\geq 1 - e^{-k^3}$.*

Proof. The total time spent by the hitchhiker is the sum of the waiting time t and the traveling time $\leq d(1 + \alpha)$, where α is the angle that the trajectories of the legs form with the source-destination axis. We want $d\alpha \approx t$. This yields an effective speed of approximately $\frac{d}{d+t} = 1 - \frac{t}{d} = 1 - \frac{k\sqrt[3]{\frac{Pd}{\rho}}}{d} = 1 - k\sqrt[3]{\frac{P}{d^2\rho}}$. The expected number of rides within the angle is then $\frac{t^2\rho}{P} \frac{t}{d} = \frac{t^3\rho}{Pd}$, and the critical time to witness at least one becomes $t_c = \sqrt[3]{\frac{Pd}{\rho}}$; therefore, when $t = kt_c$, the probability of witnessing at least one such ride is $1 - e^{-\frac{k^3 t^3 \rho}{Pd}} = 1 - e^{-3}$.

Figure 2 summarizes the results of this Section, plotting effective speed as a function of the distance between source and destination, depending on whether the hitchhiker is constrained to use at most one ride, or at most two rides. Note that, when global traffic information is available, hitchhiking on three or more rides cannot asymptotically improve effective speed: the time spent by the “two-leg” hitchhiker (see Theorem 4) is the sum of two terms — the first asymptotically equal to the time d necessary for a private car to reach the destination, and the other asymptotically equal to the time $\Theta(\sqrt{\frac{P}{\rho}})$ necessary (even with full knowledge of future car traffic) to find *any car at all*.

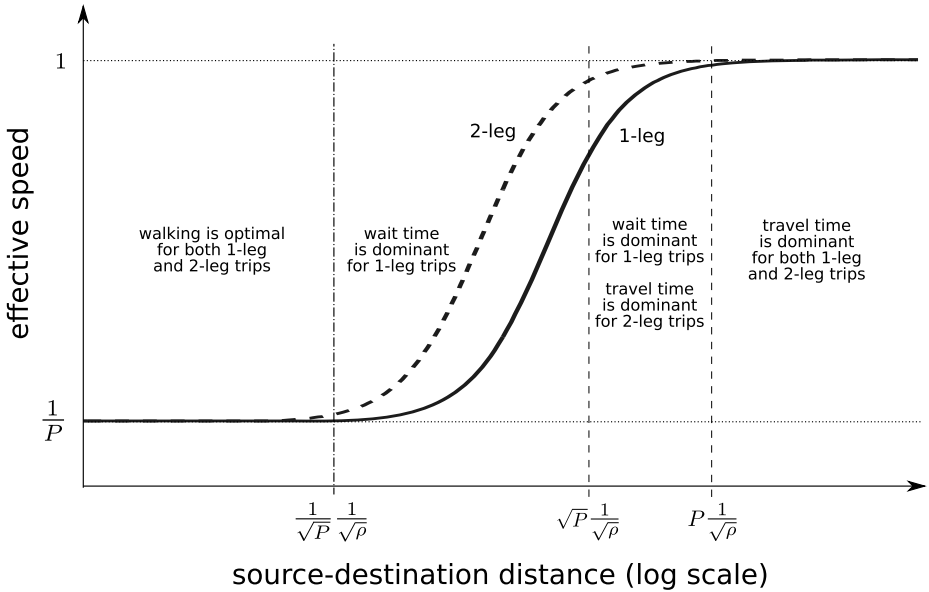


Fig. 2. The hitchhiker’s expected effective speed as a function of the source-destination distance in the case of one-leg trips (solid) and two-leg trips (dashed)

5 Conclusions

Urban hitchhiking - assuming a wide base of collaborating drivers - can be extremely efficient, particularly for long distances. A centralized infrastructure is beneficial, but even without it a hitchhiker can advance towards the destination at an effective speed within a small constant factor of that achievable with a private vehicle, changing car only $O(1)$ times - at least until he reaches within a distance $\frac{1}{\rho}$ of the destination, i.e. within the average distance a car can drive before encountering another car. Below this threshold the effective speed of a hitchhiker with purely local information decreases proportionally to the distance to the destination, until it reaches walking speed, and car changes become more frequent.

An infrastructure providing information about all moving cars improves a hitchhiker’s effective speed, allowing him to travel to the destination almost as fast as with a private vehicle - and, importantly, changing cars at most once - as long as he starts at a distance from the destination that is at least $\sqrt{\frac{P}{\rho}}$. This distance is mean proportional between the average distance $\frac{1}{\rho}$ one can drive before encountering another car, and the average distance P one can drive in the time to walk the few meters of a “contact distance”. Thus, under realistic choices of P and ρ , access to global traffic information allows one to travel almost as efficiently through hitchhiking as through the use of a private vehicle when moving more than a few hundred meters.

This brief, preliminary work can be expanded in many directions. First, it might be interesting to enrich the model to take into account that the source-destination pairs might not be chosen uniformly, but perhaps according to some “small world” power law distribution [7]; and that cars do not travel along straight lines, but tend instead to concentrate along major traffic arteries. At first glance, neither change significantly appears to affect our results, at most making hitchhiking, especially without an infrastructure, slightly more efficient. Much more interesting, however, would be validating our model and conclusions on real traffic data; these are unfortunately hard to obtain, since very few systems keep track of the trajectories of large numbers of *individual* cars. Of course, the ultimate test of our results would be an implementation of a dynamic ride-sharing system based on them!

Acknowledgments. This work was supported in part by MIUR under PRIN AlgoDEEP and by Univ. Padova under Strategic Project AACSE. Marco Bressan was supported in part by a Univ. Padova Research Fellowship.

References

1. Abdel-Naby, S., Fante, S.: Auctions negotiation for mobile rideshare service. In: Proc. IEEE Second International Conference on Pervasive Computing and Applications (2007)
2. Buliung, R., Soltys, K., Habel, C., Lanyon, R.: Driving factors behind successful carpool formation and use. Transportation Research Record: Journal of the Transportation Research Board 2118 (2009)
3. Calvo, R.W., de Luigi, F., Haastrup, P., Maniezzo, V.: A distributed geographic information system for the daily car pooling problem. Comput. Oper. Res. 31(13), 2263–2278 (2004)
4. Hartwig, S., Buchmann, M.: Empty seats traveling: Next-generation ridesharing and its potential to mitigate traffic and emission problems in the 21st century. Technical report, Nokia (2007), <http://research.nokia.com/tr/NRC-TR-2007-003.pdf>
5. Kelley, K.: Casual carpooling-enhanced. Journal of Public Transportation 10(4) (2007)
6. Klar, A., Kuehne, R., Wegener, R.: Mathematical models for vehicular traffic. Survey on Mathematics for Industry (1996)
7. Kleinberg, J.: The Small-World Phenomenon: An Algorithmic Perspective. In: Proceedings of the 32nd ACM Symposium on Theory of Computing (2000)
8. Morris, J.: Saferide: Reducing single occupancy vehicles. Technical report (2008), <http://www.cs.cmu.edu/~jhm/SafeRide.pdf>
9. Murphy, P.: The smart jitney: Rapid, realistic transport. New Solutions Journal (4) (2007)

A Fun Application of Compact Data Structures to Indexing Geographic Data*

Nieves R. Brisaboa¹, Miguel R. Luaces¹, Gonzalo Navarro², and Diego Seco¹

¹ Database Laboratory, University of A Coruña
Campus de Elviña, 15071, A Coruña, Spain
{brisaboa, luaces, dseco}@udc.es

² Department of Computer Science, University of Chile
Blanco Encalada 2120, Santiago, Chile
gnavarro@dcc.uchile.cl

Abstract. The way memory hierarchy has evolved in recent decades has opened new challenges in the development of indexing structures in general and spatial access methods in particular. In this paper we propose an original approach to represent geographic data based on compact data structures used in other fields such as text or image compression. A *wavelet tree*-based structure allows us to represent minimum bounding rectangles solving geographic range queries in logarithmic time. A comparison with classical spatial indexes, such as the R-tree, shows that our structure can be considered as a fun, yet seriously competitive, alternative to these classical approaches.

Keywords: geographic data, MBR, range query, wavelet tree.

1 Introduction

The ever-increasing demand for services that allow users to find the geographic location of some resources in a map has emphasized the interest in the field of Geographic Information Systems (GIS). The huge size of geographic databases has made the development of spatial access methods one of the most important topics of interest in this field. Even though many classical spatial indexes [7] provide an excellent performance, the way the memory hierarchy has evolved in recent decades has opened new opportunities in this topic. New levels have been added (e.g., flash storage) and the sizes at all levels have been considerably increased. In addition, access times in upper levels of the hierarchy have decreased much faster than in lower levels. Thus, reducing the size of spatial indexes is a topic of interest because placing these indexes in upper levels of the memory hierarchy reduces access times considerably, in some cases by several orders of magnitude. Nowadays it is feasible to place complete spatial indexes in

* This work has been partially supported by “Ministerio de Educación y Ciencia” (PGE y FEDER) ref. TIN2009-14560-C03-02, by “Xunta de Galicia” ref. 08SIN009CT, and by Fondecyt Grant 1-080019, Chile.

main memory. Note that spatial indexes do not contain the real geographic objects but a simplification of them. The most common simplification is the MBR (*Minimum Bounding Rectangle*).

In this paper we aim at the development of compact spatial indexes that can be placed in upper levels of the memory hierarchy. We build on previous solutions for two-dimensional points using a structure called a *wavelet tree* [9], and generalize them to an index able of answering range queries on rectangle data. Wavelet trees are interesting because they offer a compact-space solution to various point indexing problems. In previous work [3] we presented a spatial index for two-dimensional points based on wavelet trees. The generalization to support queries over MBRs, which we present here, turns out to be a rather challenging problem not arising in other domains where wavelet trees have been used. Our experiments, featuring GIS-like scenarios, show that our index is a relevant and funnier alternative to classical spatial indexes, such as the R-tree [10], and that it can take advantage of the fashionable research in compressed data structures.

2 Related Work

A great variety of spatial indexes have been proposed supporting the different kinds of queries that can be applied to spatial databases (*exact match, adjacency, nearest neighbor, etc.*). In this paper we focus on a very common kind of query, named *range query*, on collections of two-dimensional geographic objects. The problem is formalized as follows. In the 2-dimensional Euclidean space E^2 , we define the MBR of a geographic object o , $MBR(o) = I_1(o) \times I_2(o)$ where $I_i(o) = [l_i, u_i]$ ($l_i, u_i \in E^1$) is the minimum interval describing the extent of o along the dimension i . In the same way, we define a rectangle query $q = [l_1^q, u_1^q] \times [l_2^q, u_2^q]$. Finally, the range query to find all the objects o having at least one point in common with q is defined as $RQ(q) = \{o \mid q \cap MBR(o) \neq \emptyset\}$.

The R-tree [10] is one of the most popular multidimensional access methods used to solve range queries in GIS. It consists of a balanced tree “derived from the B-tree” that decomposes the space into hierarchically nested, possibly overlapping, MBRs. Object MBRs are associated with the leaf nodes, and each internal node stores the MBR that contains all the nodes in its subtree. The algorithm to solve range queries using this structure goes down the tree from the root visiting those nodes whose MBR intersects the query window. Most of the numerous variants [13] of the original Guttman’s proposal aim at improving the performance of the R-tree both in the general case and in particular applications (static collections). Two of these variants (the R*-tree [2] and the STR R-tree [12]) are used in Section 4 to compare the performance of our proposal.

The problem of solving two-dimensional range queries on points has also been tackled in other research fields. The seminal computational geometry work by Chazelle [4] offers several space-time tradeoffs, including one that in two dimensions requires $O(N \log U)$ bits of space and answers range queries in time $O(\log N + k \log^\epsilon N)$, where N is the total number of points in $[1, U] \times [1, U]$, k

is the output size, and $0 < \epsilon < 1$ is a constant affecting memory consumption. The *wavelet tree* [9] can be regarded as a compact version of Chazelle’s data structure, which requires exactly $N \log_2 U + o(N \log U)$ bits to index N points in the range $[1, U]$. Recently [3], we adapted the basic approach where the points form a permutation to handle an arbitrary set of points in a continuous space, following Gabow’s arguments [6].

A basic tool in compact data structures is the *rank* operation: given a sequence S of length N , drawn from an alphabet Σ of size σ , $rank_a$ counts the occurrences of symbol $a \in \Sigma$ in $S[1, i]$. The dual operation, $select_a(S, i)$, finds the i -th occurrence of a symbol $a \in \Sigma$ in S . For the special case $\Sigma = \{0, 1\}$ (S is a bit-vector B), both rank and select operations can be implemented in constant time and using little additional space on top of B ($o(n)$ in theory [14, 8]). For example, given a bitmap $B = 1000110$, $rank_0(B, 5) = 3$ and $select_1(B, 3) = 6$. In addition, the symbol a can be extended to a finite number of sequences with similar techniques. For instance, given two bitmaps $B = \underline{1000110}$ and $C = \underline{0011010}$, $rank_{00}(B, C, 7) = 2$ and $select_{00}(B, C, 1) = 2$ (00 represents occurrences of the symbol 0 in both bitmaps simultaneously).

3 Our Fun Structure

In this section we introduce our technique for range queries on MBRs. Recall our formal definition of the problem from the previous section. The following, easy to verify, observation provides a basis for our next developments. It says, essentially, that an intersection between a query q and an object o occurs when, across each dimension, the query finishes not before the object starts, and starts not after the object finishes.

Observation 1. $o \in RQ(q)$ iff $\forall i, u_i^q \geq l_i \wedge l_i^q \leq u_i$.

3.1 Index Construction

In the upcoming discussion, we assume that the first dimension represents the rows of the grid (y-axis or latitudes) and the second represents the columns (x-axis or longitudes). Assume now the set of MBRs $g = \{m_1, \dots, m_N\}$ does not contain any MBR m_i whose projection in the x-axis is within the projection over the x-axis of other MBR m_j in the set (i.e., $\forall i, j$ if $l_2^i < l_2^j$ then $u_2^i \leq u_2^j$). We name g a *maximal set* and describe now a structure to represent a maximal set of MBRs. If the set of MBRs is not a maximal set, the problem can be decomposed into k independent maximal sets (see Section 3.4).

Then, let N be the number of MBRs in a maximal set, each one described by two pairs $\{(l_1, l_2), (u_1, u_2)\}$ (the coordinates of two opposite vertices). These MBRs can be represented in a $2N \times 2N$ grid with only one point in each row and column. Gabow et al. [6] proved that the orthogonal nature of the problem makes possible to work with the ranks of the coordinates instead of working with the coordinates themselves.

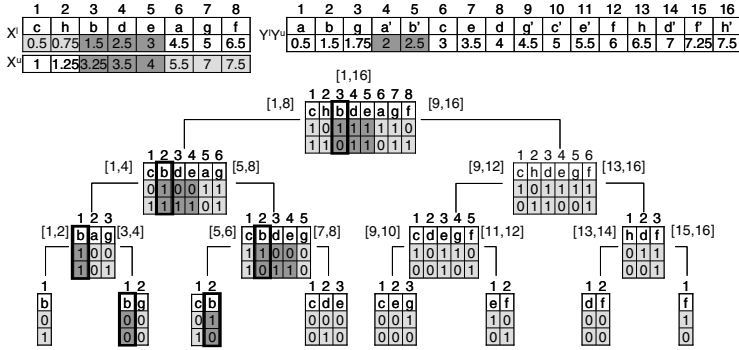


Fig. 1. Representing N MBRs using a wavelet tree

A wavelet tree with $\lceil \log_2 2N \rceil$ can be used to store this matrix (the permutation from the order of the MBRs in the x-axis to their order in the y-axis) with little storage cost (Figure 1). This is a binary tree where each node covers a range of positions in the $Y^l Y^u$ array that represents the first half of the range covered by its parent, in the case of a left child, and the second half in the case of a right child. The range covered by the root node is $[1,2N]$.

Each node in the tree stores two bitmaps B_1 and B_2 of the same length, and each position in these bitmaps corresponds with a MBR (in the figure, these positions have been annotated with the identifier of the corresponding MBR). The MBRs in each node are ordered by the x-axis. Let MBR_i be the MBR stored at the position i of a node, $b_i^{B_1}$ the bit i in the bitmap B_1 , and $b_i^{B_2}$ the bit i in the bitmap B_2 . Then, $b_i^{B_1} = 1$ if the MBR_i is processed in the left child and $b_i^{B_2} = 1$ if MBR_i is processed in the right child. A MBR is processed in a node if, in the y-axis, it finishes not before the range covered by the node starts, and starts not after the range covered by the node finishes. Let lB and uB be the lower and upper bounds of the range covered by a node in $Y^l Y^u$, then Equations 1 and 2 define the value of the bit i of this node in the first and second bitmap respectively. Note that a MBR can be processed in both the left and right child of a node and thus both $b_i^{B_1}$ and $b_i^{B_2}$ can store the value 1 simultaneously.

$$b_i^{B_1} = \begin{cases} 1 & \text{if } l_1^{MBR_i} \leq \frac{lB+uB}{2} \\ 0 & \text{otherwise} \end{cases} \quad (1) \quad b_i^{B_2} = \begin{cases} 1 & \text{if } u_1^{MBR_i} > \frac{lB+uB}{2} \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

We also need to store the real coordinates of the MBRs to perform the translation from the geographic space to the rank space. The order of the lower (X^l) and upper (X^u) coordinates in the x-axis is the same because we assume the matrix represents a maximal set. Thus, we use two sorted arrays with the lower (X^l) and upper (X^u) x-coordinates and an array storing the identifiers of the MBRs in the same order. Y-coordinates are stored also in an ordered array, $Y^l Y^u$, containing both lower (Y^l) and upper (Y^u) y-coordinates. Each position in the $Y^l Y^u$ array

of the figure has been annotated with the identifier of the corresponding MBR for clarity, but these identifiers are not stored.

As such, this structure may require quadratic space, however. The reason is that a MBR with a large extent in y can be represented in a linear number of nodes at the same level. In order to solve this problem Equation 3 presents a slight modification in the way the structure is created. When a MBR_i completely contains the range covered by the node both bitmaps store a 0 in the position i , and thus, this MBR is not stored in the nodes of this subtree. Then each MBR can be stored at most four times per level and we can guarantee logarithmic bit-space per MBR.

$$b_i^{B_1} = b_i^{B_2} = \begin{cases} 0 & \text{if } (l_1^{MBR_i} \leq lB) \text{ and } (u_1^{MBR_i} \geq uB) \\ \text{use (1) and (2) otherwise} & \end{cases} \quad (3)$$

3.2 Solving Queries

This structure can be used to solve range queries in the rank space derived from the translation of the original queries in the geographic space using the ordered arrays of coordinates (X^l , X^u , and Y^lY^u). A $leftSearch(S, t_i)$ finds the lowest $s_i \geq t_i$ in an ordered array S by means of a binary search. In a similar way, a $rightSearch(S, t_i)$ returns the largest $s_i \leq t_i$. Thus, a query in the geographic space $q = [y_l, y_u] \times [x_l, x_u]$ is translated into the equivalent query $q' = [y'_l, y'_u] \times [x'_l, x'_u]$ ($y'_l = leftSearch(Y^lY^u, y_l)$, $y'_u = rightSearch(Y^lY^u, y_u)$, $x'_l = leftSearch(X^u, x_l)$, and $x'_u = rightSearch(X^l, x_u)$) in the rank space (yes, the upper x coordinates of the MBRs are searched for the lower x coordinate of the query, and vice versa). For example, the query $q = [2.0, 2.75] \times [2.0, 3.5]$ translates into $q' = [4, 5] \times [3, 5]$ ($leftSearch(Y^lY^u, 2.0) = 4$, $rightSearch(Y^lY^u, 2.75) = 5$, $leftSearch(X^u, 2.0) = 3$, and $rightSearch(X^l, 3.5) = 5$).

Algorithm 1 shows the recursive method to solve range queries once they have been translated into the rank space. The interval $[x'_l, x'_u]$ determines the valid range inside the root node of the wavelet tree and the interval $[y'_l, y'_u]$ determines nodes that can be pruned (because the wavelet tree maps from the order in the x -axis to the order in the y -axis). This algorithm recursively projects a range, $[x'_l, x'_u]$ at the beginning, onto the child nodes using $rank_1$ operations over the two different bitmaps. The first bitmap B_1 is used to project onto the left child and the second bitmap B_2 is used to project onto the right child. The recursive traversal stops when the result of the two child nodes has been computed. Note that the same MBR can be reported by both child nodes but no repeated results should be reported by their parent node. Thus, the results of both siblings are merged to compute the result of their parent node. In addition, there can be local results in a node corresponding with MBRs that completely contain the range covered by the node (i.e., all the MBRs in a position i where $b_i^{B_1} = b_i^{B_2} = 0$), which are added to the result in the merge stage.

Figure 1 highlights the nodes visited to solve the query of the example $q = [2.0, 2.75] \times [2.0, 3.5]$. As we noted before, this query is translated into the

Algorithm 1. Range query algorithm in the rank space.

Require: $cNode, p_{min}, p_{max}, lB, uB$; current node, valid node positions $[p_{min}, p_{max}]$, query range $[lB, uB]$
 $result \leftarrow []$; $leftResult \leftarrow []$; $rightResult \leftarrow []$; $localResult \leftarrow []$
if $cNode.range \subseteq [lB, uB]$ **then**
 for $i = p_{min}$ to p_{max} **do**
 add i to $localResult$
 end for
else
 if $cNode.leftChild.range \cap [lB, uB] \neq \emptyset$ **then**
 $leftResult \leftarrow$ recursive call with:
 $p_{min} \leftarrow rank_1(cNode.B_1, p_{min} - 1) + 1$
 $p_{max} \leftarrow rank_1(cNode.B_1, p_{max})$
 $cNode \leftarrow cNode.leftChild$
 end if
 if $cNode.rightChild.range \cap [lB, uB] \neq \emptyset$ **then**
 $rightResult \leftarrow$ recursive call with:
 $p_{min} \leftarrow rank_1(cNode.B_2, p_{min} - 1) + 1$
 $p_{max} \leftarrow rank_1(cNode.B_2, p_{max})$
 $cNode \leftarrow cNode.rightChild$
 end if
 for $i = rank_{00}(cNode.B, p_{min} - 1) + 1$ to $rank_{00}(cNode.B, p_{max})$ **do**
 add $select_{00}(cNode.B, i)$ to $localResult$
 end for
 end if
 for all $lR \leftarrow leftResult.next(), j \leftarrow rightResult.next(), k \leftarrow localResult.next()$ **do**
 $merge(select_1(cNode.B_1, i), select_1(cNode.B_2, j), k)$
 end for
return $result$

ranges $[3, 5]$ (valid positions in the root node) and $[4, 5]$ (interval to prune the tree traversal). The first range is projected onto the child nodes of the root node as $[rank_1(B_1, 3 - 1) + 1, rank_1(B_1, 5)] = [2, 4]$ and $[rank_1(B_2, 3 - 1) + 1, rank_1(B_2, 5)] = [3, 4]$ but the second one is not accessed because it covers the range $[9, 16]$ which does not intersect the query range $[4, 5]$. In the same way the range $[2, 4]$ of the left child is projected onto its children as $[rank_1(B_1, 2 - 1) + 1, rank_1(B_1, 4)] = [1, 1]$ and $[rank_1(B_2, 2 - 1) + 1, rank_1(B_2, 4)] = [2, 4]$. In the next level, the first node accessed is the second one that covers the range $[3, 4]$. The result of this node comes from the local result that is computed in this way: there is one local result (because $rank_{00}(B, 1) = 1$) that is at the position 1 (because $select_{00}(B, 1) = 1$). When the recursive call returns the control to the parent of this node, its result is computed using the merge of the left child result (an empty set), the right child result ($select_1(B_2, 1) = 1$) and the local result (an empty set). In the parent of this node, there are no local results and the left result ($[1]$) and right result ($[2]$) reference the same MBR ($select_1(B_1, 1) = select_1(B_2, 2) = 2$). Finally, in the root node the result comes

from the left child and it is computed as $select_1(B_1, 2) = 3$. Note that the MBR at position 3 is b , the result of the query.

3.3 Coordinate Encoding

We introduce a compressed storage scheme to store the ordered arrays of coordinates (X^l , X^u , and Y^lY^u). We assume that these coordinates can be represented with four bytes, which is sufficient for the finite precision used in GIS. Geographic coordinates can be represented in degrees or meters and in most cases it is possible to round the coordinates to integer values, after appropriate scaling, without losing any precision. We make use of this assumption, as it holds in most practical applications.

Let $A = a_1a_2 \dots a_N$ be one of the arrays of integers to encode. Then, we encode A as a sequence of non-negative differences between consecutive values $b_{i+1} = a_{i+1} - a_i$ and $b_1 = a_1$. Let $B = b_1b_2 \dots b_N$ be this sequence, so that $a_i = \sum_{1 \leq j \leq i} b_j$. The array B is a representation of A that can be compressed by exploiting the fact that consecutive differences are smaller numbers. These small numbers can be encoded with different coding algorithms. We compare four different well-known coding algorithms [15]: Elias-Gamma, Elias-Delta, Rice, and VBytes.

Given a value v , we are interested in finding the largest $a_i \leq v$ and the lowest $a_i \geq v$. These operations are the *rightSearch* and *leftSearch* described in Section 3.2. In order to solve them efficiently we store a vector that stores the accumulated sum at regularly sampled positions (say every h th position, thus the vector stores all values $x_{i \cdot h}$). The search algorithm first performs a binary search in the vector of sampled sums, and then it carries out a sequential scan in the resulting interval of B .

3.4 Decomposition into Maximal Sets

In the general case, a maximal set is not enough to properly encompass the dataset but k maximal sets are needed. Each such set must be queried separately. We use a single shared Y^lY^u array for all of them, to reduce the number of binary searches. Thus the query time complexity can be bounded by $O(k \log N)$. Therefore, minimizing the number of maximal sets k is a key factor to improve the performance of our structure.

We can in fact decompose a general set of MBRs into the optimal number k of maximal sets, at indexing time, within $O(N \log N)$ complexity, as follows. We first order the MBRs by the left x-axis value, and process them in that order. We start with an empty set of maximal sets, which is kept sorted by rightmost x value in the set. Each new segment can be inserted into any such maximal set whose rightmost value does not exceed the rightmost x value of the new segment. From those, we search the one with maximum rightmost value. If no candidate exists, the new segment creates its own new maximal set.

This solution is not new. It is well known to find the longest increasing subsequence in a stream of numbers, and is also related to the problem of decomposing

a permutation Π over $\{1 \dots N\}$ into the minimum number of Shuffled (i.e., not consecutive) UpSequences [1] (the rightmost values of the MBRs correspond to the permutation values). Our algorithm is equivalent to Fredman’s [5] one to find the optimal number of Shuffled UpSequences.

4 Experiments

Our machine is an Intel Core2Duo with two processors Intel Pentium 4 CPU 3.00GHz, with 4GB of RAM. It runs GNU/Linux (kernel 2.6.27). We compiled with gcc version 4.3.2 and option `-O9`. Both synthetic and real datasets were used in our experiments. The three synthetic collections have one million MBRs each, the first one with a uniform distribution, the second one with a Zipf distribution (world size = 1000×1000 , $\rho = 1$), and the third one with a Gauss distribution (world size = 1000×1000 , $\mu = 500$, $\sigma = 200$). We created four query sets for each dataset, with different selectivities that represent 0.001%, 0.01%, 0.1%, and 1% of the area of the space where the MBRs are located. They contain 1,000 queries with the same distribution of the original datasets and the ratio between the horizontal and vertical extensions varies uniformly between 0.25 and 2.25. The algorithm generating these query sets is based on the one used in the evaluation of the R*-tree [2]. The first real collection, named Tiger dataset, contains 2,249,727 MBRs from California roads and it is available at the U.S. Census Bureau [4]. In addition, six smaller real collections available at the same place were used as query sets: Block (groups of buildings), BG (block groups), AIANNH (American Indian/Alaska Native/Native Hawaiian Areas), SD (elementary, secondary, and unified school districts), COUSUB (country subdivisions), and SLDL (state legislative districts). The second real collection, named EIEL dataset, contains 569,534 MBRs from buildings in the province of A Coruña, Spain [3]. Five smaller collections available at the same place were used as query sets: URBRU (urbanized rural places), URBRE (urbanized residential places), CENT (population centers), PAR (parishes), and MUN (municipalities).

4.1 Coordinate Encoding

Coordinate encoding does not have a key influence in search time performance (these arrays are only used to translate the queries from the geographic space to the rank space). Thus we can tolerate a small loss in performance in exchange for better compression. We performed experiments with four coding algorithms (Elias-Gamma, Elias-Delta, Rice, and VBytes) and five sampling rates h . Figure 2 shows the results of these experiments in the Zipf, Tiger, and EIEL datasets respectively. Query sets contained 1,000 uniformly distributed queries in the surface covered by each dataset with a selectivity that represents the 0.01% of the area. The four lines correspond to the coding algorithms and each point in these lines represents a different sampling rate (10, 50, 100, 1,000 and 10,000 are the different h values from left to right).

¹ <http://www.census.gov/geo/www/tiger>

² <http://www.dicoruna.es/webeiel>

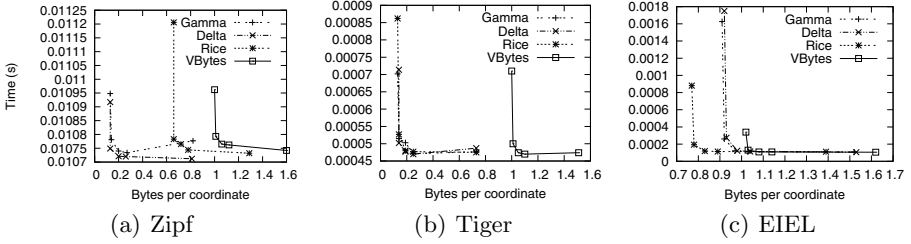


Fig. 2. Influence of the coordinate encoding

All the coding algorithms provide a good compression rate (the size is significantly lower than the 4 bytes per coordinate necessary without encoding). Elias-Gamma and Elias-Delta provide the best performance when the differences are very small (e.g., Zipf dataset), but their performance is quite worse in the EIEL dataset where the differences are larger. VBytes coding provides better time performance than the rest of the algorithms but its compression rate is not competitive. Note that VBytes works at the byte level whereas the rest work at the bit level. Hence, Rice coding can be identified as the algorithm that offers a better space/time trade-off in the majority of the situations. In addition, an interval of sampling rates providing an optimal space/time trade-off can be identified around 500. In the rest of the experiments we use a sampling rate $h = 500$ and a preprocessing stage to choose the best coding algorithm.

4.2 Space Comparison

We compare now our structure with two variants of the R-tree in terms of space needed to store the structure. The space needed by an R-tree over a collection of N MBRs can be estimated considering a certain arity (M). Dynamic versions of this structure, such as the R*-tree, estimate that nodes are 70% full whereas static versions, such as the STR R-tree, assume that nodes are full. Therefore, an R*-tree needs $\frac{N}{0.7 \times M - 1}$ nodes and an STR R-tree needs $\frac{N}{M - 1}$ nodes. Each node needs $M \times \text{sizeof}(\text{entry})$ bytes. The size of an entry is the size of an MBR plus a pointer to the child (or to the data if the node is a leaf). In order to compare these variants with our structure we assume that MBRs are stored in 16 bytes (4 coordinates with numbers of 4 bytes) and the pointer in 4 bytes. Hence, the total size of an R*-tree is $\frac{N}{0.7 \times M - 1} \times 20 \times M$ whereas the size of an STR R-tree is $\frac{N}{M - 1} \times 20 \times M$. In our experiments the best time performance of the R*-tree and STR R-tree is achieved with an effective M value of 30. Note that the coordinates stored by the R-tree are not sorted, thus it is not possible to apply our differential encoding.

On the other hand, our structure stores the encoded coordinates of the N MBRs, their identifiers (N 4-byte numbers) and the wavelet tree bitmaps (see grayed data in Figure 1). The wavelet tree needs $\lceil \log_2 2N \rceil$ levels but the number of times a MBR appears in each level is not constant (four times per level is a

pessimistic upper bound). In addition, in order to perform *rank* operations in constant time, some auxiliary structures are needed that use an additional space. In our experiments we use the classical two-level solution to perform *rank*₁ and *select*₁ over the bitmaps B_1 and B_2 (37.5% in addition to the bitmaps) and a simpler one level solution to perform *rank*₀₀ and *select*₀₀ over the virtual double bitmap that is composed of B_1 and B_2 (an additional 5%). A description and empirical comparison of these solutions can be found in [8]. As well as the size of the wavelet tree the effectiveness of the coordinates compression also varies across datasets, so we show the results for each dataset in Figure 3.

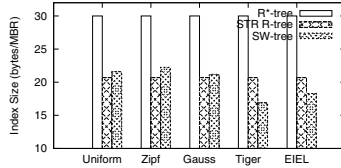


Fig. 3. Space comparison

These results show that our structure, named *SW-tree* (from *spatial wavelet tree*) in the graphs, can index collections of MBRs in less space than the R*-tree in both synthetic and real scenarios, and it also needs less space than the STR R-tree in real scenarios and a comparable space in synthetic ones. This is due to the compressed encoding of the coordinates and the little space required by the wavelet tree.

4.3 Time Comparison

To perform the time comparison we implemented our structure as described in Section 3 and used the R-tree implementation provided by the *Spatial index library* [11]. This library provides several implementations of R-tree variants such as the R*-tree and the STR packing algorithm to perform bulk loading. In addition, all these variants can run in main memory. In our experiments we run both the R*-tree and the STR R-tree in main memory with a load factor $M = 30$.

We first perform experiments with the three synthetic collections. Figures 4(a), 4(b), and 4(c) show the results obtained with uniform data, Gauss distributed data, and Zipf distributed data, respectively. The main conclusion that can be extracted from these results is that our structure is competitive with respect to query time efficiency. It outperforms both variants of the R-tree with the uniform dataset. In the other two datasets the performance of the three structures is very similar. The R-tree variants outperform our structure when the queries are very selective and in less selective queries the results are the opposite.

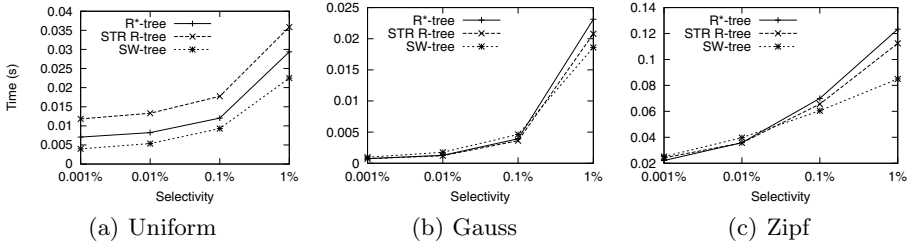


Fig. 4. Time comparison in three synthetic datasets with different distributions

Finally, we present the results with the two real datasets. Figures 5(a) and 5(b) present the results with the Tiger and EIEL datasets respectively. In these graphs the real query sets have been sorted accordingly with their selectivity (from left to right the query selectivity is looser). Note that all of them are meaningful queries. For example, in the EIEL dataset the query set CENT contains queries of the form *which buildings are contained in the population center X*. In the same way as Zipf and Gauss datasets the performance of the three structures is quite similar. Our structure outperforms both R-tree variants in less selective queries and it is less competitive in the more selective ones.

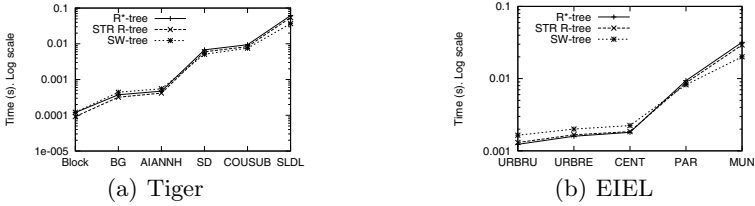


Fig. 5. Time comparison in two real datasets

5 Further Fun

The minimum number k of maximal sets that cover the MBRs can be thought of the difficulty of the problem, thus our $O(k \log N)$ time query algorithm is adaptive to this difficulty. Yet, the situation is indeed more complex (and fun). As a simple example, the number could be different if we rotated the data. For example, in the TIGER data set, we obtain 19 maximal sets in the x-axis and 36 in the y-axis. This difference is also reflected in the query time performance (for example, using the *Block* query set, the time is almost the double in the second option). A finer consideration is as follows. Assume N_1, N_2, \dots, N_k are the sizes of the k maximal sets. Then, $\sum N_i \lceil \log N_i \rceil$ is the space necessary to store the wavelet tree that solves the queries in $\sum \lceil \log N_i \rceil$ time. This is interesting because the space is a convex function whereas the time is a concave function. Therefore,

balancing the number of elements in the maximal sets improves the size of the structure whereas the opposite improves the query time performance. Hence, we can design heuristics to create the maximal sets based on this tradeoff. For example, the algorithm to create the maximal sets decomposition can choose the set that, without violating the constraints, contains fewer/more elements, minimizes $N_i \lceil \log N_i \rceil$, etc. Finally, the analysis of the query time performance can be refined by defining the complexity of the problem k as the number of maximal sets accessed to solve a query (and not all the maximal sets necessary to represent the dataset). In this case, heuristics that minimize the overlap between maximal sets can improve the query time performance. This leads us to a band-decomposition of the space very typical in some packing algorithms for spatial indexes.

References

1. Barbay, J., Navarro, G.: Compressed representations of permutations, and applications. In: Proc. 26th STACS 2009, pp. 111–122 (2009)
2. Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B.: The R*-tree: an efficient and robust access method for points and rectangles. *SIGMOD Record* 19(2), 322–331 (1990)
3. Brisaboa, N.R., Luaces, M.R., Navarro, G., Seco, D.: A new point access method based on wavelet trees. In: Proc. SeCoGIS 2009. ER 2009 Workshops, pp. 297–306 (2009)
4. Chazelle, B.: A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing* 17(3), 427–462 (1988)
5. Fredman, M.L.: On computing the length of longest increasing subsequences. *Discrete Mathematics* 11(1), 29–35 (1975)
6. Gabow, H.N., Bentley, J.L., Tarjan, R.E.: Scaling and related techniques for geometry problems. In: Proc. 16th STOC, pp. 135–143 (1984)
7. Gaede, V., Günther, O.: Multidimensional access methods. *ACM Computing Surveys* 30(2), 170–231 (1998)
8. González, R., Grabowski, S., Mäkinen, V., Navarro, G.: Practical implementation of rank and select queries. In: Proc. 4th WEA (Poster), pp. 27–38 (2005)
9. Grossi, R., Gupta, A., Vitter, J.: High-order entropy-compressed text indexes. In: Proc. 14th ACM-SIAM SODA, pp. 841–850 (2003)
10. Guttman, A.: R-Trees: A Dynamic Index Structure for Spatial Searching. In: Proc. SIGMOD, pp. 47–57. ACM Press, New York (1984)
11. Hadjieleftheriou, M.: Spatial index library, <http://research.att.com/~mariah/spatialindex/> (retrieved March 2009)
12. Leutenegger, S., Lopez, M., Edgington, J.: STR: A simple and efficient algorithm for R-tree packing. In: Proc. 13th ICDE, pp. 497–506 (1997)
13. Manolopoulos, Y., Nanopoulos, A., Papadopoulos, A.N., Theodoridis, Y.: R-Trees: Theory and Applications. Springer, Heidelberg (2005)
14. Navarro, G., Mäkinen, V.: Compressed full-text indexes. *ACM Computing Surveys* 39(1) (2007)
15. Salomon, D.: Data Compression: The Complete Reference. Springer, Heidelberg (2004)

On Table Arrangements, Scrabble Freaks, and Jumbled Pattern Matching^{*}

Péter Burcsi¹, Ferdinando Cicalese², Gabriele Fici³, and Zsuzsanna Lipták⁴

¹ Department of Computer Algebra, Eötvös Loránd University, Hungary
bupe@compalg.inf.elte.hu

² Dipartimento di Informatica ed Applicazioni, University of Salerno, Italy
cicalese@dia.unisa.it

³ I3S, UMR6070, CNRS et Université de Nice-Sophia Antipolis, France
fici@i3s.unice.fr

⁴ AG Genominformatik, Technische Fakultät, Bielefeld University, Germany
zsuzsa@cebitec.uni-bielefeld.de

Abstract. Given a string s , the Parikh vector of s , denoted $p(s)$, counts the multiplicity of each character in s . Searching for a match of Parikh vector q (a “jumbled string”) in the text s requires to find a substring t of s with $p(t) = q$. The corresponding decision problem is to verify whether at least one such match exists. So, for example for the alphabet $\Sigma = \{a, b, c\}$, the string $s = abaccbabaaa$ has Parikh vector $p(s) = (6, 3, 2)$, and the Parikh vector $q = (2, 1, 1)$ appears once in s in position $(1, 4)$. Like its more precise counterpart, the renown Exact String Matching, Jumbled Pattern Matching has ubiquitous applications, e.g., string matching with a dyslectic word processor, table rearrangements, anagram checking, Scrabble playing and, *allegedly*, also analysis of mass spectrometry data. We consider two simple algorithms for Jumbled Pattern Matching and use very complicated data structures and analytic tools to show that they are not worse than the most obvious algorithm. We also show that we can achieve non-trivial efficient average case behavior, but that’s less fun to describe in this abstract so we defer the details to the main part of the article, to be read at the reader’s risk. . . well, at the reader’s discretion.

1 Prologue

Last month, I happened to organize a workshop at my university. We ended up being 20 people, so, for a little social dinner, I decided to call a friend who owns a restaurant in town, and asked him to prepare a table for 20 people.

My friend, who always likes to make jokes, decided to tease me a bit and on arrival at the restaurant we found a table laid for 100 people. However, instead of having exactly one fork, one knife, and one spoon at each plate, my friend

^{*} Part of this work was done while F.C. and Zs.L. were visiting the Alfréd Rényi Institute of Mathematics in Budapest, Hungary, within the EU Marie Curie Transfer of Knowledge project “Hungarian Bioinformatics (HUBI).”

had put all the 300 pieces of cutlery 3 by 3 but completely at random. I was about to faint. What would my scholarly friends think of me! What a horrible impression they would get of my country's hospitality! So I hurried to my friend and told him to solve the problem immediately. I said, "unless you find a way to have us all sit next to each other, we are all going to McDonald's!"

My friend got pale at the prospect of losing 20 customers, but did not lose his spirit. He knew we were computer scientists, so, "in order to speed up things," he said, "please quickly check whether there are 20 consecutive places where you can find 20 knives, 20 forks and 20 spoons, and I will proceed to rearrange them properly." Of course we found the task amusing, and we ended up spending the rest of the evening discussing the following article

2 Definitions and Problem Statement

Caveat: Since this is a *fun* paper on jumbled pattern matching, the paper itself is also jumbled. Readers who prefer a more classic structure are advised to read Section 6 first, which details motivations and related work.

Given a finite ordered alphabet $\Sigma = \{a_1, \dots, a_\sigma\}, a_1 \leq \dots \leq a_\sigma$. For a string $s \in \Sigma^*$, $s = s_1 \dots s_n$, the *Parikh vector* $p(s) = (p_1, \dots, p_\sigma)$ of s defines the multiplicities of the characters in s , i.e. $p_i = |\{j \mid s_j = a_i\}|$, for $i = 1, \dots, \sigma$. For a Parikh vector p , the *length* $|p|$ denotes the length of a string with Parikh vector p , i.e. $|p| = \sum_i p_i$. An *occurrence* of a Parikh vector p in s is an occurrence of a substring t with $p(t) = p$. (An occurrence of t is a pair of positions $0 \leq i \leq j \leq n$, such that $s_i \dots s_j = t$.) A Parikh vector that occurs in s is called a sub-Parikh vector of s . The prefix of length i is denoted $pr(i) = pr(i, s) = s_1 \dots s_i$, and the Parikh vector of $pr(i)$ as $prv(i) = prv(i, s) = p(pr(i))$.

For two Parikh vectors $p, q \in \mathbb{N}^\sigma$, we define $p \leq q$ and $p + q$ component-wise: $p \leq q$ if and only if $p_i \leq q_i$ for all $i = 1, \dots, \sigma$, and $p + q = u$ where $u_i = p_i + q_i$ for $i = 1, \dots, \sigma$. Similarly, for $p \leq q$, we set $q - p = v$ where $v_i = q_i - p_i$ for $i = 1, \dots, \sigma$.

Jumbled Pattern Matching (JPM). Let $s \in \Sigma^*$ be given, $|s| = n$. For a Parikh vector $q \in \mathbb{N}^\sigma$ (the query), $|q| = m$, find all occurrences of q in s . The *decision version* of the problem is where we only want to know whether q occurs in s .

We assume that K many queries arrive over time, so some preprocessing may be worthwhile.

Note that for $K = 1$, both the decision version and the occurrence version can be solved worst-case optimally with a simple window algorithm, which moves a fixed size window of size m along string s . Maintain the Parikh vector c of the current window and a counter r which counts indices i such that $c_i \neq q_i$. Each sliding step costs either 0 or 2 update operations of c , and possibly one increment or decrement of r . This algorithm solves both the decision and occurrence problems and has running time $\Theta(n)$, using additional storage space $\Theta(\sigma)$.

Precomputing, sorting, and storing all sub-Parikh vectors of s would lead to $\Theta(n^2)$ storage space, since there are non-trivial strings with a quadratic number of Parikh vectors over arbitrary alphabets [11]. Such space usage is unacceptable in many applications.

For small queries, the problem can be solved exhaustively with a linear size indexing structure such as a suffix tree, which can be searched down to length $m = |q|$ (of the substrings), yielding a solution to the decision problem in time $O(\sigma^m)$. For finding occurrences, report all leaves in the subtrees below each match; this costs $O(M)$ time, where M is the number of occurrences of q in s . Constructing the suffix tree takes $O(n)$ time, so for $m = o(\log n)$, we get a total runtime of $O(n)$, since $M \leq n$ for any query q .

3 Decision Problem in the Binary Case

In [10], an algorithm (Interval Algorithm) was presented which solved the decision problem on binary alphabets in constant time per query, using linear storage space. However, it needed $\Theta(n^2)$ time for the preprocessing phase. In this section, we improve that preprocessing time to $O(n^2/\log n)$. We first recall the Interval Algorithm, which makes use of the following property of binary strings:

Lemma 1 ([10], Lemma 3). *Let $s \in \{a, b\}^*$ with $|s| = n$. Fix $1 \leq m \leq n$. If the Parikh vectors $(x_1, m - x_1)$ and $(x_2, m - x_2)$ both occur in s , then so does $(y, m - y)$ for any $x_1 \leq y \leq x_2$.*

This means that the Parikh vectors of substrings of s of length m build a set of the form $\{(x, m - x) \mid x = \text{pmin}(m), \text{pmin}(m) + 1, \dots, \text{pmax}(m)\}$ for appropriate $\text{pmin}(m)$ and $\text{pmax}(m)$. The algorithm computes these values in a preprocessing step; then, when a query $q = (x, y)$ arrives, it answers *yes* if and only if $x \in [\text{pmin}(x + y), \text{pmax}(x + y)]$. The query time is $O(1)$. We now show how to reduce the preprocessing problem to a $(\min, +)$ -convolution problem, giving subquadratic running time.

Let $\underline{x} = (x_0, x_1, \dots, x_n)$ and $\underline{y} = (y_0, y_1, \dots, y_n)$ be two real vectors. The $(\min, +)$ -convolution, minimum convolution or simply \min -convolution of \underline{x} and \underline{y} is the vector $\underline{z} = \underline{x} \star \underline{y} = (z_0, z_1, \dots, z_{2n})$ with $z_m = \min(x_i + y_{m-i})$, where the minimum is taken over all possible values of i . (Standard convolution or $(+, \cdot)$ -convolution is obtained if \min is replaced by \sum and $x_i + y_{m-i}$ by $x_i \cdot y_{m-i}$.) We note that there are variants of the definition, e.g. the index $m - i$ is sometimes understood mod n or \underline{z} is of the same length as \underline{x} and \underline{y} . These are easily reducible to each other.

Min-convolution has first been used in optimization problems [3], but it also has applications in computer vision and signal processing [2], sequence alignment [13] and sequential data analysis [15]. The currently known best algorithm (in worst-case sense and in the RAM model) was introduced in [7] and runs in slightly subquadratic time: $O(n^2/\log n)$. This algorithm reduces min-convolution to a problem in computational geometry, that of finding *dominating pairs*, which is discussed and analyzed in [9].

We briefly describe how the problem of determining the values $\text{pmin}(m)$ and $\text{pmax}(m)$ for $m = 1, 2, \dots, n$ can be reduced to min-convolution. Let s be a string of length n , and let x_i be the number of characters a in $\text{pr}(i)$, $y_{n-i} = -x_i$ for $i = 0, 1, \dots, n$. Then

$$\text{pmin}(m) = \min_{i=0}^{n-m} (x_{i+m} - x_i) = \min_{i=0}^{n-m} (x_{i+m} + y_{n-i}) = z_{n+m} \quad , \quad (1)$$

where $\underline{z} = \underline{x} \star \underline{y}$. The maximum can be calculated analogously.

In order to make the presentation self-contained, we also describe the problem of finding dominating pairs, and how min-convolution can be solved by it. For the analysis of running times, we refer to [9, 7].

An instance of the dominating pairs problem consists of a red set and a blue set of points in d -dimensional space as input, and asks for all pairs of points (α, β) , where α is red and β is blue and $\alpha \leq \beta$ componentwise, that is $\alpha_i \leq \beta_i$ for $i = 1, 2, \dots, d$. Chan [9] solves the problem with the following divide-and-conquer algorithm. If there is only one point, there's nothing to do. If $d = 0$, then all red-blue pairs are dominating. Otherwise calculate the median ζ of the d -th coordinates of all points and divide red and blue points into left and right sets R_l, R_r and B_l, B_r according to the relationship of their d -th coordinates to ζ . Finally, solve the problem recursively for $R_l \cup B_l, R_r \cup B_r$ and for the projection of $R_l \cup B_r$ to the first $d - 1$ coordinates. It turns out that for any $\varepsilon \in (0, 1)$ the running time of the algorithm is $O(c_\varepsilon^d n^{1+\varepsilon} + D)$, where n is the number of input points, D is the number of dominating pairs, and $c_\varepsilon = 2^\varepsilon / (2^\varepsilon - 1)$.

Let \underline{x} and \underline{y} be two vectors whose min-convolution has to be computed. Fix d (to be defined later). For each $\delta \in \{0, 1, \dots, d - 1\}$ we define a dominating pairs problem for the following set of red and blue points:

$$\begin{aligned} R &= \{\alpha_i = (x_{i+\delta} - x_i, x_{i+\delta} - x_{i+1}, \dots, x_{i+\delta} - x_{i+d-1}) \mid i = 0, d, \dots, \lfloor n/d \rfloor d\} \\ B &= \{\beta_j = (y_j - y_{j-\delta}, y_{j-1} - y_{j-\delta}, \dots, y_{j-d+1} - y_{j-\delta}) \mid j = 0, 1, \dots, n\} \end{aligned}$$

For indices out of range, take the components of x and y to be ∞ . We get a dominating pair (α_i, β_j) if and only if $x_{i+\delta} + y_{j-\delta} \leq x_{i+k} + y_{j-k}$ for $k = 0, 1, \dots, d - 1$. We collect all dominating pairs for all δ . Then, in the min-convolution calculation of z_{i+j} only those values $x_{i+\delta} + y_{j-\delta}$ have to be considered that come from a dominating pair. For each pair (i, j) (i a multiple of d and j arbitrary), only one such δ exists¹, therefore we gain a factor of d in the calculation of the minima. Choosing $d = \log_2 n/2$ gives an overall running time of $O(n^2 / \log n)$.

Example 2. Let the two characters of the alphabet be a and b , and let $s = aabba$. Then $\underline{x} = (0, 1, 2, 2, 2, 3)$, $\underline{y} = (-3, -2, -2, -2, -1, 0)$. We agree on breaking ties the following way: if $x_i + y_j = x_{i'} + y_{j'}$, $i + j = i' + j'$, then $x_i + y_j$ is considered smaller if and only if $i < i'$. For the presentation we choose $d = 3$. For $\delta = 0$ we list the red and blue sets $R = \{(0, -1, -2), (0, 0, -1)\}$ and $B = \{(0, \infty, \infty), (0, -1, \infty), (0, 0, -1), (0, 0, 0), (0, -1, -1), (0, -1, -2)\}$. The

¹ We assume that ties are broken in a consistent manner.

dominating pairs $(i + \delta, j - \delta)$ are $\{(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (3, 0), (3, 2), (3, 3)\}$. For $\delta = 1$ the dominating $(i + \delta, j - \delta)$ are $\{(4, 0), (4, 3), (4, 4), (4, 5)\}$ and for $\delta = 2$, $\{(5, 5)\}$. We then calculate \underline{z} based on these pairs. For example, $z_5 = \min\{x_0 + y_5, x_3 + y_2\} = 0$ and $z_7 = \min\{x_4 + y_3\} = 0$. Thus, for the computation of $\text{pmin}(2) = z_7$, we needed to consider only 1 sum instead of 4 acc. to [\[1\]](#), namely $z_7 = \min\{x_2 + y_5, x_3 + y_4, x_4 + y_3, x_5 + y_2\}$. We get $\underline{z} = (-3, -2, -2, -2, -1, 0, 0, 0, 1, 2, 3)$. We conclude that $\text{pmin}(m) = 0, 0, 1, 2, 3$ for $m = 1, 2, 3, 4, 5$. Note that if needed, the corresponding positions can be obtained.

In [\[10\]](#) it was conjectured that no subquadratic algorithm for this preprocessing phase exists. Informally, the reason to believe so was that if one computes the minimum for a single value m , the optimal running time is trivially linear (because one has to read the string), and knowing the minimum for one value does not give any useful information for the others. There is, however, some locality in the problem: if the substring $s_i \dots s_{i+m-1}$ has at least $\text{pmin}(m) + 2$ characters equal to a , then it is not the position for a minimal substring of length $m + 1$ either. This is the kind of locality that is exploited in the above algorithm. The logarithmic gain is too small for most practical applications, however. It remains open if an $O(n^{2-\varepsilon})$ algorithm exist for the calculation of pmin and pmax .

4 General Alphabets

An algorithm for the general case was presented in [\[10\]](#), whose expected running time was shown to be $O(n(\frac{\sigma}{\log \sigma})^{1/2} \frac{\log m}{\sqrt{m}})$, using $O(n)$ space, with preprocessing time $O(n)$. In this section, we show how to use wavelet trees to implement this algorithm, and thus improve the average runtime to $O(n(\frac{\sigma}{\log \sigma})^{1/2} \frac{1}{\sqrt{m}})$, making it competitive with the window algorithm as soon as $m = \omega(\sigma / \log \sigma)$, i.e. in practically all cases. Space requirements and preprocessing time remain the same.

4.1 The Jumping Algorithm

We give a brief explanation of the algorithm. For an illustration and the pseudocode, refer to Figs. [\[1\]](#) and [\[2\]](#).

Recall that $\text{prv}(j) = p(s_1 \dots s_j)$ is the Parikh vector of the prefix of s of length j . The algorithm makes use of the simple observation that q has an occurrence at position $(i + 1, j)$ if and only if $\text{prv}(j) - \text{prv}(i) = q$. Imagine moving two pointers L and R along s , which point to these potential positions i and j . We alternate in updating L and R : In each update, either L or R is moved to the right. The invariant is that $p(R - L) \geq q$ after each update of R , and $p(R - L) \leq q$ after each update of L (Lemma 2 of [\[10\]](#)). We need the following function:

$$\text{FIRSTFIT}(p) := \min\{j \mid \text{prv}(j) \geq p\}. \quad (2)$$

The update rules are as follows:

- update R : Both $prv(L)$ and q must fit before the new positions of R , so move R to the first index j s.t. $prv(j) \geq prv(L) + q$. So $R \leftarrow \text{FIRSTFIT}(prv(L) + q)$.
- update L : $L \leftarrow L + 1$ if a match was found, else:
 Some characters of $p(R - L)$ are unnecessary and have to be accommodated before the new position of L : $L \leftarrow \text{FIRSTFIT}(prv(R) - q)$.

After each update of R or L , we check whether there is a match *by checking whether $R - L = |q|$* . This is correct due to the invariants above. So no matching or accessing the string s is ever done. The complexity of the algorithm depends on how the functions $prv(j)$ and FIRSTFIT are implemented.

Example 3. Consider the string $s = bbacaccababbabccaaac$ and query $q = (3, 1, 2)$, which has 3 occurrences in s , namely $(5, 10)$, $(13, 18)$, and $(14, 19)$. R assumes the values 8, 10, 13, 18, 19, and thus, the while-loop is executed 5 times (see Fig. 2).

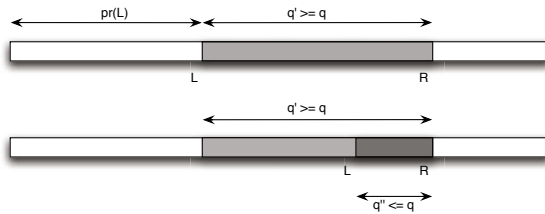


Fig. 1. The situation after the update of R (above) and after the update of L (below). R is placed at the first fit of $prv(L) + q$, thus q' is a super-Parikh vector of q . Then L is placed at the beginning of the longest good suffix ending in R , so q'' is a sub-Parikh vector of q .

4.2 Improved Running Time Using Wavelet Trees

In [10] we used an inverted table for computing the two functions $prv(j)$ and FIRSTFIT . It is very easy to understand and to implement, takes $O(n)$ space and $O(n)$ time to construct (both with constant 1), and can replace the string. Here we replace this data structure with a wavelet tree [17].

A wavelet tree on $s \in \Sigma^*$ allows *rank*, *select*, and *access* queries in time $O(\log \sigma)$. For $a_k \in \Sigma$, $rank_k(s, i) = |\{j \mid s_j = a_k, j \leq i\}|$, the number of occurrences of character a_k up to and including position i , while $select_k(s, i) = \min\{j \mid rank_k(s, j) \geq i\}$, the position of the i 'th occurrence of character a_k . When the string is clear, we just use $rank_k(i)$ and $select_k(i)$. Notice that

- $prv(j) = (rank_1(j), \dots, rank_\sigma(j))$, and
- for a Parikh vector $p = (p_1, \dots, p_\sigma)$, $\text{FIRSTFIT}(p) = \max_{k=1, \dots, \sigma} \{select_k(p_k)\}$.

So we can use a wavelet tree of string s to implement those two functions. We give a brief recap of wavelet trees, and then explain how to implement the two functions above in $O(\sigma)$ time each.

Algorithm *Jumping Algorithm***Input:** query Parikh vector q **Output:** A set Occ containing all beginning positions of occurrences of q in s

1. set $m \leftarrow |q|$; $Occ \leftarrow \emptyset$; $L \leftarrow 0$;
2. **while** $L < n - m$
3. **do** $R \leftarrow \text{FIRSTFIT}(prv(L) + q)$;
4. **if** $R - L = m$
5. **then** add $L + 1$ to Occ ;
6. $L \leftarrow L + 1$;
7. **else** $L \leftarrow \text{FIRSTFIT}(prv(R) - q)$;
8. **if** $R - L = m$
9. **then** add $L + 1$ to Occ ;
10. $L \leftarrow L + 1$;
11. **return** Occ ;

Fig. 2. Pseudocode of Jumping Algorithm

A wavelet tree is a complete binary tree with $\sigma = |\Sigma|$ many leaves. To each inner node, a bitstring is associated which is defined recursively, starting from the root, in the following way. If $|\Sigma| = 1$, then there is nothing to do (in this case, we have reached a leaf). Else split the alphabet into two roughly equal parts, Σ_{left} and Σ_{right} . Now construct a bitstring of length n from s by replacing each occurrence of a character a by 0 if $a \in \Sigma_{\text{left}}$, and by 1 if $a \in \Sigma_{\text{right}}$. Let s_{left} be the subsequence of s consisting only of characters from Σ_{left} , and s_{right} that consisting only of characters from Σ_{right} . Now recurse on the left child with string s_{left} and alphabet Σ_{left} , and on the right child with s_{right} and Σ_{right} . An illustration is given in Fig. 3. At each inner node, in addition to the bitstring B , we have a data structure of size $o(|B|)$, which allows to perform *rank* and *select* queries on bit vectors in constant time ([21][22]).

Now, using the wavelet tree of s , any *rank* or *select* operation on s takes time $O(\log \sigma)$, which would yield $O(\sigma \log \sigma)$ time for both $prv(j)$ and $\text{FIRSTFIT}(p)$. However, we can implement both in a way that they need only $O(\sigma)$ time: In order to compute $rank_k(j)$, the wavelet tree, which has $\log \sigma$ levels, has to be descended from the root to leaf k . Since for $prv(j)$, we need all values $rank_1(j), \dots, rank_\sigma(j)$ simultaneously, we traverse the complete tree in $O(\sigma)$ time.

For computing $\text{FIRSTFIT}(p)$, we need $\max_k \{select_k(p_k)\}$, which can be computed bottom-up in the following way. We define a value x_u for each node u . If u is a leaf, then u corresponds to some character $a_k \in \Sigma$; set $x_u = p_k$. For an inner node u , let B_u be the bitstring at u . We define x_u by $x_u = \max \{select_0(B_u, x_{\text{left}}), select_1(B_u, x_{\text{right}})\}$. The desired value is equal to x_{root} .

Example 4. Let $s = bbacaccabaddabccaaac$ (cp. Fig. 3). We demonstrate the computation of $\text{FIRSTFIT}(2, 3, 2, 1)$ using the wavelet tree. We have $\text{FIRSTFIT}(2, 3, 2, 1) = \max \{select_a(s, 2), select_b(s, 3), select_c(s, 2), select_d(s, 1)\}$, where in

slight abuse of notation we put the character in the subscript instead of its number. Denote the bottom left bitstring as $B_{a,b}$, the bottom right one as $B_{c,d}$, and the top bitstring as $B_{a,b,c,d}$. Then we get $\max\{\text{select}_0(B_{a,b}, 2), \text{select}_1(B_{a,b}, 3)\} = \max\{4, 6\} = 6$, and $\max\{\text{select}_0(B_{c,d}, 2), \text{select}_1(B_{c,d}, 1)\} = \max\{2, 4\} = 4$. So at the next level, we compute $\max\{\text{select}_0(B_{a,b,c,d}, 6), \text{select}_1(B_{a,b,c,d}, 4)\} = \max\{9, 11\} = 11$.

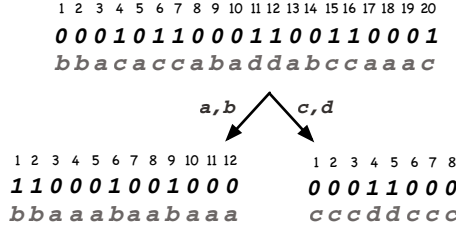


Fig. 3. The wavelet tree for string *bbacaccabaddabccaaac*. For clarity, the leaves have been omitted. Note also that the third line at each inner node (the strings over the alphabet $\{a, b, c, d\}$) are only included for illustration.

Analysis: Let J denote the number of times the while-loop in line 2 (see Fig. 2) is executed. The work done in each iteration is dominated by the computation of $\text{FIRSTFIT}(\text{prv}(L + q))$ (line 3) and $\text{FIRSTFIT}(\text{prv}(R - q))$ (line 7). Since both FIRSTFIT and prv can be computed in $O(\sigma)$ time, we have that the total runtime of the algorithm is $O(J\sigma)$. Since in every iteration, R is moved forward, $J = O(n)$; however, since there are cases where $J = n/2$, such as searching for $(2, 0)$ on string $(ab)^n$, we also have $J = \Theta(n)$. So worst-case running time of the Jumping Algorithm is $\Theta(\sigma n)$, i.e., slower than the window algorithm.

In [10], the expected value of J was shown to be $\mathbb{E}(J) = O(\frac{n}{\sqrt{m}\sqrt{\sigma \log \sigma}})$ on uniformly distributed strings and queries. This yielded an expected runtime of $O((\frac{\sigma}{\log \sigma})^{1/2} \frac{\log m}{\sqrt{m}} n)$ using the inverted prefix table. With the wavelet tree implementation, we get expected runtime $O((\frac{\sigma}{\log \sigma})^{1/2} \frac{1}{\sqrt{m}} n)$, i.e., we save a $\log m$ factor. Moreover, the new version is sublinear as soon as $m = \omega(\sigma / \log \sigma)$. Pre-processing is still linear in n , since the wavelet tree of s can be constructed in $O(n)$ time.

The space required by the wavelet tree is $\lceil \log \sigma \rceil (n + o(n))$ bits, since each level requires $n + o(n)$ bits. Our previous implementation needed $n \log n$ bits for the inverted prefix table. With the usual assumptions on the RAM model, namely that $\log n$ fits into a computer word, both are $O(n)$; however, on a bit level, we improve.

5 Variations: Sub-Parikh Vectors and Scrabble-Like Games

Consider the following game, which can be played using the Scrabble set you probably have at home. The only other thing you need is a random text, maybe from a newspaper or from the Internet. Each player draws 8 letters from the sack of letters. When it is her turn, she aligns a word made with her letters to a position in the text, trying to maximize the total score, which is the sum of the scores of the individual letters used. Then she draws new letters from the sack, until she has again 8 letters in front of her. Overlapping alignments are allowed, so the same position in the text can be matched more than once. The game ends when all letters finish, or when no player can move, and is won by the player with the highest total score.

If we refer to the player's current Parikh vector (the contents of the tray) as q , then the task is to find a jumbled match of a sub-Parikh vector q' of q to the text, under the constraint that the substring matched be a word of English. Moreover, we want to maximize the score of q' .

Note that in the game described above, maximizing the score in a single move does not necessarily result in an optimal game strategy, since you might save a letter for the next round if a lucky new letter allows for a much higher score. However, for our purposes, we will assume the simple strategy of maximizing in each move. Further, let us drop the constraint of the substring being a word of English—these simplifications will render the game more tractable (in the non-technical sense).

So, given a weight function $w : \Sigma \rightarrow \mathbb{R}^+$ on the characters (the scores), we have the following problem:

Jumbled sub-Pattern Matching (JSPM). Let $s \in \Sigma^*$ be given, $|s| = n$. For a Parikh vector $q \in \mathbb{N}^\sigma$ (the query), $|q| = m$, find all occurrences q' in s s.t. $q' \leq q$ with maximum weight.

Like the original JPM also this variant can be solved in linear time (in n) with the following simple variant of the window-algorithm. Slide a variable sized window over s , keeping the Parikh vector of the current window content in a vector c . Start with both pointers L and R pointing to the beginning of s , so $c = (0, 0, \dots, 0)$. While $c \leq q$, move R towards the right one character at a time. Now make a note of $w(c)$, the weight of the current vector, increment R , and move L to the right until again $c \leq q$. Move again R as long as $c \leq q$. Now check the value of $w(c)$ against the previous value, replace if greater, and so on.

Clearly the above procedure finds all maximal (non-extendable) sub-Parikh vectors q' of q that occur in s , and thus also those with maximum $w(q')$.

Naturally, the reader might wonder whether the Jumping Algorithm might be also adapted to this optimization problem. In fact, it is not hard to come up with a simple variant of the Jumping Algorithm which solves JSPM.

First note that the movement of L is exactly as in the Jumping Algorithm, namely $L \leftarrow \text{FIRSTFIT}(prv(R) - q)$. Next, define $\text{TIFTSRIF}(p)$ as $\text{FIRSTFIT}(p)$ on the reverse string s^{rev} , and $vrp(i)$ as the prefix Parikh vector of position $n - i + 1$ in the reverse string! Then, R is updated to $\text{TIFTSRIF}(vrp(L) - q)$. This is because R is now moved the same way as L usually is, but from the back: R is moved to the position furthest from L such that the current window is still a sub-Parikh vector of q . After each such jump of R , check $w(c)$ and keep track of maximum so far. If all occurrences are needed, make note of positions, too. So the loop is: $R++$, update L , update R , check and maybe update $w(c)$.

For the implementation, we need a way to compute TIFTSRIF and vrp , which can be easily achieved without increasing time or space complexity by using a second wavelet tree for the reverse string. However, this still results in doubling the storage space. Instead, one can implement these functions using only the wavelet tree of the string itself by noting that $\text{TIFTSRIF}(vrp(L) - q) = \min_a \{select_a(prv(L)_a + q_a + 1)\} - 1$, i.e., R is updated to the left of the first position where the allowed number of characters a is exceeded for at least one a . We can compute $prv(L)$ first in $O(\sigma)$ time, and then compute the above minimum bottom-up, also in $O(\sigma)$ time, analogously to the maximum for FIRSTFIT .

More delicate is the question of whether the above variant of the Jumping Algorithm is also competitive in running time with the window algorithm. As we will immediately see, in the worst case there is again an additional factor σ .

Again each iteration of the loop needs $O(\sigma)$ time. So we have $O(\sigma J)$, with J denoting the number of iterations of the while-loop. Since R and L are always incremented, we have $J \leq n$, thus the running time is $O(\sigma n)$. In fact, it is not hard to see that this is also a lower bound: Consider the case $\Sigma = \{a, b, c\}$, with $s = a^{n-2}bc$ and $q = (0, 1, 1)$.

5.1 Average Case: Skewed Distributions and Skewed Patterns Help

Assume that each character of the string s is generated independently according to some fixed probability distribution $\mu = (\mu_1, \dots, \mu_\sigma)$, where the probability of seeing character a_k is μ_k , for $k = 1, \dots, \sigma$.

Let $q = (x_1 - 1, \dots, x_\sigma - 1)$ be the pattern Parikh vector, thus $1 < x_k$, for each $k = 1, \dots, \sigma$. Let T_j be the random variable which takes value i if the Parikh vector of the substring $s_{j+1}s_{j+2} \dots s_{j+i-1}$ is a sub-Parikh vector of q but $p(s_{j+1} \dots s_{j+i})$ is not. Note that this means that i is the first position where one of the entries of q is exceeded. Because of the i.i.d. model we chose for the generation of the string s , it follows that $T_i \sim T_j$ and thus $\mathbb{E}[T_i] = \mathbb{E}[T_j]$, for any i, j . So we can use $\mathbb{E}[T]$, making explicit the independence from the position in the string.

By the results in [19,20] (see also [16]) we have that, if $x_k \gg \mu_k$ for $k = 1, \dots, \sigma$, then (asymptotically with $|q|$),

$$\mathbb{E}[T] \approx \min_{k=1, \dots, \sigma} \frac{x_k}{\mu_k}. \tag{3}$$

We can recast the above problem in a generalization of the classical Problem of the Points, originating from a correspondence between Pascal and Fermat or, equivalently, of the Banach match-box problem: Given is a die with σ faces, where the k 'th face appears with probability μ_k . There are σ players, and player k gets a point if the k 'th face comes up and wins as soon as she accumulates x_k points. The question is to find the expected number of tosses before the game ends (i.e., some player wins). The distribution of the finishing time for each player, i.e., the time when she has accumulated the required number of points, follows the negative binomial distribution, with expectation given by $\frac{x_k}{\mu_k}$. The results mentioned above and summarized by Eq. (3) are to the effect that the expected time when some player wins is asymptotically equal to the minimum of the expected victory time of the individual players.

We can use this to provide a bound on the expected running time of our algorithm. First assume that the minimum in (3) is attained by exactly one k^* . Then, $k^* = \operatorname{argmin}_{k=1, \dots, \sigma} \frac{x_k}{\mu_k}$, and we have that the expected position where the algorithm places the pointer R is given by $\mathbb{E}[T] = \frac{x_{k^*}}{\mu_{k^*}}$. Moreover, $s_R = a_{k^*}$ and $\operatorname{prv}(R) - \operatorname{prv}(L)$ contains exactly $x_{k^*} - 1$ many a_{k^*} 's.

Now pointer L is moved towards the right until the first new a_{k^*} is encountered. The average length of this displacement is $\frac{1}{\mu_{k^*}}$. Then the right pointer R is moved again to the furthest position from L such that the Parikh vector of the string between L and R is a sub-Parikh vector of q , and then incremented by 1. It is the difference between this new position of the right pointer and its previous position which is significant for our analysis. Because of the assumption of an i.i.d. model, the expected new position of the pointer R given by $\mathbb{E}[T_L]$ is equal to $\mathbb{E}[T] = \frac{x_{k^*}}{\mu_{k^*}}$. Thus the pointer R jumps by $\frac{1}{\mu_{k^*}}$. The same argument is clearly also valid for the following jumps.

If there is more than one index attaining the minimum in (3), then set $k^* = \operatorname{argmax}_l \{ \mu_l \mid \frac{x_l}{\mu_l} = \min_{k=1, \dots, \sigma} \frac{x_k}{\mu_k} \}$. Then the above analysis goes through as an upper bound on the expected number of jumps. The above discussion leads to:

Proposition 5. *Let s be a randomly generated string over $\Sigma = \{a_1, \dots, a_\sigma\}$ such that $\Pr(s_i = a_k) = \mu_k$, for each $i = 1, \dots, n$ and $k = 1, \dots, \sigma$. Let $q = (x_1 - 1, \dots, x_\sigma - 1)$ be the query Parikh vector, and let $k^* = \operatorname{argmax}_l \{ \mu_l \mid \frac{x_l}{\mu_l} = \min_{k=1, \dots, \sigma} \frac{x_k}{\mu_k} \}$. Then the expected running time of the Jumping Algorithm on this instance of the JSPM problem is $O(n\sigma\mu_{k^*})$.*

6 Epilogue

Armed with our solutions, we quickly solved our table rearrangement problem, had a very satisfying dinner and FUN. The next day, a famous colleague who had enjoyed the restaurant's wine assortment the most, phoned me to communicate the following—I am quoting literally, since I haven't checked, trusting the everlasting "*In vino veritas*" [Pliny the Elder, *Naturalis historia* 14, 141].

Applications of our algorithms, apart from Scrabble and table cutlery arrangement, can be found in molecular biology, notably in interpretation of mass spectrometry data. The output of an experiment consists of the molecular masses

of sample molecules, whose molecular composition (e.g., the multiplicities of the different amino acids for proteins, or of nucleotides for DNA) can in certain cases be determined efficiently up to a few candidates [6]. In other words, several candidate Parikh vectors can be computed, and then those matched against a database of sequences. Parikh vectors have also been used in other bioinformatics applications, among them alignment [4] (there referred to as compositions), SNP discovery [5] (compomers), repeated pattern discovery [14] and gene clusters [23] (permuted patterns, π patterns).

Jumbled pattern matching is a special case of approximate pattern matching. It has been used as filtering step in approximate pattern matching algorithms [18], but rarely considered in its own right.

Three of the authors of the present paper described two algorithms for JPM in [10], Interval Algorithm and Jumping Algorithm, both of which have been crucially improved here. The authors of [8] presented an algorithm for finding all occurrences of a Parikh vector in a runlength encoded text. The algorithm's time complexity is $O(n' + \sigma)$, where n' is the length of the runlength encoding of s . Obviously, if the string is not runlength encoded, a preprocessing phase of time $O(n)$ has to be added. However, this may still be feasible if many queries are expected. To the best of our knowledge, this is the only other algorithm that has been presented for the problem we treated here.

An efficient algorithm for computing all Parikh fingerprints of substrings of a given string was developed in [1]. Parikh fingerprints are Boolean vectors where the k 'th entry is 1 if and only if a_k appears in the string. The algorithm involves storing a data point for each Parikh fingerprint, of which there are at most $O(n\sigma)$ many. This approach was adapted in [14] for Parikh vectors and applied to identifying all repeated Parikh vectors within a given length range; using it to search for queries of arbitrary length would imply using $\Omega(P(s))$ space, where $P(s)$ denotes the number of different Parikh vectors of substrings of s . This is not desirable, since there are strings with quadratic $P(s)$ [11].

6.1 Postscriptum (for Those Who Read the Technical Parts)

On the agenda for future work is refining the analysis of the Jumping Algorithm and the preprocessing time for the Interval Algorithm.

We remark that our new implementation of the Jumping Algorithm using rank/select operations only, opens a new perspective on the study of Parikh vector matching. We have made another family of approximate pattern matching problems accessible to the use of self-indexing data structures [22]. We are in particular interested in compressed data structures which allow fast execution of rank and select operations, while at the same time using reduced storage space for the text. Thus, every step forward in this very active area can provide improvements for our problem.

Acknowledgements. Thanks to Gonzalo Navarro for fruitful discussions.

References

1. Amir, A., Apostolico, A., Landau, G.M., Satta, G.: Efficient text fingerprinting via Parikh mapping. *J. Discrete Algorithms* 1(5-6), 409–421 (2003)
2. Babai, L., Felzenszwalb, P.F.: Computing rank-convolutions with a mask. *ACM Trans. Algorithms* 6(1), 1–13 (2009)
3. Bellman, R., Karush, W.: Mathematical programming and the maximum transform. *Journal of the Soc. for Industrial and Applied Math.* 10(3), 550–567 (1962)
4. Benson, G.: Composition alignment. In: Benson, G., Page, R.D.M. (eds.) *WABI 2003. LNCS (LNBI)*, vol. 2812, pp. 447–461. Springer, Heidelberg (2003)
5. Böcker, S.: Simulating multiplexed SNP discovery rates using base-specific cleavage and mass spectrometry. *Bioinformatics* 23(2), 5–12 (2007)
6. Böcker, S., Lipták, Z.: A fast and simple algorithm for the Money Changing Problem. *Algorithmica* 48(4), 413–432 (2007)
7. Bremner, D., Chan, T.M., Demaine, E.D., Erickson, J., Hurtado, F., Iacono, J., Langerman, S., Taslakian, P.: Necklaces, convolutions, and $X + Y$. In: Azar, Y., Erlebach, T. (eds.) *ESA 2006. LNCS*, vol. 4168, pp. 160–171. Springer, Heidelberg (2006)
8. Butman, A., Eres, R., Landau, G.M.: Scaled and permuted string matching. *Inf. Process. Lett.* 92(6), 293–297 (2004)
9. Chan, T.M.: All-pairs shortest paths with real weights in $O(n^3/\log n)$ time. *Algorithmica* 50(2), 236–243 (2008)
10. Cicalese, F., Fici, G., Lipták, Z.: Searching for jumbled patterns in strings. In: *Proc. of the Prague Stringology Conference 2009*, pp. 105–117 (2009)
11. Cieliebak, M., Erlebach, T., Lipták, Z., Stoye, J., Welzl, E.: Algorithmic complexity of protein identification: combinatorics of weighted strings. *Discrete Applied Mathematics* 137(1), 27–46 (2004)
12. Clark, D.: Compact pat trees. PhD thesis, University of Waterloo, Canada (1996)
13. Eppstein, D.A.: Efficient algorithms for sequence analysis with concave and convex gap costs. PhD thesis, New York, NY, USA (1989)
14. Eres, R., Landau, G.M., Parida, L.: Permutation pattern discovery in biosequences. *Journal of Computational Biology* 11(6), 1050–1060 (2004)
15. Felzenszwalb, P.F., Huttenlocher, D.P., Kleinberg, J.M.: Fast algorithms for large-state-space HMMs with applications to web usage analysis. In: Thrun, S., Saul, L.K., Schölkopf, B. (eds.) *NIPS*. MIT Press, Cambridge (2003)
16. Goczyla, K.: The generalized Banach match-box problem: Application in disc storage management. *Acta Applicandae Mathematicae* 5, 27–36 (1986)
17. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: *SODA*, pp. 841–850 (2003)
18. Jokinen, P., Tarhio, J., Ukkonen, E.: A comparison of approximate string matching algorithms. *Software Practice and Experience* 26(12), 1439–1458 (1996)
19. Mendelson, H., Pliskin, J., Yechiali, U.: Optimal storage allocation for serial files. *Communications of the ACM* 22, 124–130 (1979)
20. Mendelson, H., Pliskin, J., Yechiali, U.: A stochastic allocation problem. *Operations Research* 28, 687–693 (1980)
21. Munro, J.I.: Tables. In: Chandru, V., Vinay, V. (eds.) *FSTTCS 1996. LNCS*, vol. 1180, pp. 37–42. Springer, Heidelberg (1996)
22. Navarro, G., Mäkinen, V.: Compressed full-text indexes. *ACM Comput. Surv.* 39(1) (2007)
23. Parida, L.: Gapped permutation patterns for comparative genomics. In: Bücher, P., Moret, B.M.E. (eds.) *WABI 2006. LNCS (LNBI)*, vol. 4175, pp. 376–387. Springer, Heidelberg (2006)

Cryptographic and Physical Zero-Knowledge Proof: From Sudoku to Nonogram

Yu-Feng Chien and Wing-Kai Hon

Department of Computer Science
National Tsing Hua University
{cyf,wkhon}@cs.nthu.edu.tw

Abstract. Gradwohl et al. (2007) gave a zero-knowledge proof for Sudoku that can be implemented physically using common tools like envelopes and bags, and the procedures are so simple that they can be executed solely by kids. In this paper, we work along with this direction, and first propose some simple physical zero-knowledge proofs for Nonogram (which was a very popular puzzle game in the 1990s).

1 Introduction

Since Stephen Cook's discovery of the first NP-COMPLETE problem in 1971 [1], many problems are now known to be NP-COMPLETE (See Garey and Johnson [3] for a quick reference). No polynomial-time algorithms are known for any of these problems. In other words, finding a solution to an instance of an NP-COMPLETE problem could be time-consuming.

Suppose some person has solved a particular instance of some NP-COMPLETE problem, and he would like to convince some listener that he has known the solution. The simplest way is to tell the listener directly the solution. However, this solution may be hard to obtain, so the solver would normally prefer keeping the solution a secret to himself. This gives rise to the notion of *zero-knowledge proof*, in which the target is under the condition that *no* extra knowledge about the solution is revealed, the solver can show the listener that (i) there is a solution, and (ii) he knows the solution. In particular, zero-knowledge proof for the NP-COMPLETE problem called 3-COLORABILITY is known [4]. Consequently, there exists a zero-knowledge proof for every NP-COMPLETE problems, via a reduction to the 3-COLORABILITY problem.

Some popular logic puzzles, such as Sudoku [12], Nonogram [10], and Minesweeper [7], are known to be NP-COMPLETE. However, if a solver attempts to give a proof via a reduction to 3-COLORABILITY, it will appear unnatural and unconvincing. For such kinds of puzzles which are accessible to the vast community, it is better to design a more direct and more comprehensible zero-knowledge proof. Gradwohl *et al.* [6] gave a zero-knowledge proof for Sudoku that can be implemented physically using common tools like envelopes and bags, and the procedures are so simple that they can be executed solely by kids. In this paper, we work along with this direction, and first propose a simple physical zero-knowledge proof for Nonogram.

1.1 What Is Nonogram?

Nonogram is a picture logic puzzle played on a grid with $m \times n$ cells, in which cells in the grid have to be colored black or white according to some simple rules. For a particular Nonogram puzzle, each row and each column has a sequence of numbers associated to it; the sequence is used to indicate the number of blocks formed by consecutive black cells, the number of black cells in each block, and the order in which these blocks would appear in the corresponding row or column. Precisely, if the sequence has k numbers, say (x_1, x_2, \dots, x_k) , it means there are k blocks of black cells, the lengths of the blocks are respectively x_1, x_2, \dots , and x_k , and these blocks appear in the same order as in the sequence. (See [11] for an example.)

Let NONOGRAM be the language that contains all Nonogram puzzles with a solution. It is known that NONOGRAM is NP-COMplete [10]. In this paper, we propose some simple physical zero-knowledge proofs for NONOGRAM. Essentially, the framework of our protocols are the same as the one in [6] for Sudoku puzzles. However, due to the difference in nature of the Nonogram and Sudoku puzzles, the actual implementation of the protocols are quite different.

2 Zero-Knowledge Proof

Zero-knowledge proof (ZKP) [5] is a special case of an *interactive proof system*, where the latter consists of a communication protocol (i.e., a series of well-defined steps) executed between a prover P and a verifier V which allows P to convince V that a particular statement is true. For our concern, the statement to be proven will be “the prover P knows a solution of the input Nonogram puzzle”. At the end of the execution, the verifier must output either **accept** or **reject**. The protocol is probabilistic, in a sense that the messages exchanged between the two parties P and V are functions of the input, the messages sent so far, and the private random bits associated to each party. If an interactive proof system is designed properly, then when both the prover and the verifier follow the protocol, it should achieve the following:

1. If the prover knows a solution to the Nonogram puzzle, the verifier will have “high” probability to output **accept**. This probability is called the *completeness* of the protocol.
2. If the prover does not know a solution to the Nonogram puzzle, the verifier will have “low” probability to output **accept**. This probability is called the *soundness* of the protocol.

If an interactive proof system has the following property, it will become a zero-knowledge proof:

Zero-Knowledge Property: The verifier learns nothing new from the prover about the statement; anything that the verifier acquires is beyond his computational ability.

In addition, the protocol should also be *proof-of-knowledge*: if the prover can succeed in making the verifier accept, and further if the prover always uses the same random bits, then we can run the protocol several times and obtain the solution ourselves.

2.1 Cryptographic ZKP versus Physical ZKP

Traditional ZKP may use some cryptographic tools to support the extra functionalities. For instance, Naor [9] shows that we can implement a fairly efficient *commitment protocol* based on any one-way function, so that a committed bit (or message) cannot be read or changed without the other party’s notice. We call such kind of zero-knowledge proofs which make use of the cryptographic tools the *cryptographic ZKPs*.

In contrast, there is a recent study for implementing a protocol when the prover and the verifier are actual human beings, with the help of some “physical tools”. Examples include the use of playing cards [2], sealed envelopes [8], scratch-off cards [6] and more (See [8] for a short survey). In particular, sealed envelopes and scratch-off cards allow us to achieve the same function offered by the above commitment protocol. When ZKPs are implemented based on physical tools, we call them *physical ZKPs*.

3 Framework of Our Protocols

Our physical ZKPs are implemented with two physical tools—scratch-off cards and bags. Scratch-off cards are commonly used in many lottery games, in which some information that determines if we are winning is marked on the cards, but the information is hidden by some protective coating. In our protocols, the scratch-off cards will be specially designed, which are used as a communication means between the prover and the verifier. Each card is *blank* at the beginning, which does not have any protective coating, and inside which some entries are to be filled. Once the entries are filled, the card is *sealed* so that each entry is covered by the protective coating. All sealed cards have the same appearance, so that there is no way to distinguish one from the other. After sealing, we may selectively *open* a specific entry by scratching off the corresponding coating, while keeping the other entries still covered. Bags, on the other hand, will simply be used for random shuffling of the cards. Based on these tools, we propose two physical ZKPs for the Nonogram puzzles. The common framework of our protocols is described as follows:

FRAMEWORK OF OUR PROTOCOLS

Prover Commitment Phase:

- Step 1:** Prover assigns 2 random IDs to each cell of puzzle.
- Step 2:** Prover distributes blank scratch-off cards to the cells.
- Step 3:** Prover fills out the scratch-off cards and seals them.

Verifier Checking Phase:

- Step 4:** Verifier selects a random checking.

Step 5: According to the selected checking, Prover opens some specific entries of each card.

Step 6: Verifier **accepts** if the checking passes; else **rejects**.

3.1 Our Scratch-Off Cards

For a Nonogram puzzle with $m \times n$ cells, all of our protocols will make use of $O(m \times n)$ scratch-off cards, each card has the same set of entries as shown in Figure 1. After distributing blank cards to the cells, the prover will need to fill in the entries in the cards. Before describing the meaning of the entries, we have the following two definitions:

Definition 1. Consider a row (or a column) in the Nonogram puzzle, and let (v_1, v_2, \dots, v_k) be the corresponding sequence, so that in the solution to this puzzle, the row (or the column) contains a block with v_1 black cells, followed by a block with v_2 black cells, and so on, if we scan from left to right (or from top to bottom). The j th block is called an **odd-ranked** block if j is odd; otherwise, it is called an **even-ranked** block.

Definition 2. Consider the solution of the Nonogram puzzle. If a cell is the ℓ th leftmost black cell in a row, we say its **black order** within the row is ℓ . Similarly, if a cell is the ℓ th topmost black cell in a column, we say its **black order** within the column is ℓ .

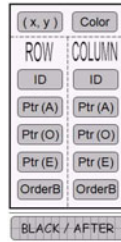


Fig. 1. Scratch-off card in our protocols

Now, we explain the commitment phase of our first protocol in details and show how the entries in the cards are filled (there are only minor variations in the other two protocols). Our intuitive idea is to create a *virtual* coordinate system for the cells, so that adjacency between cells are preserved. As a result, the original coordinates of a cell can be hidden.

Step 1: Prover prepares $2mn$ distinct numbers, and randomly distributes these numbers evenly to the mn cells in the Nonogram puzzle, so that each cell will have exactly two numbers. For each cell, one number is chosen as *row ID* while the other as *column ID*.

Step 2: Prover places one blank scratch-off card in each cell.

Step 3: Prover fills the entries of the card in each cell c as follows:

- (x, y) : The x - and y -coordinates of the cell in the Nonogram puzzle;

- COLOR: The color of the cell c in the solution;
- ROW.ID: The row ID of the cell c ;
- ROW.PTR(A): The row ID of the cell immediately on the right of c . In case c is the *rightmost* cell in the row, ROW.PTR(A) is \times .
- ROW.PTR(O): The entry is filled based on the odd-ranked blocks in the row containing c . There are three cases:
 1. If c is immediately on the left of any cell in an odd-ranked block, ROW.PTR(O) is set to ROW.PTR(A);
 2. If c is immediately on the right of the *last* cell in an odd-ranked block, ROW.PTR(O) is \times ;
 3. If neither of the above, ROW.PTR(O) is left empty.
- ROW.PTR(E): Analogous to ROW.PTR(O), filled based on the even-ranked blocks.
- ROW.ORDERB: The black order of c in the row if the color of c is black; empty otherwise.
- COLUMN.*: Filled as in ROW.* analogously (by replacing the keywords “row, left, right” to “column, top, bottom”, respectively)¹.
- BLACK/AFTER: It contains two 2-dimensional arrays $R[1..2][1..n]$ and $C[1..2][1..m]$. The row IDs of all cells to the right of c will each be filled at a random entry of $R[1][1..n]$. In addition, if c is a black cell, then $R[2][j]$ is ■ if $R[1][j]$ is the row ID of some black cell, and is left empty otherwise. If c is a white cell, all $R[2][1..n]$ are left empty. Note that some entries of $R[1][1..n]$ may not be filled. The entries of C are filled analogously.

2.1.3.1														
ID	2	4	7	8	3	5	1	10	14	9	6	11	12	15
Ptr(A)	4	7	8	x	3	5	1	10	14	9	6	11	12	15
Ptr(O)	4	7	8	x				10	14	9	6	x		
Ptr(E)					3	5	x						12	15

Fig. 2. Filling ROW.PTR(A), ROW.PTR(O), ROW.PTR(E): The first row of the figure indicates the black cells in a row of the solution. The second row indicates the row ID of each cell assigned by Prover. The remaining rows show the three values to be filled in the scratch-off cards in each cell.

ROW	15		11	9	14	6	12
			■	■			■
COLUMN	21	24	22	27	23	28	29
		■	■	■	■		

Fig. 3. Filling BLACK/AFTER in a particular scratch-off card

The row and column IDs give a virtual coordinate to a cell, and the various pointers ensure that the adjacency of the cells in the grid, or the adjacency of the black cells, are preserved. The ORDERB and BLACK/AFTER contains

¹ We use COLUMN.* as a shorthand notation to denote the collection of COLUMN.ID, COLUMN.PTR(A), COLUMN.PTR(O), COLUMN.PTR(E).

information about the order of a black cell among its row or its column. See Figures 2 and 3 for an example.

Based on the definition of the scratch-off cards, we have a necessary and sufficient condition for Prover to know the solution of the Nonogram puzzle, as shown in the two theorems below. Briefly speaking, the condition includes seven statements which are related to the random checking offered to Verifier during the checking phase.

Theorem 1. *If the scratch-off cards can be filled such that all statements below are satisfied:*

- S1:** The (x, y) entry of each card is correct;
- S2:** All IDs are different and all PTR(A), and (x, y) entries, are consistent;
- S3:** The BLACK/AFTER entry /contains the correct set of IDs;
- S4:** Each row/column contains the correct number of black cards;
- S5:** For each row/column, the IDs and ORDERB entries of all cards with black color are consistent with the set of IDs marked by ■ in their BLACK/AFTER entries;²
- S6:** For each row/column, each odd-ranked block has the correct number of black cells with correct ORDERB entries;
- S7:** For each row/column, each even-ranked block has the correct number of black cells with correct ORDERB entries;

then, Prover must know the solution of the Nonogram puzzle.

Theorem 2 (Converse of Theorem 1). *If Prover knows the solution of the Nonogram puzzle, then the scratch-off cards can be filled such that all statements S1 to S7 can be satisfied.*

3.2 Minor Modification to Input Puzzle

To simplify the design of our protocol, we hope that in the solution of the Nonogram puzzle, each block of black cells in a row is *enclosed* by two white cells, one immediately to its left and one immediately to its right. Similarly, we hope that each block of black cells in a column is *enclosed* by two white cells, one immediately to its top and one immediately to its bottom. However, this is not true in general, as the solution may require a block to be attached to the boundary of the grid. To deal with the general case, we will alter the input puzzle P slightly to create a puzzle P' with 4 more rows and 2 more columns in a straightforward manner (Details are deferred to full paper). See Figure 4 for an example.

Theorem 3. *Let P denote the input Nonogram puzzle and P' denote the transformed Nonogram puzzle. Then, a person can obtain a solution of P if and only if he can obtain a solution of P' .*

² For instance, if a black card has COLUMN.ID = 5 and COLUMN.ORDERB = 4, then COLUMN.ID, 5, must appear once in $C[1][1..m]$ entries of exactly 3 black cards, and each of the corresponding $C[2]$ entry is ■.

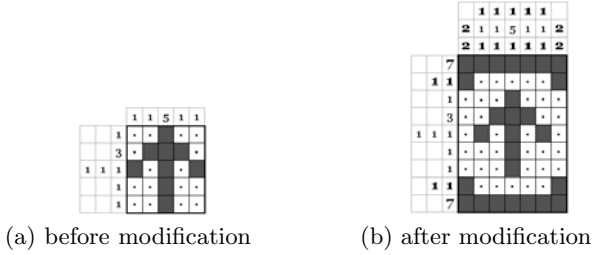


Fig. 4. Modification of Input Puzzle

4 Physical ZKP for Nonogram Puzzles

4.1 First Protocol

After the **Prover Commitment Phase**, our protocol enters the **Verifier Checking Phase**. At Step 4, Verifier begins by choosing one of the *seven* tests, uniformly at random. Then at Step 5, Prover opens specific entries in each card according to Verifier’s choice. After that, at Step 6, Verifier checks if there are inconsistencies using the opened entries, and **accept** if and only if none is found. The details of these seven tests, and their corresponding responses, are shown as follows:

(T1) Testing S1, S2, and S3

Prover’s action: Each card remains in the corresponding cell, while Prover opens the following entries of each card: (x, y) , ID, PTR(A), $R[1][1..n]$ and $C[1][1..m]$ of BLACK/AFTER.

/ $R[2][1..n]$ and $C[2][1..m]$ remain covered */*

Verifier’s action: Examine all cards and check if (i) (x, y) is correct, (ii) IDs are distinct, (iii) PTR(A) is consistent with IDs, and (iv) $R[1][1..n]$ and $C[1][1..m]$ are consistent with IDs. Output **accept** if no inconsistencies found.

(T2) Testing the row parts of S4 and S5

Prover’s action: Prepare m bags, one for each row. Collect all cards of the same row to its corresponding bag. After that, shuffle each bag, and open COLOR entry of each card.

Next, for each black card, open ROW.ID, ROW.ORDERB, and $R[2][1..n]$ of BLACK/AFTER. Finally, open $R[1][j]$ if $R[2][j]$ is ■.

Verifier’s action: Examine all cards in each bag. Check if the number of black cards in each bag (i.e., row) matches with the sum of the corresponding sequence of that row. Next, check if ROW.ID, ROW.ORDERB, and the opened entries of BLACK/AFTER in each bag are consistent. Output **accept** if no inconsistencies found.

(T3) Testing the column parts of S4 and S5 (Analogous to T2)

(T4) Testing the row part of S6

Prover's action: Prepare m bags, one for each row. Collect all cards of the same row to its corresponding bag. After that, shuffle each bag, and open ROW.PTR(O) entry of each card.

Next, for each card whose ROW.PTR(O) is not empty and not marked by \times , open COLOR, ROW.ID, ROW.PTR(A). However, if ROW.PTR(O) is \times , just open COLOR and ROW.ID.

Finally, for those cards with COLOR opened, if COLOR is black, open ROW.ORDERB.

Verifier's action: Examine all cards in each bag. Check if the entries ROW.PTR(O) and ROW.PTR(A) are consistent. Next, arrange the black cards in each row based on ROW.ORDERB, check if each odd-ranked block has the desired number of black cells, and check if each block is enclosed by two white cells at its ends. Output **accept** if no inconsistencies found.

(T5) Testing the column part of S6 (Analogous to T4)

(T6) Testing the row part of S7 (Analogous to T4)

(T7) Testing the column part of S7 (Analogous to T4)

This completes the description of our first protocol, and we have:

Theorem 4. *Our first protocol has perfect completeness, 6/7 soundness, and zero-knowledge property.*

Corollary 1. *By re-running the first protocol $\Theta(\log(1/\varepsilon))$ times, we obtain a protocol that has perfect completeness, ε soundness, and zero-knowledge property, for any $\varepsilon > 0$.*

4.2 Second Protocol: Duplicating the Cards

In the **Prover Commitment Phase** of our second protocol, after assigning the row ID and column ID to each cell of the Nonogram puzzle, Prover will now place *two* blank scratch-off cards in each cell, fill in the entries accordingly, and seal them afterwards. The entries of the two cards in each cell should be filled identically, except Prover has a freedom to fill in BLACK/AFTER in a different manner. Note that because of the use of duplicate cards, Theorems [1](#) and [2](#) will be modified slightly to include to following statement (in addition to the original statements S1 to S7):

S0: The two cards in each cell are filled identically.

After the commitment phase, our protocol enters the **Verifier Checking Phase** as in the first protocol. At Step 4, Verifier begins by choosing one of the *nine* new tests, randomly according to some predetermined probability. Then at Step 5, Prover opens specific entries in each card according to Verifier's choice. After that, at Step 6, Verifier checks if there are inconsistencies using the opened entries, and **accept** if and only if none is found. The details of these nine new tests, and their corresponding responses, are shown as follows:

(T1') Testing S1, S2, and S3, and their related part of S0

Prover's action: Each card remains in the corresponding cell, while Prover opens the following entries of each card: (x, y) , ID, PTR(A), $R[1][1..n]$ and $C[1][1..m]$ of BLACK/AFTER.

/ $R[2][1..n]$ and $C[2][1..m]$ remain covered */*

Verifier's action: Examine all cards and check if (i) (x, y) is correct, (ii) IDs are distinct, (iii) PTR(A) is consistent with IDs, and (iv) $R[1][1..n]$ and $C[1][1..m]$ of each card is consistent with IDs. Examine if the opened entries in the two cards in each cell are identical. Output **accept** if no inconsistencies found.

(T2') Testing COLOR, $R[2][1..n]$ and $C[2][1..m]$ of BLACK/AFTER, ORDERB of S0

Prover's action: Prepare mn small bags, one for each cell. Collect all two cards of the same cell to its corresponding small bag. Put the mn bags into a larger bag, and shuffle the mn bags using the large bag.

Open COLOR, $R[2][1..n]$ and $C[2][1..m]$ of BLACK/AFTER, ORDERB entries in each card. At this point, each small bag contains two cards that are originally from the same cell.

Verifier's action: Examine each bag and check if the opened entries in the two cards are identical. Output **accept** if no inconsistencies found.

(T3') Testing ROW.PTR(O) of S0 (Analogous to T2')

(T4') Testing ROW.PTR(E) of S0 (Analogous to T2')

(T5') Testing COLUMN.PTR(O) of S0 (Analogous to T2')

(T6') Testing COLUMN.PTR(E) of S0 (Analogous to T2')

(T7') Testing S4 and S5

Prover's action: Prepare m bags, one for each row. In each bag, collect *one* card of each cell from the corresponding row. Then, shuffle the cards in each bag.

Next, prepare n bags, one for each column. In each bag, collect *one* card of each cell from the corresponding column. Again, shuffle the cards in each bag.

For each bag in each row, open COLOR entries of each card. Next, for each black card, open ROW.ID, ROW.ORDERB, and $R[2][1..n]$ of BLACK/AFTER. Finally, open $R[1][j]$ if $R[2][j]$ is ■.

Similarly, for each bag in each column, open COLOR entries of each card. Next, for each black card, open COLUMN.ID, COLUMN.ORDERB, and $C[2][1..n]$ of BLACK/AFTER. Finally, open $C[1][j]$ if $C[2][j]$ is ■.

Verifier's action: For each bag in each row, examine all its cards. Check if the number of black cards in each bag matches with the sum of the corresponding sequence. Next, check if ROW.ID, ROW.ORDERB, and the opened entries of BLACK/AFTER in each bag are consistent. Similarly, check for each bag in each column. Output **accept** if no inconsistencies found.

(T8') Testing S6

Prover's action: Prepare m bags, one for each row. In each bag, collect *one* card of each cell in the corresponding row.

After that, shuffle each bag, and open ROW.PTR(O) entry of each card. Next, for each card whose ROW.PTR(O) is not empty and not marked by \times , open COLOR, ROW.ID, ROW.PTR(A). However, if ROW.PTR(O) is \times , just open COLOR and ROW.ID.

Finally, for those cards with COLOR opened, if COLOR is black, open ROW.ORDERB.

Similarly, prepare n bags, one for each column. In each bag, collect *one* card of each cell in the corresponding column. After that, we proceed analogously for the preparation for T5.

Verifier's action: For each bag in each row, examine all its cards. Check if ROW.PTR(O) and ROW.PTR(A) are consistent. Next, arrange the black cards in each row according to ROW.ORDERB order, check if each odd-ranked block has the desired number of black cells, and check if each block is enclosed by two white cells at its ends. Similarly, check for each bag in each column. Output **accept** if no inconsistencies found.

(T9') Testing S7 (Analogous to T8')

This completes the description of our second protocol, and we have:

Theorem 5. *Our second protocol has perfect completeness, 8/9 soundness, and zero-knowledge property.*

Despite using duplicate cards, our second protocol fails to achieve a better (i.e., smaller) soundness probability than the first one. Nevertheless, in our full paper, we will show that with extra resources (such as using carbon paper and having trusted third parties), we can enhance the second protocol to obtain a physical ZKP with 0 soundness.

5 Further Discussions

We have proposed two physical protocols for the Nonogram puzzle which have perfect completeness, constant soundness, and zero-knowledge property. The framework of our protocols is adapted from the one in [6] for Sudoku puzzles. We utilize the advantage of a scratch-off card that we can selectively open its entries in specific order; this may open up the possibilities to simplify some of the existing protocols. However, our use of scratch-off cards, and their preparation, are much more involved than that in [6], making our protocols much harder to be implemented³.

³ The protocol of [6] uses only 9 different kinds of scratch-off cards, so that these cards can be prepared in advance. For our protocol, the scratch-off cards have to be filled on the spot, as there are exponential number of variations. This is one of the major drawbacks of our protocol.

References

1. Cook, S.: The Complexity of Theorem Proving Procedures. In: Proceedings of ACM Symposium on the Theory of Computing (STOC), pp. 151–158 (1971)
2. Fischer, M., Wright, R.: Bounds on Secret Key Exchange Using a Random Deal of Cards. *Journal of Cryptology* 9(2), 71–99 (1996)
3. Garey, M., Johnson, D.: *Computers and Intractability: A Guide to the Theory of NP-Completeness* (1979)
4. Goldreich, O., Micali, S., Wigderson, A.: Proofs that Yield Nothing But their Validity, and a Methodology of Cryptographic Protocol Design. *Journal of the ACM* 38(3), 691–729 (1991)
5. Goldwasser, S., Micali, S., Rackoff, C.: *The Knowledge Complexity of Interactive Proof-Systems* (1985)
6. Gradwohl, R., Naor, M., Pinkas, B., Rothblum, G.: Cryptographic and Physical Zero-Knowledge Proof Systems for Solutions of Sudoku Puzzles. *Theory of Computing Systems* 44(2), 245–268 (2009)
7. Kaye, R.: Minesweeper is NP-complete. *Mathematical Intelligencer* 22(2), 9–15 (2000)
8. Moran, T., Naor, M.: Basing Cryptographic Protocols on Tamper-Evident Seals. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) *ICALP 2005*. LNCS, vol. 3580, pp. 285–297. Springer, Heidelberg (2005)
9. Naor, M.: Bit Commitment Using Pseudo-Randomness. *Journal of Cryptology* 4(2), 151–158 (1991)
10. Ueda, N., Nagao, T.: NP-completeness Results for Nonogram via Parsimonious Reductions. Technical Report TR96-0008, Department of Computer Science, Tokyo Institute of Technology (1996)
11. Wikipedia. Nonogram, <http://en.wikipedia.org/wiki/Nonogram>
12. Yato, T.: Complexity and Completeness of Finding Another Solution and its Application to Puzzles. Master’s thesis, University of Tokyo, Department of Information Science (2003)

A Better Bouncer's Algorithm

Ferdinando Cicalese¹, Travis Gagie^{2,*}, Anthony J. Macula³,
Martin Milanić⁴, and Eberhard Triesch⁵

¹ Department of Computer Science
University of Salerno
cicalese@dia.unisa.it

² Department of Computer Science
University of Chile
travis.gagie@gmail.com

³ Department of Mathematics
State University of New York at Geneseo
macula@geneseo.edu

⁴ Faculty of Mathematics, Natural Sciences and Information Technologies
University of Primorska
martin.milanic@upr.si

⁵ Department of Mathematics
RWTH Aachen University
triesch@math2.rwth-aachen.de

Abstract. Suppose we have a set of materials — e.g., drugs or genes — some combinations of which react badly together. We can experiment to see whether subsets contain any bad combinations and we want to find a maximal subset that does not. This problem is equivalent to finding a maximal independent set (or minimal vertex cover) in a hypergraph using group tests on the vertices. Consider the simple greedy algorithm that adds vertices one by one; after adding each vertex, the algorithm tests whether the subset now contains any edges and, if so, removes that vertex and discards it. We call this the “bouncer’s algorithm” because it is reminiscent of how order is maintained as patrons are admitted to some bars. If this algorithm processes the vertices according to a given total preference order, then its solution is the unique optimum with respect to that order. Our main contribution is another algorithm that produces the same solution but uses fewer tests when few vertices are discarded: if the bouncer’s algorithm discards d of the n vertices in the hypergraph, then our algorithm uses at most $d(\lceil \log_2 n \rceil + 1) + 1$ tests. It follows that, given black-box access to a monotone Boolean formula on n variables, we can find a minimal satisfying truth assignment using at most $d(\lceil \log_2 n \rceil + 1) + 1$ tests, where d is the number of variables set to true. We also prove some bounds for partially adaptive algorithms.

* Funded by the Millennium Institute for Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, Chile.

1 Introduction

Our departmental parties are plagued with arguments about politics, religion and P vs. NP. This seems not to be the fault of a single person but rather to be the effect of putting certain groups of people together in the same room. For example, it might be that any two of three particular professors get along happily but, when all three of them are present, a discussion breaks out. Unfortunately, arguments in our department tend to draw in any bystanders, making it difficult to determine exactly which groups should not be put together. While pondering this problem one evening, we glance out the window at the bar across the street and notice how orderly it seems in comparison. Curious how the bouncer keeps the peace, we wander over and ask him his secret. He replies that he admits patrons one by one — VIPs, friends and regulars first, of course — and, whenever he hears the sounds of raised voices inside, he throws them out again in the reverse order until calm is restored [1].

Back in our office, we start thinking about our next departmental party as a combinatorial problem, rather than a social one, and decide to analyze the bouncer’s algorithm. We model the faculty, staff and students as n vertices v_1, \dots, v_n in a hypergraph with unknown edges, with the edges representing the minimal groups who will argue. (We make the simplifying assumption that, once an arguing group is assembled, adding more people will not prevent the argument; this seems to be the case in many departments.) We can test whether a subset of vertices contains any edges, since this is equivalent to putting the corresponding people together and seeing if they argue. For example, the bouncer’s algorithm starts with an empty set and adds vertices one by one; after adding each vertex, the algorithm tests whether the subset now contains any edges and, if so, removes that vertex and discards it. Our abstraction makes it clear the most obvious goal — maximizing the number of people at the party while avoiding an argument — is intractable even when we know the arguing groups since it is equivalent to finding a maximum independent set in the hypergraph [4,6,7] (i.e., a subset of the vertices that does not completely contain any edge). We notice, however, that the bouncer’s algorithm always finds a *maximal* independent set using exactly n tests. This is the best bound possible in terms of only n : when each vertex is an edge by itself (i.e., when each member of our department will argue even when by himself or herself), we need n tests to rule out the individual vertices and return the empty set. When some vertices are edges by themselves and the others form an independent set, finding the unique maximal independent set is equivalent to finding the defectives in a set using group tests that indicate their presence [5]. Since the complement of a maximal independent set is a minimal vertex cover, we can also use the bouncer’s algorithm to find one of those. Finally, given black-box access to a monotone Boolean formula, we can use the bouncer’s algorithm to find a minimal satisfying truth assignment.

Thinking of the bouncer’s rule “VIPs, friends and regulars first”, we decide to impose a total preference order $v_1 \succ \dots \succ v_n$ on the vertices (i.e., we prefer each vertex to its successor). In this setting, one subset S_1 of vertices

dominates another S_2 if $S_1 \cap \{v_1, \dots, v_i\} \supset S_2 \cap \{v_1, \dots, v_i\}$ for some i . We realize that, if the bouncer’s algorithm processes the vertices in order of preference, then it finds the independent set that dominates all the others, i.e., the unique optimum with respect to that order. Naturally, we are impressed. Nevertheless, worried the people in our department might not be as patient as the average bar patron and optimistic that only a few of them argue while alone, we start thinking about whether we can find the optimum using fewer tests when few vertices are discarded. The algorithm we present in this paper produces the same solution as the bouncer’s algorithm but uses at most $d(\lceil \log_2 n \rceil + 1) + 1$ tests, where d is the number of discarded vertices. As it takes nearly $d \log_2 n$ bits in the worst case to specify the vertices discarded by the bouncer’s algorithm, our algorithm is within lower order terms of optimality. Once we recover from the stresses of our next departmental party, we hope to apply our algorithm to problems in bioinformatics where it is necessary, e.g., to combine drugs or genes while avoiding unwanted interactions. For more background, we refer readers to Damaschke and Muhammad’s recent paper [2] on using group tests to find small vertex covers in graphs, which is similar to the special case of our problem in which interactions depend only on the presence of pairs.

2 Algorithm

We start with $S = \emptyset$ and $p = 0$ and maintain the invariant that S contains the optimum independent set that is a subset of $\{v_1, \dots, v_p\}$. At each step of our algorithm, we check whether $S \cup \{v_{p+1}, \dots, v_n\}$ contains an edge and, if so, we use binary search in v_{p+1}, \dots, v_n to find the smallest value i such that $S \cup \{v_{p+1}, \dots, v_{i-1}\}$ does not contain an edge but $S \cup \{v_{p+1}, \dots, v_i\}$ does; if not, we set $i = n + 1$. Once we have found i , we add v_{p+1}, \dots, v_{i-1} to S and set $p = i$. Notice we discard one vertex at each step (except possibly the last); therefore, if we discard d vertices, we use a total of at most $d(\lceil \log_2 n \rceil + 1) + 1$ tests. If we are given d in advance, we need only $d \lceil \log_2 n \rceil$ tests: we need not test $S \cup \{v_{p+1}, \dots, v_n\}$ at the beginning of each step, because it will contain an edge until we discard d vertices.

For example, suppose the hypergraph’s vertex set is $\{v_1, \dots, v_8\}$, its edge set is $\{\{v_1, v_4, v_8\}, \{v_2, v_5\}, \{v_3, v_6, v_8\}\}$ and the preference order is $v_1 \succ \dots \succ v_8$. The bouncer’s algorithm performs the eight tests

1) $\{v_1\}$	✓	5) $\{v_1, \dots, v_5\}$	×
2) $\{v_1, v_2\}$	✓	6) $\{v_1, \dots, v_4, v_6\}$	✓
3) $\{v_1, \dots, v_3\}$	✓	7) $\{v_1, \dots, v_4, v_6, v_7\}$	✓
4) $\{v_1, \dots, v_4\}$	✓	8) $\{v_1, \dots, v_4, v_6, \dots, v_8\}$	×

and returns $\{v_1, \dots, v_4, v_6, v_7\}$; the checkmarks and crosses above indicate whether or not the induced subgraph on those vertices is an independent set, respectively.

Our algorithm, on the other hand, performs the six tests

- 1) $\{v_1, \dots, v_8\}$ ×
- 2) $\{v_1, \dots, v_4\}$ ✓
- 3) $\{v_1, \dots, v_6\}$ ×
- 4) $\{v_1, \dots, v_5\}$ ×
- 5) $\{v_1, \dots, v_4, v_6, \dots, v_8\}$ ×
- 6) $\{v_1, \dots, v_4, v_6, v_7\}$ ✓

and still returns $\{v_1, \dots, v_4, v_6, v_7\}$. The first four tests are for the first step, in which we discard v_5 , and the last two are for the second, in which we discard v_8 .

A standard induction on the number of vertices processed shows the bouncer’s algorithm selects the unique optimum with respect to the preference order, and a standard induction on the number of steps shows our algorithm produces the same solution. Naturally, we leave the details for the full version of this paper.

Theorem 1. *Consider a preference order on the n vertices of a hypergraph with unknown edges, and let d be the number of vertices not present in the independent set optimum with respect to that order. If we are allowed to perform group tests on subsets of the vertices to determine whether they contain edges then, without knowing d , we can find the optimum independent set (which is necessarily maximal) using at most $d(\lceil \log_2 n \rceil + 1) + 1$ tests.*

Notice a truth assignment satisfies the Boolean formula $(v_1 \vee v_4 \vee v_8) \wedge (v_2 \vee v_5) \wedge (v_3 \vee v_6 \vee v_8)$ if and only if it corresponds to a vertex cover of the hypergraph in our example. Having black-box access to the formula (i.e., being able to test whether truth assignments are satisfying) is equivalent to being able to test whether subsets of vertices cover all the edges; since the complement of a vertex cover is an independent set, both are equivalent to being able to test whether subsets of vertices contain any edges. It follows that we can find a minimal satisfying truth assignment using the complements of the six tests listed above:

- 1) $\overline{v_1}, \dots, \overline{v_8}$ ×
- 2) $\overline{v_1}, \dots, \overline{v_4}, v_5, \dots, v_8$ ✓
- 3) $\overline{v_1}, \dots, \overline{v_6}, v_7, v_8$ ×
- 4) $\overline{v_1}, \dots, \overline{v_5}, v_6, \dots, v_8$ ×
- 5) $\overline{v_1}, \dots, \overline{v_4}, v_5, \overline{v_6}, \dots, \overline{v_8}$ ×
- 6) $\overline{v_1}, \dots, \overline{v_4}, v_5, \overline{v_6}, \overline{v_7}, v_8$ ✓

More generally, for any monotone Boolean formula on n variables, there is a hypergraph on n vertices such that satisfying truth assignments correspond to vertex covers. (It is easy to build the hypergraph once we put the formula into conjunctive normal form; we are not concerned with the explosion of clauses that could result, as they will be contained within the black box.) Therefore, our theorem has the following corollary:

Corollary 1. *Given black-box access to a monotone Boolean formula on n variables, we can find a minimal satisfying truth assignment using at most $d(\lceil \log_2 n \rceil + 1) + 1$ tests, where d is the number of variables set to true.*

If we drop the assumption of monotonicity, of course, the problem becomes intractable, even when we know the formula and do not care whether the satisfying truth assignment is minimal [6].

3 (Non)adaptivity

Naturally, we would like to develop a nonadaptive version of our algorithm, i.e., one that uses $\mathcal{O}(d \log n)$ tests batched into as few stages as possible. It may not be feasible to batch tests if that involves, say, somehow putting a professor in several rooms at the same time — perish the thought! — but it is feasible and very useful in many bioinformatics applications [3]. When every edge contains exactly two vertices — i.e., when the hypergraph is a graph — there is a single-stage algorithm that finds all minimal vertex covers of at most a given size k using $\mathcal{O}(k^3 \log n)$ tests [2]; however, it depends on the fixed-parameter tractability of finding a minimum vertex cover in a graph and, thus, cannot be applied efficiently to hypergraphs in general [4].

So far, we have had only a little success proving upper bounds. If we use q -ary search in our algorithm for some $q > 2$, instead of binary search, then we use at most $d(\lceil \log_q n \rceil + 1) + 1$ stages but at most $(q - 1)d(\lceil \log_q n \rceil + 1) + 1$ tests. For example, if we set $q = n^\epsilon$ for some positive constant $\epsilon \leq 1$, then we use $\mathcal{O}(d)$ stages and $\mathcal{O}(dn^\epsilon)$ tests. The most interesting case may be when, in each stage, we perform enough tests to be sure of finding a vertex to discard, if there is one left. In that case, we use at most $d + 1$ stages and at most $(d + 1)n$ tests. If the last vertex is discarded, then we use only d stages; if we somehow know d in advance, then we use only d stages and need never test subsets including v_{n-d+1}, \dots, v_n .

In the first stage, we test each subset of the form $\{v_1, \dots, v_i\}$ for $1 \leq i \leq n$; if $\{v_1, \dots, v_{r_1}\}$ is the first such subset to contain an edge, then v_{r_1} is the first vertex the bouncer’s algorithm discards. For $j > 1$, in the j th stage we test each subset of the form $\{v_1, \dots, v_i\} - \{v_{r_1}, \dots, v_{r_{j-1}}\}$ for $r_{j-1} < i \leq n$, where $v_{r_1}, \dots, v_{r_{j-1}}$ are the vertices we already know are discarded by the bouncer’s algorithm; if $\{v_1, \dots, v_{r_j}\} - \{v_{r_1}, \dots, v_{r_{j-1}}\}$ is the first such subset to contain an edge, then v_{r_j} is the j th vertex the bouncer’s algorithm discards. We stop when none of the subsets we test includes an edge. On our running example, this algorithm performs the eleven tests

1) $\{v_1\}$	✓	7) $\{v_1, \dots, v_7\}$	×
2) $\{v_1, v_2\}$	✓	8) $\{v_1, \dots, v_8\}$	×
3) $\{v_1, \dots, v_3\}$	✓	9) $\{v_1, \dots, v_4, v_6\}$	✓
4) $\{v_1, \dots, v_4\}$	✓	10) $\{v_1, \dots, v_4, v_6, v_7\}$	✓
5) $\{v_1, \dots, v_5\}$	×	11) $\{v_1, \dots, v_4, v_6, \dots, v_8\}$	×
6) $\{v_1, \dots, v_6\}$	×		

and returns $\{v_1, \dots, v_4, v_6, v_7\}$. The first eight tests are performed in the first stage and the last three in the second stage; since we discard the last vertex, v_8 , we do not use a third stage.

Let $r_0 = 0$ and $r_{d+1} = n$. Then for $0 \leq j \leq d$ and $r_j < i \leq r_{j+1}$ and $0 \leq k \leq j$, we test each subset $\{v_1, \dots, v_i\} - \{v_{r_1}, \dots, v_{r_k}\}$ once. It follows that we use $\sum_{j=0}^d (j+1)(r_{j+1} - r_j) \leq (d+1)n$ tests. Calculation shows the left-hand side of this inequality is $n - d + d(d+1)/2 + b$, where b is the number of times we need to swap neighbouring bits to sort the n -bit binary string with ones in positions r_1, \dots, r_d . In our example, since we discard two of eight vertices and need to swap neighbouring bits twice in order to sort the binary string 00001001, we use eleven tests. Interestingly, if we can choose the preference order and try to guess an arrangement in which discarded vertices appear toward the end, then the better our guess, the fewer tests we use.

Suppose T_0 is the set of sets we perform in some stage and, if we then discard v_i , then T_i is the set of tests we perform in the next stage. Then we can combine the stages by performing $\bigcup_{0 \leq i \leq n} T_i$ instead of T_0 . It follows that, for any constant $c \geq 1$, we can use $\mathcal{O}(d/c + 1)$ stages and $\mathcal{O}((d/c + 1)n^c)$ tests. (For non-integer c , we combine the stages of an algorithm that uses $\mathcal{O}(d)$ stages and $\mathcal{O}(dn^\epsilon)$ tests, choosing ϵ such that c/ϵ is an integer.) Notice this tradeoff can also be viewed as a kind of q -ary search, with $q = n^c$. If we collapse all the possible stages into one, then we end up using exhaustive search: $2^n - 1$ tests if we do not know d in advance, and $\binom{n}{d}$ if we do.

Theorem 2. *Without knowing d in advance, we can find the optimum independent set using*

- $d + 1$ stages and at most $(d + 1)n$ tests;
- for any positive constant $\epsilon \leq 1$, $\mathcal{O}(d)$ stages and $\mathcal{O}(dn^\epsilon)$ tests;
- for any constant $c \geq 1$, $\mathcal{O}(d/c + 1)$ stages and $\mathcal{O}((d/c + 1)n^c)$ tests.

If we know d , we need only d stages and at most $d(n - d)$ tests.

Sadly, we have had more luck with lower bounds. For example, if we use a single stage without knowing d in advance, then we must test all $2^n - 1$ non-empty subsets in order to be certain of finding a maximal independent set. To see why, suppose we do not test some subset S and that, of those we do test, we find that only subsets of S are independent. Furthermore, suppose every vertex in $\{v_1, \dots, v_n\} - S$ is an edge by itself. Then we cannot tell whether S is the unique maximal independent set or whether it is an edge.

If we use a single stage but somehow know d in advance, then we must still test all but one of the subsets of size $n - d$. To see why, suppose we do not test two subsets S_1 and S_2 of size $n - d$ and that, of those we do test, we find that only the subsets of S_1 and S_2 are independent. Furthermore, suppose every vertex in $\{v_1, \dots, v_n\} - S_1 - S_2$ is an edge by itself, one of S_1 and S_2 is an independent set and the other is an edge. We cannot tell which of S_1 and S_2 is independent and, so, cannot return either; for the sake of guaranteeing maximality, we also cannot return a strict subset of either. It follows that we must test at least $\binom{n}{d} - 1$ subsets, a bound which is achievable via exhaustive search but exponential in $d \log n$ when $d \ll n$. Notice that, if we know $d = 1$, then the algorithm we described earlier in this section uses one stage and $n - 1$ tests, matching our lower bound.

If we use a single stage but know only an upper bound d' on d , then we must test all but one of the subsets of size $n - d'$ (for the same reason we must test them when we know d exactly) and also all the subsets of size between $n - d' + 1$ and n . To see why we need to test the latter subsets, suppose we do not test a subset S of size between $n - d' + 1$ and n and that, of those we do test, we find that only the subsets of S are independent. Furthermore, suppose every vertex in $\{v_1, \dots, v_n\} - S$ is an edge by itself. Then we cannot tell whether S is independent and $d = n - |S|$ or whether S is an edge and $d = n - |S| + 1$. It follows that we must test at least $\sum_{i=0}^{d'} \binom{n}{i} - 1$ subsets, a bound which is again achievable by exhaustive search but exponential in $d \log n$ when $d \ll n$. Notice that, if we know $d \leq 1$, then algorithm we described earlier in this section uses one stage and n tests, again matching our lower bound.

Lemma 1. *If we use a single stage without knowing d in advance, then we need $2^n - 1$ tests. If we know d in advance, then we still need $\binom{n}{d} - 1$ tests. If we know only an upper bound d' on d , then we need $\sum_{i=0}^{d'} \binom{n}{i} - 1$ tests.*

Now suppose we use two stages without knowing d in advance, and test t subsets in the first stage, each of which contains an edge. We claim that, after the first stage, there remains a subset of size at least $n - t$ about which we know nothing. To see why, let v'_1, \dots, v'_t be a list of vertices, one from each tested subset. Every superset of a tested subset is also a superset of at least one of $\{v'_1\}, \dots, \{v'_t\}$ and, therefore, not $\{v_1, \dots, v_n\} - \{v'_1, \dots, v'_t\}$. Since all the subsets we test contain edges and we do not know d , we learn nothing about $\{v_1, \dots, v_n\} - \{v'_1, \dots, v'_t\}$. Therefore, by Lemma II, we still need at least $2^{n-t} - 1$ tests to finish in the second stage.

Finally, suppose we want to be sure of using $\mathcal{O}(d \log n)$ tests in total but without knowing d in advance. As we noted in Section II, when some vertices are edges by themselves and the others form an independent set, finding the unique maximal independent set is equivalent to finding the defectives in a set using group tests that indicate their presence. Therefore, since we need $\Omega(\log d / \log \log d)$ stages to find d defectives with $\mathcal{O}(d \log n)$ tests [2], the same lower bound holds for finding a maximal independent set. Suppose we want to be sure of using at most $c_1 d \log n + c_2$ tests in total, for some constants c_1 and c_2 , and let $t_i = c_1 \left(\sum_{j=1}^{i-1} t_j \right) \log n + c_2$. If we do not know d in advance then, for any i , we can use at most t_i tests in the i th stage, since we cannot be certain at that point whether d is greater than the number of tests we have already performed. It follows that, after $o(\log n / \log \log n)$ stages, there remains a subset of size at least $n - n^{o(1)}$ such that none of its subsets have been tested; therefore, by Lemma II, in the worst case we still need at least $2^{n-n^{o(1)}}$ tests to finish in the next stage — even though $d \log n$ could be in $n^{o(1)}$.

Lemma 2. *If we use $o(\log n / \log \log n)$ stages without knowing d in advance, then we need $2^{n-n^{o(1)}}$ tests.*

Combining Lemmas 1 and 2 — and rephrasing them slightly to highlight, rather optimistically, their similarity to lower bounds for group testing with defectives — we have the following theorem:

Theorem 3. *If we use a single stage then, whether we know d in advance or not, we need $\omega(d \log n)$ tests. If we use $o(\log n / \log \log n)$ stages but without knowing d in advance, then we still need $\omega(d \log n)$ tests.*

Naturally, we leave as future work proving tighter bounds for the case in which we use multiple stages while knowing d in advance, as that turned out to be the most interesting case for group testing with defectives [3] but this section is already somewhat longer than we anticipated.

Acknowledgments

Many thanks to the other organizers and participants of Dagstuhl Seminar 09281, “Search Methodologies”.

References

1. Bouncer, A.: Personal Communication (2010)
2. Damaschke, P., Muhammad, A.S.: Competitive group testing and learning hidden vertex covers with minimum adaptivity. In: Gębala, M. (ed.) FCT 2009. LNCS, vol. 5699, pp. 84–95. Springer, Heidelberg (2009)
3. De Bonis, A., Gasieniec, L., Vaccaro, U.: Optimal two-stage algorithms for group testing problems. *SIAM Journal on Computing* 34(5), 1253–1270 (2005)
4. Downey, R.G., Fellows, M.R.: *Parameterized Complexity*. Springer, Heidelberg (1999)
5. Du, D.-Z., Hwang, F.K.: *Combinatorial Group Testing and Its Applications*, 2nd edn. World Scientific, Singapore (2000)
6. Garey, M.R., Johnson, D.S.: *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York (1979)
7. Håstad, J.: Clique is hard to approximate within $n^{1-\epsilon}$. *Acta Mathematica* 182(1), 105–142 (1999)

Tradeoffs in Process Strategy Games with Application in the WDM Reconfiguration Problem^{*}

Nathann Cohen¹, David Coudert¹, Dorian Mazauric¹,
Napoleão Nepomuceno^{1,2}, and Nicolas Nisse¹

¹ MASCOTTE, INRIA, I3S, CNRS, UNS, Sophia Antipolis, France

² Universidade Federal do Ceará, Fortaleza-CE, Brazil

Abstract. We consider a variant of the graph searching games that is closely related to the routing reconfiguration problem in WDM networks. In the digraph processing game, a team of agents is aiming at clearing, or *processing*, the vertices of a digraph D . In this game, two important measures arise: 1) the total number of agents used, and 2) the total number of vertices occupied by an agent during the processing of D . Previous works have studied the problem of minimizing each of these parameters independently. In particular, both of these optimization problems are not in APX. In this paper, we study the tradeoff between both these conflicting objectives. More precisely, we prove that there exist some instances for which minimizing one of these objectives arbitrarily impairs the quality of the solution for the other one. We show that such bad tradeoffs may happen even in the case of basic network topologies. On the other hand, we exhibit classes of instances where good tradeoffs can be achieved. We also show that minimizing one of these parameters while the other is constrained is not in APX.

Keywords: Graph searching, process number, routing reconfiguration.

1 Introduction

In this paper, we study the *digraph processing* game analogous to graph searching games [9]. This game aims at *clearing* the vertices of a contaminated directed graph D . For this, we use mobile agents that are sequentially put to and removed from the vertices of D . We are interested in two different measures and their tradeoffs: the minimum number of vertices that must be *covered* (i.e., visited by an agent) and the minimum number of agents required to *clear* D . This game is closely related to the routing reconfiguration problem in Wavelength Division Multiplexing (WDM) networks. In this context, the goal is to reroute some connections that are established between pairs of nodes in a communication network, which unfortunately can lead to interruptions of service. Each instance of this problem may be represented by a directed graph called its *dependency*

^{*} This work was partially funded by région PACA, ANRs AGAPE, and DIMAGREEN.

digraph D such that the reconfiguration problem is equivalent to the clearing of D . More precisely, the two measures presented above respectively correspond to the total number of requests disrupted during the rerouting of the connections, and to the number of simultaneous disruptions during the whole process. The equivalence between these two problems is detailed in Section 5.

The digraph processing game has been introduced in 5 for its relationship with the routing reconfiguration problem. This game is defined by the three following operations (or rules), which are very similar to the ones defining the *node search number* [1, 9, 12, 14] of a graph and whose goal is to clear, or to *process*, all the vertices of a digraph D :

- R_1 Put an agent at a vertex v of D ;
- R_2 Remove an agent from a vertex v of D if all its outneighbors are either processed or occupied by an agent, and process v ;
- R_3 Process an unoccupied vertex v of D if all its outneighbors are either processed or occupied by an agent.

A graph whose vertices have all been processed is said *processed*. A sequence of such operations resulting in processing all vertices of D is called a *process strategy*. Note that, during a process strategy, an agent that has been removed from a vertex can be reused. The number of agents used by a strategy on a digraph D is the maximum number of agents present at the vertices of D during the process strategy. A vertex is *covered* during a strategy if it is occupied by an agent at some step of the process strategy.

Fig. 1 illustrates two process strategies for a symmetric digraph D of 7 vertices. The strategy depicted in Fig. 1(a) first put an agent at vertex x_1 (R_1), which enables to process y_1 (R_3). A second agent is then put at r allowing the vertex x_1 to be processed, and the agent on it to be removed (R_2). The procedure goes on iteratively, until all the vertices are processed after 11 steps. The depicted strategy uses 2 agents and covers 4 vertices. Another process strategy is depicted, Fig. 1(b), uses 3 agents and covers 3 vertices. Note that the latter strategy consists in placing agents at the vertices of a feedback vertex set (FVS)¹ of minimum size.

Clearly, any digraph can be processed by placing simultaneously an agent at every node. However, Rule R_3 allows to process some vertices without placing an agent on it. More precisely, to process a digraph D , it is sufficient to put an agent at every vertex of a feedback vertex set F of D , then the vertices of $V(D) \setminus F$ can be processed using Rule R_3 , and finally all agents can be removed. In particular, a Directed Acyclic Graph (DAG) can be processed using 0 agents and thus covering no vertices. Indeed, to process a DAG, it is sufficient to process sequentially its vertices starting from the leaves. Remark that any process strategy for a digraph D must cover all vertices of a feedback vertex set of D (not necessarily simultaneously). In general, the minimum number of agents required to process a digraph D (without constraint on the number of covered

¹ $F \subseteq V$ of a digraph $D = (V, A)$ is a FVS if removing all vertices in F makes D acyclic.

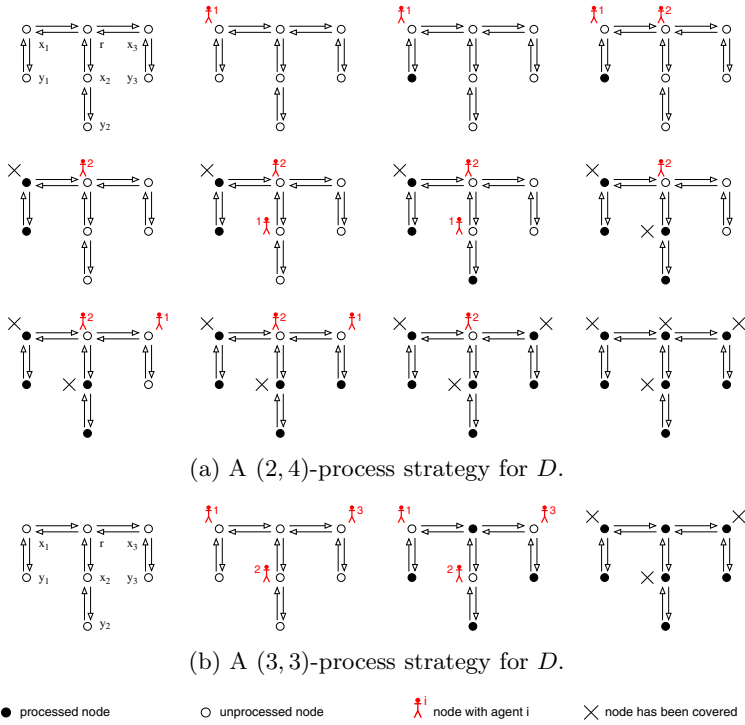


Fig. 1. Different process strategies for a symmetric digraph D

vertices) is called the *process number* [5, 6, 4], while the minimum number of covered vertices required to process D (without constraint on the number of agents) equals the size of a *minimum feedback vertex set* of D .

We are interested in tradeoffs between the minimum number of agents used by a strategy and the minimum number of vertices covered during it.

1.1 Definitions and Previous Results

Let D be a n -node digraph. A (p, q) -*process strategy* denotes a process strategy for D using at most p agents and covering at most q vertices. When the number of covered vertices is not constrained, we write *p -process strategy* instead of (p, n) -process strategy. Similarly, when the number of agents is not constrained, *q -process strategy* replaces (n, q) -process strategy.

Process Number. The problem of finding the *process number* of a digraph D was introduced in [5] as a metric of the routing reconfiguration problem (see Section 5). Formally :

Definition 1. Let $pn(D)$ denote the smallest p such that there exists a p -process strategy for D .

For instance, the digraph Fig. 1 satisfies $pn(D) = 2$. Indeed, Fig. 1(a) describes a process strategy using 2 agents, and it is easy to check that no strategy can process D using at most 1 agent. While digraphs with process number 0, 1, and 2 can be recognized in polynomial time [6], computing the process number is NP-complete and not in APX (i.e., admitting no polynomial-time approximation algorithm up to a constant factor, unless $P = NP$) [5]. A distributed polynomial-time algorithm to compute the process number of trees (or forests) with symmetric arcs has been proposed in [3]. Furthermore the first heuristic for computing the process number of any digraph is described in [4]. In [16], Solano conjectured that computing the process number of a digraph can be solved in polynomial time if the set of covered vertices is given as part of the input. We disprove this conjecture, showing that computing the process number of a digraph remains out of APX (and so is NP-complete) even when the subset of vertices at which an agent will be put is given (see Theorem 1).

The *node search number* and the *pathwidth* are graph invariants closely related to the notion of process number for undirected graph. The node search number of a graph G , denoted by $sn(G)$, is the smallest p such that rules R_1 and R_2 (R_3 is omitted) are sufficient to process G using at most p agents. See [1, 9, 12, 14] for more details. The notion of pathwidth was introduced by Robertson and Seymour in [15]. It has been proved in [8] by Ellis *et al.* that the pathwidth and the node search number are equivalent, that is for any graph G , $pw(G) = sn(G) - 1$, and in [5] that $pw(G) \leq pn(G) \leq pw(G) + 1$ (and so $sn(G) - 1 \leq pn(G) \leq sn(G)$), where the graph G is considered as a symmetric digraph. Since the problem of determining the pathwidth of a graph is NP-complete [13] and not in APX [7], these two parameters behave similarly.

Minimum Feedback Vertex Set. Given a digraph D , the problem of finding a process strategy that minimizes the number of nodes covered by agents is similar to the one of computing the size of a *minimum feedback vertex set* (MFVS) of D . Computing such a set is well known to be NP-complete and not in APX [10]. We define below the parameter $mfvs(D)$, using the notion of (p, q) -process strategy, corresponding to the size of a MFVS of D .

Definition 2. Let $mfvs(D)$ denote the smallest q such that there exists a q -process strategy for D .

As an example, for the digraph of Fig. 1, $mfvs(D) = 3$. As mentioned above, $mfvs(D) \geq pn(D)$. Moreover, the gap between these two parameters may be arbitrarily large. For example a symmetric path P_n of $n \geq 4$ nodes is such that: $mfvs(P_n) = \lfloor \frac{n}{2} \rfloor$ while $pn(P_n) = 2$.

Tradeoff Metrics. We introduce new tradeoff metrics in order to study the loss one may expect on one parameter when adding a constraint on the other. In particular, what is the minimum number of vertices that must be covered by a process strategy for D using $pn(D)$ agents? Similarly, what is the minimum number of agents that must be used to process D covering $mfvs(D)$ vertices?

Definition 3. Given an integer $q \geq mfv_s(D)$, we denote by $pn_q(D)$ the minimum p such that a (p, q) -process strategy for D exists. We define $pn_{mfv_s}(D) = pn_q(D)$ when $q = mfv_s(D)$.

Definition 4. Given an integer $p \geq pn(D)$, we denote by $mfv_{s_p}(D)$ the minimum q such that a (p, q) -process strategy for D exists. We define $mfv_{s_{pn}}(D) = mfv_{s_p}(D)$ when $p = pn(D)$.

Note that $mfv_{s_{pn}}(D)$ is precisely the minimum number of vertices that must be covered by a process strategy using the minimum number of agents, and that $pn_{mfv_s}(D)$ is the minimum number of agents required by a process strategy minimizing the number of covered vertices.

To illustrate the pertinence of these tradeoff metrics, consider the digraph D of Fig. 1. Recall that $pn(D) = 2$ and $mfv_s(D) = 3$. We can easily verify that there does not exist a $(2, 3)$ -process strategy for D , that is a process strategy minimizing both p and q . On the other hand, we can exhibit a $(2, 4)$ -process strategy (Fig. 1(a)) and a $(3, 3)$ -process strategy (Fig. 1(b)) for D . Hence, we have: $pn_{mfv_s}(D) = 3$ while $pn(D) = 2$, and $mfv_{s_{pn}}(D) = 4$ while $mfv_s(D) = 3$. Intuitively for these two process strategies, we can not decrease one value without increasing the other.

We generalize this concept through the introduction of the notion of minimal values for a digraph D . We say that (p, q) is a minimal value for a digraph D if $p = pn_q(D)$ and $q = mfv_{s_p}(D)$. Remark that $(pn(D), mfv_{s_{pn}}(D))$ and $(pn_{mfv_s}(D), mfv_s(D))$ are both minimal values by definition (and may be the same). Clearly for a given digraph D , the number of minimal values is linear in the number of nodes $n = |V(D)|$. For the digraph of Fig. 1, there are two minimal values: $(2, 4)$ and $(3, 3)$. Fig. 2 represents the shape of minimal values for a digraph D . More precisely, Fig. 2 depicts the variations of the minimum number q of vertices covered by a p -strategy for D ($p \geq pn(D)$, i.e., $mfv_{s_p}(D)$ as a function of p . Clearly, it is a non-increasing function greater than by $mfv_s(D)$.

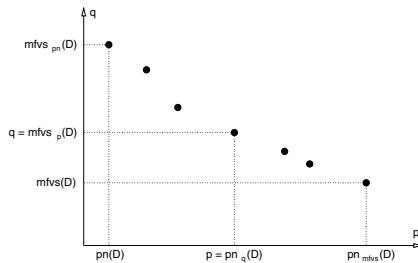


Fig. 2. Representation of minimal values

1.2 Our Results

Our results constitute an analysis of the behaviour of the two given measures both in general digraphs and in symmetric digraphs. In general, as mentioned

above, no process strategy minimizes both the number of agents and the number of covered vertices (see, e.g., Fig. 1). Hence, we are interested in the loss on one measure when the other is constrained. In particular, we are interested in the ratios $\frac{pn_{mfv_s}(D)}{pn(D)}$, and $\frac{mfv_{s_{pn}}(D)}{mfv_s(D)}$. This study involves various theorems on the complexity of estimating this loss (Sec. 2) and the existence of digraphs for which it can be arbitrarily large (Sec. 3 and Sec. 4). More precisely, we first disprove the conjecture of Solano [16] (Th. 1). Then, we prove that all parameters $pn_{mfv_s}(D)$, $mfv_{s_{pn}}(D)$, $\frac{pn_{mfv_s}(D)}{pn(D)}$, and $\frac{mfv_{s_{pn}}(D)}{mfv_s(D)}$ are not in APX (Th. 2). Then, we prove that $\frac{pn_{mfv_s}(D)}{pn(D)}$ and $\frac{mfv_{s_{pn}}(D)}{mfv_s(D)}$ are not bounded in general digraphs even in the class of bounded process number digraphs (Th. 3 and 4). However, $\frac{mfv_{s_{pn}}(D)}{mfv_s(D)} \leq pn(D)$ for any symmetric digraph D (Lemma 1). Due to lack of space most of the proofs are sketched and can be found in [2].

2 Complexity Results

Before proving that computing the tradeoff parameters introduced in Section 1.1 are NP-complete and not in APX, we disprove a conjecture of Solano about the complexity of computing the process number of a digraph D .

Indeed a possible approach for computing the process number, proposed in [16], consists of two phases: 1) finding the subset of vertices of the digraph at which an agent will be put, and 2) deciding the order in which the agents are put at these vertices. Solano conjectures that the complexity of the process number problem resides in Phase 1 and that Phase 2 can be solved or approximated in polynomial time [16]. We disprove this conjecture.

Theorem 1. *Computing the process number of a digraph D is not in APX (and thus NP-complete), even when the subset of covered vertices is given.*

Sketch of the Proof. Let $D = (V, A)$ be a symmetric digraph with $V = \{u_1, \dots, u_n\}$. Let $D' = (V', A')$ be the symmetric digraph with $V' = V \cup \{v_1, \dots, v_n\}$ obtained from D by adding 2 symmetric arcs between u_i and v_i ($i = 1, \dots, n$). It is easy to show that there exists an optimal process strategy for D' such that the set of occupied vertices is V . Now, consider the problem of computing an optimal process strategy for D' when the set of vertices covered by agents is constrained to be V . It is easy to check that this problem is equivalent to the one of computing the node search number (and so the pathwidth) of the underlying undirected graph of D which is NP-complete [13] and not in APX [7]. ■

Theorem 2. *Given a digraph D , the problems of determining $pn_{mfv_s}(D)$, $mfv_{s_{pn}}(D)$, $\frac{pn_{mfv_s}(D)}{pn(D)}$, and $\frac{mfv_{s_{pn}}(D)}{mfv_s(D)}$ are not in APX (and thus NP-complete).*

Proof. From Theorem 1, we know that the problem of determining pn_{mfv_s} is not in APX. Indeed, in the class of graphs D' defined in the proof of Theorem 1, $pn(D') = pn_{mfv_s}(D') = pw(D) + 1 = sn(D)$ (where the relationship between D and D' is described in this proof).

Let H_n be a symmetric directed star with n branches each of which contains two vertices, excluding the central node r (e.g., Fig. 1 for $n = 3$). Let K_n be a symmetric n -node clique digraph, and D be any n -node digraph.

Let D' be the disjoint union of K_n and D . Clearly, $pn(D') = pn(K_n) = n - 1$. Thus $mfvs_{pn}(D') = n - 1 + mfvs(D)$ since when we process D we can use $n - 1$ agents. Since computing $mfvs(D)$ is not in APX, computing $mfvs_{pn}$ is not in APX. To show that $\frac{pn_{mfvs}}{pn}$ is not in APX, let D' be the digraph composed of two components H_n and D . Let us do some trivial remarks: (1) the neighbors of r belong to any MFVS of D' . (2) Moreover, r does not belong to a MFVS of D' . Hence, to process r while occupying at most $mfvs(D')$ vertices, all neighbors of r must be simultaneously occupied. This leads to $pn_{mfvs}(D') = n$. To conclude, it is sufficient to remark that $pn(D') = \max\{pn(D), pn(H)\}$. Hence, $\frac{pn_{mfvs}(D')}{pn(D')} = \frac{n}{\max\{pn(D), 2\}}$. However, computing $pn(D)$ is not in APX [5].

To prove that $\frac{mfvs_{pn}}{mfvs}$ is not in APX, let D' be the digraph composed of K_n , H_n , and D . It is easy to show that $pn(D') = pn(K_n) = n - 1$. Hence, $\frac{mfvs_{pn}(D')}{mfvs(D')} = \frac{(n-1)+(n+1)+mfvs(D)}{(n-1)+n+mfvs(D)}$. Indeed to process H_n using $n - 1$ agents, we must cover $n + 1$ nodes by agents: the central node r and successively its n neighbors (see Fig. 1 for $n = 3$). Furthermore, the minimum number of nodes covered by agents when we process D is $mfvs(D)$ because we have $n - 1$ available agents. Thus $\frac{mfvs_{pn}(D')}{mfvs(D')} = \frac{2n+mfvs(D)}{2n-1+mfvs(D)}$. To get this ratio we must compute $mfvs(D)$ which is not in APX. ■

Corollary 1. *Let $p \geq pn(D)$, $q \geq mfvs(D)$ be integers, and D a digraph. Computing $pn_q(D)$, $mfvs_p(D)$, $\frac{pn_q(D)}{pn(D)}$, or $\frac{mfvs_p(D)}{mfvs(D)}$ is not in APX.*

3 Behaviour of Ratios in General Digraphs

We study in this section behaviours of parameters introduced in Section 1.1 and their ratios, showing that, in general, good tradeoffs are impossible.

Theorem 3. $\forall C > 0, q \in \mathbb{N}$, there exists a digraph D s.t. $\frac{pn_{mfvs+q}(D)}{pn(D)} > C$.

Sketch of the Proof. Let H_n be a symmetric directed star with $n \geq 3$ branches each of which containing two vertices, excluding the central node r . H_3 is represented in Fig. 1. It is easy to check that $pn(H_n) = 2$ (e.g., Fig. 1(a)). Moreover, since the single MFVS of H_n is the set X of the n vertices adjacent to r , it is easy to check that $pn_{mfvs}(H_n) = n$ (e.g., Fig. 1(b)). We now build D with $q + 1$ copies of H_n . Hence, $pn_{mfvs+q}(D) = n$ while $pn(D) = 2$. Taking $n > 2C$, we get $\frac{pn_{mfvs+q}(D)}{pn(D)} > C$. ■

Corollary 2. *For any $C > 0$, there exists a digraph D s.t. $\frac{pn_{mfvs}(D)}{pn(D)} > C$.*

We now prove similar results for the other ratio. To do it, let us consider the digraph D of Fig. 3(a). K_{n+1}^1 is a symmetric clique of $n + 1$ nodes x_1, \dots, x_n, u .

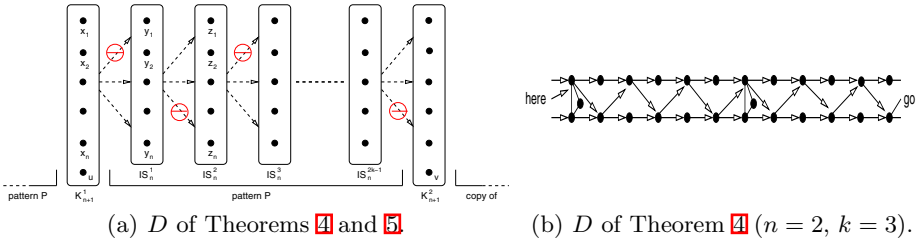


Fig. 3. Digraph D described in Theorems 4 and 5

IS_n^1 and IS_n^2 are two independent sets of n nodes each: respectively y_1, \dots, y_n and z_1, \dots, z_n . In D , there is an arc from x_i to y_j , $i = 1, \dots, n$, $j = 1, \dots, n$, if and only if $j \geq i$. There is an arc from y_i to z_j , $i = 1, \dots, n$, $j = 1, \dots, n$, if and only if $i \geq j$. The other arcs of D are built in such a way for other independent sets $IS_n^3, \dots, IS_n^{2k-1}$ and the symmetric clique K_{n+1}^2 . These arcs and the independent sets form the pattern P (see Fig. 3(a)). Between K_{n+1}^2 and K_{n+1}^1 , the same pattern is built. Fig. 3(b) represents D when $n = 2$ and $k = 3$.

Theorem 4. For any $C > 0$, there exists a digraph D s.t. $\frac{mfvs_{pn}(D)}{mfvs(D)} > C$.

Sketch of the Proof. Let D be the digraph described in Fig. 3(a) with $n = 2$. We prove that $mfvs(D) = 4$, and that for any $(3, q)$ -process strategy for D , $q \geq 2k + 3$. Taking $k > \frac{4C-3}{2}$, we get $\frac{mfvs_{pn}(D)}{mfvs(D)} \geq \frac{2k+3}{4} > C$. ■

By setting $n = p + 2$ in the digraph of Fig. 3(a) (details in [2]), we get:

Theorem 5. For any $C > 0$ and any integer $p \geq 0$, there exists a digraph D such that $\frac{mfvs_{pn+p}(D)}{mfvs(D)} > C$.

The digraph described in proof of Theorem 4 has process number 3 while $\frac{mfvs_{pn}(D)}{mfvs(D)}$ is unbounded. Lemma 1 in Section 4 shows that, in the class of symmetric digraphs with bounded process number, $\frac{mfvs_{pn}(D)}{mfvs(D)}$ is bounded.

4 Behaviour of Ratios in Symmetric Digraphs

We address the behaviour of $\frac{mfvs_{pn}(D)}{mfvs(D)}$ for symmetric digraphs D . Note that the behaviours of pn_q , pn_{mfvs} , and the different ratios, have been already studied in Sec. 3 for symmetric digraphs with bounded process number. Due to lack of space the proof of Lemma 1 is omitted and can be found in [2].

Lemma 1. For any symmetric digraph D , $\frac{mfvs_{pn}(D)}{mfvs(D)} \leq pn(D)$.

Lemma 2. $\forall \epsilon > 0$, there exists a symmetric digraph D s.t. $\frac{mfvs_{pn}(D)}{mfvs(D)} \geq 3 - \epsilon$.

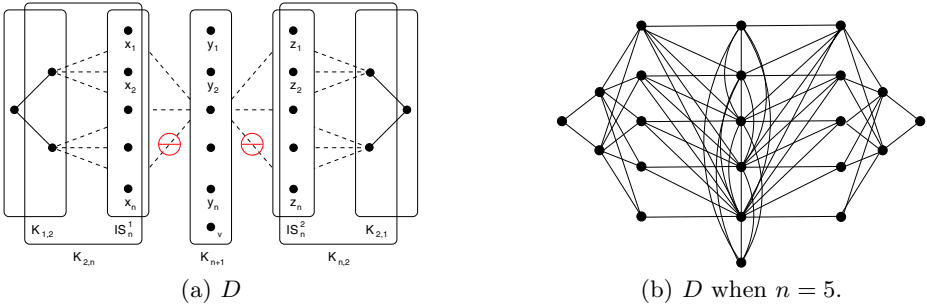


Fig. 4. Symmetric digraph D of Lemma 2 (a) and D when $n = 5$ (b)

Sketch of the Proof. Let D be the symmetric digraph of Fig. 4(a). Let IS_n^1 and IS_n^2 be two independent sets of n nodes each: respectively x_1, \dots, x_n and z_1, \dots, z_n . Let K_{n+1} be a symmetric clique of $n+1$ nodes y_1, \dots, y_n, v . In D , there are two symmetric arcs between x_i and y_j , and between z_i and y_j , $i = 1, \dots, n$, $j = 1, \dots, n$, if and only if $j \geq i$. Furthermore the two right nodes of $K_{1,2}$ and nodes of IS_n^1 form a complete symmetric bipartite subgraph (the same construction for $K_{2,1}$ and IS_n^2). The symmetric digraph of Fig. 4(b) represents D when $n = 5$. We prove that $mfvs(D) = n + 4$, and that, any $(n+1, q)$ -process strategies must cover at least $3n+2$ nodes, that is $q \geq 3n+2$. Taking $n > \frac{10}{\epsilon} - 4$, we get $\frac{mfvs_{pn}(D)}{mfvs(D)} \geq \frac{3n+2}{n+4} \geq 3 - \epsilon$. ■

Conjecture. For any symmetric digraph D , $\frac{mfvs_{pn}(D)}{mfvs(D)} \leq 3$.

5 Processing Game Out of Routing Reconfiguration

The *routing reconfiguration problem* occurs in connection-oriented networks such as telephone, MPLS, or WDM [2, 4, 5, 6, 16, 17]. In such networks, a connection corresponds to the transmission of a data flow from a source to a destination, and is usually associated with a capacitated path (or a wavelength in WDM optical networks). A *routing* is the set of paths serving the connections. To avoid confusion, we assume here that each arc of the network has capacity one, and that each connection requires one unit of capacity. Consequently, no two paths can share the same arc (valid assumption in WDM networks). When a link of the network needs to be repaired, it might be necessary to change the routing of the connection using it, and incidentally to change the routing of other connections if the network has not enough free resources. Computing a new viable routing is a well known hard problem, but it is not the concern of this paper. Indeed, this is not the end of our worries: once a new routing not using the unavailable edges is computed, it is not acceptable to stop all the connections going on, and change the routing, as it would result in a bad quality of service for the users (such operation requires minutes in WDM networks). Instead, it is preferred that each connection first establishes the new path on which it transmits data, and

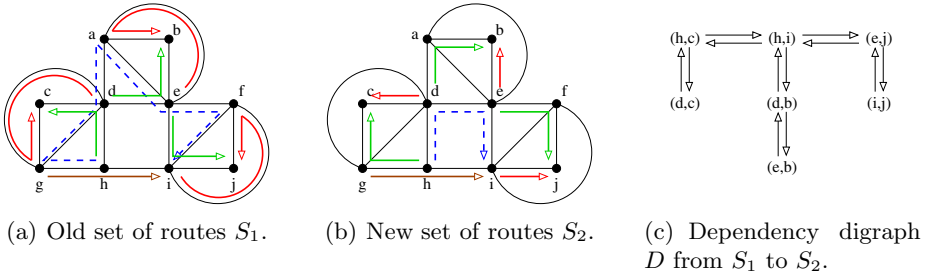


Fig. 5. Instance of the reconfiguration problem consisting of a network with 10 nodes and symmetric arcs, 8 connections $(h, i), (h, c), (d, c), (d, b), (e, b), (e, j), (i, j), (g, i)$ to be reestablished. Fig. 5(a) depicts the old set of routes S_1 , Fig. 5(b) the new set S_2 , and Fig. 5(c) the dependency digraph.

then stops the former one. This requires a proper scheduling to avoid conflicts in accessing resources (resources needed for a new path must be freed by other connections first). Furthermore, cyclic dependencies might force to interrupt some connections during that phase. The aim of the routing reconfiguration problem is to optimize tradeoffs between the total number and the concurrent number of connections to interrupt.

As an example, a way to reconfigure the instance depicted in Fig. 5 may be to interrupt connections $(h, c), (d, b), (e, j)$, then set up the new paths of all other connections, tear down their old routes, and finally, set up the new paths of connections $(h, c), (d, b), (e, j)$. Such a strategy interrupts a total of 3 connections. Another strategy may consist of interrupting the connection (h, i) , then sequentially: interrupt connection (h, c) , reconfigure (d, c) without interruption for it, set up the new route of (h, c) , then reconfigure in the same way first (d, b) and (e, b) without interruption for these two requests, and then (e, j) and (i, j) . Finally, set up the new route of (h, i) . The second strategy implies the interruption of 4 connections, but at most 2 connections are interrupted simultaneously.

Indeed, possible objectives are (1) to minimize the total number of disrupted connections [11], and (2) to minimize the maximum number of concurrent interruptions [4, 5, 16, 17]. Following [5, 11], these two problems can be expressed through the theoretical game described in this paper, on the dependency digraph [11]. Given the initial routing and the new one, the dependency digraph contains one node per connection that must be switched. There is an arc from node u to node v if the initial route of connection v uses resources that are needed by the new route of connection u . Fig. 5 shows an instance of the reconfiguration problem and its corresponding dependency digraph. In Fig. 5(c), there is an arc from vertex (d, c) to vertex (h, c) , because the new route used by connection (d, c) (Fig. 5(b)) uses resources seized by connection (h, c) in the initial configuration (Fig. 5(a)). Other arcs are built in the same way. The next theorem proves the equivalence between instances of the reconfiguration problem and dependency digraphs. Due to the lack of space, the proof can be found in [2].

Theorem 6. *Any digraph D is the dependency digraph of an instance of the routing reconfiguration problem.*

Note that a digraph may be the dependency digraph of various instances of the reconfiguration problem. Since any digraph may be the dependency digraph of a realistic instance of the reconfiguration problem, Th. 6 shows the relevance of studying these problems through the notion of dependency digraph.

A feasible reconfiguration may be defined by a (p, q) -process strategy for the corresponding dependency digraph. Problem (1) is equivalent to minimizing q (Sec. 1.1) and Problem (2) is similar to the one of minimizing p (Sec. 1.1). Consider the dependency digraph D of Fig. 5. From Sec. 1.1, we can not minimize both p and q , that is the number of simultaneous disrupted requests and the total number of interrupted connections. Indeed there does not exist a $(2, 3)$ -process strategy while $(2, 4)$ and $(3, 3)$ exist (Fig. 1).

It is now easy to make the relation between tradeoffs metrics introduced in Section 1.1 and tradeoffs for the routing reconfiguration problem. For example, pn_{mfs} introduced in Definition 3 represents the minimum number of requests that have to be simultaneously interrupted during the reconfiguration when the total number of interrupted connections is minimum. Also Section 2 shows that the problems of computing these new tradeoffs parameters for the routing reconfiguration problem are NP-complete and not in APX. Finally Section 3 proves that the loss one can expect on one parameter when minimizing the other may be arbitrarily large.

For further research, we plan to continue our study for symmetric digraphs in order to (dis)prove Conjecture 1. Moreover, it would be interesting to design exact algorithms and heuristic to compute (p, q) -process strategies.

References

- [1] Breisch, R.L.: An intuitive approach to speleotopology. *Southwestern Cavers* VI(5), 72–78 (1967)
- [2] Cohen, N., Coudert, D., Mazauric, D., Nepomuceno, N., Nisse, N.: Tradeoffs when optimizing lightpaths reconfiguration in WDM networks. RR 7047, INRIA (2009)
- [3] Coudert, D., Huc, F., Mazauric, D.: A distributed algorithm for computing and updating the process number of a forest. In: Taubenfeld, G. (ed.) DISC 2008. LNCS, vol. 5218, pp. 500–501. Springer, Heidelberg (2008)
- [4] Coudert, D., Huc, F., Mazauric, D., Nisse, N., Sereni, J.-S.: Routing reconfiguration/process number: Coping with two classes of services. In: 13th Conf. on Optical Network Design and Modeling, ONDM (2009)
- [5] Coudert, D., Perennes, S., Pham, Q.-C., Sereni, J.-S.: Rerouting requests in wdm networks. In: *AlgoTel 2005, May 2005*, pp. 17–20 (2005)
- [6] Coudert, D., Sereni, J.-S.: Characterization of graphs and digraphs with small process number. Research Report 6285, INRIA (September 2007)
- [7] Deo, N., Krishnamoorthy, S., Langston, M.A.: Exact and approximate solutions for the gate matrix layout problem. *IEEE Tr. on Comp.-Aided Design* 6, 79–84 (1987)

- [8] Ellis, J.A., Sudborough, I.H., Turner, J.S.: The vertex separation and search number of a graph. *Information and Computation* 113(1), 50–79 (1994)
- [9] Fomin, F., Thilikos, D.: An annotated bibliography on guaranteed graph searching. *Theo. Comp. Sci.* 399(3), 236–245 (2008)
- [10] Garey, M.R., Johnson, D.S.: *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York (1979)
- [11] Jose, N., Somani, A.K.: Connection rerouting/network reconfiguration. In: *Design of Reliable Communication Networks*. IEEE, Los Alamitos (2003)
- [12] Kirousis, M., Papadimitriou, C.H.: Searching and pebbling. *Theoretical Comp. Sc.* 47(2), 205–218 (1986)
- [13] Megiddo, N., Hakimi, S.L., Garey, M.R., Johnson, D.S., Papadimitriou, C.H.: The complexity of searching a graph. *J. Assoc. Comput. Mach.* 35(1), 18–44 (1988)
- [14] Parsons, T.D.: Pursuit-evasion in a graph. In: *Theory and applications of graphs*. Lecture Notes in Mathematics, vol. 642, pp. 426–441. Springer, Berlin (1978)
- [15] Robertson, N., Seymour, P.D.: Graph minors. I. Excluding a forest. *J. Comb. Th. Ser. B* 35(1), 39–61 (1983)
- [16] Solano, F.: Analyzing two different objectives of the WDM network reconfiguration problem. In: *IEEE Global Communications Conference, Globecom* (2009)
- [17] Solano, F., Pióro, M.: A mixed-integer programming formulation for the lightpath reconfiguration problem. In: *VIII Workshop on G/MPLS Networks, WGN8* (2009)

UNO Is Hard, Even for a Single Player

Erik D. Demaine¹, Martin L. Demaine¹, Ryuhei Uehara²,
Takeaki Uno³, and Yushi Uno⁴

¹ MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar St., Cambridge,
MA 02139, USA

edemaine@mit.edu, mdemaine@mit.edu

² School of Information Science, JAIST, Asahidai 1-1, Nomi, Ishikawa 923-1292, Japan
uehara@jaist.ac.jp

³ National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
uno@nii.jp

⁴ Graduate School of Science, Osaka Prefecture University, 1-1 Gakuen-cho, Naka-ku, Sakai
599-8531, Japan
uno@mi.s.osakafu-u.ac.jp

Abstract. UNO[®] is one of the world-wide well-known and popular card games. We investigate UNO from the viewpoint of combinatorial algorithmic game theory by giving some simple and concise mathematical models for it. They include cooperative and uncooperative versions of UNO, for example. As a result of analyzing their computational complexities, we prove that even a single-player version of UNO is NP-complete, while it becomes in P in some restricted cases. We also show that uncooperative two-player's version is PSPACE-complete.

1 Introduction

Playing games and puzzles is a lot of fun for everybody, and analyzing games and puzzles has long been attracted much interests of both mathematicians and computer scientists [5,8]. Among various interests and directions of researchers in mathematics and computer science, one of the central issues is their computational complexities, that is, how hard or easy to get an answer of puzzles or to decide the winner (loser) of games [2,4,10]. Such games and puzzles of interests include Nim, Hex, Peg Solitaire, Tetris, Geography, Amazons, Chess, Othello, Go, Poker, and so on. Recently, this field is sometimes called ‘algorithmic combinatorial game theory’ [2] to distinguish it from games arising from the other field, especially the classical economic game theory.

In this paper, we focus on one of the well-known and popular card games called UNO[®] and investigate it from the viewpoint of algorithmic combinatorial game theory to add it to the research list. More specifically, we propose mathematical models of UNO, which is one of the main purposes of this paper, and then examine their computational complexities. As a result, even a single-player version of UNO is computationally intractable, while we can show that the problem becomes rather easy under a certain restriction.

We organize this paper as follows: Section 2 introduces two mathematical models of UNO and their variants, and also defines UNO graphs. Among those models, Section 3

¹ UNO[®] is a registered trademark of Mattel Corporation.

focuses on a single-player version of UNO, and investigates its complexities. In Section 4, we argue with two-players' version of UNO, and show that it is PSPACE-complete. Finally, Section 5 concludes the paper.

2 Preliminaries

Games are often categorized from several aspects of properties that they have when we research it theoretically. Typical classifications are, for example, if it is multi-player or single-player, imperfect-information or perfect-information, cooperative or uncooperative, and so on [2,8]. A single-player game is automatically perfect-information and cooperative, and is sometimes called a puzzle.

2.1 Game Settings

UNO is one of the world-wide well-known and popular card games. It can be played by 2–10 players. Each player is dealt equal number of cards at the beginning of the game, where each (normal) card has its color and number (except for some special ones called 'action cards'). The basic rule is that each player plays in turn, and one can discard exactly one of his/her cards at hand in one's turn by matching the card with its color or number to the one discarded immediately before one. The objective of a single game is to be the first player to discard all the cards in one's hand before one's opponents. Thus, UNO is a (i) multi-player, (ii) imperfect-information, and (iii) uncooperative combinatorial game (see [3] for detailed rules of UNO).

In the real game setting of UNO, it is quite true that action cards play important roles to make this game complicated and interesting. However, in this paper, when we model the game mathematically, we concentrate on the most important aspect of the rules of UNO that a card has a color and a number and that one can discard a card only if its color or number match the card discarded immediately before one's turn. In addition to obeying this fundamental property, for theoretical simplicity, we set following assumptions on our mathematical models: (a) we do not take into account either action cards nor draw pile, (b) all the cards dealt to and at hand of any player are open during the game, i.e., perfect-information, (c) we do not necessarily assume that all the players have a same number of cards at the beginning of a game (unless otherwise stated), (d) any player acts rationally, e.g., any player is not allowed to skip one's turn intentionally, and (e) the first player can start a game by discarding any card he/she likes at hand.

2.2 Definitions and Notations

An UNO card has two attributes called *color* and *number*, and in general, we define a *card* to be a tuple $(x, y) \in X \times Y$, where $X = \{1, \dots, c\}$ is a set of colors and $Y = \{1, \dots, b\}$ is a set of numbers. Finite number of *players* $1, 2, \dots, p$ (≥ 1) can join an UNO game. At the beginning of a single game of UNO, each card of a set of n cards C is dealt to one player among p players, i.e., each player i is initially given a set C_i of cards; $C_i = \{t_{i,1}, \dots, t_{i,n_i}\}$ ($i = 1, \dots, p$). By definition, $\sum_{i=1}^p n_i = n$. Here, we assume that C is a

multiple set, that is, there may be more than one card with the same color and the same number. We denote a card (x, y) dealt to player i by $(x, y)_i$. When the number of players is one, we omit the subscript without any confusion. Throughout the paper, we assume without loss of generality that player 1 is the first to play, and players $1, 2, \dots, p$ play in turn in this order.

Player i can *discard* (or *play*) exactly one card currently at hand in his/her turn if the color or the number of the card is equal to each of the card discarded immediately before player i . In other words, we say that a card $t' = (x', y')_i$ can be discarded immediately after a card $t = (x, y)_i$ if and only if $(x' = x \vee y' = y) \wedge i' = i + 1 \pmod{p}$. We also say that a card t' *matches* a card t when t' can be discarded after t . A discarded card is removed from a set of cards at hand of the player. A *discarding* (or *playing*) *sequence* (of cards) of a card set C is a sequence of cards $(t_{s_1}, \dots, t_{s_k})$ such that $t_{s_i} \in C$ and $t_{s_i} \neq t_{s_j}$ ($i \neq j$). A discarding sequence $(t_{s_1}, \dots, t_{s_k})$ is *feasible* if $t_{s_{j+1}}$ matches t_{s_j} for $j = 1, \dots, k - 1$.

In our mathematical models of UNO, we specify the problems by four parameters: number of players p , number of total cards n , number of colors c and the number of numbers b . Two values c and b are assumed to be unbounded unless otherwise stated.

2.3 Models

We now define two different versions of UNO, one is cooperative and the other is uncooperative.

UNCOOPERATIVE UNO

Instance: the number of players p , and player i 's card set C_i with c colors and b numbers.

Question: determine the first player that cannot discard one's card any more.

We refer to this UNCOOPERATIVE UNO with p players as UNCOOPERATIVE UNO- p . This problem setting makes sense only if $p \geq 2$ since UNO played by a single player becomes automatically cooperative.

COOPERATIVE UNO

Instance: the number of players p , player i 's card set C_i with c colors and b numbers.

Question: can all the players make player 1 win, i.e., make player 1's card set empty before any of the other players become finished.

We abbreviate COOPERATIVE UNO played by p players as COOPERATIVE UNO- p , or simply as UNO- p . This problem setting makes sense if the number of players p is greater than or equal to 1. In UNCOOPERATIVE/COOPERATIVE UNO, when the number of players is given by a constant, such as UNO-2, it implies that p is no longer a part of the input of the problem. In addition to the assumptions (a)–(e) on game settings described in Subsec. 2.1, we set one additional assumption which changes depending on whether the game is cooperative or uncooperative: any player that cannot discard any card at hand (f1) skips one's turn but still remains in the game and waits for the next turn in cooperative games, and (f2) is a loser in uncooperative games.

We define UNO- p *graph* as a directed graph to represent 'match' relationship between two cards in the entire card set. More precisely, a vertex corresponds to a card,

and there is a directed arc from vertex u to v if and only if their corresponding cards t_v matches (can be discarded immediately after) t_u . Let us consider UNO-1 graph, i.e., UNO- p graph in case that the number of players $p = 1$. In this case, a card t' matches t if and only if t matches t' , that is, the ‘match’ relation is symmetric. This implies that UNO-1 graph becomes undirected. For UNO-2 graph, a card $t' = (x', y')_2$ matches $t = (x, y)_1$ if and only if t matches t' , and therefore, UNO-2 graph also becomes undirected. Furthermore, since a player cannot play consecutively when the number of players $p \geq 2$, UNO-2 graph becomes bipartite. In general, since n cards of a card set C is dealt to p players at the beginning of a single UNO game, i.e., C is partitioned into $C_i = \{(x, y)_i\}$, UNO- p graph becomes a (restricted) p -partite graph whose partite sets correspond to C_i .

3 Cooperative UNO

In this section, we focus on the cooperative version of UNO, and discuss its complexity when the number of players is two or one.

3.1 Two-Players’ Case

We first show that UNO-2 is intractable.

Theorem 1. *UNO-2 is NP-complete.*

Proof. Reduction from HAMILTONIAN PATH (HP).

An instance of HP is given by an undirected graph G . The problem asks if there is a Hamiltonian path in G , and it is known to be NP-complete [7]. Here, we assume without loss of generality that G is connected and is not a tree, and hence that $|V(G)| \leq |E(G)|$. We transform an instance of HP into an instance of UNO-2 as follows. Let C_1 and C_2 be the card set of players 1 and 2, respectively. We define $C_1 = \{(i, i) \mid v_i \in V(G)\}$ and $C_2 = \{(i, j) \mid \{v_i, v_j\} \in E(G)\}$. Then, notice that the resulting UNO-2 graph G' , which is bipartite, has partite sets X and Y ($X \cup Y = V(G')$) corresponding to $V(G)$ and $E(G)$, respectively, and represents vertex-edge incidence relationship of G (Fig. 1). Now we show that the answer of an instance of HP is yes if and only if the answer of an instance of UNO-2 is yes. If there is a Hamiltonian path, say $P = (v_{i_1}, v_{i_2}, \dots, v_{i_n})$, in the instance graph of HP, then there is a feasible discarding sequence alternatively by player 1’s and 2’s as $((i_1, i_1)_1, (i_1, i_2)_2, (i_2, i_2)_1, \dots, (i_{n-1}, i_{n-1})_1, (i_{n-1}, i_n)_2, (i_n, i_n)_1)$, which ends up player 1’s card before player 2’s. Conversely, if there is a feasible discarding sequence $((i_1, i_1)_1, (i_1, i_2)_2, (i_2, i_2)_1, \dots, (i_{n-1}, i_{n-1})_1, (i_{n-1}, i_n)_2, (i_n, i_n)_1)$, it visits all the vertices in X of G' exactly once, and thus the corresponding sequence of vertices $(v_{i_1}, v_{i_2}, \dots, v_{i_n})$ is a simple path visiting all the vertices in $V(G)$ exactly once, that is, a Hamiltonian path in G .

The size of an instance of UNO-2 is proportional to $|C_1| + |C_2|$. Since $|C_1| = |V(G)|$ and $|C_2| = |E(G)|$, the reduction is done in polynomial size in $|V(G)| + |E(G)|$, which is the input size of an instance of HP. This completes the proof. \square

Corollary 1. *UNO-2 is NP-complete even when the number of cards of two players are equal.*

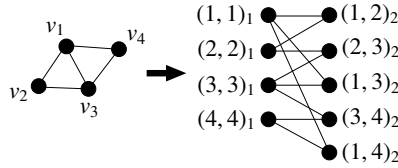


Fig. 1. Reduction from HP to UNO-2

Proof. Reduction from HAMILTONIAN PATH with specified starting vertex, which is known to be NP-complete [7].

We consider the same reduction in the proof of Theorem 1. As in that proof, we can assume $|C_1| \leq |C_2|$ without loss of generality. When $|C_1| = |C_2|$, we are done. If $|C_1| < |C_2|$, add $|C_2| - |C_1|$ cards $(n + 2, n + 2)$ and a single card $(n + 2, n + 1)$ to C_1 , a single card $(i, n + 1)$ ($i \in \{1, \dots, n\}$) to C_2 , and player 1 starts with card $(n + 2, n + 2)$. This forces the original graph G to specify a starting (or an ending) vertex of a Hamiltonian path to be v_i . □

3.2 Single-Player’s Intractable Case

In single-player’s case, two different versions of UNO, cooperative and uncooperative ones, become equivalent. We redefine this setting as the following:

UNO-1 (SOLITAIRE UNO)

Instance: a set C of n cards (x_i, y_i) ($i = 1, \dots, n$), where $x_i \in \{1, \dots, b\}$ and $y_i \in \{1, \dots, c\}$.

Question: determine if the player can discard all the cards.

Example 1. Let the card set C for player 1 is give by $C = \{(1, 3), (2, 2), (2, 3), (2, 3), (2, 4), (3, 2), (3, 4), (4, 1), (4, 3)\}$. Then, a feasible discarding sequence using all the cards is $((1, 3), (2, 3), (2, 4), (3, 4), (3, 2), (2, 2), (2, 3), (4, 3), (4, 1))$ in this order, for example, and the answer is yes. The corresponding UNO-1 graph is depicted in Fig. 2.

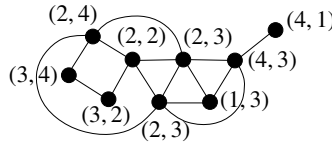


Fig. 2. An example of UNO-1 graph

We here investigate some basic properties of UNO-1 graphs. In UNO-1 graphs, all the vertices whose corresponding cards have either the same color or number form a clique. A *line graph* $L(G)$ of a given graph G is a graph whose vertices are edges of G and $\{e, e'\} \in E(L(G))$ for $e, e' \in V(L(G))$ if and only if e and e' share endpoints in G . A graph that contains no induced $K_{1,3}$ is called *claw-free*, and line graphs are claw-free.

It is not so difficult to see that UNO-1 graphs are claw-free since at least two of the three cards that match a card must have the same color or number. Furthermore, we can observe the following fact.

Observation 1. A graph is UNO-1 if and only if it is a line graph of a bipartite graph.

Now we can easily understand that UNO-1 is essentially equivalent to finding a Hamiltonian path in UNO-1 graph. However, the following fact is known.

Theorem 2. [9] HAMILTONIAN PATH for line graphs of bipartite graphs is NP-complete.

Therefore, as a corollary of this theorem, we unfortunately know that UNO is hard even for a single player.

Theorem 3. UNO-1 is NP-complete.

Here, we give a direct and concise proof of Theorem 3 for self-containedness and completeness instead of the one in [9], which further depends on [11].

Proof. A cubic graph is a graph each of whose vertex has degree 3. We reduce HAMILTONIAN PATH for cubic graphs (HP-C), which is known to be NP-complete [6], to UNO-1.

Let an instance of HP-C be G . We transform G into a graph G' , where

$$V(G') = \{(x, e) \mid x \in V(G), e = \{x, y\} \in E(G)\},$$

$$E(G') = \{((x, e), (y, e)) \mid e = \{x, y\} \in E(G)\} \cup \{((x, e_i), (x, e_j)) \mid e_i \neq e_j\}.$$

This transformation implies that any vertex $x \in V(G)$ is split into three new vertices (x, e_i) ($i = 1, 2, 3$) to form a clique (triangle), while each incident edge e_i ($i = 1, 2, 3$) to x becomes incident to a new vertex (x, e_i) . (We call it a “node gadget” as shown in Fig. 3.) Then we prepare the card set C of the player of UNO-1 to be the set $V(G')$, where the color and the number of (x, e) are x and e , respectively. We can easily confirm that there is an edge $e = (t, t')$ in G' if and only if t and t' match, i.e., G' is the corresponding UNO-1 graph for card set C . Now it suffices to show that there is a Hamiltonian path in G of an instance of HP-C if and only if there is a Hamiltonian path in G' .

Suppose there is a Hamiltonian path, say $P = (v_{i_1}, \dots, v_{i_n})$, in G . We construct a Hamiltonian path P' in G' from P as follows. Let $v_{i_{j-1}}, v_{i_j}, v_{i_{j+1}}$ be three consecutive vertices in P in this order, and let $e_1 = \{v_{i_{j-1}}, v_{i_j}\}$, $e_2 = \{v_{i_j}, v_{i_{j+1}}\}$ and $e_3 = \{v_{i_j}, v_{i_k}\}$ ($k \neq j - 1, j + 1$). Then we replace these three vertices by the sequence of vertices $(v_{i_{j-1}}, e_1), (v_{i_j}, e_1), (v_{i_j}, e_3), (v_{i_j}, e_2), (v_{i_{j+1}}, e_2)$ in G' to form a subpath in P' . For the starting two vertices v_{i_1} and v_{i_2} , we replace them by the sequence of vertices (v_{i_1}, e_1) ($e_1 \neq \{v_{i_1}, v_{i_2}\}$), (v_{i_1}, e_2) ($e_2 \neq \{v_{i_1}, v_{i_2}\}$), $(v_{i_1}, \{v_{i_1}, v_{i_2}\})$, $(v_{i_2}, \{v_{i_1}, v_{i_2}\})$ (same for the ending two vertices). We can now confirm that the resulting sequence of vertices P' in G' form a Hamiltonian path.

For the converse, we have to show that if there is a Hamiltonian path P' in UNO-1 graph G' , then there is in G . If P' visits (v, e_i) ($i = 1, 2, 3$) consecutively in any order (call it “consecutiveness”) for any v (as shown in Fig. 4 (a1) or (a2)), then P' can be transformed into a Hamiltonian path P in G in an obvious way. Suppose not, that is, a Hamiltonian path P' in G' does not visit (v, e_i) ($i = 1, 2, 3$) consecutively. It suffices to show that such P' can be transformed into another path to satisfy the consecutiveness.

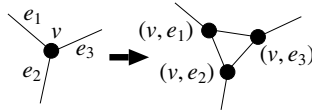


Fig. 3. A node gadget splits a vertex into three vertices to form a triangle

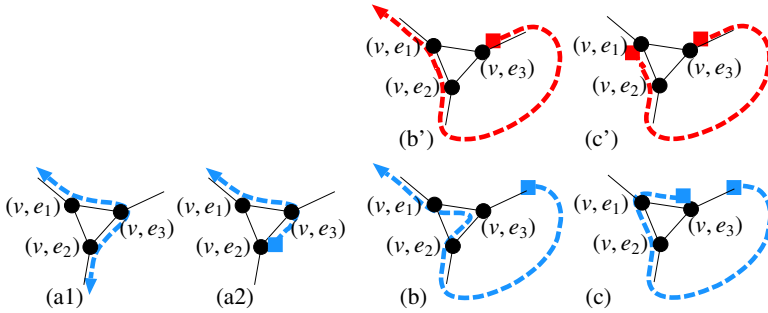


Fig. 4. Possible tours passing through a node gadget

There are two possible cases as shown in Fig. 4 (b') and (c'), both of which contain at least one end point of P' in (v, e_i) . In case (b'), we can resolve this inconsecutiveness in (v, e_i) as shown in (b), which may result in case (c') in adjacent set of three vertices. In case (c'), in order to resolve it, we can transform it into (c), which does not contain inconsecutiveness any more.

The reduction can be done in the size proportional to the size of an instance of HP-C. Thus, the proof is completed. \square

3.3 Single-Player's Tractable Case

In the remaining part of this section, we will show that such an intractable problem UNO-1 becomes tractable if the number of colors c is bounded by a constant. It will be solved by dynamic programming (DP) approach. To illustrate the DP for UNO-1, we first introduce a geometric view of UNO-1 graphs.

Since an UNO card (x, y) is an ordered pair of integer values standing for its color and number, it can be viewed as a (integer) lattice point in the 2-dimensional lattice plane. Then an UNO-1 graph is a set of points in that plane, where all the points with the same x - or y -coordinate form a clique. We call this way of interpretation a *geometric view* of UNO-1 graphs. The geometric view of an instance in Example 1 is shown in Fig. 5(a). Now the problem UNO-1, which is equivalent to finding a Hamiltonian path in UNO-1 graphs, asks if, for a given set of points in the plane and starting and ending at appropriate different points, one can visit all the points exactly once under the condition that only axis-parallel moves are allowed at each point (Fig. 5(b)).

Strategy. Let C be a set of n points and G be an UNO-1 graph defined by C . Then a subgraph P forms a Hamiltonian path if and only if it is a single path that spans G .

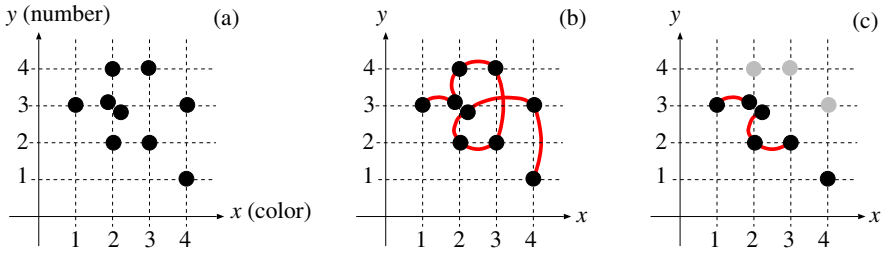


Fig. 5. (a) Geometric view of a UNO-1 graph, where all the edges are omitted, (b) a Hamiltonian path in the UNO-1 graph, and (c) a set of subpaths in the subgraph of the UNO-1 graph induced by the first 6 points; it shows $h_{\{1,2\}} = 1$, $v_{\{2,3\}} = 1$ and $d_{\{4,4\}} = 1$

Suppose a subgraph P is a spanning path of G . If we consider a subset C' of the point set C , then $P[C']$ (the subgraph of P induced by C') is a set of subpaths that spans $G[C']$ (Fig. 5(c)). We count and maintain the number of sets of subpaths by classifying subpaths into three disjoint subsets according to the types of their two endpoints.

Starting with the empty set of points, the DP proceeds by adding a new point according to a fixed order by updating the number of sets of subpaths iteratively. Finally when the set of points grows to C , we can confirm the existence of a Hamiltonian path in G by checking the number of sets of subpaths consisting of a single subpath (without isolated vertices). Remark that, throughout this DP, we regard for convenience that an isolated vertex by itself contains a (virtual) path starting and ending at itself that spans it.

Mechanism. To specify a point to be added in an iteration of the DP, we define a relation $<$ on the point set C , where $x(t)$ and $y(t)$ are x - and y -coordinates of a point t , respectively: Let t and t' be two points in C , then $t < t' \iff y(t) < y(t')$ or $(y(t) = y(t') \wedge x(t) < x(t'))$. When $t = t'$, a tie breaks arbitrary. This relation $<$ defines a total order on C , and we refer n points in C to t_1, \dots, t_n according to the increasing order of $<$. We also define $C_\ell = \{t_i \mid 1 \leq i \leq \ell\}$. Now points are added from t_1 to t_n , and consider when a new point $t_\ell = (x(t_\ell), y(t_\ell))$ is added to $C_{\ell-1}$. It must be added either to two, one or zero endpoints of different subpaths to form a new set of subpaths.

Now let $\mathcal{P}(\ell)$ be a family of sets of subpaths spanning $G[C_\ell]$. (Recall that we regard that an isolated vertex contains a path spanning itself.) Then we classify subpaths in a set of subpaths $\mathcal{P} \in \mathcal{P}(\ell)$ in the following manner: for any subpath $P \in \mathcal{P}$ and the y -coordinates of its two endpoints, either (i) both equal $y(t_\ell)$ (type-h), (ii) exactly one of two equals $y(t_\ell)$ (type-v), or (iii) none equals $y(t_\ell)$ (type-d) holds. We count the number of such three types of subpaths in \mathcal{P} further by classifying them by the x -coordinates of their endpoints. (Notice that types-h, -d are symmetric but type-v is not with respect to x -coordinate.) For this purpose, we prepare some subscript sets: a set of subscripts $K = \{1, \dots, c\}$, sets of unordered pair of subscripts $I = \binom{K}{2}$ and $I^+ = I \cup \{(i, i) \mid i \in K\}$, and sets of ordered pair of subscripts $J = K \times K$ and $J^- = J - \{(i, i) \mid i \in K\}$.

We now introduce the following parameters h , v and d to count the number of subpaths in $\mathcal{P} (\in \mathcal{P}(\ell))$ (see Fig. 5(c)):

$h_{\{i,i'\}}$: #subpaths in \mathcal{P} with endpoints $(x_i, y(t_\ell))$ and $(x_{i'}, y(t_\ell))$ for $\{i, i'\} \in I^+$,
 $v_{\{i,i'\}}$: #subpaths in \mathcal{P} with endpoints $(x_i, y(t_\ell))$ and $(x_{i'}, y')$ for $\{i, i'\} \in J$ and $y' < y(t_\ell)$,
 $d_{\{i,i'\}}$: #subpaths in \mathcal{P} with endpoints (x_i, y') and $(x_{i'}, y'')$ for $\{i, i'\} \in I^+$ and $y', y'' < y(t_\ell)$.

Then we define a $(2|I^+| + |J|)$ -dimensional vector $z(\mathcal{P})$ for a set of subpaths $\mathcal{P} (\in \mathcal{P}(\ell))$ as $z(\mathcal{P}) = (\mathbf{h}; \mathbf{v}; \mathbf{d}) = (\langle h_{\{1,1\}}, \dots, h_{\{1,c\}}, h_{\{2,2\}}, \dots, h_{\{2,c\}}, h_{\{3,3\}}, \dots, h_{\{c,c\}} \rangle; \langle v_{\{1,1\}}, \dots, v_{\{1,c\}}, v_{\{2,1\}}, v_{\{2,2\}}, \dots, v_{\{2,c\}}, v_{\{3,1\}}, \dots, v_{\{c,c\}} \rangle; \langle d_{\{1,1\}}, \dots, d_{\{1,c\}}, d_{\{2,2\}}, \dots, d_{\{2,c\}}, d_{\{3,3\}}, \dots, d_{\{c,c\}} \rangle)$. Finally, for a given vector $(\mathbf{h}; \mathbf{v}; \mathbf{d})$, we define the number of sets \mathcal{P} satisfying $z(\mathcal{P}) = (\mathbf{h}; \mathbf{v}; \mathbf{d})$ in a family $\mathcal{P}(\ell)$ by $f(\ell, (\mathbf{h}; \mathbf{v}; \mathbf{d}))$, i.e., $f(\ell, (\mathbf{h}; \mathbf{v}; \mathbf{d})) = |\{\mathcal{P} \mid \mathcal{P} \in \mathcal{P}(\ell), z(\mathcal{P}) = (\mathbf{h}; \mathbf{v}; \mathbf{d})\}|$. Now the objective of the DP is to determine if there exists a vector $(\mathbf{h}; \mathbf{v}; \mathbf{d})$ such that $f(n, (\mathbf{h}; \mathbf{v}; \mathbf{d})) \geq 1$, where all the elements in \mathbf{h} , \mathbf{v} and \mathbf{d} are 0 except for exactly one element is 1.

Recursion. As we explained, the DP proceeds by adding a new point t_ℓ to $C_{\ell-1}$. When t_ℓ is added, it is connected to either 0, 1 or 2 endpoints of existing different paths, where each endpoint has $y(t_\ell)$ or $x(t_\ell)$ in its coordinate. The recursion of the DP is described just by summing up all possible combinations of these patterns. We treat it by dividing them into three cases, one of which has two subcases: (a) a set of base cases; (b) a case in which t_ℓ is added as the first point whose y-coordinate is $y(t_\ell)$, and (b1) as an isolated vertex, or (b2) as to be connected to an existing path; (c) all the other cases.

Now we can give the DP formula for computing $f(\ell; (\mathbf{h}; \mathbf{v}; \mathbf{d}))$, however, we just explain the idea of the DP in Fig. 6 by illustrating one of the cases appearing in the DP (see [3] for full description of this recursion). In this example, consider a subpath in a graph induced by C_ℓ whose two endpoints have $x_{i'}$ and x_j in their x-coordinates. It will be counted in $h_{\{i',j\}}$. Then this subpath can be generated by adding point t_ℓ to connect to two paths in a graph induced by $C_{\ell-1}$, the one whose one endpoint is $(x_i, y(t_\ell))$ (counted in $v_{\{i,i'\}}$), and the other whose one endpoint is (k, y) ($y < y(t_\ell)$) (counted in $d_{\{j,k\}}$). The number of such paths is the sum of those for all the combinations of i, i' and j .

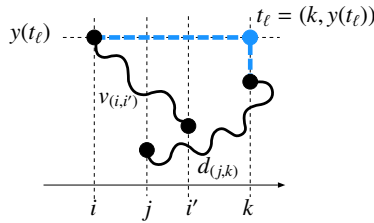


Fig. 6. An example case of the DP

Timing analysis. We first count possible combinations of arguments for f . Since ℓ varies from 0 to n , there are $\Theta(n)$ possible values. All of \mathbf{h} , \mathbf{v} and \mathbf{d} have $\Theta(c^2)$ elements, each of which can have $O(n)$ possible values, and therefore $O(n^{c^2})$ possible values in all. To compute a single value of f , it requires $O(n^4)$ lookups of previously computed values of f in case (c), while $O(n^{3c^2}) \times O(n^2)$ lookups and check-sums in cases (b1) and (b2), which is greater than $O(n^4)$. Therefore, the total running time for this DP is

$\Theta(n) \times O(n^{3c^2}) \times O(n^{3c^2+2}) = O(n^{6c^2+3}) = n^{O(c^2)}$, which is polynomial in n when c is a constant.

Since the role of colors and numbers are symmetric in UNO games, we have the following results.

Theorem 4. *UNO-1 is in P if b (the number of numbers) or c (the number of colors) is a constant.*

4 Uncooperative UNO

In this section, we deal with the uncooperative version of UNO. Especially, we show that it is intractable even for two player's case. For this purpose, we consider the following version of GENERALIZED GEOGRAPHY, which is played by two players.

GENERALIZED GEOGRAPHY

Instance: a directed graph, and a token placed on an initial vertex.

Question: a turn is to move the token to an adjacent vertex, and then to remove the vertex moved from from the graph. Player 1 and 2 take turns, and the first player unable to move loses. Determine the loser.

It is well-known that GENERALIZED GEOGRAPHY is PSPACE-complete [10], and a stronger result is presented.

Theorem 5. [10] GENERALIZED GEOGRAPHY for bipartite graphs is PSPACE-complete.

Now we show the hardness result for UNCOOPERATIVE UNO-2.

Theorem 6. UNCOOPERATIVE UNO-2 is PSPACE-complete.

Proof. Reduction from GENERALIZED GEOGRAPHY for bipartite graphs (GG-B).

Let (directed) bipartite graph G with $V(G) = X \cup Y$ be an instance of GG-B, where X and Y are two partite sets, and let $r \in X$ be an initial vertex. To construct a corresponding UNCOOPERATIVE UNO-2 instance, we first transform G into another graph G' where

$$\begin{aligned} V(G') &= \{u_s, u_t, u_c \mid u \in V(G)\}, \\ E(G') &= \{(u_t, u_c), (u_c, u_s) \mid u \in V(G)\} \cup \{(u_s, v_t) \mid (u, v) \in E(G)\} \end{aligned}$$

(Fig. 7). By construction, we can confirm that G' is a bipartite graph with $V(G') = X' \cup Y'$, where $X' = \{u_s, u_t \mid u \in X\} \cup \{u_c \mid u \in Y\}$ and $Y' = \{u_s, u_t \mid u \in Y\} \cup \{u_c \mid u \in X\}$. We let $r' = r_t \in X'$ be an initial vertex. It is easy to confirm that player 1 can win the game GG-B on G if and only if the player wins on G' . Then we prepare card sets C_i for players $i (= 1, 2)$ by

$$\begin{aligned} C_1 &= \{(x, e), (e, y) \mid e = (x, y) \in E(G'), x \in X', y \in Y'\} \\ &\quad \cup \{(e, e) \mid e = (y, x) \in E(G'), x \in X', y \in Y'\}, \\ C_2 &= \{(y, e), (e, x) \mid e = (y, x) \in E(G'), x \in X', y \in Y'\} \\ &\quad \cup \{(e, e) \mid e = (x, y) \in E(G'), x \in X', y \in Y'\}. \end{aligned}$$

This means that we prepare three cards for each arc e in $E(G')$, one for player i and two for player $3 - i$ (Fig. 8).

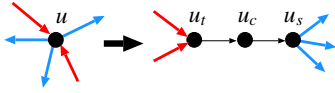


Fig. 7. Split a vertex into two edges so that edges correspond to cards

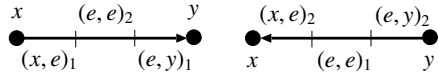


Fig. 8. Prepare three cards $(x, e)_1, (e, e)_2$ and $(e, y)_1$ for an arc $e = (x, y)$, and three cards $(e, y)_2, (e, e)_1$ and $(x, e)_2$ for an arc $e = (y, x)$

Now we show that player 1 can win in an UNCOOPERATIVE UNO-2 instance if and only if player 1 can win in an GG-B instance G' and s' . To show this, it suffices to show that any feasible playing sequence by players 1 and 2 in an GG-B instance corresponds to a feasible discarding sequence alternatively by players 1 and 2 in the corresponding UNCOOPERATIVE UNO-2 instance, and vice versa.

Suppose a situation that player 2 has just discarded a card. The discarded card belongs to either one of the following five cases: (i) (e, x) for $e = (y, x)$, (ii) (y, e) for $e = (y, x)$, (iii) (e, e) for $e = (x, y)$. Among those, for cases (ii) and (iii), since player 1 starts the game (player 1 always played before player 2's turn), there exists exactly one card (outgoing arc) that matches the one discarded by player 2 from the end vertex of the arc corresponding to the card. This forces to traverse G' along the directed arc (in forward direction), which implies to remove corresponding end vertex from G' . The only case we have to care about is case (i), where there may be multiple choices for player 1. In this case, once player 1 discarded one of match cards, the player will never play another match card afterwards, since the only card that can be discarded immediately before it has played and used up. This implies that vertex x is removed from G' . (The argument is symmetric for player 1 except that the initial card is specified.)

Now we verify that UNCOOPERATIVE UNO-2 is in PSPACE. For this, consider a search tree for UNCOOPERATIVE UNO-2, whose root is for player 1 and every node has outgoing arcs corresponding to each player's possible choices. Since the number of total cards for the two players is n , the number of choices at any turn is $O(n)$ and since at least one card is removed from either of the player's card set, the number of depth of the search tree is bounded by $O(n)$. Therefore, it requires polynomial space with respect to the input size. Thus the proof is completed. \square

5 Concluding Remarks

In this paper, we focused on UNO, the well-known card game, and gave two mathematical models for it; one is cooperative (to make a specified player win), and the other is uncooperative (to decide the player not to be able to play). As a result of analyzing their complexities, we showed that these problems are difficult in many cases, however, we also showed that a single-player's version is solvable in polynomial time under a certain restriction.

As for an obvious future work, we can try gaining speedup in dynamic programming for UNO-1 with constant number of colors by better utilizing its geometric properties. In this direction, it may be quite natural to ask if UNO-1 is fixed-parameter tractable.

Another probable direction is to investigate UNO-1 graphs from the structural point of view, since they form a subclass of claw-free graphs and seem to have interesting properties by themselves. It is also quite probable to modify our models more realistic, e.g., to take draw pile into account (as an additional player), to make all players' cards not open, and so on.

Based on our mathematical models, it is not so difficult to invent several variations or generalizations of UNO games, even for UNO-1 (single-player's version). Among them, we can generalize an UNO card from 2-tuple (2-dimensional) to d -tuple, that is, D -DIMENSIONAL UNO-1 with appropriate modifications to 'match' relation of cards. Another one is MINIMUM CARD FILL-IN, that is, given a no instance for UNO-1, find a minimum number of cards to be added to make it to be a yes instance.

Acknowledgments. We deeply appreciate Nicholas J. A. Harvey at University of Waterloo, Canada, for fruitful discussions with his deep insight at the early stage of this manuscript. We also thank for the anonymous referees for their valuable comments.

References

1. Bertossi, A.A.: The edge Hamiltonian path problem is NP-complete. *Information Processing Letters* 13, 157–159 (1981)
2. Demaine, E.D.: Playing games with algorithms: Algorithmic combinatorial game theory. In: Sgall, J., Pultr, A., Kolman, P. (eds.) *MFCS 2001*. LNCS, vol. 2136, pp. 18–32. Springer, Heidelberg (2001)
3. Demaine, E.D., Demaine, M.L., Uehara, R., Uno, T., Uno, Y.: The complexity of UNO. *CoRR abs/1003.2851* (2010)
4. Even, S., Tarjan, R.E.: A combinatorial problem which is complete in polynomial space. *J. ACM* 23, 710–719 (1976)
5. Gardner, M.: *Mathematical Games: The Entire Collection of his Scientific American Columns*. The Mathematical Association of America (2005)
6. Garey, M.R., Johnson, D.S., Tarjan, R.E.: The planar Hamiltonian circuit is NP-complete. *SIAM Journal of Computing* 5, 704–714 (1976)
7. Garey, M.R., Johnson, D.S.: *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York (1979)
8. Hearn, R.A., Demaine, E.D.: *Games, Puzzles, and Computation*. A.K. Peters (2009)
9. Lai, T.-H., Wei, S.-S.: The edge Hamiltonian path problem is NP-complete for bipartite graphs. *Information Processing Letters* 46, 21–26 (1993)
10. Lichtenstein, D., Sipser, M.: GO is polynomial-space hard. *J. ACM* 27, 393–401 (1980)

Leveling-Up in Heroes of Might and Magic III

Dimitrios I. Diochnos*

Department of Mathematics, Statistics, and Computer Science,
University of Illinois at Chicago, Chicago IL 60607, USA
diochnos@math.uic.edu

Abstract. We propose a model for level-ups in Heroes of Might and Magic III, and give an $\mathcal{O}\left(\frac{1}{\epsilon^2} \ln\left(\frac{1}{\delta}\right)\right)$ learning algorithm to estimate the probabilities of secondary skills induced by any policy in the end of the leveling-up process. We develop software and test our model in an experiment. The correlation coefficient between theory and practice is greater than 0.99. The experiment also indicates that the process responsible for the randomization that takes place on level-ups generates only a few different pseudo-random sequences. This might allow exploitation techniques in the near future; hence that process might require reengineering.

Keywords: learning, reverse engineering, inverse coupon collector's problem, software reengineering, Heroes of Might and Magic.

1 Introduction

Heroes of Might and Magic III (HoMM3) is a turn-based strategy and role-playing video game. It was developed by New World Computing for Microsoft Windows and was released by the 3DO Company in 1999. The game has been popular since its release and there is a big community worldwide. One of the major complaints of the players since the release of the game is that the *manual* was incomplete; in some cases facts were omitted, in other cases the phrasing was vague, and sometimes the descriptions were simply wrong¹. In 2003 3DO went bankrupt, the rights of the game were sold to Ubisoft, and unfortunately, there has never been an update on the manual or answers to questions about mechanisms of the game. Typically players in the online community devise techniques which aim to uncover certain mechanisms, usually through excessive testing.

This paper has similar flavor. However, we want to minimize time-consuming human testing with the aid of algorithmic techniques. Section 2 has a brief description of the game, some fundamental definitions, and the two major problems related to this paper. Section 3 gives a model regarding a (fundamental from the players' perspective) mechanism of the game. Section 4 presents a Monte Carlo approach on learning efficiently the probabilities of certain attributes under that model. Section 5 has an experiment with a dual impact. First, we examine how

* Research partially supported by NSF Grant CCF 0916708.

¹ For example, <http://heroescommunity.com/viewthread.php3?TID=17267> has a collection of more than 250 such examples.

close theory and practice are. Second, we derive quantitative estimates on the number of certain pseudo-random sequences generated by the actual game using the inverse version of the coupon collector's problem. Section 6 gives ideas about future work and extensions to our current open-source software².

Finally, note that a team of enthusiasts, since 2007, is reengineering the game under the title *Tournament Edition*. Hence, the content of the paper has independent interest since a process of the game that generates randomness might require reengineering for a more balanced game.

2 A Brief Description of the Game and Related Problems

The game allows from 2 up to 8 players to take part in the game, possibly forming allied teams. Each player rules a kingdom that belongs to one of nine different factions; not necessarily different. Each kingdom is composed primarily by different cities and armies. The goal for each player (or team of players) is to eliminate all the opponents. The army can be split into different parts and each part is guided by some *hero*. There are two classes of heroes per faction, which we call *mighty* and *magic* for reasons that will soon be apparent. Hence, we have 18 different hero classes.

Heroes have abilities, called *primary* and *secondary skills*, that mainly reinforce the battles or help in the exploration of uncharted territory. Through victories in battles heroes acquire *experience*. As more experience is accumulated and certain values are surpassed, *heroes gain levels*. This *leveling-up* process typically enhances both the primary and the secondary skills, which, in principle, results in a stronger overall army.

There are four different primary skills; **ATTACK**, **DEFENSE**, **POWER**, and **KNOWLEDGE**. Mighty heroes develop their **ATTACK** and **DEFENSE** with higher probability, while magic heroes develop their **POWER** and **KNOWLEDGE** (which are associated with magic spells) with higher probability. Moreover, there are 28 secondary skills, and each hero can acquire and store in different *slots* at most 8 during each game. Secondary skills have 3 different levels of *expertise*: **BASIC** < **ADVANCED** < **EXPERT** which are obtained in that order. Typically heroes start with two **BASIC** secondary skills or one **ADVANCED** secondary skill, and some low non-negative integer values on the primary skills. We focus on mighty heroes of these two kinds only. We examine the different heroes between the starting level 1 and level 23; at level 23 the heroes have 8 **EXPERT** secondary skills for the first time. Figure 1 gives an example of the starting configuration for one popular hero of the game that starts with two secondary skills at **BASIC** level.

Every time a hero gains a level, some primary skill is incremented by one; moreover, the user is presented with two secondary skills among which he has to choose one. We refer to the presented options as **LEFT** and **RIGHT option** since they appear respectively on the left and right part of the user's screen. Figure 2 gives an example of the dialogue that is shown on the user's screen during a

² See <http://www.math.uic.edu/~diochnos/software/games/homm3/index.php>

primary skill	value
ATTACK	4
DEFENSE	0
POWER	1
KNOWLEDGE	1

slot	secondary skill	expertise
1	OFFENSE	BASIC
2	ARTILLERY	BASIC
3		
4		
5		
6		
7		
8		

Fig. 1. Skills at level 1 for Gurnisson; hero class: Barbarian (mighty hero)

ATTACK +1	
ADVANCED OFFENSE	BASIC EARTH MAGIC

Fig. 2. Sample level-up dialogue when Gurnisson (see Fig. 1) reaches level 2. The LEFT option is ADVANCED OFFENSE; the RIGHT is a new secondary skill (BASIC EARTH MAGIC).

sample level-up. The details that determine the LEFT and RIGHT option are given in Sect. 3.

Definition 1 (Level-Up). *Level-Up is the process that determines the pair (primary skill, (LEFT secondary skill, RIGHT secondary skill)) which is presented to the user when some hero gains a new level.*

Figure 2 implies the pair (ATTACK, (OFFENSE, EARTH MAGIC)). The expertise is omitted for simplicity; it is straightforward to be computed. The pair which is presented on every level-up is called *level-up offer*. An action $a \in \mathcal{A} = \{\text{LEFT}, \text{RIGHT}\}$ determines which secondary skill is selected on a level-up. A state $\kappa \in \mathcal{K}$ on a particular level for a particular hero contains the history of all the level-up offers up to this level, as well as the actions that were performed on every level.

Definition 2 (Policy [6]). *A policy π is a mapping $\pi(\kappa, a)$ from states $\kappa \in \mathcal{K}$ to probabilities of selecting each possible action $a \in \mathcal{A}$.*

A policy is called *deterministic* if there is a unique $a \in \mathcal{A}$ with $\pi(\kappa, a) = 1$ for every $\kappa \in \mathcal{K}$. Otherwise, the policy is called *stochastic*.

Clearly, not all secondary skills have the same importance; different secondary skills enhance different abilities of the heroes. Since the release of the game there are two main problems that have tantalized the players.

Prediction Problem: The first problem has to do with the *prediction* of the offered skills during level-ups. We present a model in Sect. 3 and we focus on the secondary skills.

Evaluation Problem: The second problem has to do with the *computation* of the probabilities of acquiring secondary skills by level 23 given the policy the players are bound to follow; see Sect. 4.

3 A Model for the Prediction Problem

We are now ready to examine a model for the level-ups as this has been formed by observations and testing throughout the years. A crucial ingredient is the existence of integer weights associated with the secondary skills. These weights can be found in the file `HCTRAITS.TXT`. On every level-up the model first determines the `LEFT` option and then the `RIGHT`.

3.1 The Basic Mechanism on Secondary Skills

Case A: The hero has at least one free slot. We have two subcases.

1. At least one of the secondary skills the hero currently has is not `EXPERT`. On the next level-up the hero will be offered an upgrade of one of the existing secondary skills as the `LEFT` option, while the `RIGHT` option will be a secondary skill the hero does not already have.
2. All the secondary skills the hero currently has are `EXPERT`. On the next level-up the hero will be offered two new secondary skills.

Case B: The hero does not have a free slot. We have three subcases.

1. The hero has at least two secondary skills not `EXPERT`. On the next level-up the hero will be offered two different choices in order to upgrade one of the secondary skills that are not `EXPERT`.
2. The hero has only one secondary skill not `EXPERT`. On the next level-up the hero will be offered only this secondary skill upgraded.
3. All 8 slots of the hero are occupied by secondary skills at `EXPERT` level. No secondary skills will be offered on level-ups from now on.

3.2 Presenting Secondary Skills on Level-Ups at Random

It is unclear who discovered first the data of the file `HCTRAITS.TXT` and how. However, these *weights* appear in forums and various pages about the game for many years now. The interpretation is that the weights are directly related to the probability of acquiring a specific (`LEFT`, `RIGHT`) secondary skill offer during a level-up. In particular, consider a set \mathcal{S} of secondary skills, and to each $s \in \mathcal{S}$ we have a weight w_s associated with it. We say that a secondary skill s is *presented at random from the set \mathcal{S}* and we imply that s is selected with probability

$$\Pr(\text{selecting } s) = \frac{w_s}{\sum_{s' \in \mathcal{S}} w_{s'}} . \quad (1)$$

We implement **(II)** the usual way; i.e. a pseudo-random number generated on run-time is reduced mod $\sum_{s' \in \mathcal{S}} w_{s'}$, and then an ordering on secondary skills determines which $s \in \mathcal{S}$ is selected. In principle we have two sets of secondary skills that we are interested in; the set \mathcal{A} of secondary skills the hero already has but are not `EXPERT`, and the the set \mathcal{U} of the secondary skills that the hero does not have in any of his slots.

3.3 Two Groups of Secondary Skills Appear Periodically

Two groups of secondary skills appear periodically and hence the randomized scheme presented in Sect. 3.2 is not always applied on level-ups; the limitations of Sect. 3.1 are always applied. These two groups are:

- The *Wisdom group* which is composed by one secondary skill; `WISDOM`.
- The *Magic Schools group* which is composed by the secondary skills `AIR MAGIC`, `EARTH MAGIC`, `FIRE MAGIC`, and `WATER MAGIC`.

Let T_{Wisdom} be the period for the Wisdom group, and T_{Magic} be the period for the Magic Schools group. Hence, every at most T_{Wisdom} level-ups, if the hero does not have `EXPERT WISDOM`, `WISDOM` will be offered; as `BASIC` if `WISDOM` does not appear in one of the slots (and clearly there is at least one empty slot), otherwise as an upgrade of the current expertise. Similarly, every at most T_{Magic} level-ups a secondary skill from the Magic Schools group has to appear. We refer to these events respectively as *Wisdom Exception* and *Magic School Exception*; i.e. exceptions to the randomized scheme of Sect. 3.2. When these two exceptions coincide on a particular level-up, then Wisdom is treated first; if necessary, Magic School Exception propagates to the next level-up. Hence it might take $T_{\text{Magic}} + 1$ level-ups until Magic School Exception is applied; read below.

The model first determines the `LEFT` option and then the `RIGHT` option. Hence, the model first attempts to apply the Wisdom Exception on the `LEFT` option, and if this is impossible (e.g. Case A1 of Sect. 3.1 but the hero does not have `WISDOM` in any of the slots) then the model attempts to apply the Magic School Exception on the `LEFT` option (which might be impossible again). If the above two steps do not yield a solution, then the randomized scheme of Sect. 3.2 determines the `LEFT` option. The model then works in the same fashion in order to determine the `RIGHT` option. Note that each exception can be applied in at most one of the options on every level-up. For mighty heroes it holds $T_{\text{Wisdom}} = 6$ and $T_{\text{Magic}} = 4$.

3.4 The Leveling-Up Algorithm

Algorithm 1 gives the overall prediction scheme by incorporating the descriptions of Sects. 3.1, 3.2, and 3.3. There are four functions of primary interest during the level-ups since they handle randomness. `RNDNEW` returns a secondary skill at random from the set `U`. `RNDNEWMAGIC` returns a secondary skill at random from the set of Magic Schools that the hero does not already possess. `RNDUPGRADE` returns an upgrade of a secondary skill at random among the skills the hero has but not at `EXPERT` level. `RNDUPGRADEMAGIC` returns an upgrade of a Magic School secondary skill at random. Clearly, if there are two calls on the same function on a level-up, then, the skill that appears due to the first call is excluded from the appropriate set in the computations of the second call.

The function `WISDOMEXCEPTION` returns `TRUE` if at least T_{Wisdom} level-ups have passed since the last `WISDOM` offer and the hero does not have `EXPERT`

Algorithm 1. Determine Skills on a Level-Up.

Input: Appropriate amount of experience points to gain a level.**Output:** A level-up offer.

```

1 level ← level + 1;
2 primary ← GETPRIMARYSKILL ();
3 if ALLSECONDARYSKILLSEXPERT () then
4   if not HASFREESLOTS () then return (primary, (NULL, NULL));
5   if HASWISDOM () then
6     if MAGICEXCEPTION () and MAGICSKILLSAREAVAILABLE () then
7       LEFT ← RNDNEWMAGIC ();
8     else LEFT ← RNDNEW ();
9     RIGHT ← RNDNEW ();
10  else
11    if WISDOMEXCEPTION () then
12      LEFT ← WISDOM;
13    if MAGICEXCEPTION () and MAGICSKILLSAREAVAILABLE () then
14      RIGHT ← RNDNEWMAGIC ();
15    else RIGHT ← RNDNEW ();
16  else
17    if MAGICEXCEPTION () and MAGICSKILLSAREAVAILABLE () then
18      LEFT ← RNDNEWMAGIC ();
19    else LEFT ← RNDNEW ();
20    RIGHT ← RNDNEW ();
21 else
22   if WISDOMEXCEPTION () and HASWISDOMTOUPGRADE () then
23     LEFT ← WISDOM;
24   else if MAGICEXCEPTION () and HASMAGICTOUPGRADE () then
25     LEFT ← RNDUPGRADEMAGIC ();
26   else LEFT ← RNDUPGRADE ();
27   if CANACQUIREMORESKILLS () then
28     if WISDOMEXCEPTION () and not HASWISDOM () then
29       RIGHT ← WISDOM;
30     else if MAGICEXCEPTION () and MAGICSKILLSAREAVAILABLE () and
31       (ALLMAGICAREEXPERT () or WISDOMEXCEPTION () ) then
32       RIGHT ← RNDNEWMAGIC ();
33     else RIGHT ← RNDNEW ();
34   else if NUMBEROFSKILLSTOUPGRADE () > 1 then
35     if WISDOMEXCEPTION () and MAGICEXCEPTION () and
36       HASMAGICTOUPGRADE () then
37       RIGHT ← RNDUPGRADEMAGIC ();
38     else RIGHT ← RNDUPGRADE ();
37   else RIGHT ← NULL;
38 return (primary, (LEFT, RIGHT));
```

WISDOM. Similarly, **MAGICEXCEPTION** returns **TRUE** if at least T_{Magic} level-ups have passed since the last offer of a Magic School and the hero does not have all the Magic Schools (with non-zero weight) at **EXPERT** level. In any other case these two functions return **FALSE**. **HASWISDOMTOUPGRADE** returns **TRUE** if the hero has **WISDOM** in one of the slots but not **EXPERT**, otherwise **FALSE**. Similarly, **HASMAGICTOUPGRADE** returns **TRUE** if the hero has at least one Magic School not **EXPERT** in one of the slots, otherwise **FALSE**. **HASWISDOM** returns **TRUE** if the hero has **WISDOM** in one of the slots, otherwise **FALSE**. **CANACQUIREMORESKILLS** returns **TRUE** if the hero has at least one empty slot, otherwise **FALSE**. **MAGICSKILLSAREAVAILABLE** returns **TRUE** if there are Magic Skills with nonzero weight that the hero does not already possess, otherwise **FALSE**. **ALLMAGICAREEXPERT** returns **TRUE** if all the Magic Schools the hero has are **EXPERT**, otherwise **FALSE**. **HASFREESLOTS** returns **TRUE** if the hero has at least 1 slot empty, otherwise **FALSE**. Finally, **NUMBEROFSKILLSTOUPGRADE** returns the number of secondary skills that occupy one of the hero's slots but are not **EXPERT**.

4 Evaluating Policies

We resort to a Monte Carlo approach (Theorem [1](#)) so that we can compute efficiently the probabilities of acquiring secondary skills by level 23 given any policy with bounded error and high confidence.

Proposition 1 (Union Bound). *Let Y_1, Y_2, \dots, Y_S be S events in a probability space. Then $\Pr\left(\bigcup_{j=1}^S Y_j\right) \leq \sum_{j=1}^S \Pr(Y_j)$.*

Proposition 2 (Hoeffding Bound [3](#)). *Let X_1, \dots, X_R be R independent random variables, each taking values in the range $\mathcal{J} = [\alpha, \beta]$. Let μ denote the mean of their expectations. Then $\Pr\left(\left|\frac{1}{R} \sum_{i=1}^R X_i - \mu\right| \geq \epsilon\right) \leq e^{-2R\epsilon^2/(\beta-\alpha)^2}$.*

Theorem 1 (Monte Carlo Evaluation). *We can compute the probabilities of secondary skills induced by any policy π using $\mathcal{O}\left(\frac{1}{\epsilon^2} \ln\left(\frac{1}{\delta}\right)\right)$ simulation runs such that the aggregate error on the computed probabilities is at most ϵ with probability $1 - \delta$.*

Proof. Let $X_i^{(j)}$ be the indicator random variable that is 1 if the secondary skill j appears in the i -th run while following a policy π , and 0 otherwise. After R simulation runs, any skill j has been observed with empirical probability $\tilde{p}_j = \frac{1}{R} \sum_{i=1}^R X_i^{(j)}$. We apply Proposition [2](#) to each \tilde{p}_j with $\alpha = 0, \beta = 1, \epsilon = \epsilon/28$, we require to bound the quantity from above by $\delta/28$, and solve for R . We get $R \geq \left\lceil \frac{28^2}{2 \cdot \epsilon^2} \ln\left(\frac{28}{\delta}\right) \right\rceil$. Let Y_j be the event that \tilde{p}_j is not within $\epsilon/28$ of its true value μ_j . The above analysis implies $\Pr(Y_j) \leq \delta/28$ for every skill j . None of these bad events Y_j will take place with probability $1 - \Pr\left(\bigcup_{j=1}^{28} Y_j\right)$. By Proposition [1](#) this quantity is at least $1 - \sum_{j=1}^{28} (\delta/28) = 1 - \delta$. We now sum the errors of all the 28 computed probabilities. \square

5 An Experimental Study

In 2006 a player named `Xarfax111` suggested that the number of different sequences of secondary skill offers is actually fairly limited. An experimental study was conducted in order to verify the validity of the claim as well as test the effectiveness of the model.

The experiment consisted of 200 tests with Crag-Hack (mighty hero, class: Barbarian). Crag-Hack starts with **ADVANCED OFFENSE** but has zero weight on two secondary skills (one of which is a skill from the Magic Schools group) and hence these two can not be obtained. During the first 7 level-ups the new secondary skill that appeared was picked in every case, thereby filling all 8 slots of the hero by level 8; hence, all 8 secondary skills that appear in level 23 are determined by level 8. We call this policy *AR* (*Always Right*) since the user selects the (new) secondary skill that appears on the **RIGHT** of every level-up offer. Later in this section we examine estimates on the amount of different **RIGHT** sequences generated by the game; all imply expected values of at most 256. Note that the model allows more. For example, if we consider the cases where a Magic School appears every 4 levels and **WISDOM** every 6 levels due to exceptions (see Sect. 3.3), and allow only very heavy skills (with weights equal to 7 or 8) in between, there are $7 \cdot 6 \cdot 3 \cdot 5 \cdot 1 \cdot 4 \cdot 3 = 7,560$ different ordered combinations of new secondary skills until level 8. A similar calculation allowing all possible skills in between gives 7,325,640 different combinations.

Out of the 200 tests, only 128 yielded different sequences of new secondary skills; 76 sequences occurred once, 33 sequences occurred twice, 18 sequences occurred three times, and 1 sequence occurred four times. Figure 3 presents graphically the probabilities of the secondary skills in three cases; the expectations according to the model, the values as these were recorded on the 200 tests, and the values when we consider only the 128 distinct sequences. The correlation coefficient between the expected probabilities and the ones computed empirically after 200 tests was 0.9914. Moreover, the correlation coefficient between the expected probabilities and the ones formed by the 128 unique sequences was 0.9946.

We now turn our attention to the second part of the experiment. We want to estimate the amount of different sequences of new secondary skills that appear when we follow this policy up to level 8 based on our observations on the collisions of the various sequences. This problem is essentially the inverse version of the Coupon Collector's Problem, and is well studied in Statistics; see e.g. [12]. We will follow the simple route of working with expectations, assume equal selection probability for each coupon, and in the end we will arrive to the same formula for prediction as in [1]; see also [5, Sect. 3.6]. Let H_N be the N -th harmonic number. The expected time T_i to find the first i different coupons is given by

$$T_i = \sum_{j=N-i+1}^N \frac{N}{j} = \sum_{j=1}^N \frac{N}{j} - \sum_{j=1}^{N-i} \frac{N}{j} = N(H_N - H_{N-i}). \quad (2)$$

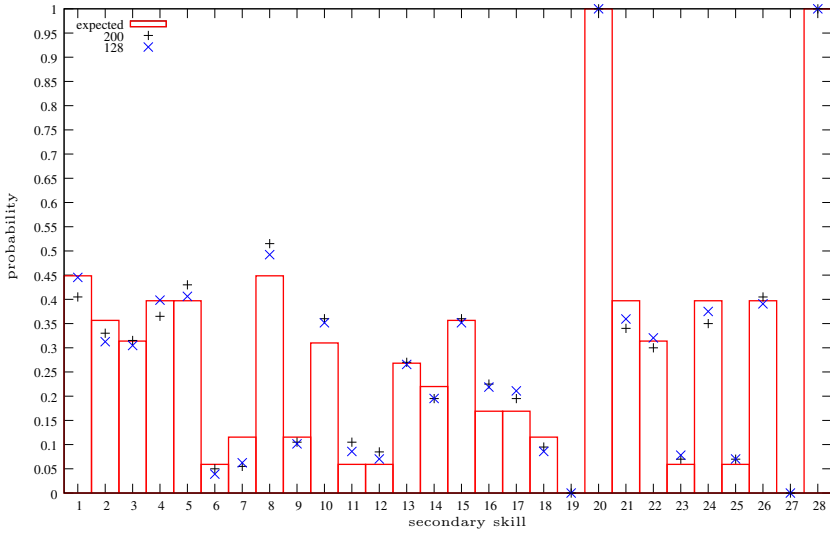


Fig. 3. Probabilities of secondary skills; Crag-Hack, AR policy. The boxes indicate the expected values based on Sect. 3, the +’s present the values computed on all 200 tests, while the x’s present the values computed on the 128 different sequences. The secondary skills are shown in lexicographic ordering; i.e. 1: AIR MAGIC, 2: ARCHERY, etc. Note that 28: WISDOM and in every test WISDOM was offered by at most level 6.

Lemma 1. Let $N \geq 3$ and k be fixed such that $k \in \{2, 3, \dots, N - 1\}$. Then, the quantity $Q(N) = N(H_N - H_{N-k})$ is monotone decreasing.

Proof. We want $Q(N) > Q(N+1)$ or equivalently $N(-\frac{1}{N+1} + \frac{1}{N+1-k}) > H_{N+1} - H_{N+1-k}$. However, $H_{N+1} - H_{N+1-k} = \sum_{i=N+2-k}^{N+1} \frac{1}{i} < \frac{1}{N+2-k}$. It suffices to show $\frac{kN}{(N+1)(N+1-k)} > \frac{k}{N+2-k}$, which holds since $k > 1$. \square

The history of the 128 different sequences of new secondary skills is shown with a thick solid line in Fig. 4. Let D_i be the number of different new secondary skill combinations that have occurred on the i -th test. Working only with expectations we want to use (2), set $T_i = D_i$, and solve for N . This is precisely the solution asserted by the maximum likelihood principle. Typically, this is a floating point number; both the floor and the ceiling of that number are candidates for the solution; see [1]. We draw the average of those candidates with a thin solid line in Fig. 4. In order to get a better picture we apply Lemma 1 and calculate all the values of N such that $T_i \in [D_i - 0.5, D_i + 0.5]$; note that T_i rounded to the closest integer is equal to D_i . We get a lower and upper bound on the above mentioned values of N and we plot them with dashed lines in Fig. 4.

Another heuristic estimate can be obtained by looking at the ratio

$$\lambda(x, t) = \lambda_x(t) = \frac{\text{new sequences found in the last } x \text{ tests}}{x}. \tag{3}$$

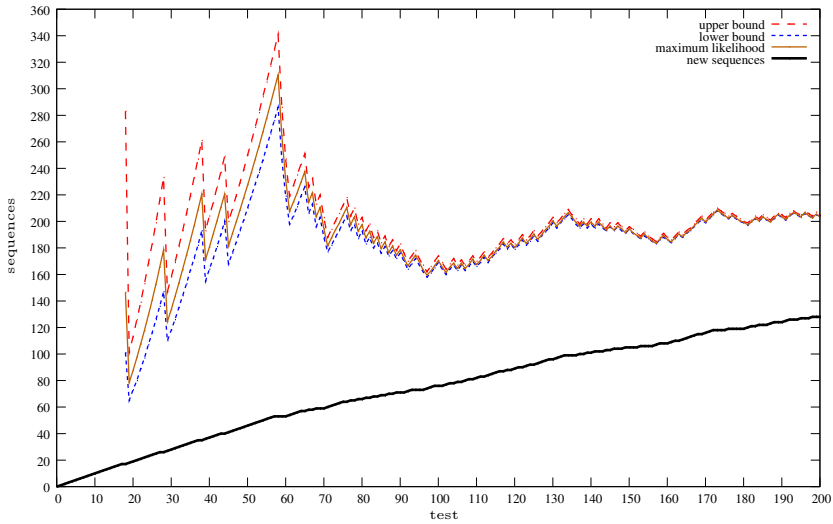


Fig. 4. New sequences and estimates on the amount of different sequences

We use the ratios $\lambda_x(200)$ for $x = 10, 20, 40$, and 80 as heuristic approximations of the true probability of discovering a new sequence at test $t = 200$. All four of them lie in the interval $[0.4, 0.5]$ which implies an estimate of $213 \leq N \leq 256$ different sequences in total.

6 A Glimpse Beyond

All the estimates of Sect. 5 are interesting since they are at most 256; a single byte can encode all of them. Quite recently (2009), a player named `AlexSpl` has developed similar software³ and according to the descriptions we have⁴ there are 255 different cases to be evaluated; there is a description of the random number generator too. Our model and `AlexSpl`'s description differ in the function `RNDUPGRADEMAGIC` of Sect. 3.4 where `AlexSpl` suggests treating all the participating skills with weights equal to 1. Compared to our 0.9914 and 0.9946 values for the correlation coefficient in the experiment of Sect. 5, `AlexSpl`'s approach achieves 0.9959 and 0.9974 respectively. `AlexSpl`'s software is not open-source, and there is no description about his method on attacking the problem. In any case, we embrace relevant software; at the very least it promotes more robust software for everyone. Both Sect. 5 and `AlexSpl`'s description imply a small space to be explored in practice. This suggests techniques of exploitation that will allow us to predict most, if not all, sequences online after a few level-ups, at least for a few popular heroes and policies followed in tournaments.

³ <http://heroescommunity.com/viewthread.php3?TID=27610>

⁴ <http://heroescommunity.com/viewthread.php3?TID=17812&pagenumber=12>

Unfortunately, this will greatly reduce the fun and the luck factor of the real game! This is why reengineering is needed.

Coming back to our approach, perhaps the most important thing is to extend the current implementations and compute probabilities for magic heroes too. Another important extension is the computation of the probabilities for all the intermediate levels. Moreover, there are policies closer to tournament play which have not been implemented yet. In addition, we would like to ask questions such as *what is the probability of acquiring (TACTICS \wedge AIR MAGIC \wedge OFFENSE) \vee (EARTH MAGIC \wedge LOGISTICS) under various policies ?* Some work has been done (`ansaExtended`); however, it is not part of the Monte Carlo approach. Also, parallelize the computations with the inclusion of a library such as [4]. Finally, is there a simpler alternative for Algorithm II of Sect. 3.4?

There are certainly exciting times ahead of both the developers and the players. We are eagerly looking forward into that future!

Acknowledgements. The author wants to thank the participants in the thread <http://heroescommunity.com/viewthread.php3?TID=17812>; a special thanks to Ecoris for his tests and techniques. His insight will affect future work for sure. Finally, the author thanks György Turán for fruitful discussions, and the referees for their insightful comments.

References

1. Dawkins, B.: Siobhan's Problem: The Coupon Collector Revisited. *The American Statistician* 45(1), 76–82 (1991)
2. Efron, B., Thisted, R.: Estimating the Number of Unseen Species: How Many Words Did Shakespeare Know? *Biometrika* 63(3), 435–447 (1976)
3. Hoeffding, W.: Probability Inequalities for Sums of Bounded Random Variables. *Journal of the American Statistical Association* 58(301), 13–30 (1963)
4. Mascagni, M.: SPRNG: A Scalable Library for Pseudorandom Number Generation. In: *Parallel Processing for Scientific Computing* (1999)
5. Motwani, R., Raghavan, P.: *Randomized Algorithms*. Cambridge University Press, Cambridge (1995)
6. Sutton, R.S., Barto, A.G.: *Reinforcement Learning: An Introduction*. MIT Press, Cambridge (1998)

The Magic of a Number System^{*}

Amr Elmasry¹, Claus Jensen², and Jyrki Katajainen³

¹ Max-Planck Institut für Informatik, Saarbrücken, Germany

² The Royal Library, Copenhagen, Denmark

³ Department of Computer Science, University of Copenhagen, Denmark

Abstract. We introduce a new number system that supports increments with a constant number of digit changes. We also give a simple method that extends any number system supporting increments to support decrements using the same number of digit changes. In the new number system the weight of the i th digit is $2^i - 1$, and hence we can implement a priority queue as a forest of heap-ordered complete binary trees. The resulting data structure guarantees $O(1)$ worst-case cost per *insert* and $O(\lg n)$ worst-case cost per *delete*, where n is the number of elements stored.

1 Introduction

The interrelationship between numerical representations and data structures is efficacious. As far as we know, the issue was first discussed in the paper by Vuillemin on binomial queues [14] and the seminar notes by Clancy and Knuth [5]. However, in many write-ups such connection has not been made explicit. In this paper, we introduce a new number system and use it to develop a priority queue that, in a sense, utilizes the structure of Williams' binary heap [15].

In the computing literature, many types of priority queues have been studied. Sometimes it is sufficient to construct priority queues that support the elementary operations *find-min*, *insert*, and *delete*. Our binary-heap variant supports *find-min* and *insert* at $O(1)$ worst-case cost, and *delete* at $O(\lg n)$ worst-case cost, n denoting the number of elements stored prior to the operation. In contrast, $\Omega(\lg \lg n)$ is known to be a lower bound on the worst-case complexity of *insert* for the standard binary heaps [9].

In a positional numeral system, a sequence of digits $\langle d_0, d_1, \dots, d_{k-1} \rangle$ is used to represent a positive integer, k being the length of the representation. By convention, d_0 is the least-significant digit and d_{k-1} the most-significant digit. If w_i is the weight of d_i , $\langle d_0, d_1, \dots, d_{k-1} \rangle$ represents the (decimal) number $\sum_{i=0}^{k-1} d_i w_i$. Different numerical representations are obtained by enforcing different invariants for the values that d_i and w_i can take for $i \in \{0, 1, \dots, k-1\}$. In accordance, the performance characteristics of some operations may vary for different number systems. Important examples of number systems include the *binary system*,

^{*} The work of the authors was partially supported by the Danish Natural Science Research Council under contract 09-060411 (project “Generic programming—algorithms and tools”). A. Elmasry was supported by the Alexander von Humboldt Foundation and the VELUX Foundation.

Table 1. The number systems used in some priority queues and their effect on the complexity of *insert*. All the mentioned structures can support *find-min* at $O(1)$ worst-case cost and *delete* at $O(\lg n)$ worst-case cost, n is the current size of the data structure.

Digits in use	Binomial-queue variants	Binary-heap variants
$\{0, 1\}$	$O(\lg n)$ worst case & $O(1)$ amortized [14]	$O(\lg^2 n)$ worst case [folklore]
$\{0, 1\}$ & first non-zero digit may be 2	$O(1)$ worst case [3]	$O(\lg n)$ worst case & $O(1)$ amortized [2][11] ^a
$\{0, 1, 2\}$	$O(1)$ worst case [4][7]	$O(\lg n)$ worst case [13] ^b & $O(1)$ amortized [this paper] ^a
$\{1, 2, 3, 4\}$	$O(1)$ worst case [8] ^a	
$\{0, 1, 2, 3, 4\}$		$O(1)$ worst case [this paper]

^a *borrow* has $O(1)$ worst-case cost.

^b *meld* has $O(\lg m \cdot \lg n)$ worst-case cost, where m and n are the sizes of the data structures melded.

where $d_i \in \{0, 1\}$ and $w_i = 2^i$; the *redundant binary system*, where $d_i \in \{0, 1, 2\}$ and $w_i = 2^i$; the *skew binary system*, where $w_i = 2^{i+1} - 1$; the *canonical skew binary system* [10], where $w_i = 2^{i+1} - 1$ and $d_i \in \{0, 1\}$ except that the first non-zero digit may also be 2; and the *zeroless variants*, where $d_i \neq 0$. These systems and some other number systems, together with some priority-queue applications, are discussed in [12, Chapter 9]. We have gathered the most relevant earlier results related to the present study in Table 1.

A binomial queue is a forest of heap-ordered binomial trees [14]. If the queue stores n elements and the binary representation of n contains a 1-bit at position i , $i \in \{0, 1, \dots, \lfloor \lg n \rfloor\}$, the queue contains a tree of size 2^i . In the binary number system, an addition of two 1-bits at position i results in a 1-bit at position $i + 1$. Correspondingly, in a binomial queue two trees of size 2^i are linked resulting in a tree of size 2^{i+1} . For binomial trees, this linking is possible at $O(1)$ worst-case cost. Since *insert* corresponds to an increment of an integer, *insert* may have logarithmic cost due to the propagation of carries. Instead of relying on the binary system, some of the aforementioned or other specialized variants could be used to avoid cascading carries. That way a binomial queue can support *insert* at $O(1)$ worst-case cost [3,4,7,8]. A binomial queue based on a zeroless system where $d_i \in \{1, 2, 3, 4\}$ also supports the removal of an unspecified element—an operation that we call *borrow*—at $O(1)$ worst-case cost [8].

An approach similar to that used for binomial queues has been proposed for binary heaps too. The components in this case are either *perfect heaps* [2][11] or *pennants* [13]. A perfect heap is a heap-ordered complete binary tree, and accordingly is of size $2^i - 1$ where $i \geq 1$. A pennant is a heap-ordered tree whose root has one subtree that is a complete binary tree, and accordingly is of size 2^i where $i \geq 0$. In contrary to binomial trees, the worst-case cost of linking two pennants of the same size is logarithmic, not constant. To link two perfect heaps of the same size, we even need to have an extra node, and if this node is arbitrary chosen the cost per link is as well logarithmic. When perfect heaps (pennants)

are used, it is natural to rely on the skew (redundant) binary system. Because of the cost of linking, this approach only guarantees $O(\lg n)$ worst-case cost [13] and $O(1)$ amortized cost per *insert* [2,11].

Our most interesting contribution is the new number system which uses five symbols and skew weights. As the title of the paper indicates, it may look mysterious why the number system works as effectively as it does; a question that we answer in Section 2. As a by-product, we show how any number system supporting increments can be extended to support decrements with the same number of digit changes; we expect this simple technique to be helpful in the design of new number systems. The application to binary heaps is discussed in Section 3.

2 The Number System

We represent an integer n as a sequence of digits $\langle d_0, d_1, \dots, d_{k-1} \rangle$, least-significant digit first, such that

- $d_i \in \{0, 1, 2, 3, 4\}$ for all $i \in \{0, 1, \dots, k-1\}$,
- $w_i = 2^{i+1} - 1$ for all $i \in \{0, 1, \dots, k-1\}$, and
- the decimal value of n is $\sum_{i=0}^{k-1} d_i w_i$.

Remark 1. In general, a skew binary number system that uses five symbols is redundant, i.e. there is possibly more than one representation for the same integer. However, for our system, the way the operations are performed guarantees a unique representation for any integer.

2.1 Operations

We define two operations on sequences of digits. An *increment* increases the value of the corresponding integer by 1, and a *decrement* decreases the value by 1. Each operation involves at most four digit changes. We say that a sequence of digits is *valid* if it can be procured by repeatedly performing the increment operation starting from zero. It follows from the correctness proof of Section 2.2 that every valid sequence in our number system has $d_i \in \{0, 1, 2, 3, 4\}$.

Increment. Assume that d_j is equal to 3 or 4. We define how to perform a *fix* for d_j as follows:

1. Decrease d_j by 3.
2. Increase d_{j+1} by 1.
3. If $j \neq 0$, increase d_{j-1} by 2.

Remark 2. Since $w_0 = 1$, $w_1 = 3$, and $3w_i = w_{i+1} + 2w_{i-1}$ for $i \geq 1$, the fix does not change the value of the number.

To increment a number, we perform the following steps:

1. Increase d_0 by 1.
2. Find the smallest j where $d_j \in \{3, 4\}$. If no such digit exists, set j to -1 .
3. If $j \neq -1$, perform a fix for d_j .
4. Push j onto an undo stack.

Remark 3. When defining all valid sequences by means of increments, we make the representation of every integer unique. For example, decimal numbers from 1 to 30 are represented by the following sequences: 1, 2, 01, 11, 21, 02, 12, 22, 03, 301, 111, 211, 021, 121, 221, 031, 302, 112, 212, 022, 122, 222, 032, 303, 113, 2301, 0401, 3111, 1211, 2211.

Remark 4. If we do not insist on doing the fix at the smallest index j where $d_j \in \{3, 4\}$, the representation may become invalid. For example, starting from 22222, which is valid, two increments will subsequently give 03222 and 30322. If we now repeatedly fix the second 3 in connection with the forthcoming increments, after three more increments we will end up at 622201. Actually, one can show that d_0 can get as high as $\Theta(k^2)$ for a k -digit representation.

Decrement. We define the *unfix*, as the reverse of the fix, as follows:

1. Increase d_j by 3.
2. Decrease d_{j+1} by 1.
3. If $j \neq 0$, decrease d_{j-1} by 2.

As a result of the increments, we maintain an undo stack containing the positions where the fixes have been performed. To decrement a number, we perform the following steps:

1. Pop the index at the top of the stack; let it be j .
2. If $j \neq -1$, perform an unfix for d_j .
3. Decrease d_0 by 1.

Remark 5. Since a decrement is the reverse of an increment, the correctness of a decrement operation (that it creates a valid sequence) directly follows from the correctness of the increment.

Remark 6. After any sequence of increments and decrements, the stack size will be equal to the difference between the number of increments and decrements, i.e. the value of the number.

2.2 Correctness

To prove that the operations work correctly, we only need to show that by applying any number of increments starting from zero every digit satisfies $d_i \in \{0, 1, 2, 3, 4\}$. In a fix, although we increase d_{j-1} by 2, no violations could happen as d_{j-1} was at most 2 before the increment. So, a violation would only be possible if, before the increment, d_0 or d_{j+1} was 4.

Define a *block* to be a maximal sequence, none of its digits is 3 or 4 except the last digit. Hence, any sequence representing a number consists of a sequence of blocks, if any, followed by a sequence of digits not in a block, if any, called the *tail*. Since every digit in the tail is less than 3, increasing any of its digits by 1 keeps the sequence valid.

To characterize valid sequences, we borrow some notions from the theory of automata and formal languages. We use d^* to denote the set containing zero or more repetitions of the digit d . Let $\mathcal{S} = \{S_1, S_2, \dots\}$ and $\mathcal{T} = \{T_1, T_2, \dots\}$ be two

sets of sequences of digits. We use $\mathcal{S} \mid \mathcal{T}$ to denote the set containing all sequences in \mathcal{S} and \mathcal{T} . We write $\mathcal{S} \subseteq \mathcal{T}$ if for every $S_i \in \mathcal{S}$ there exists $T_j \in \mathcal{T}$ such that $S_i = T_j$, and $\mathcal{S} = \mathcal{T}$ if $\mathcal{S} \subseteq \mathcal{T}$ and $\mathcal{T} \subseteq \mathcal{S}$. We also write $S \xrightarrow{+} T$ indicating that the sequence T results by applying an increment to S , and $\mathcal{S} \xrightarrow{+} \mathcal{T}$ if for each $S_i \in \mathcal{S}$ there exists $T_j \in \mathcal{T}$ such that $S_i \xrightarrow{+} T_j$. Furthermore, we write \overline{S} for a sequence that results from S by increasing its least-significant digit by 1 without performing a fix, and $\overline{\mathcal{S}}$ for $\{\overline{S_1}, \overline{S_2}, \dots\}$. To capture the intricate structure of valid sequences, we recursively define the following sets of sequences.

$$\tau \stackrel{\text{def}}{=} (1^*2^*)^* \tag{1}$$

$$\alpha \stackrel{\text{def}}{=} 2^*1\gamma \tag{2}$$

$$\beta \stackrel{\text{def}}{=} \tau \mid 2^*3\psi \tag{3}$$

$$\gamma \stackrel{\text{def}}{=} 1 \mid 2\tau \mid 3\beta \mid 4\psi \tag{4}$$

$$\psi \stackrel{\text{def}}{=} 0\gamma \mid 1\alpha \tag{5}$$

$$\phi \stackrel{\text{def}}{=} (21 \mid 02 \mid 12)\tau \tag{6}$$

Remark 7. The intersections among the defined sets are non-empty.

The above definitions imply that

$$\begin{aligned} \overline{\beta} &= 1 \mid 2\tau \mid 3\tau \mid 32^*3\psi \mid 4\psi \\ &= 1 \mid 2\tau \mid 3(\tau \mid 2^*3\psi) \mid 4\psi \\ &= 1 \mid 2\tau \mid 3\beta \mid 4\psi \\ &= \gamma \\ \overline{\psi} &= 1\gamma \mid 2\alpha \\ &= 1\gamma \mid 22^*1\gamma \\ &= \alpha \end{aligned}$$

Next, we show that any sequence representing an integer in our number system can be fully characterized. More precisely, such sequences can be classified into a fixed number of sets, that we call *states*, where every increment is equivalent to a transition whose current and resulting states are uniquely determined from the sequence. Since this state space is closed under the transitions, and each is characterized by a set of sequences of digits with $d_i \in \{0, 1, 2, 3, 4\}$, the correctness of the increment operation follows.

Define the following nine states: 12α , 22β , 03β , 30γ , 11γ , 23ψ , 04ψ , 31α , and ϕ . Next, we show that the following are the only possible transitions.

1. $\underline{12\alpha \xrightarrow{+} 22\beta}$
 $\underline{12\alpha = 122^*1\gamma = 122^*1(1 \mid 2\tau \mid 3\beta \mid 4\psi)}$
 $\xrightarrow{+} 22\tau \mid 222^*3(0\overline{\beta} \mid 1\overline{\psi}) = 22\tau \mid 222^*3(0\gamma \mid 1\alpha) = 22(\tau \mid 2^*3\psi) = 22\beta$
2. $\underline{22\beta \xrightarrow{+} 03\beta}$
 Obvious.

3. $\frac{03\beta \xrightarrow{+} 30\gamma}{03\beta \xrightarrow{+} 30\bar{\beta}} = 30\gamma$
4. $\frac{30\gamma \xrightarrow{+} 11\gamma}{\text{Obvious.}}$
5. $\frac{11\gamma \xrightarrow{+} \phi \mid 23\psi}{11\gamma = 11(1 \mid 2\tau \mid 3\beta \mid 4\psi)}$
 $\xrightarrow{+} \phi \mid 23(0\bar{\beta} \mid 1\bar{\psi}) = \phi \mid 23(0\gamma \mid 1\alpha) = \phi \mid 23\psi$
6. $\frac{23\psi \xrightarrow{+} 04\psi}{\text{Obvious.}}$
7. $\frac{04\psi \xrightarrow{+} 31\alpha}{04\psi \xrightarrow{+} 31\bar{\psi}} = 31\alpha$
8. $\frac{31\alpha \xrightarrow{+} 12\alpha}{\text{Obvious.}}$
9. $\frac{\phi \xrightarrow{+} \phi \mid 22\beta}{\phi = (21 \mid 02 \mid 12)\tau \xrightarrow{+} \phi \mid 22\beta}$

Remark 8. By Remark [3](#), the numbers from 1 up to 21 (whose decimal equivalent is 5) are valid, so we may assume that ϕ is the initial state.

2.3 Properties

The following lemma directly follows from the definition of the sequence families.

Lemma 1

- The body of a block ending with 4 constitutes either 0 or 12^*1 .
- The body of a block ending with 3 constitutes either 0, 12^*1 , or 2^* .
- Each 4, 23 and 33 is followed by either 0 or 1.
- There can be at most one 0 in the tail, which must then be its first digit.

The next lemma bounds the average of the digits of any valid sequence to be at most 2.

Lemma 2. *If $\langle d_0, d_1, \dots, d_{k-1} \rangle$ is a representation of a number in our number system, then $\sum_{i=0}^{k-1} d_i \leq 2k$. If k' denotes the number of the digits constituting the blocks of a number, then $2k' - 1 \leq \sum_{i=0}^{k'-1} d_i \leq 2k'$.*

Proof. We prove the second part of the lemma, which implies the first part following the fact that any digit in the tail is at most 2. First, we show by induction that the sum of the digits of a subsequence of the form $\alpha, \beta, \gamma, \psi$ is respectively $\sum_{\alpha} = 2\ell_{\alpha}, \sum_{\beta} = 2\ell_{\beta}, \sum_{\gamma} = 2\ell_{\gamma} + 1, \sum_{\psi} = 2\ell_{\psi} - 1$, where $\ell_{\alpha}, \ell_{\beta}, \ell_{\gamma}, \ell_{\psi}$ are the lengths of the corresponding subsequences when ignoring the trailing digits that are not in a block. The base case is for the subsequence solely consisting of the digit 3, which is a type- γ subsequence with $\ell_{\gamma} = 1$ and $\sum_{\gamma} = 3$. From definition [\(2\)](#), $\sum_{\alpha} = 2(\ell_{\alpha} - \ell_{\gamma} - 1) + 1 + \sum_{\gamma} = 2(\ell_{\alpha} - \ell_{\gamma} - 1) + 1 + 2\ell_{\gamma} + 1 = 2\ell_{\alpha}$. From definition [\(3\)](#), $\sum_{\beta} = 2(\ell_{\beta} - \ell_{\psi} - 1) + 3 + \sum_{\psi} = 2(\ell_{\beta} - \ell_{\psi} - 1) + 3 + 2\ell_{\psi} - 1 = 2\ell_{\beta}$.

From definition (4), $\sum_\gamma = 3 + \sum_\beta = 3 + 2\ell_\beta = 3 + 2(\ell_\gamma - 1) = 2\ell_\gamma + 1$. Alternatively, $\sum_\gamma = 4 + \sum_\psi = 4 + 2\ell_\psi - 1 = 4 + 2(\ell_\gamma - 1) - 1 = 2\ell_\gamma + 1$. From definition (5), $\sum_\psi = \sum_\gamma = 2\ell_\gamma + 1 = 2(\ell_\psi - 1) + 1 = 2\ell_\psi - 1$. Alternatively, $\sum_\psi = 1 + \sum_\alpha = 1 + 2\ell_\alpha = 1 + 2(\ell_\psi - 1) = 2\ell_\psi - 1$. The induction step is accordingly complete, and the above bounds follow.

Consider the subsequence that constitutes the blocks of a number. Let k' be the length of such subsequence. Since any sequence of blocks can be represented in one of the forms: $12\alpha, 22\beta, 03\beta, 30\gamma, 11\gamma, 23\psi, 04\psi, 31\alpha$ (excluding the tail). It follows that $\ell_\alpha, \ell_\beta, \ell_\gamma, \ell_\psi = k' - 2$. A case analysis implies that $\sum_{i=0}^{k'-1} d_i$ either equals $2k' - 1$ or $2k'$ for all cases. \square

3 Application: A Worst-Case Efficient Priority Queue

Let us now use the number system for developing a worst-case efficient priority queue. Recall that a binary heap [15] is a *heap-ordered* binary tree where the element stored at a node is no greater than that stored at its children. We rely on *perfect heaps* that are complete binary trees storing $2^h - 1$ elements for some integer $h \geq 1$. Moreover, our heaps are pointer-based; each node keeps pointers to its parent and children.

As in a binomial queue, which is an ordered collection of heap-ordered binomial trees, in our binary-heap variant we maintain an ordered collection of perfect heaps. A similar approach has been used in several earlier publications [2, 11, 13]. The key difference between our approach and the earlier approaches is the number system in use; we rely on our new number system. Assuming that the number of elements being stored is n and that $\langle d_0, d_1, \dots, d_{\lfloor \lg n \rfloor} \rangle$ is the representation of n in this number system, we maintain the invariant that the number of perfect heaps of size $2^{i+1} - 1$ is exactly d_i .

To keep track of the perfect heaps, we maintain a resizable array whose i th entry points to the roots of the perfect heaps of size $2^i - 1$. Since it is important to access the *big* digits 3 and 4 quickly, we maintain the big digits in a singly-linked list by having an additional *jump pointer* at each array entry. In addition, to support borrowing, we also need an *undo stack* holding the indexes corresponding to the positions where fixes were made. To facilitate fast *find-min*, we maintain a pointer to a *root* that stores the minimum among all elements.

The basic toolbox for manipulating perfect heaps is described in most textbooks on algorithms and data structures (see, for example, [6, Chapter 6]). We need the function *siftdown* to reestablish the heap order when an element at a node is made larger, and the function *siftup* to reestablish the heap order when an element at a node is made smaller. Both operations are known to have logarithmic cost in the worst case; *siftdown* performs at most $2 \lg n$ and *siftup* at most $\lg n$ element comparisons. Note that in *siftdown* and *siftup* we never move elements but whole nodes. This way the handles to nodes will always remain valid and *delete* operations can be executed without any problems.

In our data structure a fix is emulated by taking three perfect heaps of the same height h , determining which root stores the minimum element (breaking

ties arbitrarily), making this node the new root of a perfect heap of height $h + 1$, and making the roots of the other two perfect heaps the children of this new root. The old subtrees of the selected root become perfect heaps of height $h - 1$. That is, starting from three heaps of height h , one new heap of height $h + 1$ and two new heaps of height $h - 1$ are created; this is exactly corresponding to the digit changes resulting from a fix in the number system. After performing the fix on the heaps, the respective changes have to be made in the auxiliary structures (resizable array, jump pointers, and undo stack). The emulation of an unfix is a reverse of these actions. The necessary information indicating the position of the unfix is available at the undo stack. Compared to a fix, the only new ingredient is that, when the root of a perfect heap of height $h + 1$ is made the root of the two perfect heaps of height $h - 1$, *sift-down* is necessary due to the possible changes made in the perfect heaps between the fix and the corresponding unfix; otherwise, it cannot be guaranteed that the heap order is satisfied in the composed tree. Hence, a fix can be emulated at $O(1)$ worst-case cost, whereas an unfix has $O(\lg n)$ worst-case cost involving at most $2 \lg n$ element comparisons.

Because of the minimum pointer, *find-min* has $O(1)$ worst-case cost. In *insert*, a node that is a perfect heap of size 1 is first added to the collection. If the element in that node is smaller than the current minimum, the minimum pointer is updated to point to the new node. Additionally, if the added node creates a big digit, the list of big digits is updated accordingly. Thereafter, the other actions specified for an increment in the number system are emulated. The location of the desired fix can be easily determined by accessing the first in the list of big digits. The worst-case cost of *insert* is $O(1)$ and it may involve at most three element comparisons (one to compare the new element with the minimum and two when performing a fix).

When removing a node it is important that we avoid any interference with the number system and do not change the sizes of the heaps retained by the number system. Hence, we implement *delete* by borrowing a node and then using it to replace the deleted node in the associated perfect heap. This approach guarantees that the number system should only support decrements and unfixes. Now, *borrow* is performed by doing an unfix using the information available at the undo stack, and thereafter removing a perfect heap of size 1 from the data structure. Such a heap must always exist since a fix recorded in the undo stack was preceded by an increment. Due to the cost of the unfix, the worst-case cost of *borrow* is $O(\lg n)$ and it may involve at most $2 \lg n$ element comparisons.

By the aid of *borrow*, it is quite straightforward to implement *delete*. Assuming that the replacement node is different from the node to be deleted, the replacement is done, and *sift-down* or *sift-up* is executed depending on the value stored at the replacement node. Because of this process, the root of the underlying perfect heap may change. If this happens, we have to go through the resizable array pointing to the roots of the heaps and update the pointer in one of the entries to point to the new root instead of the old root. A deletion may also invalidate the minimum pointer, so we have to scan all roots to determine the current overall minimum and update the minimum pointer to point to this

root. The worst-case cost of all these operations is $O(\lg n)$. In total, the number of element comparisons performed is never larger than $6 \lg n$; *borrow* requires at most $2 \lg n$, *sift-down* (as well as *sift-up*) requires at most $2 \lg n$, and the scan over all roots requires at most $2 \lg n$ element comparisons.

Remark 9. As a consequence of Lemma 2, the number of perfect heaps in the priority queue is at most $2 \lg n$.

Remark 10. We can incorporate $O(\lg^2 m + \lg n)$ worst-case *meld*, where m and n are the number of elements in the two melded priority queues and $m \leq n$. In such case, *insert* will have $O(\lg n)$ worst-case cost. Following each *insert* or *meld*, the idea is to perform a fix on every three heaps having the same height until there are at most two heaps per height. Fixes can be performed in arbitrary order; neither the number of fixes nor the resulting representation are affected by the order in which the fixes are done.

To establish these worst-case bounds, we note that a fix on the highest index j , where $d_j > 2$, may propagate to the higher indexes at most $\lg n$ times. The worst-case bound on the cost of *insert* follows. To analyse *meld*, we distinguish between the fixes performed on the lower $\lfloor \lg m \rfloor$ indexes and those performed on the higher indexes. Since there are at most two heaps per height in each queue prior to each *meld*, the number of possible fixes on $d_{\lfloor \lg m \rfloor}$ is at most four (we leave the verification of this fact for the reader); each of these fixes may propagate forward resulting in at most $\lg n$ fixes on the higher indexes. Since the sum of the heights of the heaps corresponding to the lower $\lfloor \lg m \rfloor$ indexes is $O(\lg^2 m)$ and each fix on the lower indexes decreases this quantity by 1, there are $O(\lg^2 m)$ such fixes. The worst-case bound on the cost of *meld* follows.

By extracting the root of the smallest perfect heap and adding the subtrees of that node, if any, to the collection of perfect heaps, this data structure can support *borrow* at $O(1)$ worst-case cost. In accordance, *delete* can use this kind of borrowing instead.

For this implementation, the amortized costs are: $O(1)$ per *insert*, $O(\lg m)$ per *meld*, and $O(\lg n)$ per *borrow* and *delete*. To establish these bounds, we use a potential function that is the sum of the heights of the heaps currently in the priority queue. The key observation is that a fix decreases the potential by 1, *insert* increases it by 1, *borrow* and *delete* increase it by $O(\lg n)$, and *meld* does not change the total potential. Caveat, *meld* involves $O(\lg m)$ work since the perfect heaps have to be maintained in height order.

4 Conclusions

We gave a number system that efficiently supports increments. A disturbance in any of the digits may push the increments out of the orbit, resulting in an invalid representation. In order not to disturb this sensitive system, we implemented decrements as the reverse of increments. This undo-logging technique can be applied to other number systems as well. On the other hand, one may still ask whether there is another way of extending our system to incorporate efficient decrements without using an undo stack.

When applying the number system to implement a variant of Williams' binary heap, we obtained a priority queue that performs *insert* at $O(1)$ worst-case cost and *delete* at $O(\lg n)$ worst-case cost. This improves all earlier approaches that implement binary-heap variants. However, there are other ways of achieving the same bounds by using specialized data structures like binomial queues [3,4,7,8], or via general data-structural transformations [1]. Also, the price we pay for $O(1)$ worst-case *insert* is relatively high; we cannot support polylogarithmic *meld*, and the bound on the number of element comparisons performed by *delete* is increased from $2 \lg n$ to $6 \lg n$. It would be natural to ask whether our approach can be improved in any of these aspects.

References

1. Alstrup, S., Husfeld, T., Rauhe, T., Thorup, M.: Black box for constant-time insertion in priority queues. *ACM Transactions on Algorithms* 1(1), 102–106 (2005)
2. Bansal, S., Sreekanth, S., Gupta, P.: M-heap: A modified heap data structure. *International Journal of Foundations of Computer Science* 14(3), 491–502 (2003)
3. Brodal, G.S., Okasaki, C.: Optimal purely functional priority queues. *Journal of Functional Programming* 6(6), 839–857 (1996)
4. Carlsson, S., Munro, J.I., Poblete, P.V.: An implicit binomial queue with constant insertion time. In: Karlsson, R., Lingas, A. (eds.) *SWAT 1988*. LNCS, vol. 318, pp. 1–13. Springer, Heidelberg (1988)
5. Clancy, M.J., Knuth, D.E.: A programming and problem-solving seminar. Technical Report STAN-CS-77-606, Stanford University (1977)
6. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: *Introduction to Algorithms*, 3rd edn. MIT Press, Cambridge (2009)
7. Elmasry, A., Jensen, C., Katajainen, J.: Multipartite priority queues. *ACM Transactions on Algorithms* Article 14, 5(1) (2008)
8. Elmasry, A., Jensen, C., Katajainen, J.: Two-tier relaxed heaps. *Acta Informatica* 45(3), 193–210 (2008)
9. Gonnet, G.H., Munro, J.I.: Heaps on heaps. *SIAM Journal on Computing* 15(4), 964–971 (1986)
10. Myers, E.: An applicative random-access stack. *Information Processing Letters* 17, 241–248 (1983)
11. Harvey, N.J.A., Zatloukal, K.: The post-order heap. In: *Proceedings of the 3rd International Conference on Fun with Algorithms* (2004)
12. Okasaki, C.: *Purely Functional Data Structures*. Cambridge University Press, Cambridge (1998)
13. Sack, J.-R., Strothotte, T.: A characterization of heaps and its applications. *Information and Computation* 86(1), 69–86 (1990)
14. Vuillemin, J.: A data structure for manipulating priority queues. *Communications of the ACM* 21(4), 309–315 (1978)
15. Williams, J.W.J.: Algorithm 232: Heapsort. *Communications of the ACM* 7(6), 347–348 (1964)

Bit-(Parallelism)²: Getting to the Next Level of Parallelism

Domenico Cantone, Simone Faro, and Emanuele Giaquinta

Dipartimento di Matematica e Informatica, Università di Catania, Italy
{cantone,faro,giaquinta}@dmi.unict.it

Abstract. We investigate the problem of getting to a higher instruction-level parallelism in string matching algorithms. In particular, starting from an algorithm based on bit-parallelism, we propose two flexible approaches for boosting it with a higher level of parallelism. These approaches are general enough to be applied to other bit-parallel algorithms. It turns out that higher levels of parallelism lead to more efficient solutions in practical cases, as demonstrated by an extensive experimentation.

Keywords: string matching, bit parallelism, text processing, design and analysis of algorithms, instruction-level parallelism.

1 Introduction

Given a text t of length n and a pattern p of length m over some alphabet Σ of size σ , the *string matching problem* consists in finding *all* occurrences of the pattern p in the text t . This problem has been extensively studied in computer science because of its direct applications to such diverse areas as text, image and signal processing, speech analysis and recognition, information retrieval, computational biology and chemistry. String matching algorithms are also basic components in many software applications. Moreover, they emphasize programming methods that serve as paradigms in other fields of computer science. Finally, they also play an important rôle in theoretical computer science by providing challenging problems.

In this paper we focus on one of such engaging problems, namely the problem of enhancing the instruction-level parallelism of string matching algorithms.

The *instruction-level parallelism* (ILP) is a measure of how many operations in an algorithm can be performed simultaneously. Ordinary programs are typically written under a sequential execution model, in which instructions are executed one after the other and in the order specified by the programmer. ILP allows one to overlap the execution of multiple instructions or even to change the order in which instructions are executed. The extent to which ILP is present in programs heavily depends on the application. In certain fields, such as graphics and scientific computing, ILP is largely used. However, workloads such as cryptography exhibit much less parallelism.

Consider, for instance, the sequences of instructions shown in Fig. 1. In sequence **a**, operation a_3 depends on the results of operations a_1 and a_2 , and thus it

sequence a	sequence b	sequence c
a ₁ . $p[j] \leftarrow p[j] + 2$	b ₁ . $a \leftarrow b \leftarrow 0$	c ₁ . $j \leftarrow 0$
a ₂ . $h \leftarrow p[i] \times 3$	b ₂ . for $i = 1$ to n do	c ₂ . for $i = 1$ to n do
a ₃ . $p[i] \leftarrow p[j] + h$	b ₃ . $a \leftarrow a + (q[i] \times p[i])$	c ₃ . $j \leftarrow j + (p[i] \times (3000 + q[i]))$
	b ₄ . $b \leftarrow b + (3 \times p[i])$	c ₄ . $a \leftarrow j \bmod 1000$
		c ₅ . $b \leftarrow \lfloor j/1000 \rfloor$

Fig. 1. Three sequences of instructions

cannot be calculated until both operations are completed. However, operations a_1 and a_2 can be calculated simultaneously as they are independent of each other. If we assume that each operation can be completed in one time unit, then these three instructions can be completed in two time units, yielding an ILP of 3/2.

Two main techniques can be adopted to increase the ILP of a sequence of instructions: micro-architectural and software techniques.

Micro-architectural techniques used to exploit ILP include, among others, *instruction pipelining*, where the execution of multiple instructions can be partially overlapped, and *superscalar execution*, in which multiple execution units are used to execute multiple instructions in parallel. For instance, in **sequence b**, operation b_3 and b_4 are independent, so (if two different processors are available) they can be calculated in parallel for each iteration of the cycle in b_2 , thus halving the time needed for their execution.

Instead, software techniques are generally more challenging as they strongly depend on the processed data. Consider again **sequence b** shown in Fig. 1. Assuming that the sum $\sum_{i=1}^n p[i]q[i]$ is less than 1000, a smart programmer could modify the sequence in the form proposed in **sequence c**, achieving again an ILP of 2 while using a single processor.

Several string matching algorithms have been proposed to take advantage of micro-architectural techniques for increasing ILP (see for instance [6,7,10,5]). However, most of the work has been devoted to develop software techniques for ILP to simulate efficiently the parallel computation of nondeterministic finite automata (NFAs) related to the search pattern, whose number of states is about the pattern size (see for instance [2,8,9,3]). Such simulations can be done efficiently using the *bit-parallelism* technique, which consists in exploiting the intrinsic parallelism of the bit operations inside a computer word [2]. In some cases, bit-parallelism allows to reduce the overall number of operations up to a factor equal to the number of bits in a computer word. Thus, although string matching algorithms based on bit-parallelism are usually simple and have very low memory requirements, they generally work well with patterns of moderate length only.

When the pattern size is small enough, in favorable situations it becomes possible to carry on in parallel the simulation of multiple copies of a same NFA or of multiple distinct NFAs, thus getting to a second level of parallelism.

In this paper we illustrate this idea in the case of BNDM-like algorithms. More specifically, not satisfied with the degree of parallelism of the bit-parallel implementation of a variant of the Wide-Window algorithm, we present two different approaches which yield a better ILP if compared with the original algorithm.

The methods we present turn out to be quite flexible and, as such, can be applied to other bit-parallel algorithms as well¹

The rest of the paper is organized as follows. In Section 2 we introduce the notations and terminology used in the paper. In Section 3, we introduce the bit-parallel technique and describe some string matching algorithms based on it. Next, in Section 4, we present two techniques to enhance ILP and illustrate two new algorithms resulting from their application. Experimental data obtained by running the algorithms under various conditions are presented and compared in Section 5. Finally, we draw our conclusions in Section 6.

2 Notations and Terminology

Throughout the paper we will make use of the following notations and terminology. A string p of length $m \geq 0$ is represented as a finite array $p[0..m-1]$ of characters from a finite alphabet Σ of size σ (in particular, for $m = 0$ we obtain the empty string, also denoted by ε). Thus $p[i]$ will denote the $(i+1)$ -st character of p , for $0 \leq i < m$, and $p[i..j]$ will denote the *factor* or *substring* of p contained between the $(i+1)$ -st and the $(j+1)$ -st characters of p , for $0 \leq i \leq j < m$. A factor of the form $p[0..i]$, also written p_i , is called a *prefix* of p and a factor of the form $p[i..m-1]$ is called a *suffix* of p for $0 \leq i \leq m-1$. We write $Suff(p)$ for the collection of all suffixes of p . In addition, we write $p.p'$ or, more simply, pp' to denote the concatenation of the strings p and p' . Finally, we denote the reverse of the string p by \bar{p} , i.e. $\bar{p} = p[m-1]p[m-2] \dots p[0]$.

The *nondeterministic suffix automaton* with ε -transition $NSA(p) = (Q, \Sigma, \delta, I, F)$ for the language $Suff(P)$ of the suffixes of p is defined as follows:

- $Q = \{I, q_0, q_1, \dots, q_m\}$ (I is the initial state)
- $F = \{q_m\}$ (F is the set of final states)
- the transition function $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow Pow(Q)$ is such that $\delta(q, c) = \{q_{i+1}\}$, if $q = q_i$ and $c = p[i]$ ($0 \leq i < m$); $\delta(q, c) = \{q_0, q_1, \dots, q_m\}$, if $q = I$ and $c = \varepsilon$; and $\delta(q, c) = \emptyset$, otherwise.

We will also make use of the following C-like notations to represent some bitwise operations. In particular, “|” represents the bitwise Or; “&” denotes the bitwise And; “~” represents the one’s complement; “>>” and “<<” denote respectively the bitwise right shift and the bitwise left shift.

3 Bit-Parallelism: Starting from the First Level

Bit-parallelism exploits the intrinsic parallelism of bit operations inside computer words, allowing in favorable cases to cut down the overall number of operations up to the number of bits in a computer word.

¹ A similar technique has been exploited in [4] to boost the approximate search of a pattern, under the edit distance; the algorithm proposed in [4] performs a left to right scan of the text while processing $\lfloor w/m \rfloor$ text segments simultaneously (w is the word size in bits). We thank an anonymous referee for pointing this out to us.

Among the several algorithms based on bit-parallelism which have been developed over the years, the **Backward-Nondeterministic-DAWG-Matching** algorithm (BNDM, for short) deserves particular attention as it has inspired various other algorithms and is still considered one of the fastest algorithms based on bit-parallelism [8].

The BNDM algorithm uses bit-parallelism to simulate the nondeterministic suffix automaton $NSA(\bar{p})$ for the reverse of the search pattern p .

To this purpose, the states of the automaton (except states I and q_0) are put in a one-one correspondence with the bits of a bit mask D , having size equal to the length L of the pattern p .² In this context, in any automaton configuration bits corresponding to active states are set to 1, while bits corresponding to inactive states are set to 0. Additionally, for each character c of the alphabet Σ , the algorithm maintains a bit mask $B[c]$ whose i -th bit is set to 1 iff $p[i] = c$. Thus, the array B requires $|\Sigma| \cdot L$ bits.

The BNDM algorithm works by shifting a window of length m over the text. For each window alignment, it searches the pattern by scanning the current window backwards, updating the automaton configuration for each character read. Whenever bit $(m - 1)$ of D is set, a suffix of \bar{p} (i.e., a prefix of p) has been found and the current position is recorded. A search ends when either D becomes zero or the algorithm has performed m iterations. The window is then shifted to the start position of the longest recognized proper suffix of \bar{p} .

Because of the ϵ -closure of the initial state I , at the beginning of any search all states are active, i.e. $D = 1^m$. A transition on character c is implemented (with the exception of the first one³) as follows:

$$D \leftarrow (D \ll 1) \ \& \ B[c].$$

The BNDM algorithm scales in function of the number $\lceil m/\omega \rceil$ of words needed to represent each of D and $B[c]$, for $c \in \Sigma$, where ω is the size of the computer word in bits. Its worst case time complexity is $\mathcal{O}(nm \lceil m/\omega \rceil)$, though in practice exhibits a sublinear behavior.

Several variants of the BNDM algorithm have been proposed over the years. In what follows we briefly describe an efficient variant, the **Wide-Window** algorithm (WW, for short) [3], which is particularly suitable for our purposes. However, we slightly depart from its original version so as to make the algorithm parallelizable in ways that will be explained in the next section.

3.1 The Wide-Window Algorithm

Let p be a pattern of length m and let t be a text of length n . The WW algorithm locates $\lfloor n/m \rfloor$ *attempt positions* in t , namely positions $j = km - 1$, for $k = 1, \dots, \lfloor n/m \rfloor$. For each such position j , the pattern p is searched for in the *attempt window* of size $2m - 1$ centered at j , i.e. in the substring $t[j - m + 1 .. j + m - 1]$. Each of such search phases is divided into two steps.

² Note that if $L \leq \omega$ the entire bit mask D fits in a single computer word, otherwise $\lceil L/\omega \rceil$ computer words are needed to represent it.

³ The first transition is simply encoded as $D \leftarrow D \ \& \ B[c]$.

In the first step, the right side of the attempt window, consisting of the last m characters, is scanned from left to right with the automaton $NSA(p)$. In this step, the start positions (in p) of the suffixes of p aligned with position j in t are collected in a set

$$\mathcal{S}_j = \{0 \leq i < m \mid p[i..m-1] = t[j..j+m-1-i]\}.$$

In the second step, the left half of the attempt window, consisting of the first m symbols, is scanned from right to left with the automaton $NSA(\bar{p})$. During this step, the end positions (in p) of the prefixes of p aligned with position j in t are collected in a set

$$\mathcal{P}_j = \{0 \leq i < m \mid p[0..i] = t[j-i..j]\}.$$

Taking advantage of the fact that an occurrence of p is located at position $(j - k)$ of t if and only if $k \in \mathcal{S}_j \cap \mathcal{P}_j$, for $k = 0, \dots, m - 1$, the number of all the occurrences of p in the attempt window centered at j is readily given by the cardinality $|\mathcal{S}_j \cap \mathcal{P}_j|$.

Fig. 2(A) shows a simple schematization of the structure of an iteration of the WW algorithm at a given position j in t . The two sequential phases are represented by the arrows labeled 1 and 2, respectively.

It is straightforward to devise a bit-parallel implementation of the WW algorithm. The sets \mathcal{P} and \mathcal{S} can be encoded by two bit masks P and S , respectively. The nondeterministic automata $NSA(p)$ and $NSA(\bar{p})$ are then used for searching the suffixes and prefixes of p on the right and on the left parts of the window, respectively. Both automata state configurations and final state configuration can be encoded by the bit masks D and $M = (1 \ll (m - 1))$, so that $(D \& M) \neq 0$ will mean that a suffix or a prefix of the search pattern p has been found, depending on whether D is encoding a state configuration of the automaton $NSA(p)$ or of the automaton $NSA(\bar{p})$. Whenever a suffix (resp., a prefix) of length $(\ell + 1)$ is found (with $\ell = 0, 1, \dots, m - 1$), the bit $S[m - 1 - \ell]$ (resp., the bit $P[\ell]$) is set by one of the following bitwise operations:

$$\begin{aligned} S &\leftarrow S \mid ((D \& M) \gg \ell) && \text{(in the suffix case)} \\ P &\leftarrow P \mid ((D \& M) \gg (m - 1 - \ell)) && \text{(in the prefix case)}. \end{aligned}$$

If we are only interested in counting the number of occurrences of p in t , we can just count the number of bits set in $(S \& P)$. This can be done in $\log_2(\omega)$ operations by using a *population count* function, where ω is the size of the computer word in bits (see [1]). Otherwise, if we want also to retrieve the matching positions of p in t , we can iterate over the bits set in $(S \& P)$ by repeatedly computing the index of the highest bit set and then masking it. The function that computes the highest bit set of a register x is $\lfloor \log_2(x) \rfloor$, and can be implemented efficiently in either a machine dependent or machine independent way (see again [1]).

The resulting algorithm based on bit parallelism is named **Bit-Parallel Wide-Window algorithm** (B_pW_w , for short). It needs $\lceil m/\omega \rceil$ words to represent the bit

masks D , S , P , and $B[c]$, for $c \in \Sigma$. The worst case time complexity of the B_pW_w algorithm is $\mathcal{O}(n\lceil m/\omega \rceil + \lfloor n/m \rfloor \log_2(\omega))$.

Additionally, we observe that the B_pW_w algorithm can be easily modified so as to work on windows of size $2m$. For the sake of clarity, we have just discussed a simpler but slightly less efficient variant.

4 Bit-(Parallelism)²: Getting to the Second Level

In this section we present two different approaches which lead to a higher level of parallelism. By way of demonstration we apply them to a bit-parallel version of the **Wide-Window** algorithm,⁴ but our approaches can be applied as well to other (more efficient) solutions based on bit-parallelism.

The two approaches can be summarized as follows:

- **first approach:** if the algorithm searches for the pattern in fixed-size text windows then, at each attempt, process simultaneously two (adjacent or partially overlapping) text windows by using in parallel two copies of a same automaton;
- **second approach:** if each search attempt of the algorithm can be divided into two steps (which possibly make use of two different automata) then execute simultaneously the two steps, by running the two automata in parallel.

Both variants use the **SIMD** (Single Instruction Multiple Data) paradigm. This approach, on which vectorial instructions sets like **MMX** and **SSE** are based, consists in executing the same instructions on multiple data in a parallel way. Typically, a register of size ω is logically divided into i blocks of k bits which are then updated simultaneously.

In both variants of the B_pW_w algorithm, we divide a word of ω bits into *two* blocks, each being used to encode a suffix automaton. Thus, the maximum length of the pattern gets restricted to $\lfloor \omega/2 \rfloor$. We denote with B the array of bit masks encoding the suffix automaton $NSA(p)$ and with C the array of bit masks encoding the suffix automaton $NSA(\bar{p})$.

4.1 The Bit-Parallel (Wide-Window)² Algorithm

In the first variant, named **Bit-Parallel (Wide-Window)²** ($B_pW_w^2$, for short), two partially overlapping windows in t , each of size $2m - 1$, centered at consecutive attempt positions $j - m$ and j , are processed simultaneously. For the parallel simulation two automata are represented in a single word and updated in parallel.

Specifically, each search phase is again divided into two steps. During the first step, two copies of $NSA(p)$ are operated in parallel to compute simultaneously the sets \mathcal{S}_{j-m} and \mathcal{S}_j (lines 13-18). Likewise, in the second step, two copies of $NSA(\bar{p})$ are operated in parallel to compute the sets \mathcal{P}_{j-m} and \mathcal{P}_j (lines 20-25).

⁴ We chose the **Wide-Window** algorithm in our case study since its structure makes its parallelization simpler.

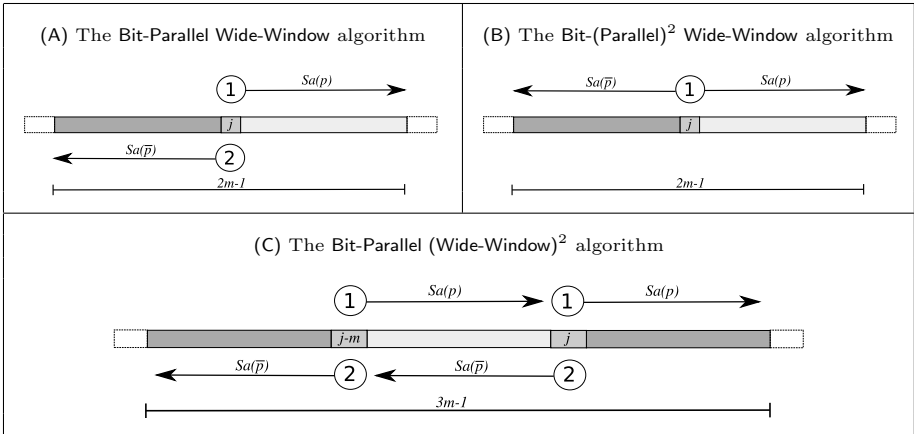


Fig. 2. Structure of a searching iteration at a given position j in the text t of (A) the B_pW_w algorithm, (B) the $B_p^2W_w$ algorithm, and (C) the $B_pW_w^2$ algorithm

To represent the automata with a single word, the bit masks D , M , S , and P are logically divided into two blocks, each of $k = \omega/2$ bits.

During the first step, the most significant k bits of D encode the state of the suffix automaton $NSA(p)$ that scan the attempt window centered at $j - m$. Similarly, the least significant k bits of D encode the state of the suffix automaton $NSA(p)$ that scans the attempt window centered at j . An analogous encoding is used in the second step, but with the automaton $NSA(\bar{p})$ in place of $NSA(p)$. Fig. 2(C) schematizes the structure of a search iteration of the $B_pW_w^2$ algorithm, at given attempt positions $j - m$ and j of t .

The most significant k bits of the bit mask S (resp., P) encode the set \mathcal{S}_{j-m} (resp., \mathcal{P}_{j-m}), while the least significant k bits encode the set \mathcal{S}_j (resp., \mathcal{P}_j). Thus, to properly detect suffixes in both windows, the bit mask M is initialized (lines 8-9) with the value

$$M \leftarrow (1 \ll (m + k - 1)) | (1 \ll (m - 1))$$

and transitions of the automata are performed in parallel with the following bitwise operations (lines 14-15 and lines 21-22)

$$\begin{aligned} D &\leftarrow (D \ll 1) \& ((B[t[j - m + \ell]] \ll k) | B[t[j + \ell]]) && \text{(in the first phase)} \\ D &\leftarrow (D \ll 1) \& ((C[t[j - m - \ell]] \ll k) | C[t[j - \ell]]) && \text{(in the second phase),} \end{aligned}$$

for $\ell = 1, \dots, m - 1$ (when $\ell = 0$, the left shift of D does not take place).

The remaining bitwise operations are left unchanged, as the automata configurations are updated using the same instructions. Since two windows are simultaneously scanned at each search iteration, the shift becomes $2m$, therefore doubling the length of the shift with respect to the WW algorithm. The pseudocode of the algorithm $B_pW_w^2$ is reported in Fig. 3 (on the left).

4.2 The Bit-(Parallel)² Wide-Window Algorithm

The second variant of the WW algorithm which we present next is called Bit-(Parallel)² Wide-Window algorithm ($B_p^2W_w$, for short). The idea behind it consists in processing a single window at each attempt (as in the original WW algorithm) but this time by scanning its left and right sides simultaneously. Fig. 2(B) schematizes the structure of a searching iteration of the $B_p^2W_w$ algorithm, while Fig. 3 (on the right) shows the pseudocode of the $B_p^2W_w$ algorithm.

<p>Bit-Parallel (Wide-Window)² (p, m, t, n)</p> <ol style="list-style-type: none"> 1. $count \leftarrow 0$ 2. $k \leftarrow \omega/2$ 3. for $c \in \Sigma$ do $B[c] \leftarrow 0$ 4. for $c \in \Sigma$ do $C[c] \leftarrow 0$ 5. for $i \leftarrow 0$ to $m - 1$ do 6. $B[p[i]] \leftarrow B[p[i]] \mid (1 \ll i)$ 7. $C[p[m - 1 - i]] \leftarrow C[p[m - 1 - i]] \mid (1 \ll i)$ 8. $H \leftarrow 1 \ll (m - 1)$ 9. $M \leftarrow (H \ll k) \mid H$ 10. $j \leftarrow 2m - 1$ 11. while $j < n - m$ do 12. $D \leftarrow \sim 0, l \leftarrow 0, S \leftarrow 0$ 13. while $D \neq 0$ do 14. $H \leftarrow (B[t[j - m + l]] \ll k) \mid B[t[j + l]]$ 15. $D \leftarrow D \& H$ 16. $S \leftarrow S \mid ((D \& M) \gg l)$ 17. $D \leftarrow D \ll 1$ 18. $l \leftarrow l + 1$ 19. $D \leftarrow \sim 0, l \leftarrow 0, P \leftarrow 0$ 20. while $D \neq 0$ do 21. $H \leftarrow (C[t[j - m - l]] \ll k) \mid C[t[j - l]]$ 22. $D \leftarrow D \& H$ 23. $P \leftarrow P \mid ((D \& M) \gg (m - 1 - l))$ 24. $D \leftarrow D \ll 1$ 25. $l \leftarrow l + 1$ 26. $count \leftarrow count + \text{popcount}(P \& S)$ 27. $j \leftarrow j + 2m$ 	<p>Bit-(Parallel)² Wide-Window (p, m, t, n)</p> <ol style="list-style-type: none"> 1. $count \leftarrow 0$ 2. $k \leftarrow \omega/2$ 3. for $c \in \Sigma$ do $B[c] \leftarrow 0$ 4. for $c \in \Sigma$ do $C[c] \leftarrow 0$ 5. for $i \leftarrow 0$ to $m - 1$ do 6. $B[p[i]] \leftarrow B[p[i]] \mid (1 \ll (k + i))$ 7. $C[p[m - 1 - i]] \leftarrow C[p[m - 1 - i]] \mid (1 \ll i)$ 8. $H \leftarrow 1 \ll (m - 1)$ 9. $M \leftarrow (H \ll k) \mid H$ 10. $j \leftarrow m - 1$ 11. while $j < n - m$ do 12. $D \leftarrow \sim 0, l \leftarrow 0, PS \leftarrow 0$ 13. while $D \neq 0$ do 14. $H \leftarrow C[t[j - l]] \mid B[t[j + l]]$ 15. $D \leftarrow D \& H$ 16. $PS \leftarrow PS \mid ((D \& M) \gg l)$ 17. $D \leftarrow D \ll 1$ 18. $l \leftarrow l + 1$ 19. $P \leftarrow \text{reverse}(PS) \gg (\omega - m)$ 20. $S \leftarrow PS \gg k$ 21. $count \leftarrow count + \text{popcount}(P \& S)$ 22. $j \leftarrow j + m$
---	--

Fig. 3. The Bit-Parallel (Wide-Window)² algorithm (on the left) and the Bit-(Parallel)² Wide-Window algorithm (on the right) for the exact string matching problem

As above, let p be a pattern of length m , and t be a text of length n . The bit masks B and C which are used to perform the transitions on both automata $NSA(p)$ and $NSA(\bar{p})$ are computed as in the B_pW_w algorithm (lines 3-7).

Automata state configurations are again encoded simultaneously in a same bit mask D . Specifically, the most significant k bits of D encode the state of the suffix automaton $NSA(p)$, while the least significant k bits of D encode the state of the suffix automaton $NSA(\bar{p})$. The $B_p^2W_w$ algorithm uses the following bitwise operations to perform transitions⁵ of both automata in parallel (lines 14-15,17):

$$D \leftarrow (D \ll 1) \& ((B[t[j + \ell]] \ll k) \mid C[t[j - \ell]]),$$

for $\ell = 1, \dots, m - 1$. Note that in this case the left shift of k positions can be precomputed in B by setting $B[c] \leftarrow B[c] \ll k$, for each $c \in \Sigma$.

⁵ For $\ell = 0$, D is simply updated by $D \leftarrow D \& ((B[t[j + l]] \ll k) \mid C[t[j - l]])$.

Using the same representation, the final-states bit mask M is initialized as

$$M \leftarrow (1 \ll (m + k - 1)) | (1 \ll (m - 1)) \quad (\text{lines 8-9}).$$

At each iteration around an attempt position j of t , the sets \mathcal{S}_j and \mathcal{P}_j^* are computed, where \mathcal{S}_j is defined as in the case of the B_pW_w algorithm, and \mathcal{P}_j^* is defined as $\mathcal{P}_j^* = \{0 \leq i < m \mid p[0..m-1-i] = t[j-(m-1-i)..j]\}$, so that $\mathcal{P}_j = \{0 \leq i < m \mid (m-1-i) \in \mathcal{P}_j^*\}$.

The sets \mathcal{S}_j and \mathcal{P}_j^* can be encoded with a single bit mask PS , in the rightmost and the leftmost k bits, respectively. Positions in \mathcal{S}_j and \mathcal{P}_j^* are then updated simultaneously in PS by executing the following operation (line 16):

$$PS \leftarrow PS | ((D \& M) \ggg l).$$

At the end of each iteration, the bit masks S and P are retrieved from PS with the following bitwise operations (lines 19-20):

$$P \leftarrow \text{reverse}(PS) \ggg (\omega - m), \quad S \leftarrow PS \ggg k,$$

where `reverse` denotes the bit-reversal function, which satisfies $\text{reverse}(x)[i] = x[\omega - 1 - i]$, for $i = 0, \dots, \omega - 1$ and any bit mask x . In fact, to obtain the correct value of P we used bit-reversal modulo m , which has been easily achieved by right shifting $\text{reverse}(PS)$ by $(\omega - m)$ positions. We recall that the `reverse` function can be implemented efficiently with $\mathcal{O}(\log_2(\omega))$ operations (see [11]).

5 Experimental Results

We present next the results of an extensive experimental comparison of our proposed variants $B_p^2W_w$ and $B_pW_w^2$ with the B_pW_w and $BNDM$ algorithms. In particular, we have tested two different implementations of the $B_p^2W_w$ and $B_pW_w^2$ algorithms, characterized by a different implementation of the *population-count* function. One implementation uses the builtin version of the GNU C compiler (algorithms $B_p^2W_w$ and $B_pW_w^2$), while the second implementation uses the population-count function described in [11] (algorithms $B_p^2W_w^{bc}$ and $B_pW_w^{2bc}$). Thus, we compared the following string matching algorithms, in terms of running time:

- the Bit-Parallel Wide-Window algorithm (B_pW_w)
- the Bit-(Parallel)² Wide-Window algorithm ($B_p^2W_w$)
- the Bit-(Parallel)² Wide-Window algorithm with bit-count ($B_p^2W_w^{bc}$)
- the Bit-Parallel (Wide-Window)² algorithm ($B_pW_w^2$)
- the Bit-Parallel (Wide-Window)² algorithm with bit-count ($B_pW_w^{2bc}$)
- the Backward-Nondeterministic-DAWG-Matching algorithm ($BNDM$).

All algorithms have been implemented in the C programming language and tested on a PC with Intel Core2 processor of 1.66GHz running Linux and with a 32 bit word. In particular, all algorithms have been tested on seven $\text{Rand}\sigma$ problems, for $\sigma = 2, 4, 8, 16, 32, 64, 128$, where a $\text{Rand}\sigma$ problem consists of searching a set of 400 random patterns of a given length in a 5Mb random text over a common alphabet of size σ , with a uniform character distribution.

Only short patterns of length $m = 2, 4, 6, 8, 10, 12, 14, 16$ have been considered in our tests, since the bit size of a word was 32 in our case. However, the same

approach could be applied with 64-bit processors or using Intel Processors with SSE instructions on 128 bit registers, to process patterns up to a length of 32 and of 64, respectively. Moreover, we observe that $\lceil 2m/\omega \rceil$ different words could be used for representing our suffix automata in case of longer patterns, overcoming the bound on the value of m , though at the price of an increased running time.

In the following tables, running times are expressed in hundredths of seconds. The best results among all bit-parallel WW variants have been boldfaced and underlined. Additionally, running times relative to the BNDM algorithm have been boldfaced and underlined when the BNDM algorithm outperforms the other algorithms.

m	2	4	6	8	10	12	14	16
$B_p W_w$	816.50	634.00	521.25	442.50	381.50	334.25	300.75	271.00
$B_p^2 W_w$	775.00	481.50	<u>332.25</u>	<u>255.50</u>	<u>211.75</u>	<u>183.00</u>	<u>162.25</u>	<u>146.00</u>
$B_p^2 W_w^{bc}$	<u>642.25</u>	<u>455.50</u>	353.50	287.50	243.00	210.75	186.50	167.25
$B_p W_w^2$	905.75	700.75	554.50	455.25	383.00	331.50	291.75	268.50
$B_p W_w^{2bc}$	659.75	566.75	478.50	404.50	349.25	306.75	274.50	256.50
BNDM	690.75	545.75	410.50	308.00	244.50	201.50	172.00	150.00

Results for a Rand2 problem

m	2	4	6	8	10	12	14	16
$B_p W_w$	615.75	381.25	278.00	221.50	185.50	159.25	140.25	125.50
$B_p^2 W_w$	527.00	<u>311.75</u>	<u>232.75</u>	<u>186.75</u>	<u>155.75</u>	<u>134.00</u>	<u>118.25</u>	<u>106.25</u>
$B_p^2 W_w^{bc}$	<u>503.50</u>	342.75	260.00	207.50	172.75	148.25	130.50	116.50
$B_p W_w^2$	645.00	386.25	279.25	219.50	179.75	152.25	132.75	119.25
$B_p W_w^{2bc}$	557.75	375.50	276.25	217.25	180.00	154.00	135.50	122.25
BNDM	555.50	312.00	<u>215.75</u>	<u>166.25</u>	<u>137.00</u>	<u>117.00</u>	<u>102.00</u>	<u>90.50</u>

Results for a Rand4 problem

m	2	4	6	8	10	12	14	16
$B_p W_w$	422.75	273.50	199.75	153.75	123.75	103.50	89.25	79.00
$B_p^2 W_w$	<u>375.75</u>	<u>234.50</u>	173.25	137.50	115.25	100.50	89.50	80.75
$B_p^2 W_w^{bc}$	378.25	252.25	186.00	147.50	123.25	107.25	95.25	86.00
$B_p W_w^2$	407.75	239.00	<u>164.75</u>	<u>126.25</u>	<u>103.25</u>	<u>89.00</u>	<u>78.25</u>	<u>70.75</u>
$B_p W_w^{2bc}$	398.50	249.25	172.25	131.00	107.50	92.00	81.50	73.25
BNDM	<u>369.25</u>	236.50	167.75	<u>124.50</u>	<u>99.25</u>	<u>81.25</u>	<u>69.75</u>	<u>61.00</u>

Results for a Rand8 problem

m	2	4	6	8	10	12	14	16
$B_p W_w$	353.75	204.00	154.75	127.50	108.25	92.75	80.75	69.50
$B_p^2 W_w$	258.25	185.00	145.00	114.25	94.00	78.75	67.75	59.00
$B_p^2 W_w^{bc}$	<u>265.00</u>	193.50	151.50	118.50	98.25	82.00	71.00	62.00
$B_p W_w^2$	278.50	<u>173.50</u>	<u>129.00</u>	<u>99.25</u>	<u>79.75</u>	<u>65.75</u>	<u>56.25</u>	<u>48.50</u>
$B_p W_w^{2bc}$	282.75	181.25	134.25	103.25	82.75	68.25	58.00	50.25
BNDM	<u>265.00</u>	<u>169.0</u>	132.00	108.50	91.00	77.00	66.00	57.25

Results for a Rand16 problem

m	2	4	6	8	10	12	14	16
B_pW_w	273.75	160.75	119.25	98.25	85.00	76.00	68.50	62.75
$B_p^2W_w$	<u>191.50</u>	136.50	114.75	95.75	81.50	71.50	63.00	56.00
$B_p^2W_w^{bc}$	197.00	140.50	119.00	98.00	84.50	73.50	64.25	56.75
$B_pW_w^2$	215.00	<u>129.00</u>	<u>98.75</u>	<u>82.00</u>	<u>70.50</u>	<u>61.00</u>	<u>53.50</u>	<u>46.25</u>
$B_pW_w^{2bc}$	219.50	132.50	101.25	83.25	72.00	62.50	54.75	47.50
BNDM	218.75	<u>128.25</u>	<u>98.00</u>	<u>82.00</u>	72.00	64.50	58.25	52.75

Results for a Rand32 problem

m	2	4	6	8	10	12	14	16
B_pW_w	250.00	136.75	98.50	79.50	67.50	59.50	53.75	49.25
$B_p^2W_w$	<u>160.00</u>	107.00	89.00	75.00	65.25	59.75	54.00	50.25
$B_p^2W_w^{bc}$	162.75	111.50	91.50	75.00	67.50	60.50	54.25	50.00
$B_pW_w^2$	184.25	<u>104.25</u>	<u>77.00</u>	<u>63.25</u>	<u>55.25</u>	<u>49.50</u>	<u>45.00</u>	<u>41.25</u>
$B_pW_w^{2bc}$	186.00	105.75	78.25	64.25	55.75	49.75	45.50	41.75
BNDM	197.50	109.00	79.25	64.25	<u>55.25</u>	<u>49.50</u>	<u>45.00</u>	41.75

Results for a Rand64 problem

m	2	4	6	8	10	12	14	16
B_pW_w	238.50	126.25	88.25	69.25	58.00	50.00	44.50	40.25
$B_p^2W_w$	<u>145.75</u>	92.75	73.50	61.50	53.00	49.00	43.50	41.50
$B_p^2W_w^{bc}$	148.00	96.00	76.00	61.50	54.75	49.00	43.00	41.00
$B_pW_w^2$	168.25	<u>91.25</u>	<u>65.25</u>	<u>52.50</u>	<u>44.25</u>	<u>39.25</u>	<u>36.00</u>	<u>33.00</u>
$B_pW_w^{2bc}$	169.00	92.00	65.75	<u>52.50</u>	45.00	39.75	<u>36.00</u>	<u>33.00</u>
BNDM	187.25	99.50	70.25	55.50	46.75	40.75	36.50	33.50

Results for a Rand128 problem

The above experimental results show that the algorithms obtained by applying a second level of parallelism perform always better than the original B_pW_w algorithm. The gap is more evident in the case of short patterns or small alphabets. In particular the $B_p^2W_w$ algorithm achieves its best performances with small alphabets, while the $B_pW_w^2$ algorithm turns out to be the best choice for patterns with a length greater than 4.

The BNDM algorithm obtains the best results in some cases and it performs always better than the B_pW_w algorithm. It is interesting to observe that the BNDM algorithm is outperformed by the $B_p^2W_w$ algorithm when the alphabet is small and by the $B_pW_w^2$ algorithm in the case of large alphabets.

6 Conclusions

We have presented two variants of the Bit-Parallel Wide-Window algorithm which use a second level of parallelization inspired by the *Single Instruction Multiple Data* paradigm. While the $B_p^2W_w$ variant is quite entangled to the original algorithm, as it uses two different automata in parallel, the other one ($B_pW_w^2$) is quite general and much the same approach can possibly be applied to other bit-parallel algorithms. As the experimental results show, this technique provides

a non negligible speedup; this is particularly true for the second variant as it allows to double the size of window shifts.

References

1. Arndt, J.: *Matters Computational* (2009), <http://www.jjj.de/fxt/>
2. Baeza-Yates, R., Gonnet, G.H.: A new approach to text searching. *Commun. ACM* 35(10), 74–82 (1992)
3. He, L., Fang, B., Sui, J.: The wide window string matching algorithm. *Theor. Comput. Sci.* 332(1-3), 391–404 (2005)
4. Hyyrö, H., Fredriksson, K., Navarro, G.: Increased bit-parallelism for approximate and multiple string matching. *J. Exp. Algorithmics* 10(2.6), 1–27 (2005)
5. Kaplan, K., Burge III, L.L., Garuba, M.: A parallel algorithm for approximate string matching. In: Arabnia, H.R., Mun, Y. (eds.) *Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2003*, pp. 1844–1848 (2003)
6. Lin, W., Liu, B.: Pipelined parallel AC-based approach for multi-string matching. In: *Proc. 14th International Conference on Parallel and Distributed Systems (ICPADS 2008)*, pp. 665–672 (2008)
7. Michailidis, P.D., Margaritis, K.G.: A programmable array processor architecture for flexible approximate string matching algorithms. *J. Parallel Distrib. Comput.* 67(2), 131–141 (2007)
8. Navarro, G., Raffinot, M.: A bit-parallel approach to suffix automata: Fast extended string matching. In: Farach-Colton, M. (ed.) *CPM 1998*. LNCS, vol. 1448, pp. 14–33. Springer, Heidelberg (1998)
9. Peltola, H., Tarhio, J.: Alternative algorithms for bit-parallel string matching. In: Nascimento, M.A., de Moura, E.S., Oliveira, A.L. (eds.) *SPIRE 2003*. LNCS, vol. 2857, pp. 80–94. Springer, Heidelberg (2003)
10. Semé, D., Youlou, S.: Parallel algorithms for string matching problems on a linear array with reconfigurable pipelined bus system. In: Oudshoorn, M.J., Rajasekaran, S. (eds.) *Proc. of the ISCA 18th International Conference on Parallel and Distributed Computing Systems*, pp. 55–60 (2005)

An Algorithmic Analysis of the Honey-Bee Game^{*}

Rudolf Fleischer¹ and Gerhard J. Woeginger²

¹ School of Computer Science, IPL, Fudan University, Shanghai 200433, China
rudolf@fudan.edu.cn

² Department of Mathematics and Computer Science, TU Eindhoven, 5600 MB
Eindhoven, The Netherlands
gwoegi@win.tue.nl

Abstract. The HONEY-BEE game is a two-player board game that is played on a connected hexagonal colored grid, or in a generalized setting, on a connected graph with colored nodes. In a single move, a player calls a color and thereby conquers all nodes of that color that are adjacent to his own territory. Both players want to conquer the majority of the nodes. We show that winning the game is PSPACE-hard in general, NP-hard on series-parallel graphs, but easy on outerplanar graphs. The solitaire version, where the goal is to conquer the entire graph with a minimum number of moves, is NP-hard on trees and split graphs, but can be solved in polynomial time on co-comparability graphs.

1 Introduction

The HONEY-BEE game is a popular two-player board game that shows up in many different variants and at many different places on the web (the game is best played on a computer); for a playable version we refer the reader for instance to Axel Born's web-page [1], see Fig. 1 for a screenshot. The playing field in HONEY-BEE is a connected grid of hexagonal honey-comb cells that come in various colors. At the beginning, each player controls a single cell in some corner of the playing field. Usually, the playing area is symmetric and the two players face each other from symmetrically opposite start cells. In every move a player may call a color c and thereby gain control over all connected regions of color c that have a common border with the area already under his control. The only restriction on c is that it cannot be one of the two colors used by the two players in their most recent move before the current move, respectively. A player

^{*} RF acknowledges support by the National Natural Science Foundation of China (No. 60973026), the Shanghai Leading Academic Discipline Project (project number B114), the Shanghai Committee of Science and Technology of China (09DZ2272800), and the Robert Bosch Foundation (Science Bridge China 32.5.8003.0040.0). GJW acknowledges support by the Netherlands Organisation for Scientific Research (NWO grant 639.033.403), and by BSIK grant 03018 (BRICKS: Basic Research in Informatics for Creating the Knowledge Society).

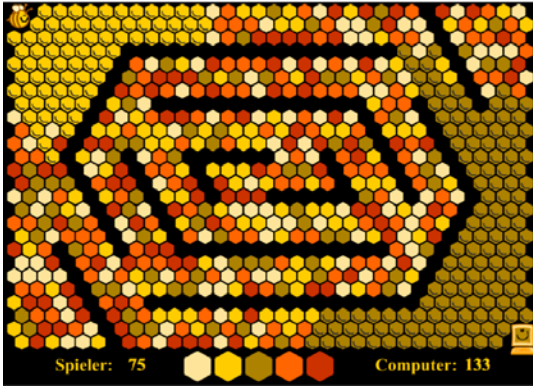


Fig. 1. Born’s Biene game. The human started at the top-left, the computer at the bottom-right corner.

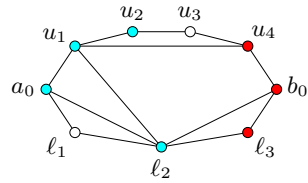


Fig. 2. An outerplanar graph. A has conquered the cyan nodes, B the red nodes. We have $U = 2$ and $L = 2$, and B cannot reach ℓ_1 .

wins when he controls the majority of all cells. On Born’s web-page [1] one can play against a computer, choosing between four different layouts for the playing field. The computer uses a simple greedy strategy: “Always call the color c that maximizes the immediate gain.” This strategy is not strong, and an alert human player usually beats the computer after a few practice matches.

In this paper we study the complexity of finding winning strategies for the HONEY-BEE game when played on arbitrary connected graphs instead of the hex-grid of the original game. We will show in Section 4 that the game is NP-hard even on series-parallel graphs, and that it is PSPACE-complete in general. On outerplanar graphs, however, it is easy to compute a winning strategy.

In the *solitaire* (single-player) version of HONEY-BEE the goal is to conquer the entire playing field as quickly as possible. Intuitively, a good strategy for the solitaire game will usually be a strong heuristic for the two-player game. For the solitaire version, our results draw a sharp separation line between easy and difficult cases. In particular, we show in Section 3 that HONEY-BEE-SOLITAIRE is NP-hard for split graphs and for trees (and thus also for bipartite, chordal, and perfect graphs), but polynomial-time solvable on co-comparability graphs (which include interval graphs and permutation graphs).

2 Definitions

We model HONEY-BEE in the following graph-theoretic setting. The playing field is a connected, simple, loopless, undirected graph $G = (V, E)$. There is a set C of k colors, and every node $v \in V$ is colored by some color $col(v) \in C$; we stress that this coloring does not need to be proper, that is, there may be edges $[u, v] \in E$ with $col(u) = col(v)$. For a color $c \in C$, the subset $V_c \subseteq V$ contains the nodes of color c . For a node $v \in V$ and a color $c \in C$, we define the *color- c -neighborhood* $\Gamma(v, c)$ as the set of nodes in V_c either adjacent to v or connected to v by a

path of nodes of color c . Similarly, we denote by $\Gamma(W, c) = \bigcup_{w \in W} \Gamma(w, c)$ the color- c -neighborhood of a subset $W \subseteq V$. For a subset $W \subseteq V$ and a sequence $\gamma = \langle \gamma_1, \dots, \gamma_b \rangle$ of colors in C , we define a corresponding sequence of node sets $W_1 = W$ and $W_{i+1} = W_i \cup \Gamma(W_i, \gamma_i)$, for $1 \leq i \leq b$. That is, calling a color c always conquers all connected components induced by V_c that are adjacent to the current territory. We say that sequence γ started on W *conquers* the final node set W_{b+1} in b moves, and we denote this situation by $W \rightarrow_\gamma W_{b+1}$. Nodes in $V - W_{b+1}$ are called *free* nodes. It would be easier to define the game on *weighted* nodes, instead, and with a proper node coloring; however, not all graph classes we study later are closed under edge contractions (e.g., the hex-grid graph of the original HONEY-BEE game).

In the *solitaire* version of HONEY-BEE, the goal is to conquer the entire playing field with the smallest possible number of moves. Note that HONEY-BEE-SOLITAIRE is trivial in the case of only two colors. As we will see in Section 3, the case of three colors can already be difficult.

Problem HONEY-BEE-SOLITAIRE

Input: A graph $G = (V, E)$; a set C of k colors and a coloring $col : V \rightarrow C$; a start node $v_0 \in V$; and a bound b .

Question: Does there exist a color sequence $\gamma = \langle \gamma_1, \dots, \gamma_b \rangle$ of length b such that $\{v_0\} \rightarrow_\gamma V$?

In the *two-player* version of HONEY-BEE, the two players A and B start from two distinct nodes a_0 and b_0 and then extend their regions step by step by alternately calling colors. Player A makes the first move. One *round* of the game consists of a move of A followed by a move of B . Consider a round, where at the beginning the two players control node sets W_A and W_B , respectively. If player A calls color c , then he extends his region W_A to $W'_A = W_A \cup (\Gamma(W_A, c) - W_B)$. If afterwards player B calls color d , then he extends his region W_B to $W'_B = W_B \cup (\Gamma(W_B, d) - W'_A)$. Note that once a player controls a node, he can never lose it again. The game terminates as soon as one player controls more than half of all nodes. This player wins the game.

There are two important restrictions on the colors that a player is allowed to call:

1. A player may never call the color that had just been called by the other player. This is a technical condition that arises from the graphical implementation of the game II: Whenever a player calls a color c , his current region is entirely recolored to color c . This makes it visually easier to recognize the regions controlled by both players.
2. A player may never call the color that he had called in his preceding move. A player could not gain new territory by repeatedly calling the same color, but with rule (1) he could block his opponent forever from calling this color.

Problem HONEY-BEE-2-PLAYERS

Input: A graph $G = (V, E)$ with an odd number of nodes; a set C of colors and a coloring $col : V \rightarrow C$; two start nodes $a_0, b_0 \in V$.

Question: Can player A enforce a win when the game is played according to the above rules?

Note that HONEY-BEE-2-PLAYERS is trivial in the case of only three colors: The players have no freedom to choose the next color, and always must call the only available color that is neither blocked by rule (1) nor by rule (2). However, we will see in Section 4 that the case of four colors can already be difficult.

3 The Solitaire Game

3.1 The Solitaire Game on Co-comparability Graphs

A *co-comparability graph* $G = (V, E)$ is an undirected graph whose nodes V correspond to the elements of some partial order, $<$, and whose edges E connect any two elements that are incomparable in that partial order, i.e., $[u, v] \in E$ if neither $u < v$ nor $v < u$ holds. For simplicity, we identify the nodes with the elements of the partial order. Golumbic et al. [3] showed that co-comparability graphs are exactly the intersection graphs of continuous real-valued functions over some interval I . If two function curves intersect, the corresponding elements are incomparable in the partial order; otherwise, the curve that lies completely above the other one corresponds to the larger element in the partial order. The function graph representation readily implies that the class of co-comparability graphs is closed under edge contractions. This can be seen as follows. Assume curves f and g intersect. Contracting the edge (f, g) corresponds to replacing f and g by a single curve h that rapidly zig-zags between f and g . Clearly, h intersects exactly those curves that intersect f or g (or both).

Therefore, we may w.l.o.g. restrict our analysis of HONEY-BEE-SOLITAIRE to co-comparability graphs with a proper node coloring, i.e., adjacent nodes have distinct colors (in the solitaire game we do not care about the weight of a node after an edge contraction). In this case, every color class is totally ordered because incomparable node pairs have been contracted. For any color c , let $Min(c)$ and $Max(c)$ denote the smallest and the largest node of color c , respectively. We can further assume that the start node v_0 is colorless (because we do not need to conquer it anymore).

Consider an instance of HONEY-BEE-SOLITAIRE with a start node v_0 . The function graph representation implies that conquering a node v will simultaneously conquer all nodes of the same color between (in the total order of the color class) v and v_0 .

For a color sequence $\gamma = \langle \gamma_1, \dots, \gamma_b \rangle$, we define the *length* of γ as $|\gamma| = b$. We also define the *essential length* $ess(\gamma)$ of γ as $|\gamma|$ minus the number of steps where γ conquers the second extremal node of some color class (even if both extremal nodes are conquered in the same step). This definition is motivated by the fact

that any color sequence γ conquering the entire graph must spend one step to conquer the second extremal node of each color class, i.e., $|\gamma| = \text{ess}(\gamma) + k$.

For a node v below v_0 or incomparable to v_0 and a node w above v_0 or incomparable to v_0 , let $D(v, w)$ denote the essential length of the shortest color sequence γ that can conquer v and w when starting at v_0 . Note that we do not need to keep track of which first extremal nodes of a color class have been conquered because there is no difference in the cost between conquering the second extremal node in the same step together with the first extremal node, and the first extremal node having been conquered in an earlier step. In particular, we can compute $D(v, w)$ recursively by $D(v_0, v_0) = 0$ and

$$D(v, w) = \min_c (D(v, \min_w(c)) + \delta_w(v), D(\max_v(c), w) + \delta_v(w)),$$

where $\min_w(c)$ denotes the smallest node of color c connected to w , if such nodes exist, $\max_v(c)$ denotes the largest node of color c connected to v , if such nodes exist, and $\delta_v(w) = 0$ if and only if w is an extremal node of some color class c and the other extremal node of color class c is either between v and w , or incomparable to either v or w , or both (it was either conquered earlier, or it will be conquered in this step); otherwise, $\delta_v(w) = 1$. Then we can compute the optimal cost of solving HONEY-BEE-SOLITAIRE as $\min_{v,w} (D(v, w) + k)$, where we minimize over all minimal nodes v and all maximal nodes w . This is true because after conquering v and w , γ only needs to conquer all free nodes $\text{Min}(c)$ and $\text{Max}(c)$ to conquer the entire graph, and all these nodes are connected to the conquered subgraph at that time.

Theorem 1. HONEY-BEE-SOLITAIRE can be solved in polynomial time on co-comparability graphs. □

3.2 The Solitaire Game on Split Graphs

A *split graph* is a graph whose node set can be partitioned into an induced clique and into an induced independent set. We will show that HONEY-BEE-SOLITAIRE is NP-hard on split graphs. Our reduction is from the NP-hard **Feedback Vertex Set** (FVS) problem in directed graphs; see for instance Garey and Johnson [2].

Problem FVS

Input: A directed graph (X, A) ; a bound $t < |X|$.

Question: Does there exist a subset $X' \subseteq X$ with $|X'| = t$ such that the directed graph induced by $X - X'$ is acyclic?

Theorem 2. HONEY-BEE-SOLITAIRE on split graphs is NP-hard.

Proof. Consider an instance (X, A, t) of FVS. We construct an instance (V, E, b) of HONEY-BEE-SOLITAIRE by forming a clique from the vertices in X together

with a new node v_0 , where node x has color c_x . Each arc $(x, y) \in A$ becomes a node of color c_y with a single incident edge to x in the clique. Finally, we set $b = |X| + t$. We claim that the constructed instance of HONEY-BEE-SOLITAIRE has answer YES, if and only if the instance of FVS has answer YES.

Assume that the FVS instance has answer YES. Let X' be a smallest feedback set whose removal makes (X, A) acyclic. Let π be a topological order of the nodes in $X - X'$, and let τ be an arbitrary ordering of the nodes in X' . Consider the color sequence γ of length $|X| + t$ that starts with τ , followed by π , and followed by τ again. It is easy to see that $\{v_0\} \rightarrow_\gamma V$.

Next assume that the instance of HONEY-BEE-SOLITAIRE has answer YES. Let γ be a color sequence of length $b = |X| + t$ conquering V . Define X' as the set of nodes x such that color c_x occurs at least twice in γ ; clearly, $|X'| \leq t$. Consider an arc $(x, y) \in A$ with $x, y \in X - X'$. Since γ contains color c_y only once, it must conquer the independent node (x, y) (of color c_y) after the clique node x (of color c_x). Hence, γ induces a topological order of $X - X'$. \square

The construction in the proof above uses linearly many colors. What about the case of few colors? On split graphs, HONEY-BEE-SOLITAIRE can always be solved by traversing the color set C twice; the first traversal conquers all clique nodes, and the second traversal conquers all remaining free independent set nodes. If we have only a few colors, we can just try all color sequences of length $2|C|$.

Theorem 3. *If the number of colors is bounded by a fixed constant, HONEY-BEE-SOLITAIRE on split graphs is polynomial-time solvable.* \square

3.3 The Solitaire Game on Trees

In this section we will show that HONEY-BEE-SOLITAIRE is NP-hard on trees, even if there are at only three colors. We reduce HONEY-BEE-SOLITAIRE from a variant of the Shortest Common Supersequence (SCS) problem which is known to be NP-complete (see Middendorf [4]).

Problem SCS

Input: A positive integer t ; finite sequences $\sigma_1, \dots, \sigma_s$ with elements from $\{0, 1\}$ with the following properties: (i) All sequences have the same length. (ii) Every sequence contains exactly two 1s, and these two 1s are separated by at least one 0.

Question: Does there exist a sequence σ of length t that contains $\sigma_1, \dots, \sigma_s$ as subsequences?

If we replace every occurrence of the element 0 by two consecutive elements 0 and 2, we see that the following variant of SCS is also NP-complete.

Problem Modified SCS (MSCS)

Input: A positive integer t ; finite sequences $\sigma_1, \dots, \sigma_s$ with elements from $\{0, 1, 2\}$ with the following property: In every sequence any two consecutive elements are distinct, and no sequence starts with 2.

Question: Does there exist a sequence σ of length t that contains $\sigma_1, \dots, \sigma_s$ as subsequences?

Since we can interpret the sequences as paths of labeled nodes attached to a common root, we see that HONEY-BEE-SOLITAIRE is difficult on trees.

Theorem 4. HONEY-BEE-SOLITAIRE is NP-hard on trees, even in case of only three colors. \square

4 The Two-Player Game

4.1 The Two-Player Game on Outer-Planar Graphs

A graph is *outer-planar* if it contains neither K_4 nor $K_{2,3}$ as a minor. Outer-planar graphs have a planar embedding in which every node lies on the boundary of the so-called *outer face*. For example, every tree is an outer-planar graph.

Consider an outer-planar graph $G = (V, E)$ as an instance of HONEY-BEE-2-PLAYERS with starting nodes a_0 and b_0 in V , respectively. The starting nodes divide the nodes on the boundary of the outer face F into an upper chain u_1, \dots, u_s and a lower chain ℓ_1, \dots, ℓ_t , where u_1 and ℓ_1 are the two neighbors of a_0 on F , while u_s and ℓ_t are the two neighbors of b_0 on F . We stress that the upper and lower chains are not necessarily disjoint (for instance, articulation nodes may occur in both chains).

Now consider an arbitrary situation in the middle of the game. Let U (respectively L) denote the largest index k such that player A has conquered node u_k (respectively node ℓ_k). See Fig. 2 to illustrate these definitions. Since outer-planar graphs are K_4 -minor free, none of the free nodes among u_1, \dots, u_U and ℓ_1, \dots, ℓ_L can have a neighbor among $u_{U+1}, \dots, u_s, b_0, \ell_t, \dots, \ell_{L+1}$.

Theorem 5. HONEY-BEE-2-PLAYERS on outer-planar graphs is polynomial-time solvable.

Proof. The two indices U and L encode all necessary information on the future behavior of player A . Eventually, he will own all nodes u_1, \dots, u_U and ℓ_1, \dots, ℓ_L , and the possible future expansions of his area beyond u_U and ℓ_L only depend on U and L . Symmetric observations hold true for player B .

As every game situation can be concisely described by just four indices, there is only a polynomial number $O(|V|^4)$ of relevant game situations. The rest is routine work in combinatorial game theory: We first determine the winner for every end-situation, and then by working backwards in time we can determine the winners for the remaining game situations. \square

Note that the game is even easier for trees. The crucial battlefield is the path between a_0 and b_0 , and for both players the optimal strategy is to move along this path as quickly as possible. Thus, we can solve HONEY-BEE-2-PLAYERS on trees in linear time.

4.2 The Two-Player Game on Series-Parallel Graphs

A graph is *series-parallel* if it does not contain K_4 as a minor. Equivalently, a series-parallel graph can be constructed from a single edge by repeatedly doubling edges, or removing edges, or replacing edges by a path of two edges with a new node in the middle of the path. Note that we do not know whether the two-player game is in the class NP on series-parallel graphs (we believe it is not), so we can only claim NP-hardness.

Theorem 6. *For four (or more) colors, problem HONEY-BEE-2-PLAYERS on series-parallel graphs is NP-hard.*

Proof. We use the color set $C = \{0, 1, 2, 3\}$. A central feature of our construction is that player B will have no real decision power: In even rounds, player A will call color 0 or 1, and player B must call the other color in $\{0, 1\}$, or waste his move. In odd rounds, player A will call color 2 or 3, and player B must call the other color in $\{2, 3\}$, or waste his move. Note that wasting a move by calling a ‘wrong’ color may actually block the other player in the next round; in this case we may have one round where both players do not make progress, but then the next round will be back to normal.

The proof is by reduction from the supersequence problem SCS with binary sequences, see Fig. 3 for an example. Consider an instance $(\sigma_1, \dots, \sigma_s, t)$ of SCS, and let n denote the common length of all sequences σ_i . We first construct two start nodes a_0 and b_0 of colors 2 and 3, respectively. For each sequence σ_i we construct a path P_i consisting of $2n - 1$ nodes which is attached to a_0 . The n nodes with odd numbers mimic sequence σ_i , while the $n - 1$ nodes with even numbers all receive color 2. The first node of P_i is adjacent to a_0 , and its last node is connected to a *honey pot* H_i . This is a long path consisting of $4st$ nodes of color 3. Intuitively, we may think of a honey pot as a single node of large weight, because conquering one of the nodes will simultaneously conquer the entire path.

Each H_i can also be reached from b_0 by a path Q_i consisting of $2t - 1$ nodes. Nodes with odd numbers get color 0, and nodes with even numbers get color 3. The first node of Q_i is adjacent to b_0 , and its last node is connected to H_i . Furthermore, we create for each node of color 0 a (new) twin node of color 1 that has the same two neighbors as the color 0 node. Note that for every path Q_i there are t twin pairs. Finally, we connect b_0 to a private honey pot H_B of color 2 for player B that consists of $4s(s - 1)t + (2n - 1)s$ nodes.

It is easy to verify that A wins if and only if he conquers all the H_i . It is then straightforward to show that A has a winning strategy if and only if the SCS instance has answer YES. \square

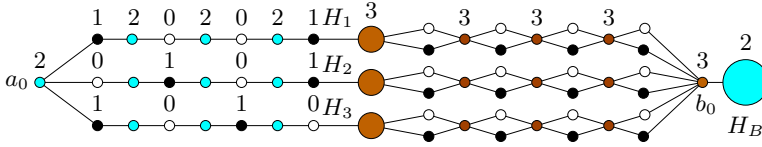


Fig. 3. The graph constructed in the proof of Thm. 6 for $\sigma_1 = 1001$, $\sigma_2 = 0101$, $\sigma_3 = 1010$, and $t = 4$. The optimal SCS solution is 10101. Thus, B can win this game.

4.3 The Two-Player Game on Arbitrary Graphs

In this section we will show that problem HONEY-BEE-2-PLAYERS is PSPACE-complete on arbitrary graphs. Our reduction is from the PSPACE-complete Quantified Boolean Formula (QBF) problem; see for instance Garey & Johnson [2].

Problem QBF

Input: A quantified Boolean formula with $2n$ variables in conjunctive normal form: $\exists x_1 \forall x_2 \cdots \exists x_{2n-1} \forall x_{2n} \wedge_j C_j$, where the C_j are clauses of the form $\vee_k l_{jk}$, where the l_{jk} are literals.

Question: Is the formula true?

Theorem 7. For four (or more) colors, problem HONEY-BEE-2-PLAYERS on arbitrary graphs is PSPACE-complete.

Proof. We prove the claim by reduction from QBF. Let $F = \exists x_1 \forall x_2 \cdots \exists x_{2n-1} \forall x_{2n} (\wedge_j C_j)$ be an instance of QBF. We now construct a bee graph $G_F = (V, E)$ with four colors (white, cyan, red, and black) such that player A has a winning strategy if and only if F is true. Let a_0 , colored cyan, and b_0 , colored red, denote the start nodes of players A and B , respectively. Each player controls a pseudo-path, called P_A and P_B . On such a path some nodes may be duplicated as parallel nodes in a diamond-shaped structure, called a choice pair; see Fig. 4(a). The start nodes are at one end of the respective pseudo-paths, and each player can conquer (nearly all) the nodes on his own path without interference from the opponent. However, they must do so in a timely manner because each path ends at a large honey pot, denoted by H_A and H_B , respectively, which is a large clique of equally-colored nodes. Both cliques will have the same size but different color, namely black (H_A) and white (H_B), and they are connected by an edge. Therefore, both players will try to reach their honey pot in the same round to prevent the other player from winning by conquering both pots. The nodes between the last variable gadgets and the honey pots are denoted by a_f and b_f , respectively.

A variable gadget (see Fig. 4(a)) is a part of the two pseudo-paths corresponding to a pair of variables $\exists x_{2i-1} \forall x_{2i}$, for some $i \geq 1$. On P_A , the gadget starts at node a_{i-1} with a choice pair a_{2i-1}^F and a_{2i-1}^T , colored white and black, respectively. The first node conquered by A will determine the truth value for

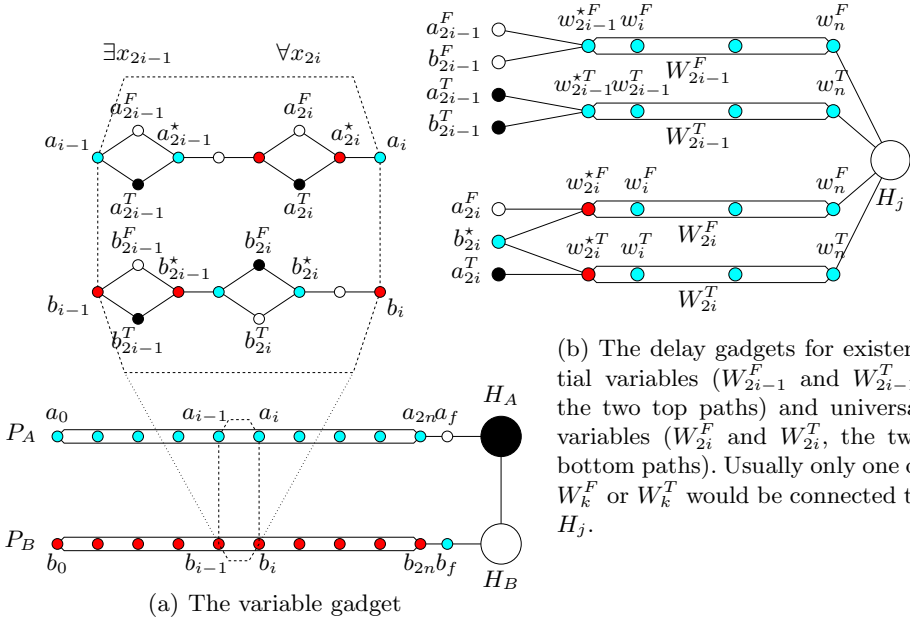


Fig. 4. The gadgets in the proof of Thm. 7

variable x_{2i-1} . In the same round, player B has a choice on P_B between nodes b_{2i-1}^F and b_{2i-1}^T , also colored white and black, so he must select the other color not chosen by A .

Three rounds later, there is a choice pair b_{2i}^F and b_{2i}^T on P_B , where player B can assign a truth value to variable x_{2i} . In the next step (which is in the next round), player A has a choice pair a_{2i}^F and a_{2i}^T with the same colors as B 's choice pair for x_{2i} , so he must select the color not chosen by B in the previous step. Since we want A to conquer those clauses containing a literal set true by player B (the clause gadgets are defined below), the colors in B 's choice pair have been switched, i.e., b_{2i}^F is black and b_{2i}^T is white.

Note that all the nodes a_0, a_1, \dots, a_n have color cyan and all the nodes b_0, b_1, \dots, b_n have color red. This allows us to concatenate as many variable gadgets as needed. Further note that a_f is white, while b_f is cyan.

For each clause C_j , there is a *clause gadget* consisting of a small honey pot H_j of color white. These honey pots are smaller than the large pots H_A and H_B , but large enough such that player A loses if he misses one of them. Player A should conquer H_j if and only if C_j is true in the assignment chosen by the players while conquering their respective pseudo-paths. Therefore, the variable gadgets are connected to the clause gadgets via *delay gadgets*. Let a_k^* denote the node on P_A right after the choice pair a_k^F and a_k^T , for $k = 1, \dots, 2n$; similarly, b_k^* are the nodes on P_B right after B 's choice pairs. A delay gadget W_k consists of two copies W_k^F and W_k^T of the sub-path of P_A starting at a_k^* and ending at

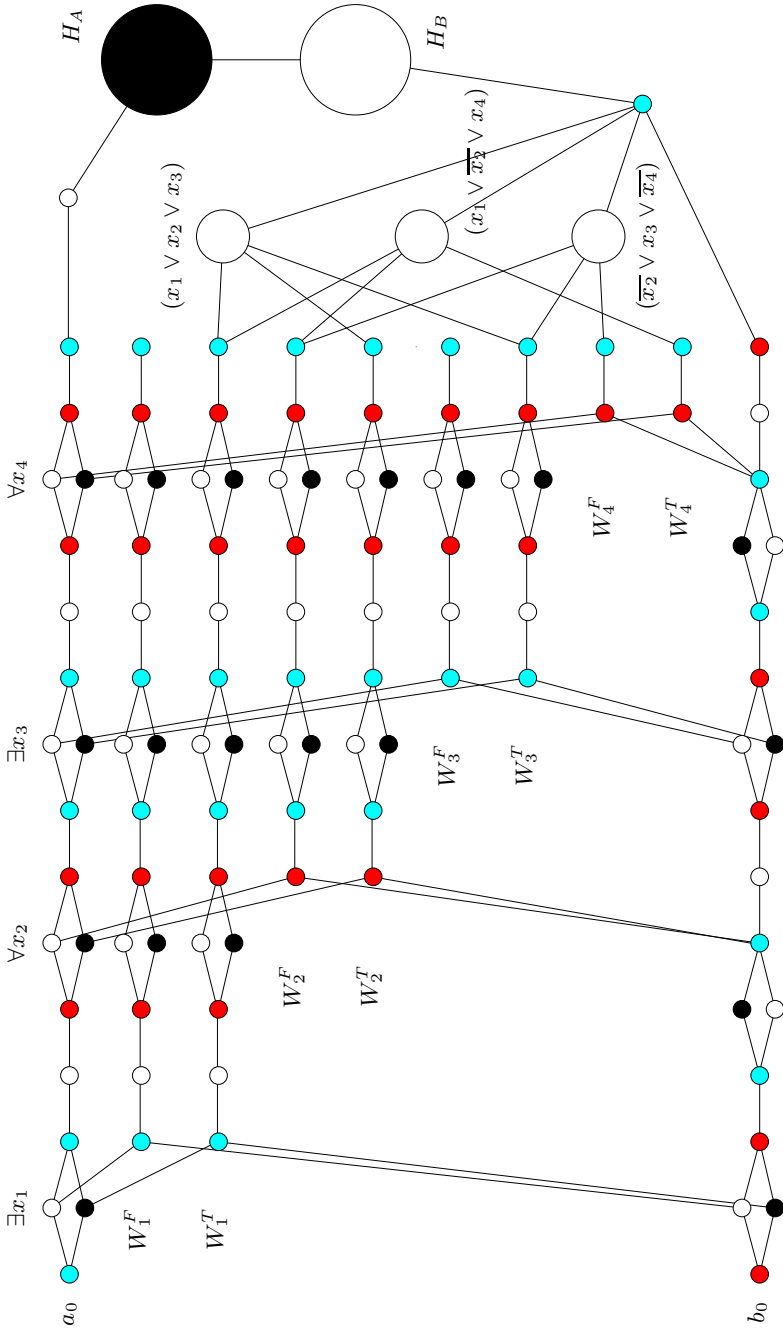


Fig. 5. The reduction in the proof of Thm. 7 would construct this graph for the formula $F = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_4) \wedge (\overline{x_2} \vee x_3 \vee \overline{x_4})$

a_n , see Fig. 4(b). If clause C_j contains literal x_k , H_j is connected to the node w_n^T corresponding to a_n in W_k^T ; if C_j contains literal $\overline{x_k}$, H_j is connected to the node w_n^F corresponding to a_n in W_k^F . If $k = 2i - 1$ (i.e., an existential variable x_{2i-1} whose value is assigned by player A), then a_{2i-1}^F and b_{2i-1}^F are connected to w_{2i-1}^{*F} , and a_{2i-1}^T and b_{2i-1}^T are connected to w_{2i-1}^{*T} . If $k = 2i$ (i.e., a universal variable x_{2i} whose value is assigned by player B), then a_{2i}^F and b_{2i}^* are connected to w_{2i}^{*F} , and a_{2i}^T and b_{2i-1}^* are connected to w_{2i}^{*T} .

Finally, we connect b_f with all clause honey pots H_j so that player B can conquer all clauses without a true literal. This finishes the construction of G_F , see Fig. 5 for an example.

We claim that player A has a winning strategy on G_F if and only if F is true if the clause honey pots have size $2n^2$ and the two large honey pots have size $2n^3$. It is easy to verify that player A can indeed conquer all honey pots and win if F is true. It is important to note that player B can never block a move by player A (except when player B selects a value for a universal variable) \square \square

5 Conclusions

We have modeled the Honey Bee game as a game on colored graphs. We have analyzed the complexity of the solitaire version on many classes of perfect graphs. We have also shown that it is hard to compute a winning strategy in the two-player version, even in the highly restricted case of series-parallel graphs.

Acknowledgements

Part of this research was done while G. Woeginger was visiting Fudan University in 2009. We would like to thank an unknown reviewer who pointed out a few unclear definitions in our original manuscript.

References

1. Born, A.: Flash application for the computer game “Biene” (Honey-Bee) (2009), <http://www.ursulinen.asn-graz.ac.at/Bugs/html/games/biene.htm>
2. Garey, M.R., Johnson, D.S.: Computers and Intractability — A Guide to the Theory of NP-Completeness. W. H. Freeman and Company, New York (1979)
3. Golumbic, M.C., Rotem, D., Urrutia, J.: Comparability graphs and intersection graphs. *Discrete Mathematics* 43(1), 37–46 (1983)
4. Middendorf, M.: More on the complexity of common superstring and supersequence problems. *Theoretical Computer Science* 125, 205–228 (1994)

¹ Actually, B could block A 's next move by choosing a color which does not make progress on his own pseudo-path; in this case, A can make an arbitrary move, and then in the next move B cannot block A again.

Mapping an Unfriendly Subway System

Paola Flocchini¹, Matthew Kellett², Peter C. Mason², and Nicola Santoro³

¹ School of Information Technology and Engineering, University of Ottawa, Canada

flocchin@site.uottawa.ca

² Defence R&D Canada – Ottawa, Government of Canada, Canada

{matthew.kellett,peter.mason}@drdc-rddc.gc.ca

³ School of Computer Science, Carleton University, Canada

santoro@scs.carleton.ca

Abstract. We consider a class of highly dynamic networks modelled on an urban subway system. We examine the problem of creating a map of such a subway in less than ideal conditions, where the local residents are not enthusiastic about the process and there is a limited ability to communicate amongst the mappers. More precisely, we study the problem of a team of asynchronous computational entities (the mapping agents) determining the location of black holes in a highly dynamic graph, whose edges are defined by the asynchronous movements of mobile entities (the subway carriers). We present and analyze a solution protocol. The algorithm solves the problem with the minimum number of agents possible. We also establish lower bounds on the number of carrier moves in the worst case, showing that our protocol is also move-optimal.

1 Introduction

Computer networks are not necessarily safe. They often contain dangerous elements such as computers that have undetectably crashed or network equipment that is malfunctioning or misconfigured. There is a large body of research into distributed algorithms for finding these faults, which are often referred to in the literature as black holes and black links, or, more generally, as dangerous elements (e.g., see [2,3,4,5,6,7,8,9,10,12,13,14,15]). All these investigations on finding dangerous elements assume that the network itself is static and connected.

There are several classes of networks that have dynamic topologies that change as a function of time, and that might be disconnected at times. They include wireless mobile ad hoc networks where the network's topology may change dramatically over time due to the movement of the network's nodes; sensor networks where links only exist when two neighbouring sensors are awake and have power; and vehicular networks, similar to mobile ad hoc networks, where the topology changes constantly as vehicles move. Indeed there is a large amount of research on these networks (which are called *delay-tolerant*, *challenged*, *opportunistic*, *evolving*, etc.) focusing mostly on broadcasting and routing (e.g., see [11,16,17,18,19]). At least one study [11] has looked at how to *explore* one class of these networks: periodically-varying graphs. In the periodically-varying

graph (PV graph) exploration problem, agents ride carriers between sites in the network. A link only exists between sites when a carrier is passing between them. The agents explore the network by moving from carrier to carrier when their routes meet at a site. These dynamic networks are no less prone to faults than static networks. The question is, how does one find a dangerous element in a time-varying network?

Imagine a group of tourists are visiting the unfriendly capital of Dystopia—perhaps not the best travel destination. Although the city has a subway system, there are no maps because the local population wants to limit their capital’s appeal to tourists (Dystopians are grumpy by nature). The tourists want to map the subway system without the local population knowing. They agree on a location in each station where they will leave notes for each other. The problem is that there are good stations and there are bad stations. Tourists arriving at good stations can easily leave notes for each other. Tourists arriving at bad stations get lost when they try to leave notes, eventually giving up on the whole map-making process. The group wants to complete the map while minimizing the number of their group lost to the frustration of the bad stations.

We look at black hole search in a class of time-varying network based on a similar scenario. Instead of tourists, we have agents. Instead of subway trains, we have carriers. Instead of stations, we have sites, where the bad stations are black hole sites that eliminate the agents arriving at them without leaving a discernable trace. The class of networks described by this *subway model* is much larger than the set of real subway systems. We look at the asynchronous version of the black hole search problem where the calculations of the agents and movements of the carriers take a finite but unpredictable amount of time. To measure complexity in this environment, we look at the number of carrier moves needed to complete the search. We show that our solution has a complexity $O(k \cdot n_C^2 \cdot l_R) + O(n_C \cdot l_R^2)$ carrier moves where n_C is the number of carriers and l_R is the length of the longest carrier route. We prove that the lower bound on the worst case complexity is $\Omega(k \cdot n_C^2 \cdot l_R) + \Omega(n_C \cdot l_R^2)$ carrier moves, making our solution worst-case optimal.

2 Model

We consider a set C of n_C carriers that move among a set S of n_S sites. A carrier $c \in C$ follows a route $R(c)$ between all the sites in its *domain* $S(c) = \{s_0, s_1, \dots, s_{n_S(c)-1}\} \subseteq S$. A carrier’s *route* $R(c) = \langle r_0, r_1, \dots, r_{l(c)-1} \rangle$ is a cyclic sequence of *stops*: after stopping at site $r_i \in S(c)$, the carrier moves to $r_{i+1} \in S(c)$, where operations on the indices are modulo $l(c) = |R(c)|$ called the *length* of the route. Carriers move *asynchronously*, taking a finite but unpredictable amount of time to move between stops. Each carrier is labelled with a distinct id and the length of its route. A route is *simple* if $n_S(c) = l(c)$, and *complex* otherwise. A *transfer site* is any site that is in the domain of two or more carriers; each transfer site is labelled with the number of carriers stopping at it. A carrier’s route $R(c) = \langle r_0, r_1, \dots, r_{l(c)-1} \rangle$ defines an edge-labelled directed

multigraph $\mathbf{G}(c) = (S(c), \mathbf{E}(c), \lambda(c))$, called *carrier graph*, where there is an edge labeled $(c, i + 1)$ from r_i to r_{i+1} , and the operations on indices and inside labels are modulo $l(c)$. The entire network is then represented by the edge-labelled directed multigraph $\mathbf{G} = (R, \mathbf{E}, \lambda)$, called *subway graph*, where $R = \cup_{c \in C} R(c)$, $\mathbf{E} = \cup_{c \in C} \mathbf{E}(c)$, and $\lambda = \{\lambda(c) : c \in C\}$. Associated with the subway graph is the *transfer graph* of \mathbf{G} which we define as the edge-labelled undirected multigraph $H(\mathbf{G}) = (C, E_T)$ where the nodes are the carriers and, $\forall c, c' \in C, s \in S$, there is an edge between c and c' labeled s iff $s \in S(c) \cap S(c')$, i.e., s is a transfer site between c and c' . In the following, when no ambiguity arises, we will omit the edge labels in all graphs. See Figure 10. Working in the network is a team A of k computational *agents* that start at unpredictable times from the same site, called the *homebase*. Agents can only communicate with each other using shared memory, available at each site in the form of a *whiteboard*, which is accessed in fair mutual exclusion. The agents are *asynchronous* in that they take a finite but unpredictable amount of time to perform computations at a site. All agents execute the same protocol and know the number of carriers n_C . The agents move around the network using the carriers. An agent can move from a carrier to a site (*disembark* at a stop) or from a site to a carrier (*board* a carrier), but not from one carrier to another directly. An agent on a transfer site can board any carrier stopping at it. When on a site, an agent can access the whiteboard, and can read the number of the carriers stopping at that site; furthermore, it can read the labels of any carrier stopping there. When travelling on a carrier, an agent can count the number of stops that the carrier has passed, and can decide whether or not to disembark at the next stop. Once disembarked, the agent can later board the same carrier at the same point in its route.

Among the sites there are $n_B < n_S$ *black holes*: sites that eliminate agents disembarking on them without leaving a discernable trace; black holes do not affect carriers. The *black hole search* (BHS) problem is that of the agents determining the locations of the black holes in the subway graph. A protocol solves the BHS problem if within finite time at least one agent survives and all the surviving agents know which *stops* are black holes. There are some basic limitations for the BHS problem to be solvable: the transfer graph must be connected once the black holes are removed, and the homebase must be a safe site. Hence we will assume these conditions to hold. Because of asynchrony, slow computation by an agent exploring a safe stop is indistinguishable from an agent having been eliminated by a black hole stop, so, with only knowledge of n_C , termination is impossible unless each carrier has at least one safe transfer site. Hence the agents' knowledge of n_C will be assumed.

As in traditional mobile agent algorithms, the basic cost measure used to evaluate the *efficiency* of a BHS solution protocol is the *size* of the team that is the number k of agents needed by the protocol. Let $\gamma(c) = |\{i : r_i \in R(c) \text{ is a black hole}\}|$ be the number of black holes among the stops of c ; and let $\gamma(\mathbf{G}) = \sum_{c \in C} \gamma(c)$, called the *faulty load* of subway graph \mathbf{G} , be the total number of stops that are black holes. The *faulty load* $\gamma(\mathbf{G})$ of subway graph \mathbf{G} is the number of stops that are black holes. To solve BHS, it is obviously necessary

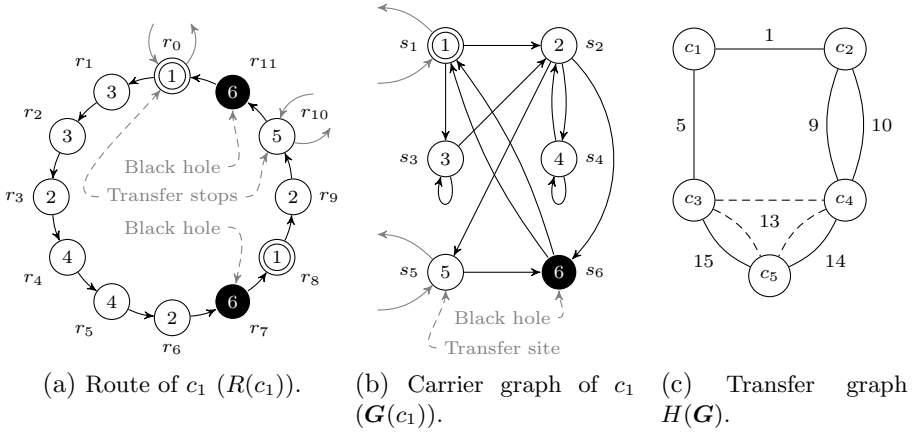


Fig. 1. A route, the corresponding carrier graph, and a transfer graph (edge labels are the corresponding transfer site ids)

to have more agents than the faulty load of the network, i.e. $k > \gamma$. A solution protocol is *agent optimal* if it solves the BHS problem for $k = \gamma + 1$. The other cost measure is the number of *carrier moves*: when an agent is waiting for a carrier or riding on a carrier, each move made by that carrier is counted as a carrier move for the agent. A solution protocol is *move optimal* in the worst case if the total number of carrier moves incurred by all agents in solving the BHS problem is the best possible.

3 Exploration Algorithm

In this section we present the proposed algorithm *SubwayExplore*; as we will show later, our algorithm works correctly with any number of agents $k \geq \gamma + 1$.

Informally, algorithm *SubwayExplore* works as follows. All the agents start at unpredictable times from the same site s , called the *homebase*. An agent’s work involves visiting a previously unexplored stop on a carrier’s route and returning, if possible, to report what was found there. Every carrier is searched from a *work site* and the work sites are organized into a logical *work tree* that is rooted in the homebase. The first agent to access the homebase’s whiteboard sets up the homebase as a work site (Sect. 3.1). It and the agents awaking after it then begin to do work by visiting the stops of the carriers stopping at the homebase (Sect. 3.2). If an exploring agent finds a previously unexplored transfer site, the agent “competes” to add the transfer site to the work tree. If the agent succeeds, the transfer site becomes a work site for other carriers stopping at it and the work site from which it was discovered becomes its parent in the work tree (Sect. 3.3). When the carrier that the agent is exploring has no more unexplored stops, it tries to find another carrier with work in the work tree. It looks for work in the subtree rooted in its current work site and if there is no work available it moves

to the work site's parent and tries again (Sect. 3.4). An agent terminates if it is at the homebase, there is no work, and there are n_C carriers in the work tree.

3.1 Initialization

When an agent awakes for the first time on the homebase, it tries to initialize the homebase as a work site. Only the first agent succeeds and executes INITIALIZE WORK SITE. All other agents proceed directly to trying to find work.

The INITIALIZE WORK SITE procedure is used to set up each work site in the work tree. The procedure takes as input the parent of the work site and the carriers to be worked on or serviced from the work site. For the homebase, the parent is *null* and the carriers to be accessed are all those stopping at s . The procedure initializes the work site's whiteboard with the information needed to find work, do work, and compete to add work. More precisely, when a work site ws is initialized, its parent is set to the work site from which it was discovered (*null* in the homebase's case) and its children are initially *null*. The carriers it will service are added to $C_{subtree}$, the set of carriers in the work tree at and below this work site. The same carriers are also added to C_{work} , the set of carriers in the subtree with unexplored stops, and C_{local} , the set of carriers serviced by this work site. For each carrier c added to C_{local} , the agent setting up the whiteboard creates three sets U_c , D_c , and E_c . The set E_c of *explored stops* is initialized with the work site at $r_0 = ws$ (r_0 is always the work site servicing the carrier). The set U_c of *unexplored stops* is initialized with the rest of the stops on the carrier's route $\{r_1, r_2, \dots, r_{l(c)-1}\}$, which is possible because each carrier is labelled with its length as well as its id. The set D_c of *stops being explored* (and therefore potentially dangerous sites) is initially empty.

3.2 Do Work

We now discuss how the agents do their exploration of unexplored stops (procedure DO WORK shown in Algorithm 1). To limit the number of agents eliminated by black holes, we use a technique similar to the cautious walk technique used in static networks. Consider an agent a on the work site ws of a carrier c that still has unexplored stops, i.e. $U_c \neq \emptyset$. The agent does the following. It chooses an unexplored stop $r \in U_c$ for exploration, removes r from U_c , and adds it to the set D_c of stops being explored. It then takes c to r and disembarks. If the agent survives, it returns to ws using the same carrier c and disembarks. The agent can make the trip back to ws because it knows the index of r and $l(c)$ and can therefore calculate the number of stops between r and ws . At ws , it removes r from D_c and adds it to the set E_c of explored stops. At this point, the agent also adds the site id and any other information of interest. If r is a transfer site and a is the first to visit it (its whiteboard is blank), then, before returning to ws , the agent proceeds as follows. It records on r 's whiteboard all the carriers that pass by r , including their id and lengths of their route. It initializes two sets in its own memory: the set of *new carriers* initially containing all the carriers stopping at r ; and the set of *existing carriers*, initially empty. These sets are used in the next procedure that we discuss: competing to add work.

Algorithm 1. Do work

Agent a is working on carrier c from work site ws .

- 1: **procedure** DO WORK(carrier c)
- 2: **while** $U_c \neq \emptyset$ **do**
- 3: choose a stop r from U_c
- 4: $U_c \leftarrow U_c \setminus \{r\}$ ▷ Remove r from the set of unexplored stops
- 5: $D_c \leftarrow D_c \cup \{r\}$ ▷ Add r to the set of stops being explored
- 6: take c to r and disembark
- 7: ▷ If not eliminated by black hole
- 8: **if** r is a transfer site \wedge whiteboard is blank **then**
- 9: $a.newC \leftarrow \emptyset$ ▷ Initialize agent's set of new carriers
- 10: $a.existingC \leftarrow \emptyset$ ▷ Initialize agent's set of existing carriers
- 11: **for** each carrier c stopping at r **do**
- 12: record c on whiteboard
- 13: $a.newC \leftarrow a.newC \cup \{c\}$ ▷ Add carrier to agent's set of new carriers
- 14: **end for**
- 15: **end if**
- 16: take c to ws and disembark
- 17: $D_c \leftarrow D_c \setminus \{r\}$ ▷ Remove r from the set of stops being explored
- 18: $E_c \leftarrow E_c \cup \{r\}$ ▷ Add r to the set of explored stops
- 19: **if** r was a transfer site **then**
- 20: COMPETE TO ADD WORK
- 21: **end if**
- 22: **end while**
- 23: **end procedure**

3.3 Compete to Add Work

When an agent a discovers that a stop r is an unvisited transfer site, that stop is a potential new work site for the other carriers stopping at it. There is a problem, however: other agents may have independently discovered some or all of those carriers stopping at r . To ensure that each carrier has only one associated work site in the work tree, in our algorithm agent a must compete with all those other agents before it can add r as the new work site in the tree for those carriers. We use $C_{subtree}$ on the work sites in the work tree to decide the competition (if any).

Let us describe the actions that agent a performs; let a have just finished exploring r on carrier c_{ws} from work site ws and found that r is a new transfer site. The agent has a set of *new carriers* that initially contains all the carriers stopping at r , a set of *existing carriers* that is initially empty, and is currently on its work site ws . The agent walks up the work tree from ws to s checking the set of new carriers against $C_{subtree}$ on each work site. If a new carrier is not in $C_{subtree}$, the agent adds it. If a new carrier is in $C_{subtree}$, the agent moves it to the set of existing carriers. The agent continues until it reaches s or its set of new carriers is empty. The agent then walks down the work tree to ws . It adds each carrier in its set of new carriers to C_{work} on each work site on the way down to ws . For each carrier in its set of existing carriers, it removes the carrier from $C_{subtree}$ on the work site if it was the agent that added it. When

it reaches ws , it removes the existing carriers and if there are no new carriers, it continues its work on c_{ws} . If there are new carriers, the agent adds r as a child of ws and goes to r . At r , the agent initializes it as a work site using the INITIALIZE WORK SITE procedure with ws as its parent and the set of new carriers as its carriers. The agent then returns to ws and continues its work on c_{ws} . The procedure COMPETE TO ADD WORK, shown in Algorithm 2, ensures the following properties:

Lemma 1. *All new work is reported to the root.*

Lemma 2. *If a new carrier is discovered, it is added to the work tree within finite time.*

Lemma 3. *A carrier is always serviced from a single work site.*

3.4 Find Work

Now that we have seen work being done and new work added to the tree, it is easy to discuss how an agent a finds work. When a work site is initialized, its parent is set to the work site from which it was discovered (*null* in the homebase's case) and its children are initially *null*. As mentioned before, each work site has a set C_{work} that contains the carriers in its subtree with unexplored stops.

If C_{work} on the current work site is not empty, an agent a looking for work chooses a carrier c and walks down the tree until it reaches the work site ws servicing c or it finds that c is no longer in C_{work} . Assume that agent a reaches ws without finding c missing from C_{work} . Then a works on c until it is either eliminated by a black hole or U_c is empty. If the agent survives and is the first agent to discover that U_c is empty, it walks up the tree from ws to s removing c from C_{work} along the way. So, it is possible for an agent descending to do work on c to find out before it reaches ws that the work on c is finished. In that case, the agent starts over trying to find work.

If agent a looking for work finds that C_{work} at the current work site is empty, it moves to the work site's parent and tries again. If it reaches the root without finding work but the termination condition is not met (there are fewer than n_C carriers in the work tree), the agent waits (loops) until new work arrives or the termination condition is finally met. The procedure FIND WORK, shown in Algorithm 3, ensures the following property:

Lemma 4. *Within finite time, an agent looking for work either finds it or waits on the root.*

3.5 Correctness

Let us now prove the correctness of algorithm *SubwayExplore*. To do so, we need to establish some additional properties of the Algorithm:

Lemma 5. *Let $r_i \in R(c)$ be a black hole. At most one agent is eliminated by stopping at r_i when riding c .*

Algorithm 2. Compete to Add Work

Agent a has found a new transfer site r while exploring carrier c_{ws} from work site ws and is competing to add it to the work tree with ws as r 's parent.

```

1: procedure COMPETE TO ADD WORK
  ▷ Walk up tree
2: repeat
3:   take the appropriate carrier to parent and disembark
4:   for  $c \in a.newC$  do
5:     if  $c \in C_{subtree}$  then
6:        $a.newC \leftarrow a.newC \setminus \{c\}$       ▷ Remove from set of new carriers
7:        $a.existingC \leftarrow a.existingC \cup \{c\}$   ▷ Add to set of existing carriers
8:     else
9:        $C_{subtree} \leftarrow C_{subtree} \cup \{c\}$ 
10:    end if
11:  end for
12: until (on  $s$ )  $\vee$  ( $a.newC = \emptyset$ )
  ▷ Walk down tree
13: while not on  $ws$  do
14:   for  $c \in a.newC$  do
15:      $C_{work} \leftarrow C_{work} \cup \{c\}$       ▷ Add new carriers with work in subtree
16:   end for
17:   for  $c \in a.existingC$  do
18:     if  $a$  added  $c$  to  $C_{subtree}$  then
19:        $C_{subtree} \leftarrow C_{subtree} \setminus \{c\}$   ▷ Remove carrier from subtree set
20:     end if
21:   end for
22:   take appropriate carrier to child in direction of  $ws$  and disembark
23: end while
  ▷ Remove any existing carriers on  $ws$ 
24: for  $c \in a.existingC$  do
25:   if  $a$  added  $c$  to  $C_{subtree}$  then
26:      $C_{subtree} \leftarrow C_{subtree} \setminus \{c\}$   ▷ Remove carrier from subtree set
27:   end if
28: end for
  ▷ Add any new carriers to the tree with  $r$  as their work site
29: if  $a.newC \neq \emptyset$  then
30:    $children \leftarrow children \cup \{r\}$ 
31:   for  $c \in a.newC$  do
32:      $C_{work} \leftarrow C_{work} \cup \{c\}$ 
33:   end for
34:   take carrier  $c_{ws}$  to  $r$  and disembark
35:   INITIALIZE WORK SITE( $ws, a.newC$ )
36:   take carrier  $c_{ws}$  to  $ws$  and disembark
37: end if
38: DO WORK( $c_{ws}$ )      ▷ Keep working on original carrier
39: end procedure

```

Algorithm 3. Find work

Agent a is looking for work in the work tree. The agent knows n_C , the number of carriers, which is needed for termination. Let ws be the current work site.

```

1: procedure FIND WORK
  ▷ Main loop
2:   while (not on  $s$ )  $\vee$  ( $C_{work} \neq \emptyset$ )  $\vee$  ( $|C_{subtree}| < n_C$ ) do
  ▷ Choose carrier to work on and go there
3:     if  $C_{work} \neq \emptyset$  then
4:       choose carrier  $c$  from  $C_{work}$ 
5:       while ( $c \notin C_{local}$ )  $\wedge$  ( $c \in C_{work}$ ) do
6:         take appropriate carrier to child in direction of  $c$  and disembark
7:       end while
8:       if  $c \in C_{local}$  then                                ▷ On the work site servicing  $c$ 
9:         DO WORK( $c$ )
10:      if  $c \in C_{work}$  then                                ▷ The first agent to find no work left on  $c$ 
11:        while not on  $s$  do
12:           $C_{work} \leftarrow C_{work} \setminus \{c\}$ 
13:          take appropriate carrier to parent and disembark
14:        end while
15:         $C_{work} \leftarrow C_{work} \setminus \{c\}$ 
16:      end if
17:    end if
18:    else                                                ▷ No work in subtree
19:      take appropriate carrier to parent and disembark
20:    end if
21:  end while
22: end procedure

```

Lemma 6. *There is at least one agent alive at all times before termination.*

Lemma 7. *An agent that undertakes work completes it within finite time.*

Lemma 8. *If there is work available, an agent eventually does it.*

Lemma 9. *All carriers are eventually added to the tree.*

We can now state the correctness of our algorithm:

Theorem 1 *Protocol SubwayExplore correctly and in finite time solves the mapping problem with $k \geq \gamma(\mathbf{G}) + 1$ agents in any subway graph \mathbf{G} .*

4 Bounds and Optimality

We now analyze the costs of our algorithm, establish lower bounds on the complexity of the problem and prove that they are tight, showing the optimality of our protocol.

Theorem 2 *The algorithm solves black hole search in a connected dangerous asynchronous subway graph in $O(k \cdot n_C^2 \cdot l_R + n_C \cdot l_R^2)$ carrier moves in the worst case.*

We now establish some *lower bounds* on the worst case complexity of any protocol using the minimal number of agents.

Theorem 3 *For any α, β, γ , where $\alpha, \beta > 2$ and $1 < \gamma < 2\alpha\beta$, there exists a simple subway graph \mathbf{G} with α carriers with maximum route length β and faulty load γ in which every agent-optimal subway mapping protocol \mathcal{P} requires $\Omega(\alpha^2 \cdot \beta \cdot \gamma)$ carrier moves in the worst case. This result holds even if the topology of \mathbf{G} is known to the agents.*

Proof. Consider a subway graph \mathbf{G} whose transfer graph is a line graph; all α routes are simple and have the same length β ; there exists a unique transfer stop between neighbouring carriers in the line graph; no transfer site is a black hole, and the number of black holes is γ . The agents have all this information, but do not know the order of the carriers in the line.

Let \mathcal{P} be a subway mapping protocol that always correctly solves the problem within finite time with the minimal number of agents $k = \gamma + 1$. Consider an adversary \mathcal{A} playing against the protocol \mathcal{P} . The power of the adversary is the following: 1) it can choose which stops are transfers and which are black holes; 2) it can “block” a site being explored by an agent (i.e., delay the agent exploring the stop) for an arbitrary (but finite) amount of time; 3) it can choose the order of the carriers in the line graph. The order of the carrier will be revealed to the agents incrementally, with each revelation consistent with all previous ones; at the end the entire order must be known to the surviving agents.

Let the agents start at the homebase on carrier c_1 . Let $q = \lceil \frac{k-2}{\beta-2} \rceil$. Assume that the system is in the following configuration, which we shall call *Flip*(i), for some $i \geq 1$: (1) carrier c_1 is connected to c_2 , and carrier c_j ($j < i$) is connected to c_{j+2} ; (2) all stops of carriers c_1, c_2, \dots, c_i have been explored, except the transfer stop r_{i+1} , leading from carrier c_{i-1} to carrier c_{i+1} , and the stop r_{i+2} on carrier c_{i+1} , which are currently being explored and are blocked by the adversary; and (3) all agents, except the ones blocked at stops r_{i+1} and r_{i+2} , are on carrier c_i . See Fig. 2. If the system is in configuration *Flip*(i), with $i < \alpha - q$, the adversary operates as follows.

(1) The adversary unblocks r_{i+1} , the transfer site leading to carrier c_{i+1} . At this point, all $k - 1$ unblocked agents (including the $k - 2$ currently on c_i) must move to r_{i+1} to explore c_{i+1} without waiting for the agent blocked at r_{i+2} to come back. To see that all must go within finite time, assume by contradiction that only $1 \leq k' \leq k - 2$ agents go to explore c_{i+1} within finite time, while the others never go to r_{i+1} . In this case, the adversary first reveals the order of the carriers in the line graph by assigning carrier c_j to be connected to c_{j+1} for $\alpha > j > i$. Then the adversary chooses the following stops to be black holes: r_{i+2} , the first k' non-transfer stops visited by the k' agents, and other $k - k' - 2$

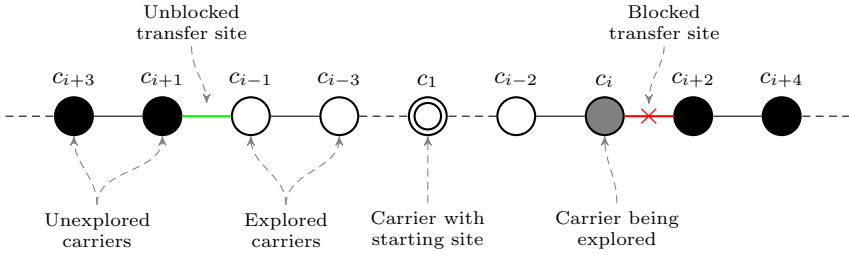


Fig. 2. Transfer graph in the lowerbound proof

non-transfer stops arbitrarily chosen in those carriers. Notice this can be done because, since $q = \lceil \frac{k-2}{\beta-2} \rceil$, the number of non-transfer stops among these carriers is $q(l-2) + 1 \geq k-1$. Thus all k' agents will enter a black hole. Since none of the other agents will ever go to c_{i+1} , the mapping will never be completed. Hence, within finite time all $k-1$ non blocked agents must go to r_{i+1} , with a total cost of $O(k \cdot i \cdot \beta)$ carrier moves.

(2) The adversary blocks each stop of c_{i+1} being explored, until $k-1$ stops are being explored. At that point, it unblocks all those stops except one, r_{i+3} . Furthermore, it makes r_{i+2} the transfer stop leading to carrier c_{i+2} .

Notice that after these operations, the system is precisely in configuration $Flip(i+1)$. Further observe now that, from the initial configuration, when all agents are at the homebase and the protocol starts, the adversary can create configuration $Flip(0)$ by simply blocking the first two stops of c_1 being explored, and making one of them the transfer to c_2 . In other words, within finite time, the adversary can create configuration $Flip(0)$; it can then transform configuration $Flip(i)$ into $Flip(i+1)$, until configuration $Flip(\alpha-q-1)$ is reached. At this point the adversary reveals the entire graph as follows: it unblocks $r_{\alpha-q+1}$, the transfer site leading to carrier $c_{\alpha-q+1}$; it assigns carrier c_j to be connected to c_{j+1} for $\alpha > j > \alpha-q$; finally it chooses $k-1$ non-transfer stops of these carriers to be black holes; notice that they can be chosen because, since $q = \lceil \frac{k-2}{\beta-2} \rceil$, the number of non-transfer stops among these carriers is $q(l-2) + 1 \geq k-1$.

The transformation from $Flip(i)$ into $Flip(i+1)$ costs the solution protocol \mathcal{P} at least $\Omega(k \cdot i \cdot \beta)$ carrier moves, and this is done for $1 \leq i \leq \alpha-q$; since $\alpha(l-2) \geq (k-2)l$ it follows that $\alpha-q = \alpha - \lceil \frac{k-2}{\beta-2} \rceil \geq \alpha - \frac{k-2}{\beta-2} \geq \frac{\alpha}{2}$; hence, $\sum_{1 \leq i \leq \alpha-q} i = O(\alpha^2)$. In other words, the adversary can force any solution protocol to use $\Omega(\alpha^2 \cdot \beta \cdot \gamma)$ carrier moves.

Theorem 4 *For any α, β, γ , where $\alpha, \beta > 2$ and $1 < \gamma < \beta - 1$, there exists a simple subway graph \mathbf{G} with α carriers with maximum route length β and faulty load γ in which every subway mapping protocol \mathcal{P} requires $\Omega(\alpha \cdot \beta^2)$ carrier moves in the worst case. This result holds even if the topology of \mathbf{G} is known to the agents,*

Theorem 5 *Protocol SubwayExplore is agent-optimal and move-optimal.*

References

1. Bui-Xuan, B., Ferreira, A., Jarry, A.: Computing shortest, fastest, and foremost journeys in dynamic networks. *Int. J. Found. Comp. Sci.* 14(2), 267–285 (2003)
2. Chalopin, J., Das, S., Santoro, N.: Rendezvous of mobile agents in unknown graphs with faulty links. In: Pelc, A. (ed.) *DISC 2007*. LNCS, vol. 4731, pp. 108–122. Springer, Heidelberg (2007)
3. Cooper, C., Klasing, R., Radzik, T.: Searching for black-hole faults in a network using multiple agents. In: Shvartsman, M.M.A.A. (ed.) *OPODIS 2006*. LNCS, vol. 4305, pp. 320–332. Springer, Heidelberg (2006)
4. Cooper, C., Klasing, R., Radzik, T.: Locating and repairing faults in a network with mobile agents. *Theoretical Computer Science* (to appear 2010)
5. Czyzowicz, J., Kowalski, D., Markou, E., Pelc, A.: Complexity of searching for a black hole. *Fundamenta Informaticae* 71(2,3), 229–242 (2006)
6. Czyzowicz, J., Kowalski, D., Markou, E., Pelc, A.: Searching for a black hole in synchronous tree networks. *Combin. Probab. Comput.* 16(4), 595–619 (2007)
7. Dobrev, S., Flocchini, P., Prencipe, G., Santoro, N.: Searching for a black hole in arbitrary networks. *Distributed Computing* 19(1), 1–19 (2006)
8. Dobrev, S., Flocchini, P., Prencipe, G., Santoro, N.: Mobile search for a black hole in an anonymous ring. *Algorithmica* 48(1), 67–90 (2007)
9. Flocchini, P., Ilcinkas, D., Santoro, N.: Ping pong in dangerous graphs: Optimal black hole search with pure tokens. In: Taubenfeld, G. (ed.) *DISC 2008*. LNCS, vol. 5218, pp. 227–241. Springer, Heidelberg (2008)
10. Flocchini, P., Kellett, M., Mason, P., Santoro, N.: Map construction and exploration by mobile agents scattered in a dangerous network. In: *IPDPS 2009*, pp. 1–10 (2009)
11. Flocchini, P., Mans, B., Santoro, N.: Exploration of periodically varying graphs. In: Dong, Y., Du, D.-Z., Ibarra, O. (eds.) *ISAAC 2009*. LNCS, vol. 5878, pp. 534–543. Springer, Heidelberg (2009)
12. Glaus, P.: Locating a black hole without the knowledge of incoming links. In: Lerner, A. (ed.) *ALGOSENSORS 2009*. LNCS, vol. 5804, pp. 128–138. Springer, Heidelberg (2009)
13. Klasing, R., Markou, E., Radzik, T., Sarracco, F.: Hardness and approximation results for black hole search in arbitrary networks. *Theoretical Computer Science* 384(2-3), 201–221 (2007)
14. Klasing, R., Markou, E., Radzik, T., Sarracco, F.: Approximation bounds for black hole search problems. *Networks* 54(4), 216–226 (2008)
15. Kosowski, A., Navarra, A., Pinotti, M.C.: Synchronization helps robots to detect black holes in directed graphs. In: Abdelzaher, T., Raynal, M., Santoro, N. (eds.) *OPODIS 2009*. LNCS, vol. 5923, pp. 86–98. Springer, Heidelberg (2009)
16. Liu, C., Wu, J.: Scalable routing in cyclic mobile networks. *IEEE Trans. Parallel Distrib. Syst.* 20(9), 1325–1338 (2009)
17. O’Dell, R., Wattenhofer, R.: Information dissemination in highly dynamic graphs. In: *2005 Joint Work. on Foundations of Mobile Computing*, pp. 104–110 (2005)
18. Zhang, X., Kurose, J., Levine, B., Towsley, D., Zhang, H.: Study of a bus-based disruption-tolerant network. In: *13th Int. Conf. on Mobile Computing and Networking*, p. 206 (2007)
19. Zhang, Z.: Routing in intermittently connected mobile ad hoc networks and delay tolerant networks: Overview and challenges. *IEEE Communications Surveys & Tutorials* 8(1), 24–37 (2006)

Cracking Bank PINs by Playing Mastermind*

Riccardo Focardi and Flaminia L. Luccio

Università Ca' Foscari Venezia
{focardi,luccio}@dsi.unive.it

Abstract. The bank director was pretty upset noticing Joe, the system administrator, spending his spare time playing Mastermind, an old useless game of the 70ies. He had fought the instinct of telling him how to better spend his life, just limiting to look at him in disgust long enough to be certain to be noticed. No wonder when the next day the director fell on his chair astonished while reading, on the newspaper, about a huge digital fraud on the ATMs of his bank, with millions of Euros stolen by a team of hackers all around the world. The article mentioned how the hackers had ‘played with the bank computers just like playing Mastermind’, being able to disclose thousands of user PINs during the one-hour lunch break. That precise moment, a second before falling senseless, he understood the subtle smile on Joe’s face the day before, while training at his preferred game, Mastermind.

Keywords: Security APIs, PIN processing, Hardware Security Modules, Mastermind.

1 Introduction

The Mastermind game was invented in 1970 by Mordecai Meirowitz. The game is played as a board game between two players or as a one player game between a single player and the computer (in both cases called the *codebreaker* and the *codemaker*, respectively) [19]. The codemaker chooses a linear sequence of colored pegs and conceals them behind a screen. Duplicates are allowed. The codebreaker has to guess, in different trials, both the color and the position of the pegs. During each trial he learns something and based on this he decides the next guess: in particular, a response consisting of a *black peg* (which we will call *black marker*) represents a right guess of the color and the position of a peg (but the marker does not indicate which one is correct), a response consisting of a *white peg* (called *white marker*) represents only the right guess of a color but at the wrong position.

An apparently completely unrelated problem is the one of protecting user’s Personal Identification Number (PIN) when withdrawing some money at an Automated Teller Machine (ATM). International bank networks are structured in such a way that an access to an ATM implies that the user’s PIN is sent to the issuing bank for the verification. While travelling, the PIN is decrypted and

* Work partially supported by Miur’07 Project SOFT: “*Security Oriented Formal Techniques*”.

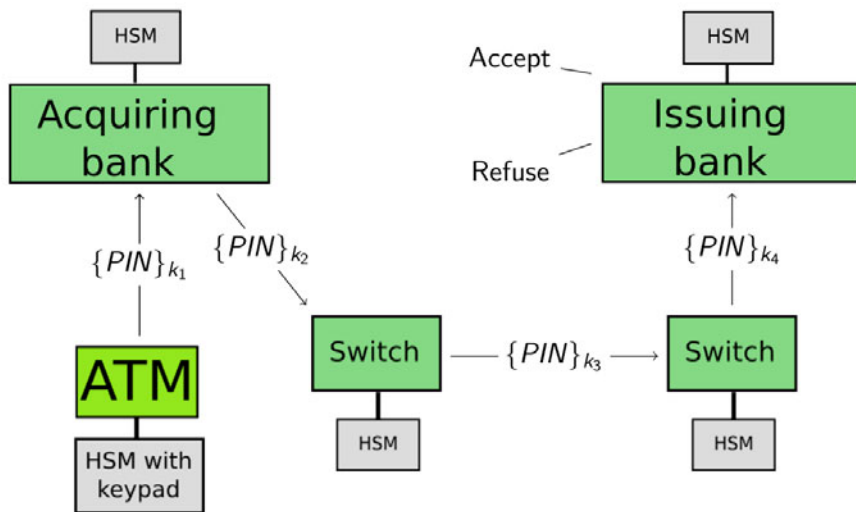


Fig. 1. Bank network

re-encrypted by special tamper-resistant devices called Hardware Security Modules (HSMs) which are placed on the traversed network switches, as illustrated in figure 1. The first PIN encryption is performed by the ATM keypad which is an HSM itself, using a symmetric key k_1 shared with the neighbour acquiring bank. While travelling from node to node, the encrypted PIN is decrypted and re-encrypted with another key shared with the destination node, by the HSM located in the switch. The final verification and acceptance/refusal of the PIN is done by the issuing bank.

Although this setting seems to be secure, several API-level attacks have been discovered on these HSMs in the last years [5,6,10]. These attacks work by assuming that the attacker is an insider gaining access to the HSM at some bank switch and performing subtle sequences of API calls from which he is able to deduce the value of the PIN. There are many examples of such attacks, the one we are considering in this paper is the so-called *dectab attack* [6], which we will illustrate in the detail in the next section. Intuitively, while verifying the PIN, the PIN verification API at the issuing bank HSM takes as an input different parameters, some of which are public. One of these parameters is a decimalization table that maps an intermediate hexadecimal representation of the user PIN into a decimal number. By manipulating some information, e.g., by modifying the way numbers are decimalized and by observing if this affects the result of the verification, the attacker can deduce which are the actual PIN digits. The position of the guessed PIN digits is reconstructed by manipulation another public parameter, i.e., the *offset* of the PIN. By combining all this information the attacker is able to reconstruct the whole PIN.

Our contribution. In this paper we show that decimalization attacks can be seen as playing an extended Mastermind game. Each API call represents a trial

of the codebreaker and the API return value is the corresponding answer of the codemaker. Manipulating the dectab and the offset together is similar to asking the codemaker to disclose the color and the position of one PIN digit, in case the guess is correct, similarly to what happens with a black marker of Mastermind. Modifying the dectab only, instead, corresponds to asking for the presence of certain digits in the PIN, analogously to the white marker in the game.

We make the above intuition formal by showing how PIN cracking and Mastermind can be seen as instances of a more general problem, or game. This extended problem suggests a new way of improving the dectab attack. The idea is to allow the player (i.e., the attacker) to ask for sets of colors (i.e., digits), instead of just single colors, for each position. This, in fact, can be implemented in the PIN cracking setting by modifying multiple entries of the dectab, as we will show in detail. We show that this reduces the known bounds on the number of average API calls for performing the attacks from 16.145 to 13.463 which is very close to the optimal value of 13.362.

To this aim, we develop a computer program that optimizes a well known technique presented by Knuth in [15] for the standard Mastermind game and extend it to our setting. We perform experiments showing that the program is almost as precise as state-of-the-art Mastermind solvers [13] but faster, being it able to compute strategies for cases not yet covered. More interestingly, the very same solving strategy is adapted to the PIN cracking problem proving the above mentioned new bound on the average number of API calls required in dectab attacks.

Paper structure. In section [1] we briefly summarize the related literature. In section 2 we formally define the two problems, i.e., the Generalized Mastermind Problem and the PIN Cracking Problem. In section 3 we introduce the Extended Mastermind Problem, i.e., a general problem whose instances are the Generalized Mastermind Problem and the PIN Cracking Problem. In section 4 we expose some experimental results, and we conclude in section 5.

1.1 Related Literature

Mastermind. In [15] Donald Knuth presented an algorithm for the solution of the standard Mastermind game, which is played using pegs of 6 different colors, in a sequence of length 4. He showed how the codebreaker can find the pattern in five moves or fewer, using an algorithm that progressively reduces the number of possible patterns. Each guess is made so that it minimizes the maximum number of remaining possibilities. The expected number of guesses is 4.478. In 1993 Kenji Koyama and Tony W. Lai proposed a technique that uses at most 6 guesses but decreases the expected number to 4.340 or to 4.341 if only 5 guesses are allowed [17]. In [4,14], the authors apply evolutionary and genetic algorithms to solve the Mastermind problem.

Different variants of the game have been proposed, e.g. in [9], Chvatal mentions a problem, suggested by Pierre Duchet, called the *static Mastermind*. This problem consists of finding the minimum number of guesses made all at once (i.e., without waiting for the responses), that are required to determine the code.

In [8] the authors propose a bound for finding a hidden code by asking questions. This problem relates to the Generalized Mastermind Game with N colors and sequences of length k . The authors show that $\lceil \frac{k}{N} \rceil + 2N \log N + 2N + 2$ guesses are sufficient to solve the problem.

Finally, in [13] the authors present some new bounds to the Generalized Mastermind Game. Using a computer program they compute some new exact values of maximum number of guesses. They also provide theoretical bounds for the case of sequences of length 2, 3 and 4, and for the general case of N colors and length k .

PIN cracking. API-level attacks on PINs have recently attracted attention from the media [13]. This has increased the interest in studying formal methods for analysing PIN recovery attacks and API-level attacks in general [18]. In particular, different models have been proposed, e.g., in [6] the authors prove that in average 16.5 API calls are required to reconstruct the PIN and this bound was decreased to 16.145 in [18]. In [7] we have presented, together with other authors, a language-based setting for analysing PIN processing API via a type-system. We have formally modelled existing attacks, proposed some fixes and proved them correct via type-checking. These fixes typically require to reduce and modify the HSM functionality by, e.g., sticking on a single format of the transmitted PIN or adding MACs for the integrity of user data. Notice, in fact, that the above mentioned attack is based on the absence of integrity on public user data such as the *dectab* and the *offset*. As upgrading the bank network HSMs worldwide is complex and very expensive in [11] we have also have proposed a low-impact, easily implementable fix requiring no hardware upgrade which makes attacks 50000 times slower, but yet not impossible.

2 The Two Problems

In this section we give a formal definition of the two problems we will be relating. We first define the *Generalized Mastermind Problem (GMP)*, i.e., the problem of solving a Generalized Mastermind Game, and we then present the problem of attacking a PIN using the decimalization table, and we call it the *PIN Cracking Problem (PCP)*.

2.1 The Generalized Mastermind Game

The Generalized Mastermind Problem is a game that is played between a player (the “codebreaker”) and a computer or another player (the “codemaker”). The codemaker chooses a linear sequence of k colored pegs, which we call *secret* and conceals them behind a screen. The colors range in a set $\{0, 1, \dots, N - 1\}$. The codebreaker has to guess the secret, i.e., both the color of the pegs and their exact position. The game is played in steps, each of which consists of a guess of the codebreaker and a response of the codemaker. The response can be empty, can contain a black or a white marker, i.e., is a sequence of at most 4 markers chosen in the set $\{B, W\}$. The black marker represents a correct guess both of



Fig. 2. An example of a Mastermind game

the color and the position of a peg, there is no indication however of its position, the white marker only represents the correct guess of the color.

An example of the standard Mastermind game, i.e., played with $N = 6$ colors and $k = 4$ pegs, is shown in figure 2 taken from [2]. In this example black markers are depicted in red. We have added numbers to identify different colors. At the first step the codebreaker only finds a right color, i.e., a cyan peg (2), in a wrong position, thus the response is a white marker, i.e., W . At the next step he correctly guesses a red peg (3) in the right position and a purple peg (4) in a wrong position, thus the response is a black and a white peg, i.e., B, W , an so on. At the last step the response are 4 black markers, i.e., B, B, B, B .

Note that in the standard Mastermind game the set of all possible solutions has size 6^4 , in the Generalized Mastermind Game the size explodes to N^k , thus running plain exhaustive search techniques might become problematic when N and k increase too much.

We can now formulate our problem.

The Generalized Mastermind Problem (GMP). Given a Generalized Mastermind Game played on N colors and k pegs, devise a minimal sequence of guesses for the correct disclosure of the secret.

2.2 API-Level Attacks in Bank Networks

In this section we show in detail a real API-level attack to the bank PINs. As we have mentioned in the introduction, a PIN travelling along the network has to be decrypted and re-encrypted under a different key, and this is done using a so called *translation* API. While the PIN reaches the issuing bank, its correspondence with the *validation data*, i.e., a value that is typically an encoding of

Table 1. The *verification* API

```

PIN_V(PAN, EPB, len, offset, vdata, dectab) {
   $x_1 := \text{enc}_{pdk}(vdata)$ ;
   $x_2 := \text{left}(len, x_1)$ ;
   $x_3 := \text{decimalize}(dectab, x_2)$ ;
   $x_4 := \text{sum\_mod10}(x_3, offset)$ ;
   $x_5 := \text{dec}_k(EPB)$ ;
   $x_6 := \text{fcheck}(x_5)$ ;
  if ( $x_6 = \perp$ ) then return("format wrong");
  if ( $x_4 = x_6$ ) then return("PIN correct");
  else return("PIN wrong")}

```

the user Personal Account Number (PAN) and possibly other ‘public’ data, such as the card expiration date or the customer name, is checked via a *verification* API. We focus on this latter API, called `PIN_V` and reported in table 1, that checks the equality of the actual *user* PIN, derived through the PIN derivation key *pdk*, from the public data *offset*, *vdata*, *dectab*, and the *trial* PIN inserted at the ATM that arrives encrypted under key *k* as *EPB* (Encrypted PIN block). The API returns the result of the verification or an error code.

`PIN_V` behaves as follows:

- The user PIN of length *len* is computed by first encrypting validation data *vdata* with the PIN derivation key *pdk* (x_1) and obtaining a 16 hexadecimal digit string. Then, the first *len* hexadecimal digits are chosen (x_2), and decimalised through *dectab* (x_3), obtaining the ‘natural’ PIN assigned by the issuing bank to the user. `decimalize` is a function that associates to each possible hexadecimal digit (of its second input) a decimal one as specified by its first parameter (*dectab*). Finally, if the user wants to choose her own PIN, an *offset* is calculated by digit-wise subtracting (modulo 10) the natural PIN from the user-selected one (x_4).
- To recover the trial PIN *EPB* is first decrypted with key *k* (x_5), then the PIN is extracted by the formatted decrypted message (x_6). This last operation depends on the specific PIN format adopted by the bank. In some cases, for example, the PIN is padded with random digits so to make its encryption immune from codebook attacks. In this case, extracting the PIN involves removing this random padding.
- Finally, if x_6 fails (\perp represents failure) then a message is returned, moreover the equality between the user PIN and the trial PIN is verified.

An API attack on `PIN_V`. We now illustrate a real attack on `PIN_V` first reported in [6]. The attack works by iterating the following two steps, until the whole PIN is recovered:

1. To discover whether or not a decimal digit *d* is present in the user ‘natural’ PIN contained in x_3 the intruder picks digit *d*, changes the *dectab* function so that

values previously mapped to d now map to $d+1 \pmod{10}$, and then checks whether the system still returns ‘PIN correct’. If this is the case d is not contained in the ‘natural’ PIN.

2. To locate the position of the digit previously discovered by a ‘PIN wrong’ output the intruder also changes the *offset* until the API returns again that the PIN is correct.

We illustrate the attack through a simple example.

Example 1. Assume $len=4$, $dectab=5753108642143210$, as

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
5	7	5	3	1	0	8	6	4	2	1	4	3	2	1	0

$offset=4732$. The correct solution, unknown to the intruder, is the following.

$$\left. \begin{array}{l}
 x_2 = \text{left}(4, AD7295FDE32BA101) \\
 x_3 = \text{decimalize}(dectab, AD72) \\
 x_4 = \text{sum_mod10}(1265, 4732) \\
 x_5 = \text{dec}_k(\{5997, r\}_k) \\
 x_6 = \text{fcheck}(5997, r)
 \end{array} \right| \begin{array}{l}
 = AD72 \\
 = 1265 \\
 = 5997 \\
 = (5997, r) \\
 = 5997
 \end{array}$$

Since x_6 is different from \perp and $x_4 = x_6$, the API returns ‘PIN correct’.

The attacker, unaware of the value of the PIN, first changes the *dectab*, which is a public parameter, as $dectab'=5753118642143211$, i.e., it replaces the two 0’s by 1’s. The aim is to discover whether or not 0 appears in x_3 . Invoking the API with $dectab'$ we obtain $\text{decimalize}(dectab', AD72) = \text{decimalize}(dectab, AD72) = 1265$, that is 0 does not appear in x_3 . The attacker proceeds by replacing the 1’s of $dectab$ by 2’s: with $dectab''=5753208642243220$ he has $\text{decimalize}(dectab'', AD72) = 2265 \neq \text{decimalize}(dectab, AD72)=1265$, reflecting the presence of 1’s in the original value of x_3 . Then, $x_4=\text{sum_mod10}(2265, 4732) =6997$ instead of 5997 returning ‘PIN wrong’.

The intruder now knows that digit 1 occurs in x_3 , and to discover its position and multiplicity, he now varies the *offset* so to ‘compensate’ for the modification of the *dectab*. In particular, he tries to decrement each *offset* digit by 1. For example, testing the position of one occurrence of one digit amounts to trying the following *offset* variations: 3732, 4632, 4722, 4731. Notice that, in this specific case, *offset* value 3732 makes the API return again ‘PIN correct’.

The attacker now knows that the first digit of x_3 is 1. Given that the *offset* is public, he also calculates the first digit of the user PIN as $1 + 4 \pmod{10} = 5$.

We can now formulate our problem.

The PIN Cracking Problem (PCP). Given a bank network the *PCP* consists of recovering an encrypted (i.e., secret) PIN by devising a malicious sequence of calls to the *verification* API.

3 Extended Mastermind

We extend the Generalized Mastermind Problem presented in previous section, by allowing the codebreaker to pose an extended guess composed of k sets of colored pegs, instead of just k pegs. Intuitively, the sets represent alternative guesses, i.e., it is enough that one of the peg in the set is correct to get a black or a white marker.

More formally, let $\mathcal{C} = \{0, 1, \dots, N - 1\}$ be the set of colors. We note (S_1, S_2, \dots, S_k) , with $S_1, \dots, S_k \subseteq \mathcal{C}$, an extended guess, and (c_1, c_2, \dots, c_k) , with $c_1, \dots, c_k \in \mathcal{C}$, the secret.

Intuitively, the number of black markers represents the number of colors in the secret belonging to the corresponding set. Formally:

Definition 1 (Black markers). *The number b of black markers is computed as $b = |\{i \in [1, k] \mid c_i \in S_i\}|$.*

The number of white markers, instead, corresponds to the number of colors in the secret belonging to sets in the guess, but not the ones in the corresponding position. To formalize this we first compute the number of occurrences of a color $j \in \mathcal{C}$ in the secret code as $p_j = |\{i \in [1, k] \mid j = c_i\}|$, and in the guess as $q_j = |\{i \in [1, k] \mid j \in S_i\}|$. Now $\min(p_j, q_j)$ represents the number of matching pegs of color j . If we sum over all the colors we obtain the overall number of matching pegs. From this we need to subtract the ones giving black markers, in order to obtain the number of white markers.

Definition 2 (White markers). *The number w of white markers is computed as $w = \sum_{j=1}^N \min(p_j, q_j) - b$.*

Let show the above definitions with a simple example

Example 2. Let $N = 6$, $(1, 2, 3, 1)$ be the secret and $(1, 3, 1, 3)$ be the guess. We compute $b = |\{i \in [1, k] \mid c_i \in S_i\}| = |\{1\}| = 1$. In fact only the first ‘1’ is in the right position, giving a black marker. Then we have

$$\begin{array}{lcl}
 p_0 = |\{\}\!| & = & 0 \\
 p_1 = |\{1, 4\}\!| & = & 2 \\
 p_2 = |\{2\}\!| & = & 1 \\
 p_3 = |\{3\}\!| & = & 1 \\
 p_4 = |\{\}\!| & = & 0 \\
 p_5 = |\{\}\!| & = & 0
 \end{array}
 \quad \left| \quad \begin{array}{lcl}
 q_0 = |\{\}\!| & = & 0 \\
 q_1 = |\{1, 3\}\!| & = & 2 \\
 q_2 = |\{\}\!| & = & 0 \\
 q_3 = |\{2, 4\}\!| & = & 2 \\
 q_4 = |\{\}\!| & = & 0 \\
 q_5 = |\{\}\!| & = & 0
 \end{array}$$

Now $\sum_{j=1}^N \min(p_j, q_j) = 3$ meaning there are 3 matching pegs (the two 1’s and one of the 3), but one of them is already counted as a black. Thus we obtain $w = 3 - b = 2$. Notice that the two 3’s in the guess are counted just once, as only one 3 appears in the secret code. This is why we need to take $\min(p_j, q_j)$.

¹ We omit the set notation for singletons, i.e., we write $(1, 3, 1, 3)$ in place of $(\{1\}, \{3\}, \{1\}, \{3\})$.

To see how this scales to set consider the extended guess $(1, 3, 1, \{1, 3\})$. In this case we have $b = 2$ (the first and the last pegs) and $w = 3 - b = 1$, i.e., there is one peg in the wrong position (i.e., the ‘3’).

Definition 3 (The Extended Mastermind Problem - EMP). *Given an Extended Mastermind Game played on N colors and k pegs, devise a minimal sequence of guesses for the correct disclosure of the secret.*

We now show how the two previous problem can be seen as instances of EMP.

Lemma 1. *GMP is an instance of EMP.*

Proof. It is sufficient to restrict sets in guesses to singletons to recover the Generalized Mastermind Game.

More interestingly, we see how the PIN cracking problem can be seen as particular instance of GMP.

Lemma 2. *PCP is an instance of EMP.*

Proof. We restrict the extended guesses (S_1, \dots, S_k) so that $\bigcap_{i=1}^k S_i = \emptyset$, i.e., the sets are disjoint. This is related to how the guess is implemented via the PIN verification API. Given a guess (S_1, \dots, S_k) , with disjoint sets, we modify the decTab of each digit $d \in S_i$ by mapping it into $d+i$. This mapping is well-defined given that sets S_i are disjoint. At the same time we modify the i -th digit of the offset by decreasing it by i . As a result, since we have changed the offset, the only way to obtain a ‘PIN correct’ is that the digit of the PIN at the i -th position has been increased by i and this only holds if it appears in S_i . Iterating this on all the sets we easily see that ‘PIN correct’ corresponds to having $c_i \in S_i$ for all $i = 1, \dots, k$, i.e., having four black markers. Thus, we say that the codemaker answers ‘yes’ when the answer is 4 black markers and ‘no’ otherwise. Notice that the player can use extended guesses and ask for sets of values and not just singletons, so it is not necessary to guess the exact code to get a ‘yes’.

4 Experimental Results

We have devised a program which is an optimized extension of the original program for Mastermind presented by Knuth in [15]. It works as follows:

1. Tries all the possible guesses. For each guess, computes the number of ‘surviving’ solutions related to each possible outcome of the guess;
2. Picks the guess from the previous step which minimizes the maximum number of surviving solutions among all the possible outcomes and performs the guess:
 - (a) For each possible outcome, stores the corresponding surviving solutions and recursively calls this algorithm;
 - (b) stops whenever the number of surviving solutions is 0 (impossible outcome) or 1 (guessed the right sequence).

Table 2. Our optimization of Knuth’s algorithm

Colours/Pegs	2	3	4	5	6	7	8	9	10
2		3	4	4	5	6	6	7	8
3		4	4	4	4	5	6	6	
4		4	4	4	5	6			
5		5	5	5					
6		5	5	5					
7		6	6	6					
8		6	6	6					
9		7	7	7					
10		7	7	8					

Table 3. Bounds from [13]

Colours/Pegs	2	3	4	5	6	7	8
2		3	3	4	4	5	5
3		3	4	4	4	5	5
4		4	4	4	5	5	
5		4	5	5	5		
6		5	5	5			
7		5	6	6			
8		6	6	6			
9		6	6	7			
10		7	7	7			

In order to reduce the complexity of the exhaustive search over all possible guesses we have implemented an optimization which starts working on a subset of the colors (the one used up to the current guess) and adds new colors only when needed by the guesses. This is similar, in the spirit, to what is done in [13].

We first show some results obtained by running this optimized algorithm to the Generalized Mastermind Problem. Note that most computations took few seconds, others few minutes, and we were also able to find new upper bounds on the minimal number of moves for unknown values (see Table 2, values in bold). As a matter of fact, as it is mentioned in [12], Knuth’s idea does not define an optimal strategy, it is however very close to the optimal. In [13] some empirical optimal values were computed (see Table 3) and some theoretical bounds were presented. Note that our values differ at most by one from the exact ones. Moreover, we were able to efficiently find bounds on 2 colors and 9 and 10 pegs and 3 colors and 8 pegs, and to list the exact sequence of moves to be followed, whereas the program of [13], as the authors state, would probably take “many weeks” of computation.

We have then applied the very same algorithm to the PIN cracking problem. In this case we have noticed that using sets with more than two elements in the guesses did not improve the solutions. With sets of size at most two the

algorithm performs quite well and we have been able to slightly improve the results of [18] by finding a strategy with an average number of calls of 15.2 instead of 16.145. This improvement is based on the idea of extended guesses, in which sets of values can be queried by simultaneously changing their mapping in the dectab of a same quantity. This idea is new, and extends the attack strategy illustrated in [6] and studied in [18]. We have then extended the algorithm so to also consider a special ‘dectab-only’ API call, where the offset is left untouched. This kind of call, exploited in [6,18], allows for discovering whenever a digit appears as one of the PIN digits. By extending this kind of call to sets we have been able to lower the bound to an average of 13.463 calls with a worst case of 15. We also found a new bound for PINs of length 5 giving an average of 16.81 calls and a worst case of 19. It is worth noticing that our results are very close to the optimal partitioning of the solutions into an almost-balanced binary tree. In fact, it can be easily computed that this would give a number of average calls of 13.362 and 16.689 for PINs of length 4 and 5, respectively.

All the files containing the detailed strategies for Mastermind and PIN cracking can be downloaded at http://www.dsi.unive.it/~focardi/MM_PIN/ .

5 Conclusion

In this paper we have considered two rather different problems, Mastermind and PIN cracking, and we have shown how they can be seen as instances of an extended Mastermind game in which guesses can contain sets of pegs. We have implemented an optimized version of a classic solver for Mastermind and we have applied it to PIN cracking, improving the known bound on the number of API calls. The idea of using sets in the guesses has in fact suggested a new attacking strategy that reduces the number of required calls. By combining ‘standard’ attacks with this new strategy we have been able to reduce the average number of calls from 16.145 to 13.463, with a new worst case of 15. We also found a new bound for PINs of length 5 giving an average of 16.81 calls and a worst case of 19. Both average cases are very close to the optimum.

As a future work we intend to study the extension of more involved techniques such as the ones of [13] to the PIN cracking setting. As a final note, we would solicit the bank director of our abstract, and other serious people to be more open-minded and never assume that something is useless just because it is funny, “sooner or later society will realize that certain kinds of hard work are in fact admirable even though they are more fun than just about anything else” [16].

Acknowledgements. We would like to thank Graham Steel for his helpful comments and suggestions.

References

1. Hackers crack cash machine PIN codes to steal millions. The Times online, http://www.timesonline.co.uk/tol/money/consumer_affairs/article4259009.ece
2. Mastermind, <http://commons.wikimedia.org/wiki/File:Mastermind.jpg>

3. PIN Crackers Nab Holy Grail of Bank Card Security. Wired Magazine Blog 'Threat Level', <http://blog.wired.com/27bstroke6/2009/04/pins.html>
4. Bento, L., Pereira, L., Rosa, A.: Mastermind by evolutionary algorithms. In: Proc. ACM Symp. Applied Computing, San Antonio, Texas, February 28-March 2, pp. 307–311. ACM Press, New York (1999)
5. Berkman, O., Ostrovsky, O.M.: The unbearable lightness of PIN cracking. In: Dietrich, S., Dhamija, R. (eds.) FC 2007 and USEC 2007. LNCS, vol. 4886, pp. 224–238. Springer, Heidelberg (2007)
6. Bond, M., Zielinski, P.: Decimalization table attacks for pin cracking. Technical Report UCAM-CL-TR-560, University of Cambridge, Computer Laboratory (2003), <http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-560.pdf>
7. Centenaro, M., Focardi, R., Luccio, F., Steel, G.: Type-based analysis of PIN processing APIs. In: Backes, M., Ning, P. (eds.) ESORICS 2009. LNCS, vol. 5789, pp. 53–68. Springer, Heidelberg (2009)
8. Chen, Z., Cunha, C., Homer, S.: Finding a hidden code by asking questions. In: Cai, J.-Y., Wong, C.K. (eds.) COCOON 1996. LNCS, vol. 1090, pp. 50–55. Springer, Heidelberg (1996)
9. Chvatal, V.: Mastermind. *Combinatorica* 3, 325–329 (1983)
10. Clulow, J.: The design and analysis of cryptographic APIs for security devices. Master's thesis, University of Natal, Durban (2003)
11. Focardi, R., Luccio, F.L., Steel, G.: Blunting differential attacks on PIN processing APIs. In: Jøsang, A., Maseng, T., Knapskong, S.J. (eds.) NordSec 2009. LNCS, vol. 5838, pp. 88–103. Springer, Heidelberg (2009)
12. Goddard, W.: Mastermind revisited. *J. Combin. Math. Combin. Comput.* 51, 215–220 (2004)
13. Jäger, G., Pezarski, M.: The number of pessimistic guesses in generalized mastermind. *Information Processing Letters* 109, 635–641 (2009)
14. Kalisker, T., Camens, D.: Solving mastermind using genetic algorithms. In: Cantú-Paz, E., et al. (eds.) GECCO 2003. LNCS, vol. 2724, pp. 1590–1591. Springer, Heidelberg (2003)
15. Knuth, D.: The Computer as a Master Mind. *Journal of Recreational Mathematics* 9, 1–6 (1976)
16. Knuth, D.E.: *The Stanford GraphBase: a platform for combinatorial computing*. Addison-Wesley Professional, Reading (1993)
17. Koyama, M., Lai, T.: An Optimal Mastermind Strategy. *Journal of Recreational Mathematics* 25, 251–256 (1993)
18. Steel, G.: Formal Analysis of PIN Block Attacks. *Theoretical Computer Science* 367(1-2), 257–270 (2006)
19. Stuckman, J., Zhang, G.: Mastermind is NP-Complete. *INFOCOMP Journal of Computer Science* 5, 25–28 (2006)

Computational Complexity of Two-Dimensional Platform Games

Michal Forišek*

Comenius University, Bratislava, Slovakia
forisek@dcs.fmph.uniba.sk

Abstract. We analyze the computational complexity of various two-dimensional platform games. We state and prove several meta-theorems that identify a class of these games for which the set of solvable levels is NP-hard, and another class for which the set is even PSPACE-hard. Notably *COMMANDERKEEN* is shown to be NP-hard, and *PRINCEOFPERIA* is shown to be PSPACE-complete.

We then analyze the related game *Lemmings*, where we construct a set of instances which only have exponentially long solutions. This shows that an assumption by Cormode in [3] is false and invalidates the proof that the general version of the *LEMMINGS* decision problem is in NP. We then augment our construction to only include one entrance, which makes our instances perfectly natural within the context of the original game.

1 Overview

The area of two-player combinatorial games, and one-player games and puzzles has already been well researched. The earliest mathematical results are more than a century old (e.g., the analysis of the game *NIM* [1]). The most important mathematical results related to two-player combinatorial games are the Sprague-Grundy theory [18,8] and Conway’s surreal numbers [2].

In recent decades researchers returned to this area with a new point of view: investigating the computational complexity of these puzzles and games. It turned out that almost all “interesting” puzzles and games are hard – if $P \neq NP$ then there is no efficient algorithm to solve/play them optimally. It was shown that many of the two-player games are EXPTIME-complete (e.g., the generalizations of *Go* and *Checkers* to an $n \times n$ board, [15,16]), and many single-player puzzles are either PSPACE-complete (e.g., solving *Sokoban* [4]) or NP-complete (e.g., checking whether a *Minesweeper* configuration is valid [10]). For more of these results, we recommend one of the survey papers [11,6,7].

In Section 2 of this article we analyze the computational complexity of a class of previously ignored single-player games: two-dimensional platform games.

* This work was supported by the grant VEGA 1/0726/09

¹ A convention for easier reading of this article: Throughout the entire article, italics (*SomeGame*) are used for the names of actual games, and small capitals (SOMEGAME) are used for the names of the underlying decision problems.

One of the already researched games is the game *Lemmings*, initially addressed by Cormode [3] and later by Spoerer [17]. In Section 3 we continue in the analysis of this game, disproving an assumption made by Cormode, and thereby invalidating his proof that LEMMINGS is in NP.

2 Hardness of 2D Platform Games

When researching the computational complexity of a given puzzle, we usually expect the puzzle to be hard – after all, the puzzles are designed with the goal to challenge the solver and to push her cognitive processes to the limit. However, when we turn to computer games, not all of them are puzzles. Many other games are perceived as simple in terms of necessary thinking. An example of such a set of games are the 2-dimensional platform games. In this section we will take a closer look at the computational complexity of these games, and surprisingly we will show that many of these games are actually very difficult to solve in general.

For many platform games we will prove that the set of all solvable instances (levels) is difficult: in some cases NP-hard, in some cases even PSPACE-hard. We will now define 2-dimensional platform games, identify a set of common features they exhibit, and then prove that some subsets of these features imply that solving the game is difficult.

A 2-dimensional platform game is a single-player game in which the player sequentially solves a set of levels. Each level is represented by a 2-dimensional map representing a vertical slice of a virtual world.² The player’s goal is to move her avatar (i.e., the game character controlled by her) from its starting location into the designated final location. The player controls the avatar by issuing simple commands (step, jump, climb up/down, etc.), usually by pressing keys or buttons of an input device. Within the game world, the avatar is affected by some form of gravity. As a consequence the maximum jump height is limited.

Additionally, many of these games share the following features:

- **long fall:** The height of the longest safe fall (that does not hurt the avatar) is larger than the maximum jump height.
- **opening doors:** The game world may contain a variable number of doors and suitable mechanisms to open them.
- **closing doors:** The game world contains a mechanism to close doors, and a way to force the player to trigger such a mechanism.
- **collecting items:** The game world contains items that must be collected.
- **enemies:** The game world contains enemy characters that must be killed or avoided in order to solve the level.

We will now show that some of these features are easy from the algorithmic point of view, but others imply that solving the game automatically is hard – regardless of other details of the particular game. These results can be seen as “meta-theorems”: for any particular game we can take the construction and adjust it to the details of the particular game.

² The world usually consists of horizontal platforms, hence the name “platform game”.

Meta-theorem 1. *A 2D platform game where the levels are constant and there is no time limit is in P , even if the **collecting items** feature is present.*

Proof. The level and the set of allowed movements of the avatar uniquely determine a directed reachability graph. In this graph we label the vertices that correspond to locations of items. Additionally, we execute two depth-first searches to determine the set of vertices that are both reachable from the entrance and allow us to reach the exit. If there is a labeled vertex that is not in this set, we reject. Otherwise we drop the vertices that are not in the reachable set and then contract each strongly connected component into a single vertex, preserving labels. We accept iff all the labeled vertices in the resulting DAG lie on a single path from the entrance to the exit. This can be verified by considering the labeled vertices in topological order and verifying reachability for each adjacent pair. The algorithm is linear in the instance size if implemented properly. \square

Meta-theorem 2. *A 2D platform game where the **collecting items** feature is present and a time limit is present as a part of the instance is NP-hard.*

Proof. Itai et al. in [9] prove that the set of grid graphs with Hamilton cycles is NP-complete. This problem can be reduced to solving our platform game as follows: Given a grid graph G , locate the leftmost of its topmost vertices and call it u . The vertex u has degree at most 2. If this degree is less than 2, we reject. (I.e., produce an unsolvable instance of our game and terminate.) Otherwise, let v be the right neighbor of u . Obviously the edge uv must be a part of every Hamilton cycle in G . Hence there is a Hamilton cycle in G iff there is a Hamilton path that starts at u and ends at v . We now design the level in such a way that the map of the level corresponds to G . We place an item into each location that represents a vertex of G , except for u and v . We place the level entrance at u , the level exit at v , and set the time limit to $\alpha(|V_G| - 1)$, where α is the time needed to travel between any two adjacent locations. \square

Meta-theorem 3. *Any 2D platform game that exhibits the features **long fall** and **opening doors** is NP-hard.*

Proof. We will show how to reduce 3-CNF-SAT to such a game. The main idea of the proof is that the door opening mechanism can be used for a non-local transfer of information. Given an instance of 3-CNF-SAT, we can construct an instance of the game as follows:

The instance will be divided into two parts: the key part, where the avatar starts, and the door part it reaches after exiting the key part. The door part will be a vertical concatenation of door gadgets (Figure 1 left), each corresponding to a single clause in the 3-CNF-SAT instance. The door gadget for the clause $c_n \equiv (l_{n,1} \vee l_{n,2} \vee l_{n,3})$ contains three doors labeled (n, l_1) , (n, l_2) , and (n, l_3) .

The key part will be a vertical concatenation of variable gadgets (Figure 1 right). Each variable gadget will correspond to a single variable used in the 3-CNF-SAT instance. For the variable x_i the keys represent mechanisms that unlock doors. The keys in the left part unlock the doors (n, x_i) for all n , and the keys in the right part unlock the doors $(n, \neg x_i)$ for all n .

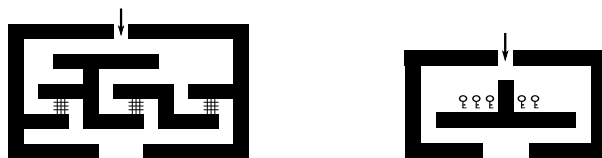


Fig. 1. The door gadget (left) and the variable gadget (right)

Clearly each valid solution of the 3-CNF-SAT instance corresponds to a valid way how to traverse the level. On the other hand, if the vertical unit size of our gadgets exceeds the maximum jump height, we can easily verify that the player has no other reasonable choice. Finally, the game level can easily be created from the 3-CNF-SAT instance in polynomial time. \square

Note 1. Both gadgets in Figure 1 were designed so that no jumping is necessary anywhere in the final instance. Hence Meta-theorem 3 does also apply to games that do not include any jumping. For example, it can be extended to dungeon exploration games that involve one-way trapdoors to lower levels.

Corollary 1. *The following famous 2D platform games are NP-hard:*

Commander Keen, Crystal Caves, Secret Agent, Bio Menace (switches that activate moving platforms), *Jill of the Jungle, Hocus Pocus* (switches that toggle walls), generalized³ *Duke Nukem* (keycards), *Crash Bandicoot, Jazz Jackrabbit 2* (crates that activate sets of new floor tiles once broken).

2.1 Prince of Persia Is PSPACE-Complete

In this section we state our main meta-theorem. Instead of a generic proof we opt to present the construction for the popular platform game Prince of Persia.

Meta-theorem 4. *Any 2D platform game that exhibits the features **long fall**, **opening doors** and **closing doors** is PSPACE-hard.*

Proof. The proof is a natural generalization of the proof presented below. \square

We will now define the PRINCEOFPERISIA decision problem. An instance of this problem is a 2-dimensional map of the level, with some additional information. On the map, each cell contains one of a fixed set of tiles. For our construction, we need the following types: {nothing, entrance, exit, floor tile, floor tile with a pressure plate, door}⁴. Additionally, the doors have unique labels from some set \mathcal{D} , and each pressure plate is assigned a single label: “ d ” if it opens door d , “ $-d$ ” if it closes the door d . Multiple plates may open/close the same door. The avatar (Prince) moves in the following ways:⁵ single step left/right; jump over

³ The original only has keycards of 4 colors and it is in P. From Meta-theorem 1 it follows that the set of perfectly solvable Duke Nukem levels is in P as well.

⁴ The actual game includes other tiles as well, such as walls and spikes.

⁵ Again, the description is simplified, but the differences do not matter for our proof.

at most three empty tiles left/right; climb to a floor tile diagonally above, if the above tile is empty; and descend into a pit ≤ 2 tiles deep.

Theorem 1. *The set PRINCEOFPERSIA of solvable instances of the game defined above is PSPACE-hard.*

Proof. To prove that PRINCEOFPERSIA is PSPACE-hard, we reduce the word problem for linear bounded automata WORDLBA to our problem.

Consider an instance (M, w) of WORDLBA with $M = (Q, \Sigma, \Gamma, \delta, q_0, L, R, F)$ and $|w| = n$. W.l.o.g. we may assume that the input and work alphabets are binary, i.e., $\Sigma = \Gamma = \{0, 1\}$. Q is the set of states, $F \subseteq Q$ is the set of final states. The symbols L and R are the left and right endmarker, respectively, δ is the non-deterministic transition function, and w is the input word.

We will now construct an instance of PRINCEOFPERSIA that will be solvable if and only if M accepts w . The general layout of our instance is shown in Figure 2. The instance will consist of several gadgets. The gadgets will be designed in such a way that the only allowed way in which the avatar will be able to navigate the level will correspond to arrows in Figure 2.

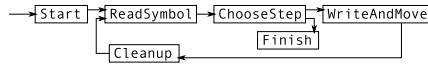


Fig. 2. Layout of gadgets in the PRINCEOFPERSIA instance

Our instance will contain several sets of doors that will represent various parts of the given word and LBA:

- the “accept” door a ,
- two sets of “head location” doors $\{h_i, \overline{h_i} \mid 0 \leq i \leq n + 1\}$,
- a set of “stored symbol” doors $\{s_{i,j} \mid 1 \leq i \leq n, j \in \{0, 1\}\}$,
- a set of “ δ -function state” and “ δ -function symbol” doors: $\{q_\pi, x_\pi \mid \pi \in \delta\}$,
- a set of “head movement” doors $\{m_{i,j} \mid 0 \leq i \leq n + 1, -1 \leq j \leq 1\}$,
- and a set of “write symbol at location” doors $\{w_{i,j} \mid 1 \leq i \leq n, j \in \{0, 1\}\}$.

The gadgets shown in Figure 2 will be constructed in such a way that the following invariant holds for each x : Let S be the set of doors open at the moment when the avatar enters the ReadSymbol gadget for the x -th time. Then there is a configuration (q, \overline{w}, k) of M such that:

- (q, \overline{w}, k) is reachable in $x - 1$ steps from $(q_0, w, 1)$,
- out of the head location doors, only doors h_k and $\overline{h_k}$ may be open,
- out of the stored symbol doors, only doors s_{i, \overline{w}_i} may be open, for all i ,
- out of the state doors q_π , only doors that correspond to q may be open,
- all other doors, including the accept door a , are closed.

From this invariant it immediately follows that the only way in which the avatar can solve the level is by simulating an accepting computation of M . The only thing left to prove is to show how to construct the individual gadgets that will enforce the above invariant.

Figure 3 shows the set of tiles we will use in our constructions, and the construction of two helper gadgets. The one in the center (denoted MC below) forces the avatar to choose one of the available paths, without the option to return later and change the choice. The right one (denoted $FA(x)$) forces the avatar to activate the pressure plate $-x$. This gadget will be used to enforce closing doors. Without the pressure plate this gadget can be used as a simple one-way wire (denoted by an arrow outside of a box). Using these helper gadgets we will now construct the gadgets in Figure 2.

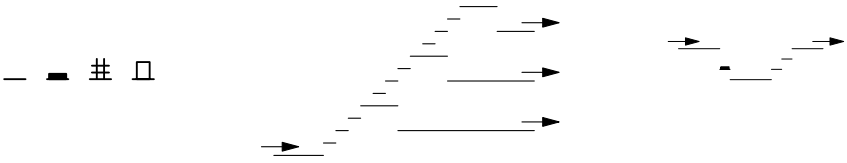


Fig. 3. Left: the set of four tiles we use – floor, pressure plate, door, and entrance/exit. Center and right: helper gadgets: multiple choice and forced activation of a plate.

Three of the gadgets are simple. The Start gadget contains pressure plates that open doors corresponding to the configuration $(q_0, w, 1)$. The Finish gadget contains the door a , blocking the way to the level exit. The Cleanup gadget consists of a series of FA gadgets that close all of the doors $m_{i,j}$, $w_{i,j}$, and x_π .

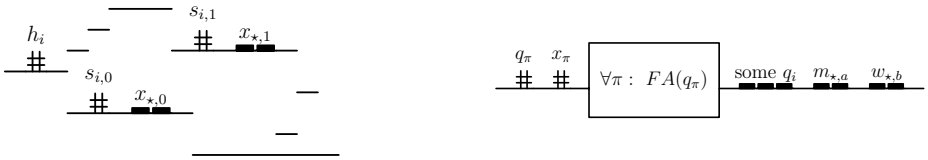


Fig. 4. Gadgets: a part of ReadSymbol (left); a part of ChooseStep (right)

The ReadSymbol gadget contains a MC gadget with $n + 2$ outputs. Each of these outputs starts with one of the h_i doors. The rest of the gadget for one h_i door is shown in Figure 4 on the left. (Doors h_0 and h_{n+1} are special, their parts do not contain the symbol doors.)

Clearly, if the avatar wants to traverse this gadget, it must follow a path that corresponds to the LBA’s current head location and the symbol underneath the head. On this path, the avatar is allowed to open any or all of those x_π doors that correspond to the current symbol.

The ChooseStep gadget contains a MC gadget with $|\delta|$ outputs, each of them corresponding to one element $\pi \in \delta$. When crossing this gadget the avatar is forced to choose one of the (possibly multiple) paths that correspond to elements of δ for the LBA’s current state and read symbol. If the new chosen state is final, the avatar is allowed to open the door a . Otherwise the avatar is forced to enter a gadget shown in Figure 4 on the right. When crossing this gadget the avatar

is forced to close all open q_i doors, and then allowed to open those doors q_i , $m_{i,a}$ and $w_{i,b}$ that correspond to the new state, symbol b to write, and head movement a .

The WriteAndMove gadget contains a MC gadget with $n + 2$ outputs. Each output starts with one of the $\overline{h_i}$ doors, which forces the avatar to pick a path corresponding to the old head location. The rest of the gadget for a single output is shown in Figure 5. In the first part (omitted for head positions 0 and $n + 1$) the avatar crosses the correct write symbol door $w_{i,b}$, closes the door $s_{i,1-b}$ and opens $s_{i,b}$. In the second part the avatar closes the current head location doors h_i and $\overline{h_i}$, crosses the correct movement door $m_{i,a}$, and opens the new head location doors h_{i+a} and $\overline{h_{i+a}}$.

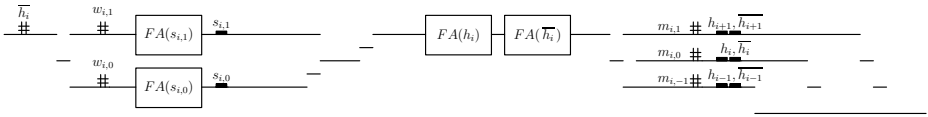


Fig. 5. Gadget: a part of WriteAndMove

That concludes the construction. An example of a full instance constructed in this way is shown in Appendix A. □

Note 2. In the original *Prince of Persia* game, there is an additional parameter of the instance: the amount of steps T after which an open door starts to close. Our construction can easily be modified to include this parameter, and it is even possible to set T to a small value approximately equal to the map size by adding a “refresh” block where the avatar can re-open any currently open door.

Corollary 2. PRINCEOFPERسيا is PSPACE-complete.

Proof. Follows from Theorem 1, Savitch’s theorem that NSPACE=PSPACE, and the obvious fact that PRINCEOFPERسيا is in NSPACE. □

3 Hardness of the Lemmings Problem

One particular 2D game that is in many ways similar to our 2D platform games from the previous section is the game Lemmings. Already in 1998 McCarthy [13] mentions this game as a challenge for artificial intelligence. In [3] Cormode defines the LEMMINGS decision problem and argues that this problem is NP-complete. This claim is later repeated in [17] and it is conjectured that a variation of this problem is NP-complete as well.

However, Cormode’s proof only holds for a restricted version of the LEMMINGS problem. This restriction is introduced on page 3 of [3] where an instance is defined. One element of the instance is the time limit, which Cormode defines as follows: “*limit*, the time limit, in discrete time units. For technical reasons, we will insist that the time limit is bounded by a polynomial in the size of the

level. We conjecture that this restriction is unnecessary, by claiming that if it is not possible to complete the level in time polynomial in the level size, then it is not possible to complete the level at all.”

The “technical reasons” are, in fact, just one reason: Cormode’s proof that $\text{LEMMINGS} \in \text{NP}$ is trivial – guess the solution and verify it. This approach obviously fails if the number of theoretically possible configurations is not polynomial in the input size. Hence Cormode needs the restriction to artificially limit the number of reachable configurations.

In this section, we show that Cormode’s conjecture is false by constructing a class of solvable LEMMINGS instances of increasing sizes such that their shortest solution length can not be bounded by a polynomial in the input size. We then conclude that the general LEMMINGS problem is NP-hard, as proved by Cormode, but so far we cannot tell whether it is in NP. The existence of instances with only exponentially long solutions leads us to conjecture that the general version of this decision problem is in fact not in NP.

3.1 Constructing the Instances

In this section we construct a class of LEMMINGS instances for which the length of the shortest correct solution is not polynomial in the input size. Our construction is based on the idea of lemming synchronization – in order to solve the level, events in different places will have to happen at the same time. And for our instances, this will only occur after an exponential number of steps has elapsed.

To construct our instances, we will only need a very small subset of the Lemmings universe: entrances, permeable walls, and an exit. The only lemming skill we will use will be the digger. We will build two types of gadgets: a release gadget, that will only be able to release a lemming in certain points in time, and a synchronization gadget that must be reached by all lemmings in (almost) the same moment in time. The release gadget is shown in Figure 6 on the left, and the synchronization gadget is shown in Figure 6 on the right.

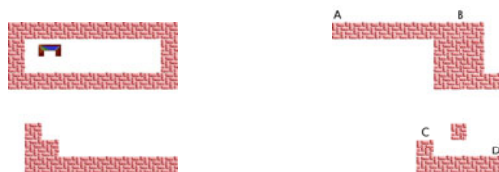


Fig. 6. The release gadget (left) and the synchronization gadget (right)

Lemma 1. *Let the release gadget be x steps wide. Assuming that no other lemming can reach the gadget, the lemming inside must start digging at a time that is an integer multiple of $2x$ in order to survive.*

Proof. At time 0 the entrance releases a lemming, at time 1 the lemming reaches the ground and starts walking towards the right. Without any interaction the

lemming will keep on bouncing off walls, and reenter the same state every $2x$ ticks. In order to save it we have to give it the digger skill sooner or later. The bottom part of the gadget is in such a height that the lemming must dig in the leftmost place, otherwise the fall will kill it. Additionally, the lemming must be walking in the correct direction (left to right), otherwise it falls to its death immediately after reaching the bottom part of this gadget. This configuration will appear precisely at ticks that are positive integer multiples of $2x$. \square

Lemma 2. *For a single lemming entering the synchronization gadget, the only way to survive is to dig at B and leave the gadget at D .*

Proof. If the lemming does not dig anywhere, it will reach the right end of the top ledge and fall to its death. The only safe place where to dig is point B , anywhere else the dig ends in a fall that is too high. \square

Lemma 3. *If for each lemming the only way to the exit leads through the point A of a single synchronization gadget, then the lemmings can only survive if all other lemmings arrive at A at 2 to 8 ticks after the first one.*

Proof. Consider the first lemming that arrives at A . From Lemma 2 it follows that this lemming has to start digging at B . As soon as this lemming digs a hole that is deep enough, no other lemming will be able to cross this gadget – regardless of what it does, it will fall and die somewhere. Hence there is only one way how the other lemmings may survive: they must arrive during the constant amount of time when the first lemming digs – and enter the hole below B while it is shallow enough. On the other hand, a delay of at least 2 ticks is necessary. The hole must be already deep enough so that the other lemmings can not escape it.

Note that when the digger finishes the hole, some of the other lemmings fall out of it walking in the opposite direction. This is fixed by the wall at C that turns these lemmings around. \square

Given an integer n , we will now create an instance I_n of LEMMINGS as follows: The only skill available will be the digger, and the number of times it can be used will be $n + 1$. We will have n lemmings, each of them starting in a separate release gadget. The lengths of the release gadgets will be $5p_1, \dots, 5p_n$, where p_i is the i -th prime number. The outgoing paths from these gadgets will be merged together in such a way that the number of steps from each of the gadgets to the common meeting point will be the same – except for one gadget that will be two steps closer. The path from the meeting point will lead onto point A of a single synchronization gadget, and the synchronization gadget will output the lemmings from point D straight to the exit.

The instance I_4 is shown in Appendix B.

Lemma 4. *Let $p_n\#$ be the primorial, i.e., the product of the first n primes. Then $n^{\lfloor n/2 \rfloor} \leq p_n\# \leq 2 \cdot n^{2n}$.*

Proof. Obviously follows from the bounds on p_n proved in [14]. \square

Theorem 2. *Each instance I_n is solvable. The shortest number of ticks in which I_n can be solved exceeds $10n^{\lfloor n/2 \rfloor}$.*

Proof. In order to save all lemmings in I_n they all have to start digging out of their release gadgets at the same time. To prove this, assume the contrary. The release gadget lengths are multiples of 5, so by Lemma 1 the period of each lemming in the gadget is a multiple of 10. Hence if lemmings do not start digging at the same time, then the last lemming will leave the release gadget at least 10 ticks after the first one. Then the last lemming will arrive at A at least 8 ticks after the first one, and by Lemma 3 it will die, which is a contradiction.

Hence the lemmings must start digging at some time T that is a multiple of the period of each lemming. By the Chinese Remainder Theorem, the smallest such T is $T = 10p_n\#$, and by Lemma 4 we have $T > 10n^{\lfloor n/2 \rfloor}$.

We can easily verify that once all lemmings start digging at time T , the rest of the solution is short and unique. □

Theorem 3. *The instance I_n can be described by $\Theta(n^2 \log n)$ bits.*

Proof. The width of the level is $\Theta(n+p_n) = \Theta(n \log n)$, hence we need $\Theta(n^2 \log n)$ bits to store the map. We also need to set a suitable time limit for the level. Clearly, the time limit $10p_n\# + 47n \log n + 47$ is sufficient. By Lemma 4 this number is $O(n^{2n})$, hence $O(\log(n^{2n})) = O(n \log n)$ bits are sufficient to encode it. The skills vector, the total number of lemmings, and the number of lemmings to save can all be stored in $O(n)$ bits, even if using unary coding. □

Corollary 3. *From Theorems 2 and 3 it follows that for the sequence of instances $\{I_n\}_{n=1}^\infty$ the length of the shortest solution grows exponentially in the instances' input sizes. Hence this sequence of instances is a counterexample to Cormode's proof that LEMMINGS \in NP.*

Note 3. Our construction requires the player to do n actions at the same time when releasing lemmings from their gadgets. If we assume that the player can only make one action per tick, this is easily fixed by moving release gadget x exactly x steps away from the meeting point, for each x . In this situation the player's actions must occur in immediately consecutive steps.

Note 4. Both our construction and Cormode's original proof of NP-hardness involve multiple entrances. This is an unnatural construction, as most levels in the original Lemmings games only have a single entrance that releases all lemmings sequentially at a fixed rate. Below we prove that our result are true for LEMMINGS instances with a single entrance.

3.2 The Distribution Gadget

Our proof is based on the construction of a distribution gadget that takes a stream of lemmings (such as the one leaving a single entrance) and breaks it into individual lemmings, each on a separate path. The construction uses a

method similar to the one used by the first lemming in the synchronization gadget: digging in a suitable place may decrease the height of the lemming’s fall.

The distribution gadget for n lemmings is a generalization of the one shown in Figure 7. When employing this gadget we need to add n additional diggers to the skill vector, and create a separate exit path for each of the lemmings. Note that the bottom part of the gadget is placed exactly in such a height that a fall from the *top* of the top ledge would kill a lemming, whereas the fall from the *bottom* of the ledge is safe.

Theorem 4. *The only way in which all lemmings survive the distribution gadget is that the player makes each of them dig above one of the exit points.*

Proof. A lemming that is never assigned the digger skill will fall to its death – either at the end of the top ledge, or into a hole dug by some previous lemming. Hence each lemming must dig in order to survive, and there are precisely n places where a lemming can dig and survive the resulting fall. Note that the places for digging must be assigned right to left, i.e., the first lemming to get out of the entrance must be the one reaching point 1 in Figure 7. □

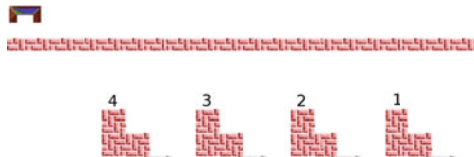


Fig. 7. The distribution gadget for four lemmings

Corollary 4. *In our construction of an instance I_n , we can start by using the distribution gadget. Then we can easily construct paths that will lead the separated lemmings to enter each of the individual release gadgets from above. Hence even if we restrict LEMMINGS to instances with only one entrance, there will still be instances with exponentially long solutions only.*

4 Conclusions and Open Problems for Further Research

We have shown a general point of view on platform games that allowed us to find two classes of NP-hard platform games and one class of PSPACE-hard platform games. These classes include many well-known examples.

For the LEMMINGS problem, we have disproved Cormode’s original assumption by constructing instances with only exponentially long solutions. Our construction works even if we limit ourselves to instances with only one entrance.

Clearly, LEMMINGS \in PSPACE, as the number of reachable configurations is always at most exponential in the input size. An open question is whether LEMMINGS can actually be shown to be PSPACE-complete. Our intuition suggests that this may indeed be the case. One observation is that the miner skill together with a combination of permeable and impermeable terrain can be used

to free a trapped lemming from the outside. This may be exploited to “store” lemmings in various part of the level, to use their presence/absence as memory, and to have one “master” lemming that can alter this memory.

As for the general research of the computational complexity of games and puzzles, we are convinced that our approach from Section 2 is the correct direction for the future. As we documented in the overview, there are already many results on the hardness of individual puzzles. What we now need to discover are patterns in these results. What are the common features that make games and puzzles hard? Can then these observations help us analyze other games and puzzles? The recent Constraint Logic framework by Demaine and Hearn [5] is probably one of the first steps in this direction.

The topic of platform games is not exhausted in this article. While most of the games we examined are either obviously in P or covered by one of our meta-theorems, there are some exceptions. Notably, so far we ignored the presence of enemies. We expect that in some cases the presence of enemies makes the games hard to solve. If enemies are present, the number of possible configurations may become exponential in the input size. If the enemies must be avoided, it should be possible to create instances that require enemy synchronization in order to be solvable, and this can force the solution to be exponentially long.

References

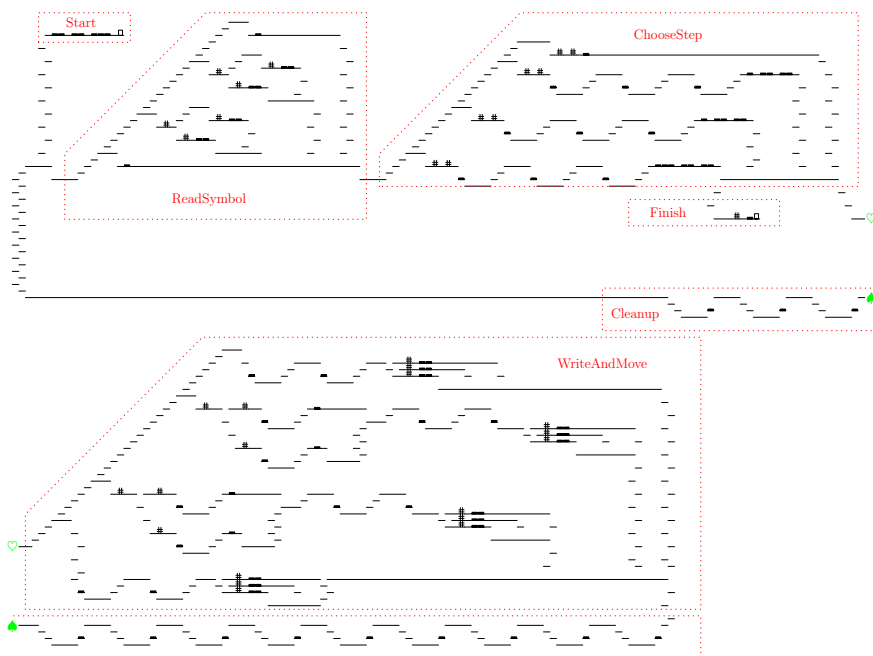
1. Bouton, C.L.: Nim, a game with a complete mathematical theory. *Annals of Mathematics* 3, 35–39 (1901/2002)
2. Conway, J.H.: *On Numbers and Games*. Academic Press, London (1976)
3. Cormode, G.: The Hardness of the Lemmings Game, or Oh no, more NP-Completeness Proofs. In: *Proceedings of Third International Conference on Fun with Algorithms*, pp. 65–76 (2004)
4. Culbertson, J.: Sokoban is PSPACE-complete. In: *Proceedings of the International Conference on Fun with Algorithms*, pp. 65–76 (1998)
5. Demaine, E.D., Hearn, R.A.: Constraint logic: A uniform framework for modeling computation as games. In: *Proceedings of the 23rd Annual IEEE Conference on Computational Complexity* (2008)
6. Demaine, E.D., Hearn, R.A.: Playing games with algorithms: Algorithmic combinatorial game theory. In: Albert, M.H., Nowakowski, R.J. (eds.) *Games of No Chance 3*. Mathematical Sciences Research Institute Publications, vol. 56, pp. 3–56. Cambridge University Press, Cambridge (2009)
7. Eppstein, D.: *Computational Complexity of Games and Puzzles* (2009), <http://www.ics.uci.edu/~eppstein/cgt/hard.html>
8. Grundy, P.M.: Mathematics and games. *Eureka* 2, 6–8 (1939)
9. Itai, A., Papadimitriou, C.H., Szwarcfter, J.L.: Hamilton Paths in Grid Graphs. *SIAM Journal on Computing* 11(4), 676–686 (1982)
10. Kaye, R.: Minesweeper is NP-complete. *Mathematical Intelligencer* 22(2), 9–15 (2000)
11. Kendall, G., Parkes, A., Spoerer, K.: A Survey of NP-Complete Puzzles. *International Computer Games Association Journal* 31(1), 13–34 (2008)

12. Lichtenstein, D.: Planar Formulae and Their Uses. *SIAM Journal on Computing* 11(2), 329–343 (1982)
13. McCarthy, J.: Partial formalizations and the Lemmings game, Technical report, Stanford University, Formal Reasoning Group (1998)
14. Robin, G.: Estimation de la fonction de Tchebychef θ sur le k -ième nombre premier et grandes valeurs de la fonction $\omega(n)$ nombre de diviseurs premiers de n . *Acta Arith.* 42(4), 367–389 (1983)
15. Robson, J.M.: The complexity of Go. In: *Proceedings of the IFIP 9th World Computer Congress on Information Processing*, pp. 413–417 (1983)
16. Robson, J.M.: N by N Checkers is EXPTIME complete. *SIAM Journal on Computing* 13(2), 252–267 (1984)
17. Spoerer, K.: The Lemmings Puzzle: Computational Complexity of an Approach and Identification of Difficult Instances. PhD thesis (2007)
18. Sprague, R.P.: Ueber mathematische Kampfspiele. *Tohoku Mathematical Journal* 41, 438–444 (1935/1936)

A Example Construction for Prince of Persia

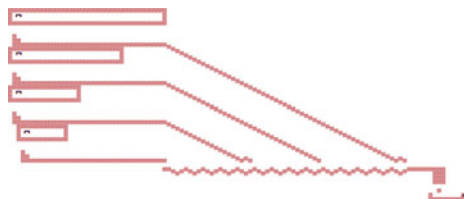
The following is a complete level for Prince of Persia, representing the instance (M, w) , where $|w| = 2$ and M is the smallest possible LBA with starting state q_0 , single final state q_2 and the δ -function: $\delta(q_0, 0) = \{(q_0, 0, 1)\}$, $\delta(q_0, 1) = \{(q_1, 0, 1)\}$, $\delta(q_0, R) = \{(q_1, R, -1)\}$, and $\delta(q_1, 0) = \{(q_2, 1, 0)\}$.

The level is broken into two parts to make it fit on a single page. The places marked by a heart and a spade should be attached to form the correct picture. Door and pressure plate labels were intentionally omitted, they can easily be reconstructed from the description in the article, if needed.



B Example Instance for Lemmings

The image below shows the complete instance I_4 . The release gadgets have lengths 10, 15, 25, and 35. The smallest release gadget is shifted 2 steps to the right – the lemming from this gadget will be the one digging at the synchronization gadget.



Christmas Gift Exchange Games

Arpita Ghosh and Mohammad Mahdian

Yahoo! Research

Santa Clara, CA, USA

arpita@yahoo-inc.com, mahdian@alum.mit.edu

Abstract. The Christmas gift exchange is a popular party game played around Christmas. Each participant brings a Christmas present to the party, and a random ordering of the participants, according to which they will choose gifts, is announced. When a participant's turn comes, she can either open a new gift with unknown value, or steal an already opened gift with known value from someone before her in the ordering; in the second case, the person whose gift was stolen gets to make the same choice.

We model the gift exchange as a sequential game of perfect information and characterize its equilibria, showing that each player plays a *threshold* strategy in the subgame perfect equilibrium of the game. We compute the expected utility of players as a function of the position in the random ordering; the first player's utility is vanishingly small relative to every other player. We then analyze a different version of the game, also played in practice, where the first player is allowed an extra turn after all presents have been opened—we show that all players have equal utility in the equilibrium of this game.

1 Introduction

The practice of giving gifts to friends and relatives at Christmas is a centuries-old tradition. Contrary to popular belief, gifts are not conjured up by Santa at the North Pole, and the actual buying of gifts unfortunately must be done by real people in real local marketplaces with no aid from Santa whatsoever. As a result, the Christmas gift industry is now a huge multibillion dollar industry much larger than online advertising, and is therefore clearly a subject deserving of serious study.

The problem of Christmas gifting admits many interesting directions of research drawing on various disciplines. For instance, the phenomenon of stores beginning to play Christmas jingles annoyingly earlier and earlier in the year (to subtly indicate gift-shopping time) is related to the phenomena of unraveling markets studied in the economics literature [5]. Another obvious area is psychology and sociology, analyzing the very noticeable impact of gifting and associated customs on the behavior of otherwise normal individuals. We will, however, focus on a particular Christmas gifting tradition, the *gift exchange*, that leads to increased welfare and significant computational and storage savings for each

gifter: instead of buying one gift individually for each giftee, every gifter brings a single gift, which are then collectively exchanged amongst the group.

The gift exchange mechanism represents a significant improvement for gifters over the tradition of buying individual gifts, since only 1 present need be bought rather than n — that is, the number of gift-choosing problems that need to be solved, and the fear struck by the prospect of Christmas gift shopping, is now reduced to a constant rather than growing linearly with the size of an player's neighborhood in the Christmas-gifting-graph¹. This also has the second-order benefit of removing the incentive to limit the size of one's social network due to the large psychological and economic gifting costs, which in the limit could lead to smaller and smaller tight-knit groups, eventually resulting in a catastrophic unraveling of society. The gift exchange scheme also improves *efficiency* relative to the scheme of one gift per neighboring node used widely in practice. Empirical and anecdotal evidence suggests that most people, including the President [1], believe they give better gifts than they get; the value obtained from the average received gift is much lower than the sum of the costs of planning to buy, buying, and giving the gift (and recovering from the experience). Since each gifting transaction leads to negative utility on average, welfare is maximized by minimizing the number of transactions— this means that, subject to the constraint that each person must receive a gift, the gift exchange mechanism actually has optimal welfare.²

The protocol used to assign the n gifts amongst the n participants is known by several different names including Chinese Christmas and White Elephant gift exchange, and is a common party game around Christmas time [2,3,4]. Each participant brings one gift (whose value is in some range prespecified by the host) to the party, and the gifts are all placed in a pile (presumably under a suitably large and well-decorated Christmas tree). We'll assume that everyone uses identical wrapping, so that they cannot identify the gift they brought once it's been put in the pile; the reason for this assumption will become clear once we present our model. A random ordering of the participants is chosen, and when a participant's turn comes, she either opens a new gift, or steals an already opened gift from someone before her in the ordering, in which case the person whose gift was stolen gets to make the same choice. A person cannot steal back the gift that was stolen from her immediately; also, to ensure that the last person is not guaranteed to walk away with the best present, a rule that no present can be stolen more than a certain number of times, say three or four times, is enforced.

In this paper, we study this gift exchange mechanism from a game theoretic point of view. Since the mechanism itself is rooted in tradition, we do not address

¹ This is based on the fairly reasonable assumption that the number of distinct Christmas parties an player must attend and buy gifts for is 1 (or very small), that is, it does not scale with the size of his Christmas-gifting neighborhood.

² There is an interesting parallel between the VCG mechanism and the gift exchange scheme — both mechanisms have excellent efficiency properties, but are nonetheless not popular in the industry. While the reason for this is very obvious for the gift exchange scheme (the gift producing industry has no reason to like this scheme which leads to fewer gifts being bought), the case for the VCG auction is far more interesting and subtle, see The Lovely but Lonely Vickrey Auction [6].

the question of whether or not this is a good mechanism to fairly distribute the gifts. Rather, we analyze the game from the participants perspective, and investigate best response—given the gift exchange mechanism, how best should a utility maximizing partygoer behave? (We mean best behavior not in the sense of being on one’s best behavior at the family party, but rather from the point of view of maximizing the expected value of the final present she goes home with). To do this, we model the Christmas party gift exchange as a sequential game where each gift’s value is drawn uniformly at random and unknown until unwrapped, and use backward induction to reason about the behavior of each player in this game. The problem becomes technically interesting because in addition to not knowing the value of an unopened present, a player also has to contend with the fact that his present may be stolen away from him in the future, depending on its desirability.

The remainder of the paper is organized as follows. We first present a formal model for the gift exchange game in §2 and then derive the best response and equilibrium in §3. Our results allow us to quantify the benefit of drawing a position towards the end of the list of participants, answering the question of how loudly to sigh or squeal when learning of one’s position in the random order. Finally, we analyze a version of the game with slightly modified rules where the first player gets a chance to steal a present at the end if she has not done so already; this version is also sometimes played in practice. In contrast with the original version, all players have equal expected utility in the equilibrium of this game.

2 Model

There are n players, and n unopened gifts. The gifts have a common value to each player, that is, different players do not value the same gift differently (for instance, this common value could be exactly the dollar value of the gift). Each gift has a value v_i drawn independently and uniformly at random from $[0, 1]$. The value of a gift remains unknown until the gift is opened, at which point its value v_i becomes known to all n players.

A random ordering is chosen amongst the n players and announced publicly—this is the order in which players get to choose presents. We number the players according to this order. We also adopt the rule from the actual party game that each gift can be stolen only a limited number of times—we restrict the number of times a gift can be stolen to 1, i.e., if a present has been stolen once, it cannot be stolen again, and stays with its current owner. We will call a present that has been opened but never stolen *available*, and an open present that has been stolen once already *unavailable*. Limiting the number of times a present can be stolen to 1 keeps the analysis tractable while still preserving the feature that an player’s current choice affects the future decision of whether other players will want to steal her gift in the future, affecting her final value.

When player i ’s turn arrives, she has a choice between picking an unopened present (with unknown value drawn UAR $[0, 1]$) from the pile, or stealing an available present from the players $1, \dots, i - 1$. If she opens a new gift, the game

proceeds to player $i + 1$. If she steals an open available gift from some player $j < i$, j again gets a choice between stealing an available present and opening a new gift from the pile (note that j cannot steal back her own gift since it has been stolen once and is now unavailable). Note that when player i 's turn arrives, there are exactly $n - (i - 1)$ unopened gifts in the pile, and each player $1, \dots, i - 1$ has an opened gift. The game continues this way until there is only one unopened gift, at which point player n takes her turn and follows the same sequence. Define a step to mean each time a gift has a new owner: the game is guaranteed to terminate in at most $2n$ steps, since there are n gifts, and each gift can be stolen only once, corresponding to at most 2 steps per gift in the game.

We analyze the gift exchange game G as a sequential game with perfect information, where the players are rational utility maximizers—each player tries to maximize the expected value of the final gift she is left with, given that each player after her is a rational utility maximizer as well (the expectation is taken over the random draws of unopened gifts). We point out that our model assumes that players only value gifts and not time, and does not address players who are running out of patience (or lacked it to start with), or want to get the gift exchange over with quickly. These can be modeled with a discount factor; we leave this as an open direction for future work. Our results also only apply when there are no externalities—they do not, for instance, predict the outcome of a game where your coworker five places down the line might steal your present either out of love for the present, or hate for you. While these assumptions are common in the research literature, they (especially rationality) may not hold in practice—to ensure applicability in practice, it is adequate to have your fellow partygoers read and understand the best response derived in §3 and instruct them to act according to it, before starting the gift exchange.

3 Analysis of the Game

In this section, we analyze the equilibrium of the gift exchange game. Before beginning with the analysis, we first make some simple observations about the game. We define round i as the sequence of steps starting from when player i first gets a turn to the step immediately before player $i + 1$ first gets her turn. Note that a new gift is opened in the last step of a round (a round can have only one step), and exactly one gift is opened in each round. Round i has no more than i steps, and the entire game terminates in no more than $2n$ steps. The last player plays exactly once; the player in the i th position in the ordering plays at most $n - i + 1$ times. Once a player steals a gift, she never plays another turn, and the value of the gift she steals is her final value from the game.

We now give a complete analysis of the game G . We prove that in the solution of the game, each player plays according to a *threshold strategy* of the following form: if the value of the most valuable available gift is at least θ (where θ is the threshold), then steal that gift; otherwise, open an unopened gift. The value of the threshold θ depends on which round in the game is in progress—specifically,

θ is a function of the number i of *unopened gifts* at the time. We define a sequence $\theta_1, \dots, \theta_n$ recursively as follows:

$$\theta_1 = 1/2, \quad \theta_i = \theta_{i-1} - \frac{\theta_{i-1}^2}{2} \text{ for } i > 1. \quad (1)$$

Note that this recurrence defines a decreasing sequence. We prove the following result.

Theorem 1. *The following is a subgame perfect equilibrium of the gift exchange game: for any player p and any time p gets to play, play the threshold strategy with threshold θ_i , where i is the number of unopened gifts at the time. Furthermore, the expected value that p receives by playing this strategy is equal to $\max(\theta_i, v)$, where v is the value of the most valuable available gift at the time.*

Proof. We prove this by induction on i . We start with $i = 1$. This means that at the time player p gets to play, only one unopened present is left. This player has two choices: either to steal the most valuable available gift (of value v), or to open the only remaining unopened gift, after which the game will end. Since the value of a gift is drawn uniformly from $[0, 1]$, the expected value of opening the unopened gift is $1/2$. Thus, the player must steal if $v \geq 1/2$ and open the unopened gift if $v < 1/2$. The value that this strategy gets is precisely $\max(1/2, v)$.

Now, we assume the statement is proved for $i - 1$, and prove it for i . Consider the player p that is playing at a time that there are exactly i unopened gifts, and the value of the most valuable available gift is v . At this point, p has two options: either to steal the gift of value v , or to open an unopened gift. If p opens an unopened gift, we denote the value of this gift by x , drawn uniformly from $[0, 1]$. In the next step, by the induction hypothesis, the next player will steal the highest value available gift if this gift has value at least θ_{i-1} . If she does so, the player whose gift is just stolen will get to play, and again, by the induction hypothesis, will steal the highest value available gift if its value is at least θ_{i-1} . This ensues a sequence of stealing the highest value available gifts, until we reach a point that the highest value available gift has value less than θ_{i-1} , at which point the person whose turn it is to play will open a new gift.

We now consider two cases: either the value x of the gift p just opened is at least θ_{i-1} , or it is less than θ_{i-1} . In the former case, the sequence of stealings will at some point include p . At this point, all the available gifts of value more than x are already stolen. We denote by $v(x)$ the highest value of an available gift of value less than x . This is precisely the value of the most valuable gift that is still available at the time that the sequence of stealings reaches p . By induction hypothesis, at this point, the maximum value that p gets is equal to $\max(\theta_{i-1}, v(x))$. In expectation, the value of p in this case is $\max(\theta_{i-1}, \text{Exp}[v(x)])$, where the expectation is over drawing x uniformly at random from $[\theta_{i-1}, 1]$.

The other case is when x is less than θ_{i-1} . In this case, since the sequence θ is decreasing, by induction hypothesis the gift that p just opened will never be stolen. Therefore, the expected value of the gift that p ends up with in this case is precisely $\theta_{i-1}/2$.

Putting these together, the overall value of p , if she decides to open a new gift can be written as

$$(1 - \theta_{i-1}) \max(\theta_{i-1}, \text{Exp}_{x \leftarrow U[\theta_{i-1}, 1]}[v(x)]) + \theta_{i-1} \times \frac{\theta_{i-1}}{2}.$$

Now, we consider two cases: if $v > \theta_{i-1}$, then we have $v > \theta_{i-1}/2$ and $v \geq v(x)$ (the latter inequality by the definition of v and $v(x)$). Therefore, the above expression is less than $(1 - \theta_{i-1})v + \theta_{i-1}v = v$, meaning that in this case, it is p 's optimal strategy to steal the gift of value v . In the other case ($v \leq \theta_{i-1}$), by the definition of $v(x)$, for every $x \in [\theta_{i-1}, 1]$, $v(x) = v \leq \theta_{i-1}$. Therefore, the utility that p obtains by opening a new present can be written as

$$(1 - \theta_{i-1})\theta_{i-1} + \theta_{i-1} \times \frac{\theta_{i-1}}{2} = \theta_i.$$

Putting everything together, we obtain that the maximum utility p can obtain is $\max(v, \theta_i)$, and this utility is obtained by playing the threshold strategy with threshold θ_i .

We cannot obtain an explicit formula for θ_i from the recurrence relation (II), but the following theorem gives us the asymptotics.

Theorem 2. *For every i , we have $\frac{2}{i+2+H_i} \leq \theta_i \leq \frac{2}{i+3}$, where $H_i \approx \ln(i) + \gamma$ is the i 'th harmonic number.*

Proof. Let $y_i = 2/\theta_i$. The recurrence (II) gives us:

$$y_i = \frac{2}{2/y_{i-1} - (2/y_{i-1})^2/2} = \frac{y_{i-1}^2}{y_{i-1} - 1} = y_{i-1} + 1 + \frac{1}{y_{i-1} - 1}. \tag{2}$$

Thus, since the term $1/(y_{i-1} - 1)$ is non-negative, we have $y_i > y_{i-1} + 1$, which together with $y_1 = 4$ implies that $y_i > i + 3$, proving the upper bound on θ_i . To prove the lower bound, we use the inequality $y_i > i + 3$ we just proved in combination with (2). This gives us $y_i < y_{i-1} + 1 + \frac{1}{i+2}$. This implies $y_i < i + 3 + \sum_{j=4}^{i+2} \frac{1}{j} < i + 2 + H_i$, proving the lower bound on θ_i .

That is, as we move along the random ordering, a player's threshold for stealing a gift keeps increasing: early in the game, players are willing to settle for gifts of lower value than later in the game (recall that if a gift is stolen, that gift's value is the final utility to the player who steals the gift).

We make the following observations about the equilibrium play of the game, which follow from the fact that the optimal strategy for each player is a threshold strategy, and these thresholds increase through the play of the game:

- If a gift is not stolen immediately after it is opened, it is never stolen, since the thresholds θ increase as the number of unopened gifts decreases.
- If a gift is stolen from a player, this player does not continue stealing, but rather opens a new gift. That is, each round is of length at most 2, *i.e.*,

there are no 'chains' of gift stealing in any round. Each player i stealing a gift therefore steals either from $i - 1$, if $i - 1$ opened a new gift, or else from the player $j < i - 1$ from whom $i - 1$ stole her gift (and who consequently opened a new gift), in this case, all players $j + 1, j + 2, \dots, i$ have stolen the gifts opened by j .

Therefore, when players play according to their optimal strategy (the threshold strategies prescribed by Theorem 1), the game will proceed as follows: first, player 1 opens a new present. If the value of this present is less than θ_{n-1} , this present will not be stolen by player 2 (and by no other player, since $\theta_{n-1} < \theta_i$ for $i > n - 1$), and player 2 opens a new present; otherwise, this present will be stolen by player 2, and player 1 will open a new present. In either case, if the value of the newly opened gift is less than θ_{n-2} , it will not be stolen by player 3 (and therefore by no other player after that), and instead, player 3 opens a new present; but if this value is greater than θ_{n-2} , it will be stolen by player 3, and the player who used to hold that gift will open a new present, and so on. When it is turn for player i 's to play for the first time, it must be that in the last step, one of the players has opened a new gift (unless $i = 1$). If the value of this gift is more than θ_{n-i+1} , player i steals it, and the player who used to hold that gift will open a new gift. Otherwise, player i opens a new gift.

Given this, we can calculate the expected utility of each player in this game: for every $i > 1$, when player i gets to play for the first time, the only way the value v of the most valuable available gift is greater than θ_{n-i+1} is if this gift is the one just opened by the last player who played before i . This happens with probability $1 - \theta_{n-i+1}$, and in this case, the value of the gift is distributed uniformly in $[\theta_{n-i+1}, 1]$. Therefore, by Theorem 1, the expected value that player i derives in this game is precisely

$$\text{Exp}[\max(\theta_{n-i+1}, v)] = (1 - \theta_{n-i+1}) \times \frac{1 + \theta_{n-i+1}}{2} + \theta_{n-i+1} \times \theta_{n-i+1} = \frac{1}{2} + \frac{\theta_{n-i+1}^2}{2}.$$

Therefore, all players $i > 1$ derive a utility more than $1/2$. This, however, is at the expense of the first player. When player 1 plays for the first time, there is no available gift. Therefore, by Theorem 1, the expected utility of player 1 is precisely θ_n , which by Theorem 2 is $\frac{2}{n}(1 + o(1))$. This is summarized in the following theorem.

Theorem 3. *The expected utility of player i for $i > 1$ in the gift exchange game is $\frac{1}{2} + \frac{\theta_{n-i+1}^2}{2}$. For player 1, the expected utility of playing the game is $\theta_n = \frac{2}{n}(1 + o(1))$.*

3.1 A Fairer Game

The first player in the ordering might never get to steal a gift in the game G , and as we saw above, receives very low utility relative to all other players: in this sense, the game G is not very fair. To be more fair to the first player, a version of the game is sometimes played where the first player gets a turn at the end to

steal a gift. We next analyze this version of the game, and show that it is indeed more fair for the first player — every player has equal expected utility in the equilibrium of this game.

We define the game G' to be the following modification of G : if the first player never gets a chance to steal a gift through the course of the play, she gets a turn at the end after all gifts have been opened, and can steal from amongst the available gifts if she wishes. (Note that if player 1 has never stolen a gift, the gift she holds is always available; if she chooses to keep her own gift at the end of the game, we will call this equivalent to stealing her own available gift.) We show the following about G' .

Theorem 4. *In the subgame perfect equilibrium of G' , every player has expected utility $1/2$.*

Proof. We claim that the following is a subgame perfect equilibrium of the modified game G' . Each player i other than player 1 uses the following strategy. If player 1 has already stolen a gift, then play according to the optimal strategy for G ; if player 1 has not yet stolen a gift, steal the maximum value available gift. For player 1, if her gift is stolen when there are 2 or more unopened items, she steals the highest value available gift if its value v is greater than $1/2$, else opens a new gift. If there is only one unopened gift when her gift is stolen, she opens the new gift, and if she gets a turn when there are no unopened gifts, she steals the highest value available gift (including her own).

We prove this claim by backward induction. First, note that if player 1 does play after all gifts have been opened, she must steal the highest value gift from amongst the available gifts (including her own). If there is exactly one unopened gift when her gift is stolen, there is no player who can steal the new gift she opens from her: if the best available gift has value v_1 , she can get a value of v_1 by stealing, or $\max\{v_1, x\} \geq v_1$ if she opens a new gift (since v_1 will still be available). So she must open the new gift. Also, once player 1 steals a gift, it is optimal for every player to play according to the strategy described for G in Theorem 1. Consider a player $j \neq 1$ when there is just one remaining unopened gift, when 1 has not stolen a gift yet. She can either open a new gift with value x , or steal the best available gift of value v_1 . If she steals, this gift cannot be stolen from her, so her final value is v_1 . If she opens a new gift with value $x > v_1$, this gift becomes the highest value available gift, and will be stolen by player 1 in the next round, leaving her to steal the gift of value v_1 . If $x < v_1$, she either retains this gift or gets the next available gift with value less than x , depending on whether x is smaller or larger than g , the value of the gift currently held by player 1. In either case, if she opens a new gift, the final value she receives is no larger than v_1 , so her best response is to steal the gift with value v_1 . (The argument for player 1 is the same for all rounds in the game, and we do not repeat it for this case with only one unopened gift).

Now assume that it is some player $j \neq 1$'s turn to play when there is more than one unopened gift, and the induction hypothesis holds for the remainder of the game. Again, consider the case where 1 has not yet stolen a gift, so that her gift is still available. Suppose the values of the available gifts are $v_1 \geq v_2 \dots$ and so on.

If j steals, she gets a final value of v_1 . If she opens a new gift of value x , this gift becomes available and can be stolen in the remainder of the game. If $x > v_1$, by the induction hypothesis x is immediately stolen by the next player and j steals v_1 , for a final value of v_1 . If $x \leq v_1$, there is a sequence of stealing v_1, v_2, \dots ; either j 's gift of value x is never stolen in the remainder of the game, in which case her final value is $x \leq v_1$, or it is stolen. Irrespective of when it is stolen— either when player 1 has not yet stolen a gift, or after 1 steals a gift— j 's final value is no larger than x : suppose x is stolen when 1 has not yet stolen a gift; by the induction hypothesis, j must steal the highest value available gift, which has value $v(x) \leq x$. (Since x was stolen, it was the highest value available gift at that point. Also note that such a gift definitely exists since 1's gift is available.) If j 's gift is stolen after player 1 steals a gift and there are i' unopened gifts at this time, we must have $x \geq \theta_{i'}$ since x was stolen, because all players are playing according to the optimal strategy in G . By Theorem [11](#), j 's expected utility from playing her optimal strategy at this point is $\max(v(x), \theta_{i'}) \leq x \leq v_1$. Therefore, player j can never get expected utility better than v_1 , so she should steal the highest value available gift.

For player 1, when there are 2 or more unopened gifts, she can either steal the highest value available gift to obtain utility v_1 , or open a new gift of value x . If she does not steal a gift now and never steals a gift in later rounds, by the induction hypothesis, every remaining gift including the last one is opened by 1 (since other players will steal the highest value available gift and 1's gift is always available). Also, by the induction hypothesis, if she steals a gift after this it has value greater than $1/2$. In either case, she can ensure a expected utility of at least $1/2$, so she should not steal if $v_1 \leq 1/2$. If $v_1 > 1/2$, then by opening the new gift of value x , the maximum value she can hope to get is $\max(v(x), 1/2)$, where $v(x)$ is the value of the best available gift after x . Since $v(x) \leq v_1$, her optimal strategy is to steal v_1 if $v_1 > 1/2$.

With these strategies, the optimal play of the game proceeds as follows: in the first step, player 1 opens a gift, which is stolen immediately by player 2; since the available set is empty, 1 opens a new gift, which is stolen immediately by 3, and so on; finally, the $n - 1$ th opened gift is stolen from 1 by player n . At this point, there are no available gifts, so 1 opens and keeps the last unopened gift; each gift, and therefore every player has expected value $1/2$.

References

1. <http://extratv.warnerbros.com/2009/12/president-obama-christmas-gifts-white-house.php>
2. http://www.associatedcontent.com/article/83228/the_craziest_christmas_party_game_the.html?cat=74
3. http://www.partygameideas.com/christmas-games/gift_exchange_2.php
4. http://en.wikipedia.org/wiki/White_elephant_gift_exchange
5. Roth, A.: Unraveling, Market Design, <http://marketdesigner.blogspot.com/2009/06/unraveling.html>
6. Ausubel, L.M., Milgrom, P.: The Lovely but Lonely Vickrey Auction. *Combinatorial Auctions*, 17–40 (2006)

Return of the Boss Problem: Competing Online against a Non-adaptive Adversary

Magnús M. Halldórsson¹ and Hadas Shachnai²

¹ School of Computer Science, Reykjavik University, 101 Reykjavik, Iceland
`mmh@ru.is`

² Department of Computer Science, The Technion, Haifa 32000, Israel
`hadas@cs.technion.ac.il`

Abstract. We follow the travails of Enzo the baker, Orsino the oven man, and Beppe the planner. Their situation have a common theme: They know the input, in the form of a sequence of items, and they are not computationally constrained. Their issue is that they don't know in advance the time of reckoning, i.e. when their boss might show up, when they will be measured in terms of their progress on the prefix of the input sequence seen so far. Their goal is therefore to find a particular solution whose size on any prefix of the known input sequence is within best possible performance guarantees.

1 Doing OK When the Boss Shows Up: Prefix Optimization

1.1 Enzo's Order Signups: Prefix Interval Selection

Enzo groaned with his arms curled over his head: "I'm in a fix – big-time. He's going to catch me at the worst possible moment."

"Who is?", I inquire.

"My boss could show up at any moment, and if I haven't signed up for my share, I can kiss my *confetteria* dream goodbye."

"So, why don't you? You can do it. You've got the brawn to handle any set of orders, and you've got the brains to figure out what is the maximum set that can be handled by a single person. What's holding you back?"

He slumps still lower in the seat. "It's not a matter of processing or computational power. Yes, I can figure it all out. I even know all the orders in advance; we always get the same set of orders on Fridays. So, of course, I could just find an optimal solution. But that's of no use."

"Now you really got me. You know everything and you can do anything, what could possibly be the problem?"

"Yeah, it's kind of funny. Look, let me explain the whole setup. Orders to the bakery arrive in a sequence. Each order has a given pickup time, and since people expect it to straight from the oven when they pick it up, it really means that the time for making it is fixed. "

Instance spec: Given is a sequence $\mathcal{I}_n = \langle I_1, I_2, \dots, I_n \rangle$, where each I_i is an interval in \mathfrak{R} .

“As soon as an order arrives, I have to either sign up for it or assign it to somebody else. I can’t work on two orders at the same time; they need my total concentration. We’ve got plenty of other guys that can handle those that I don’t do.”

Solution spec: Produce a subsequence \mathcal{I}' of intervals from \mathcal{I}_n such that no pair of intervals I_a, I_b in \mathcal{I}' overlap, i.e. $I_a \cap I_b = \emptyset$. More generally, we seek an *independent set* in a given graph G ; in the above situation G is an interval graph.

“Actually, I can decide the whole thing in advance, because I do know all the orders that arrive — we always get the same set of orders on that day of the week. But, that stupid boss doesn’t know it, and he won’t believe it.”

“He insists on checking on me at any time to ‘size me up for the big job’, as he calls it. And, the frustrating thing is that when he does, all that matters to him is how many orders I’ve then signed up for.”

This sort of makes sense to me. “So, you want to maintain always a good *efficiency index*. I mean, that what you’ve gotten, when the boss shows up, should be a large fraction of what could possibly be gotten if you’d knew when he shows up. Right?”

“Yeah, that’s it. Or, maybe it’s more of a *sloth index*.”

“There’s a catch here, though: you can’t ever hope to get a good ratio! You see, suppose the first order was so long that it took all of your available time. Well, if you take it, you can’t take anything else, and you’re doomed. But if you don’t take it, then if the boss shows up then, you’ve got an infinitely poor sloth index!”

“I knew it, I’m doomed” he whined.

“No, relax. You just need to modify your expectations a bit. You do try to minimize the ratio, but you count an empty solution as of size one. Which is fair, what diff does a single order make?”

Performance evaluation: For each prefix $\mathcal{I}_p = \langle I_1, I_2, \dots, I_p \rangle$, the *performance* on the prefix is the quantity

$$\rho_p = \rho_p(\mathcal{I}') = \frac{\alpha(\mathcal{I}_p)}{|\mathcal{I}' \cap \mathcal{I}_p| + 1},$$

where $\alpha(\mathcal{I}_p)$ is the maximum set of disjoint intervals in \mathcal{I}_p . The objective is to minimize $\rho = \max_{p=1}^n \rho_p$, the worst performance on any prefix.

“Yeah, ok. But it’s still a Catch-22 situation: no matter what I do, I’m doomed. If I start grabbing orders as soon as they arrive, I won’t be able to take on so many of the later orders, and he’ll take me to task at the end of the day. But if I try to wait until the best set of orders starts showing up, he’ll think I’m a lazy SOB that can’t get started in the morning. There’s no point even trying to

explain to him how thinking ahead could help. It's so Kafkaesque it's not even funny," he grumbles and shakes his head.

"Right, well, let's think constructively, and at least try to do the best we can. There is, by the way, a possibility that you can do better if you flip coins. "

"I'm not interested in some kind of average case."

"This is different. It means that no matter what how tricky your boss may be and even if he knows your solution strategy, you will always achieve some performance guarantee at the time he stops by. However, the guarantee is in expectation over the random coin flips, and not worst case..."

Randomized performance: In the randomized case, a solution is a probability distribution π over the independent sets of the input graph G . The expected solution size $E_\pi[\mathcal{I}_p]$ on prefix \mathcal{I}_p is the weighted sum $E_\pi[\mathcal{I}_p] = \sum_{I'} \Pr_\pi[I'] \cdot |I' \cap \mathcal{I}_p|$, where I' ranges over all independent sets in G . The performance ratio is then $\rho = \max_{p=1}^n \frac{\alpha(\mathcal{I}_p)}{\mathbb{E}_\pi[\mathcal{I}_p]}$.

1.2 Other Prefix Problems

"Right. BTW, some of the other guys are in similar situations, although they all have different types of tasks. For instance, Orsino the oven guy has to lay out the goods to be baked onto the baking plates. Not everything can go onto the same plate; it's not just the temperature, but, for instance, the slushy items can't go with the dry ones, square items will mess up the round ones, and the fragrance of certain items will affect other items, and so on. So, he needs to lay out all the goods onto the plates, and to do so as soon as they've been prepared."

"The problem is that if he uses more plates than necessary for the items ready at that time, the boss will get angry."

Prefix Coloring: Given an ordered graph G , i.e. with an ordered vertex set $V = \langle v_1, v_2, \dots, v_n \rangle$, find a coloring C of G such that $\rho = \max_{p=1}^n \frac{C(G_p)}{\chi(G_p)}$ is minimized, where G_p is the graph induced by $\langle v_1, v_2, \dots, v_p \rangle$, $C(G_p)$ is the number of colors that C uses on G_p and $\chi(G_p)$ is the chromatic number of G_p .

Something about this rang a bell with me. "Actually, this really reminds me of this chap Beppe doing urban planning for the city. They had these new servers being set up all the time, each having links to some of the earlier ones. They had to have guards on one side of each link to protect against unauthorized entry. As soon as a server was set up, they had to decide there and then whether to make it a guard, because the cost of converting an older server into a guard was prohibitive. Of course, they could have made every server a guard, but that not only was time consuming but looked spectacularly stupid."

"Every now and then, some wise-crack newspaper reporter would look at the guard installations in the city and try to score point by discovering 'waste in the system', that much fewer were needed *for the situation at that time*. Of course, there never was any point in try to counteract by showing that this would be

needed in the future; by the time such corrections came to light, nobody was interested in the story any more.”

Prefix Vertex Cover: Given an ordered graph G , find a vertex cover C of G such that $\rho = \max_{p=1}^n \frac{C(G_p)}{VC(G_p)}$ is minimized, $C(G_p) = C \cap V_p$ is the number of nodes from C in the prefix set V_p , and $VC(G_p)$ is the vertex cover number of G_p .

1.3 Related Work

Enzo now stands up and looks me straight in the eye: “What should I do? You’re the math guy, can you solve this?”

I instinctively curl my shoulders, “I’m more of a CS guy, actually. In any case, your problem doesn’t really fall into any of our usual categories. I mean, it’s not really *online*, since you know the whole input in advance. It’s also not *offline*, since we don’t know the length of the prefix where we will be measured. It’s also not really a computational issue, since you’re not computationally bounded. It is a question about *robustness*: performing well on all of a set of sub-instances; here the sub-instances are the prefixes.”

Hassin and Rubinfeld [12] gave a weighted matching algorithm that gave a $\sqrt{2}$ -performance guarantee for the total weight of the p -heaviest edges, for all p .

There are various works where robustness is with respect to a class of objective function. One prominent examples are those of scheduling under any L_p -norm [3,2]. Goel and Meyerson [11] gave a general scheme for minimizing convex cost functions, such as in load balancing problems. Robust colorings of intervals were considered in [9], where throughput in any prefix of the coloring classes went into the objective function.

Prefix optimization can be viewed as online algorithm with complete knowledge of the future, or competing against a *non-adaptive* adversary. Several lower bound constructions for online computation are actually lower bounds on prefix computation; most prominent cases are for the classical machine scheduling problems [8,4,1]. Prefix optimization corresponds to the extreme case of lookahead, and thus can perhaps shed some light on properties of online computation.

As of yet, only few papers have explicitly addressed prefix optimization. Faigle, Kern and Turán [8] considered online algorithms and lower bounds for various online problems, giving a number of lower bounds in the prefix style. They posed the question of a constant factor approximation for the Prefix Coloring problem. Dani and Hayes [6] considered online algorithms for a geometric optimization problem and explicitly compared the competitive ratios possible against adaptive and non-adaptive adversaries.

“So, it relates to this online algorithms racket, but basically nobody has done exactly this. I sure hope you can at least come up with some ideas, man.”

1.4 Our Results

We give nearly tight results for the best possible performance ratios for prefix versions of several fundamental optimization problems (in particular, those of Enzo, Beppe and Orsino).

We first consider the PREFIX IS problem and give an algorithm whose performance ratio on interval graphs is $O(\log \alpha)$, where $\alpha = \alpha(G)$ is the independence number of the graph. We derive a matching lower bound for any (randomized or deterministic) algorithm for the problem on interval graphs. We further give a randomized algorithm that achieves this ratio on general graphs. The algorithm is shown to achieve a logarithmic performance ratio for a wide class of maximum subset selection problems in the prefix model, including maximum clique, 0/1-knapsack, maximum coverage, and maximum k -colorable subgraph. For all of these problems the performance ratio is $O(\log(\alpha(I)))$, where $\alpha(I)$ is the value of an optimal solution for the complete input sequence I .

For the PREFIX VERTEX COVER problem we show that any algorithm has a performance ratio of $\Omega(\sqrt{n})$, where n is the size of G . We give a deterministic algorithm that achieves this bound.

Finally, for PREFIX COLORING we give an algorithm whose performance ratio is 4. We further show that no algorithm achieves a ratio better than 2.

2 Enzo's Effectiveness Issue: Prefix Independent Set

“So, my friend, here's what I can tell you about your problem.”

Let $\alpha = \alpha(G)$ be the independence number of the given graph G . We first present an approximation algorithm for PREFIX IS in interval graphs and then give a simple algorithm for general graphs.

Suppose that G is an interval graph. Consider the following algorithm, A_{int} , which uses as additional input parameter some $t \geq 1$. For the ordered sequence of intervals in G , let G_i be the shortest prefix satisfying $\alpha(G_i) \geq \alpha^{i/t}$, for $1 \leq i \leq t$. Algorithm A_{int} proceeds in the following steps.

1. Find in G_1 an IS of size $\alpha(G_1)$.
2. For $2 \leq i \leq t$ in sequence, find for G_i an IS, I_i , of size at least $\alpha(G_i)/t$, such that I_i conflicts with at most a fraction of $1/t$ of the vertices in each I_r , $1 \leq r \leq i - 1$, and delete from I_r the vertices conflicting with I_i .
3. Output the solution $I = \cup_{i=1}^t I_i$.

“Was there a message to this? I thought you'd come up with something simpler. Why this sequence of solutions that you keep chipping off?” Enzo quizzed.

“When you put it formally it starts to look more complicated than it really is. The point is that we have to come up with some solution early on, even if it would be better for the long haul to just wait. This early piece of the input is G_1 . We try to find a small part of the solution that doesn't mess up the rest of the input sequence by too much.”

“We then need to repeat this on several prefixes, picking some portion of the available solution; if it requires smashing some of the family treasures, so be it, but make sure it’s only a small fraction, ‘minor collateral damage’ ”.

In analyzing A_{int} , we first need to show that a ‘good’ independent set can be found in each iteration, as specified in Step 2.

Lemma 1. *Let t and the interval graph G_i be defined as above, and let k be a number between 1 and t . Let I be an independent set of G_{k-1} , partitioned into sets I_1, I_2, \dots, I_{k-1} where $I_r \subseteq V(G_r)$, for $1 \leq r \leq k - 1$. Then, there exists an IS I_k in G_k of size at least $(\alpha(G_k) - 2\alpha(G_{k-1}))/t$ which conflicts with at most $1/t$ -fraction of the intervals in each I_r , $1 \leq r < k$.*

Proof. We say that an interval a flanks an interval b if a overlaps one of the endpoints of b , i.e., a and b overlap but a is not contained in b . Let J be a maximum IS of G_k , and let J' be the set of intervals in J that do not flank any interval in I . Observe that $|J'| \geq |J| - 2|I| \geq \alpha(G_k) - 2\alpha(G_{k-1})$. Note that each interval in J' intersects at most one interval in I .

For sets A and B of intervals, let $N_B[A] = \{b \in B \mid \exists a \in A, a \cap b \neq \emptyset\}$ be the set of intervals in A that overlap some interval in B . For each interval a in I let the weight of a be the number of intervals in J' intersecting a , or $wt(a) = |\{b \in J' : a \cap b \neq \emptyset\}| = |N_{J'}[\{a\}]|$. For each $j = 1, \dots, k - 1$ in parallel, find a maximum weight subset Q_j in I_j of size $|I_j|/t$, and let $Q = \cup_{j=1}^{k-1} Q_j$. Finally, let $I_k = N_{J'}[Q]$ be the claimed set of intervals from G_k .

By construction, I_k is an IS and is of size at least $\sum_{a \in I} wt(a)/t = |J'|/t \geq (\alpha(G_k) - 2\alpha(G_{k-1}))/t$. Also, by definition, I_k conflicts with a set of size $|I_r|/t$ from each set I_r , $r = 1, \dots, k - 1$. Hence, the claim. ■

Theorem 1. *A_{int} yields a performance ratio of $O(\log \alpha)$ for PREFIX IS on interval graphs.*

Proof. Let $t = \log_3 \alpha$. Then, $\alpha(G_k) = 3\alpha(G_{k-1})$, for each $1 < k \leq t$. We distinguish between two cases.

- (i) Suppose that the prefix \hat{G} presented to A_{int} is shorter than G_1 , then A_{int} outputs an empty set, while OPT has an IS of size at most $\alpha^{1/t}$. We get that

$$\frac{OPT(\hat{G})}{A_{int}(\hat{G}) + 1} \leq \alpha^{1/t} \leq 3 .$$

- (ii) Otherwise, let k be the maximum value such that $G_k \subseteq \hat{G}$. By the construction in Step 2, each IS reduces the size of any preceding IS at most by factor $1/t$. Hence, by Lemma 1 we get that

$$|I_k| \geq \frac{\alpha(G_k) - 2\alpha(G_{k-1})}{t} \left(1 - \frac{1}{t}\right)^{t-k} \geq \frac{\alpha^{k/t}}{3t} \left(1 - \frac{1}{t}\right)^{t-k} \geq \frac{\alpha^{k/t}}{3t} \cdot e^{-1} .$$

Since $OPT(\hat{G}) \leq \alpha(G_{k+1}) = \alpha^{(k+1)/t}$ and $A_{int}(\hat{G}) \geq |I_k|$, we get a ratio of $O(t\alpha^{1/t}) = O(\log \alpha)$. ■

“This may not be exactly what you were looking for, Enzo, but it is quite interesting to some of us. Particularly the fact that the ratio is in terms of the optimal solution size, α , rather than some general property of the input, like the number of items, n .”

2.1 Matching Lower Bound

“You know, what you’ve come up with is alright. But, maybe if we think a bit harder, we could get solutions that are always just a few percent off the best possible. Can’t you get one of those really smart guys, like Luca or Pino, to help you out? Or better yet, one of those hot-shot women,” Enzo grinned.

“Easy, easy. There’re limits to everything. In fact, what we outlined above is essentially the best possible.

Theorem 2. *Any algorithm (even randomized) for PREFIX IS in interval graphs has performance ratio of $\Omega(\log \alpha)$.*

Proof. Let α be a number, and let $k = \log \alpha + 1$. Consider the graph $G = (V, E)$ which consists of k subsets of vertices, V_1, \dots, V_k . The subset V_i consists of 2^{i-1} vertices numbered $\{v_{i,1}, \dots, v_{i,2^{i-1}}\}$. The set of edges in G is given by $E = \{(v_{i,h}, v_{j,\ell}) \mid 1 \leq i < j \leq k, 1 \leq h \leq 2^{i-1}, 2^{j-i}(h-1) + 1 \leq \ell \leq 2^{j-i}h\}$. In the ordered sequence representing G , all the vertices in V_i precede those in V_{i+1} , for $i = 1, 2, \dots, k-1$. The graph G can be represented as an interval graph where a vertex $v_{i,h}$ corresponds to the interval $[(h-1)2^{k-i}, h2^{k-i}]$; see Fig. [1](#).

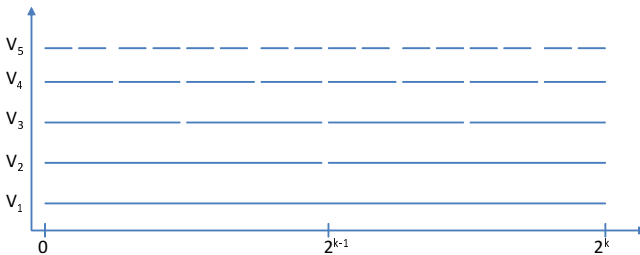


Fig. 1. Interval graph that gives a $\Omega(\log \alpha)$ lower bound

In view of Yao’s lemma we give a probability distribution over the prefixes of G and upper bound the expected performance of any deterministic algorithm over this distribution. Let $U_i = \cup_{j=1}^i V_j$. With probability $1/(2^i(1 - 2^{-k}))$ the prefix is $\hat{G} = G[U_i]$, for $i = 1, 2, \dots, k$.

First observe that

$$\mathbb{E}[OPT(\hat{G})] = \sum_{i=1}^k \Pr[\hat{G} = G[U_i]] \cdot |U_i| = \sum_{i=1}^k \frac{1}{2^i} \cdot 2^{i-1} \geq \frac{\log \alpha}{2}.$$

We now show that $\mathbb{E}[A(\hat{G})] \leq 1$, which yields the theorem.

Consider any given deterministic algorithm A that produces an independent set I_A on G . For any $v \in V_i$, let $N_{IS}(v)$ be the set of neighbors of v in $V \setminus U_i$ that are selected by A into I_A , and let $\hat{N}_{IS}(v) = N_{IS}(v) \cup \{v\}$. We denote by $A(v) = |\hat{N}_{IS}(v)|$ the increase in $|I_A|$ due to the selection of vertices in $\hat{N}_{IS}(v)$. We say that vertex $v \in V_i$ is *possible* for A if there are no edges between v and any of the vertices selected by A in U_{i-1} .

Claim 1. Let $v \in V_i$ be a possible vertex for A , $1 \leq i \leq k$. Then, if v is not selected for I_A , we have that $\mathbb{E}[A(v)|v \text{ not selected}] \leq \mathbb{E}[A(v)|v \text{ is selected}] = 1$.

Proof. The proof is by backward induction. For the base case we take $i = k$. Clearly, if A does not select the vertex v then $A(v) = 0$, since V_k is the last subset of vertices, and the claim holds. Now, assume that the claim holds for all vertices v in V_i , then we show that it holds also for the vertices in V_{i-1} . It is easy to show (details omitted) that $\Pr[V_{i+1} \subseteq \hat{G} | V_i \subseteq \hat{G}] = 1/2$, for $i = 1, 2, \dots, k-1$. Observe that G is constructed so that if a vertex $u \in V_i$ is adjacent to a vertex $v \in V_j$ and v is adjacent to vertex $w \in V_k$, for $i < j < k$, then u is also adjacent to w ; namely, the presentation ordering of the graph is transitive. If a vertex $v_{i-1,h}$ is not selected, then A can select its neighbors in V_i : $v_{i,2h-1}, v_{i,2h}$, and we get that

$$A(v_{i-1,h}) \leq \frac{1}{2} (\mathbb{E}[A(v_i, 2h-1)] + \mathbb{E}[A(v_i, 2h)]) .$$

From the induction hypothesis we have $\mathbb{E}[A(v_i, 2h-1)] = \mathbb{E}[A(v_i, 2h)] \leq 1$. ■

To complete the proof of the theorem we note that, if A selects for the solution the single vertex in V_1 then $A(\hat{G}) = 1$; else, by Claim 1, we get that $A(\hat{G}) \leq 1$. ■

2.2 Prefix IS in General Graphs

“Allow me to entertain ourselves by generalizing the problem you posed to independent sets in general graphs.

For general graphs, we can argue tight bounds on performance guarantees. Let G_1 be the shortest prefix of the input graph G for which $\alpha(G_1) = \alpha_1$ where $\alpha_1 = \lceil \sqrt{\alpha} \rceil$. The algorithm A_{gen} finds an independent set of size α_1 in G_1 and simply outputs this set.

Theorem 3. *Algorithm A_{gen} yields a performance ratio of at most $\sqrt{\alpha}$ for PREFIX IS.*

Proof. We distinguish between two types of prefixes given to the algorithm. Suppose the prefix is strictly shorter than G_1 . Then an optimal algorithm yields an IS of size at most $\alpha_1 - 1$, for a performance ratio of at most $\alpha_1 - 1 \leq \sqrt{\alpha}$. On the other hand, if G_1 is contained in the prefix \hat{G} then the approximation ratio is at most $\alpha/\alpha_1 \leq \sqrt{\alpha}$. ■

We can easily argue a matching lower bound.

Theorem 4. *The performance ratio of any deterministic algorithm for PREFIX IS is at least $\lfloor \sqrt{\alpha} \rfloor$. It is also $\Omega(\sqrt{n})$.*

Proof. Let $N = \lfloor \sqrt{n} \rfloor$. Consider the following complete bipartite graph $G = (U, V, E)$. The vertices U are $1, 2, \dots, N$, while V has $N + 1, \dots, n$. The ordering of the graph is by vertex number.

A deterministic algorithm A run on G can pick either vertices from U or from V . If it picks vertices from U , then on the prefix $B = G$ the optimal solution is of size $n - N$, while the algorithm solution is of size at most N , giving a ratio of at least $n/N - 1 \geq \sqrt{n} - 1$. The ratio is also at least $\alpha/N \geq \alpha/\lfloor \sqrt{\alpha} \rfloor \geq \sqrt{\alpha}$. On the other hand, if it picks vertices from V , then on the prefix $B' = G[U]$, the subgraph of G induced by U , $A(B') = 0$ while $\alpha(G') = N$, for a ratio of $N = \lfloor \sqrt{n} \rfloor \geq \lfloor \sqrt{\alpha} \rfloor$. Hence, the performance ratio of A on G is at least $N = \sqrt{n} \geq \sqrt{\alpha}$. ■

“You haven’t said anything here about coin flips. Can they help?”, Enzo inquired.

“For your problem of intervals, they don’t. But, for the general problem, on general graphs, a simple randomized approach does improve the situation dramatically. In fact, it’s kind of neat to phrase it in terms of still more general problem framework.”

In the following we consider a wide class of maximization subset selection problems in the prefix model.

Definition 1. *A problem Π is hereditary if for any input I of Π , if $I' \subseteq I$ is a feasible solution for Π , then any subset $I'' \subseteq I'$ is also a feasible solution.*

Note that many subset selection problems are hereditary. This includes maximum independent set, maximum clique, 0/1-knapsack, maximum coverage, and maximum k -colorable subgraph, among others.

Given an input I for a subset selection problem Π , let $\alpha = \alpha(I)$ be the size of an optimal solution for I .

Theorem 5. *Let Π be a hereditary maximum subset selection problem. Then, there is an algorithm for Π with a performance ratio of $O(\log(\alpha))$ for the PREFIX- Π problem.*

Proof. Consider the following algorithm. Let P be an input for Π with optimal value α and let $k = \lceil \lg \alpha \rceil$. Define prefixes P_1, \dots, P_k of P where $P_k = P$ and for $i = 1, \dots, k - 1$, P_i is the shortest prefix with $\alpha(P_i) \geq 2^{i-1}$. The algorithm now selects one of the prefixes P_i uniformly at random, each with probability $1/k$.

Let j be such that the prefix \hat{P} presented satisfies $P_j \subseteq \hat{P} \subseteq P_{j+1}$. Since \hat{P} is non-empty, it must contain the unit prefix P_1 . Then, the value of the optimal solution is at most twice the value of the solution for P_j . With probability $1/k$ our algorithm obtains an optimal solution on P_j . Hence, the expected size of the solution found by the algorithm is at least $1/(2k)$ fraction of optimal. ■

Corollary 1. *There is a randomized $O(\log \alpha)$ -approximation algorithm for PREFIX IS.*

3 Beppe's Guarding Business: Prefix Vertex Cover

"Beppe situation must be easier. I've heard this vertex cover problem, as you call it, is much easier" Enzo remarked philosophically.

"It is in many respects easier. For instance, it's easy when the optimal solution is small (or Fixed Parameter Tractable, as we say), which the general independent set problem is not. And it's easily approximable in polynomial time within factor 2, while the IS problem is notoriously hard."

"Yeah, you should know. You're the one who keeps doing IS in one way or another, in spite of these pathetic approximations."

"Hey, no need to get personal here, buddy. We all do what we can. But, back to Beppe's problem, it turns out to be actually harder than your prefix IS problem! Randomization won't help him at all."

Theorem 6. *Any algorithm for PREFIX VERTEX COVER has performance ratio of at least \sqrt{n} .*

Proof. We use the complete bipartite graph $G = (U, V, E)$ from Theorem 4, where U has nodes $1, 2, \dots, N = \lfloor \sqrt{n} \rfloor$ and V nodes $N + 1, N + 2, \dots, n$.

If any vertex in U is missing in a cover, then we need to select all the vertices in V . Thus, the only minimal vertex covers for B are U and V . Any randomized algorithm R_{vc} selects one of these solutions with probability at least $1/2$. If $C = U$ with probability at least $1/2$, then for a prefix \hat{B} that consists of all the vertices in U we get that $\frac{R_{vc}(\hat{B})}{OPT(\hat{B})+1} = \Omega(\sqrt{n})$, since $E = \emptyset$. On the other hand, if V is selected for the solution with probability at least $1/2$, then for the prefix $\hat{B} = B$, we have that $\frac{R_{vc}(\hat{B})}{OPT(\hat{B})+1} \geq \frac{n-\sqrt{n}}{2(\sqrt{n}+1)} = \Omega(\sqrt{n})$, since an optimal solution is $C = U$. ■

We now show that a matching upper bound is obtained by a deterministic algorithm. Let $G_1 = (V_1, E_1)$ be the prefix graph for which the minimum vertex cover is of size at least \sqrt{n} for the first time. A_{vc} finds a minimum vertex cover $S \subseteq V_1$ for G_1 and outputs $S \cup (V \setminus V_1)$.

Theorem 7. *Algorithm A_{vc} yields a ratio of \sqrt{n} to the optimal for PREFIX VERTEX COVER.*

Proof. We note that if $\hat{G} \subseteq G_1$ then A_{vc} outputs $S' \subseteq S$ where $|S'| \leq \sqrt{n}$, while an optimal algorithm may output an empty set. If $G_1 \subseteq \hat{G}$ then $|S \cup (V \setminus V_1)| \leq n$, while $OPT(\hat{G}) \geq OPT(G_1) \geq \sqrt{n}$. The claim follows. ■

4 Orsino's Oven Schedule: Prefix Graph Coloring

"What's your take then on Orso's situation?" Enzo asked me the following evening. Is he dug as deep as Beppe?"

“Actually, this looks much more promising” I replied. “We can use a standard trick of the trade called *doubling* to come out pretty well.”

“Play double-or-nothing until we finally win” he suggested hopefully.

“If you like. We use the nice property of geometric sums, i.e. $1 + 2 + 4 + 8 + \dots + 2^k$ that they add up to not too much, or only twice the last term.”

“Ah, so we first handle the first node, then the next two, then the next four, and so on?”

“You’re catching on, Enzo, but we actually need to use it slightly differently. We first find the initial set of order that can be laid on a single plate. Then, the next prefix that can be laid onto two plates. Third, the sequence of the following orders for which four plates suffice. And so on, doubling the number of plates in each step.”

“Ok! So we use new set of plates for each of these, uh, *groups*.”

“Exactly. And why is that ok?”

“Because they add up to not too much! Maybe I should try this CS business; you think I might have a shot at a Gödel award?”

For $k = 1, 2, \dots$, let t_k be the largest value such that $\alpha(G_{t_k}) \leq 2^k$, and let $t_0 = 0$ for convenience. The algorithm A simply colors each set $V_{t_k} \setminus V_{t_{k-1}}$ with 2^k fresh colors.

Theorem 8. *The algorithm A yields a performance guarantee of 4 for PREFIX COLORING.*

Proof. Let p be a number, $1 \leq p \leq n$, and let k be the smallest number such that $2^k \geq \alpha(G_p)$. So, $\alpha(G_p) \geq 2^{k-1} + 1$. Observe that

$$A(G_p) \leq \sum_{i=0}^k 2^i = 2^{k+1} - 1 < 4 \cdot 2^{k-1} < 4\alpha(G_p).$$

Since this holds for all p simultaneously, the theorem follows. ■

“Can we also do better here in the randomized case?”

“Actually, yes, a little.”

Let β be a uniformly random value from $[0, 1]$, and let a_0, a_1, a_2, \dots be the sequence given by $a_i = \lceil \beta e^i \rceil$. Let t_k be the largest value such that $\alpha(G_{t_k}) \leq a_k$. Modify the algorithm A to use a_k colors on each set $V_{t_k} \setminus V_{t_{k-1}}$.

Theorem 9. *The modified algorithm A has randomized performance guarantee of e for Prefix Graph Coloring.*

The proof is similar to the arguments used for certain coloring problems with demands [7,10].

“So, is that the best we can do.”

“It’s the best that I can come up with, but it’s also provably close to the best possible.”

Proposition 1. *There is no algorithm with performance guarantee less than 2 for Prefix Graph Coloring.*

“But, I’ll leave that for you to figure out, hot shot :-)”

5 Epilogue

“This is all cute’n stuff, but it ain’t mean nothin out there in the field, does it?”

“It’s always hard to say where theoretical results kick in. It does though tell us something about how robust we can make computation. It doesn’t have to involve your boss, of course; it could be any unpredictable event like the electricity going off. You could think of this as a sort of defensive problem-solving. In these days of global security threats, ain’t that what we all have to concern ourselves with?”

“Yeah, or you might have your little island economy suddenly going off the cliff”, Enzo chuckles. “Good luck in cashing in on these ideas...”

References

1. Albers, S.: Better bounds for online scheduling. In: STOC, pp. 130–139 (1997)
2. Azar, Y., Epstein, A.: Convex programming for scheduling unrelated parallel machines. In: STOC (2005)
3. Azar, Y., Epstein, L., Richter, Y., Woeginger, G.J.: All-norm approximation algorithms. *J. Algorithms* 52, 120–133 (2004)
4. Bartal, Y., Karloff, H.J., Rabani, Y.: A better lower bound for on-line scheduling. *Inf. Process. Lett* 50(3), 113–116 (1994)
5. Borodin, A., El-Yaniv, R.: Online computation and competitive analysis. Cambridge University Press, Cambridge (1998)
6. Dani, V., Hayes, T.P.: Robbing the bandit: Less regret in online geometric optimization against an adaptive adversary. In: SODA (2006)
7. Epstein, L., Levin, A.: On the Max Coloring Problem. In: Kaklamanis, C., Skutella, M. (eds.) WAOA 2007. LNCS, vol. 4927, pp. 142–155. Springer, Heidelberg (2008)
8. Faigle, U., Kern, W., Turán, G.: On the performance of on-line algorithms for partition problems. *Acta Cybernetica* 9, 107–119 (1989)
9. Fukunaga, T., Halldórsson, M.M., Nagamochi, H.: Robust cost colorings. In: SODA (2008)
10. Gandhi, R., Halldórsson, M.M., Kortsarz, G., Shachnai, H.: Approximating non-preemptive open-shop scheduling and related problems. *ACM Transactions on Algorithms* 2(1), 116–129 (2006)
11. Goel, A., Meyerson, A.: Simultaneous optimization via approximate majorization for concave profits or convex costs. *Algorithmica* 44, 301–323 (2006)
12. Hassin, R., Rubinstein, S.: Robust matchings. *SIAM J. Disc. Math.* 15(4), 530–537 (2002)

Managing Change in the Era of the iPhone

Patrick Healy

Computer Science Department,
University of Limerick, Ireland
patrick.healy@ul.ie

Abstract. In spite of futurologists' best predictions change has not gone away, and it is probably prudent that we not expect this to change any time soon. In this paper we consider the challenges of change, particularly in today's era of smartphones. We consider different types of change that can arise such as when in the presence of cooperative or uncooperative agents. We propose practical solutions for those who may have difficulty coping with change or with feeling weighed down by it.

1 Introduction

It is likely that every person at some time in their life has, as they have stood idly in line waiting to pay for items they wish to buy, mused on the different ways they could pay for their goods. “Cash or credit card?” is the way this is usually put when they reach the shop assistant at the head of the queue. Although the futurologists have long predicted a cashless society [6,11] it has not come to pass and we believe that that future is a long way off yet; in the meantime there will be cash. With cash comes change, and it is the management of this change that we consider here.

As the purchaser stands in line if cash is their preferred option it is also likely that they will have looked in their purses or fished out a handful of coins from their pocket and pondered exactly which coins to use for payment. Some are happy to tender a coin or paper money that covers the amount and happily accept the change, perhaps emptying their purse's contents into a jar nightly. Others, however, may be more weight-conscious. For them the goal is to keep to a minimum at all times what is jangling in their pockets or purses. So they find the selection of coins that adds up to the amount of the purchase and that is heaviest. But this may not be a trivial task.

The more enterprising punter may even spot an opportunity: a 1.80€ cup of coffee can certainly be paid for with a 2€ coin but with a 20c and 10c coin also weighing their pocket down, tendering 2.30€ yields a single 50c coin in return. *Laissez les bon temps rouler.* But how should we most advantageously select from the many probable ways of paying? And then, even, can we be sure of the shop assistant's assistance in our little scheme: after all, we rely on their willingness to give us the change our way. Further, if we *could* devise an optimal algorithm would it be transferable to an arbitrary currency: plug-and-pay?

In this paper we consider these questions. We develop optimal algorithms for solving them, some being possible to execute by hand, with others requiring the assistance of an electronic computing device. Such a computing device will need to be readily available so we consider how some of today’s portable, hand-held computers such as PDAs or smartphones could assist.

The remainder of the paper is as follows. In the following section we provide background and related work, outlining the problem(s) and what is known about them. We then propose a simple ILP-based model in Section 3 that, though not practical, will be motivational for what comes later. In Section 4 we then propose an efficient solution to the case of the cooperating sales assistant when currencies allow, and more computationally intensive solutions otherwise. Since a principle aim of this work was that it be implementable on one of today’s hand-held computing devices, we then discuss in general terms the design of an “app”, that we call **ChangeManager** in Section 5. We look at how the user might interact with one and, in particular, how the information in the device is kept current. We conclude the paper in Section 6.

2 Background and Related Work

As we have described, our goal can be simply described as paying an amount using a collection of coins so that the resulting change in our pocket is as light as possible. This may involve overpaying using heavier coins. However, there may be other constraints on how we can select our coins. In Ireland, for one, a transaction may be limited to 50 coins¹, no matter how weight-advantageous it would be to the buyer to pay with more. (Pity the poor punter who has just won big on the “Penny Falls” in the local amusement arcade.)

Our problem has similarities to several classical (and related) problems. We assume that we are using a single currency \mathcal{C} comprising m coins of value $1 = c_1, c_2, \dots, c_m$ where $c_i < c_{i+1}$. Using the m -vector $C = (c_1, c_2, \dots, c_m)$ and the inner (dot) product operation we can express any amount K as another m -vector $X = (x_1, x_2, \dots, x_m)$ called a *representation* with $K = X \cdot C$. Since $c_1 = 1$ there is always at least one representation of K and there may be several. The *size* of a representation, $|X| = X \cdot (1, 1, \dots, 1)$, is the number of coins used.

We might ask about the number of coins used to represent K and this gives rise to the change making problem, our first classical problem.

CHANGE MAKING

Find the representation of K of minimum size. Written as an integer linear program this is

¹ From Wikipedia [13]: *According to the Economic and Monetary Union Act, 1998 of the Republic of Ireland . . . No person, other than the Central Bank of Ireland and such persons as may be designated by the Minister by order, shall be obliged to accept more than 50 coins denominated in euro or in cent in any single transaction.*

$$\begin{aligned} & \min \sum_{i=1}^m x_i \\ \text{s. t. } & \sum_{i=1}^m c_i x_i = K \qquad x_i \in \mathbb{N} \end{aligned}$$

In this case note that there is no upper bound on the usage of a coin. The *greedy* strategy turns out to be quite effective for solving CHANGE MAKING heuristically, although it is no panacea. In order to make change for an amount K a greedy solution uses the largest coin not greater than K and repeats recursively on the remainder. To see that this strategy is not perfect consider a currency comprising three coins 1, 5 and 6 and the amount $K = 10$. A *greedy* strategy would take the 6-coin once and 4 uses of the 1-coin for a solution that uses 5 coins while the optimal strategy is to use the 5-coin twice.

The following definition clarifies this distinction.

Definition 1. *Given a coin system, \mathcal{C} , comprising coins $1 = c_1 < c_2 < \dots < c_m$, of weight w_i respectively, we denote by $M(x)$ the number of coins in the representation of x of smallest size. $G(x)$ denotes the size of the representation of x found by the greedy strategy.*

A coin system (currency) is called *canonical* if the greedy coin selection strategy yields the minimum number of coins. Canonical coin systems have been considered by many people. Chang and Gill [4] first studied such coin systems and identified upper and lower bounds by showing that a system of coins $1 = c_1 < c_2 < \dots < c_m$ is canonical when $M(x) = G(x)$ for all x in the range $c_3 \leq x < \frac{c_m(c_m c_{m-1} + c_m - 3c_{m-1})}{c_m - c_{m-1}}$. This interval is $O(c_m^3)$; Kozen and Zaks [7] narrow this interval considerably by showing that if a counterexample to $M(x) = G(x)$ exists then it must exist in the range $c_3 + 1 < x < c_m + c_{m-1}$. They show, further, that it is possible to avoid having to compute the optimal representation of every x in this range while searching for a counterexample, which results in an $O(m c_m)$ -time algorithm. Pearson [9] presents a strongly polynomial $O(m^3)$ -time algorithm. For the case of a *tight* coin system, Cai improves this to $O(m^2)$ [3]; a system is tight if there exists no counterexample smaller than c_m .

Complete characterizations of coin systems for fixed m have been considered by several authors [7, 3]. Adamaszek and Adamaszek [1] provide such a characterization for $m = 5$. They also consider the circumstances under which a sub-currency (a subset of the currency’s coinage) of a canonical currency is, in turn, canonical.

We will return to this problem later when we consider the optimality of our algorithm.

In order to pay for an item we will have a fixed number of coins of each type in our pocket. We may express these bounds by a vector $B = (b_1, b_2, \dots, b_m)$ ². We can tell if we can pay for goods *exactly* with the coins in our pocket and this is called the subset sum problem.

SUBSET SUM

Given a set of numbers $C = (c_1, c_2, \dots, c_m)$, the number of occurrences of each number $B = (b_1, b_2, \dots, b_m)$, and a number K , find a vector $X = (x_1, x_2, \dots, x_m)$, $0 \leq x_i \leq b_i$, and $X \cdot C = K$.

$$\begin{aligned} \min \quad & 1 \\ \text{s. t.} \quad & \sum_{i=1}^m c_i x_i = K && 0 \leq x_i \leq b_i \\ & && x_i \in \mathbb{N} \end{aligned}$$

The CHANGE MAKING and SUBSET SUM problems are special cases of the knapsack problem.

KNAPSACK

Given a set of items $C = \{c_i : 1 \leq i \leq m\}$, each item c_i of weight w_i and of profit p_i , respectively, and a container of weight-limit W , find the most profitable way of filling the container while not over-filling it.

$$\max \sum_{i=1}^m p_i x_i \tag{1}$$

$$\text{s. t.} \sum_{i=1}^m w_i x_i \leq W \tag{2} \qquad 0 \leq x_i \leq b_i$$

$$x_i \in \mathbb{N} \tag{3}$$

By equating p_i and w_i we get SUBSET SUM and by setting $p_i = -1$ with sufficiently large b_i , say, $b_i = \lceil \frac{W}{w_i} \rceil$, we get CHANGE MAKING. An alternative formulation called 0-1 KNAPSACK distinguishes between the b_i occurrences of item c_i in a multiset and restricts the resulting x_i s 0 or 1.

3 ILP-Based Solutions

KNAPSACK is a much-studied problem dating back to Dantzig’s 1957³ dynamic programming solution⁵. We can adapt the KNAPSACK formulation of Equations (1) – (3) to make a first model of our problem. Given a set of coins $C = \{c_i : 1 \leq i \leq m\}$ of weight w_i and an amount K , find the heaviest set of coins that sums to K .

² Technically, the problems we present here have bounds included in their definitions. Similar problem statements exist without the bounds; their complexity is unchanged.

³ Among the items to feature in his knapsack in this paper was a geiger counter – for the times that were in it, he wrote!

$$\max \sum_{i=1}^m w_i x_i \tag{4}$$

$$\text{s. t. } \sum_{i=1}^m c_i x_i = K \qquad 0 \leq x_i \leq b_i \tag{5}$$

$$x_i \in \mathbb{Z} \tag{6}$$

With respect to the KNAPSACK formulation note that the roles of w_i and c_i ($= p_i$) have been reversed. In order to avoid confusion with a reversed knapsack (which would be a forward facing knapsack) we call this problem COIN-KNAPSACK, to coin a phrase. Equation (6) now allows x_i to take on negative values, for otherwise we would be forced to have exact change always. In tandem with this, negative coefficients permit changing the \leq relation of Equation (2) to equality.

This represents a nice, compact modelling of our problem. However, a goal of ours is to implement **ChangeManager**TM on a modern smartphone-type device and to the best of our knowledge there is no implementation of *any* mathematical programming library on devices of this size. In spite of this seeming dead end it provides useful motivation.

The formulation of COIN-KNAPSACK given above in (4) – (6) can be more usefully expressed by separating the positive x_i s from the negatives. This can be written as

$$\begin{aligned} \max \quad & \sum_{i=1}^m w_i x_i - \sum_{i=1}^m w_i y_i \\ \text{s. t. } \quad & \sum_{i=1}^m c_i x_i - \sum_{i=1}^m c_i y_i = K \qquad 0 \leq x_i \leq b_i \\ & x_i, y_i \in \mathbb{N} \end{aligned}$$

The x variables now encode coins tendered by the buyer while y variables encode those returned as change. We can now add restrictions such as the 50-coin ($\sum_{i=1}^m x_i \leq 50$) limit we saw earlier. Since the x and y variables are not mutually constrained we can also separate the problem into the following pair of optimization problems to be solved in sequence.

$$\begin{aligned} \max \quad & z = \sum_{i=1}^m w_i x_i \\ \text{s. t. } \quad & \sum_{i=1}^m c_i x_i \geq K \qquad 0 \leq x_i \leq b_i \\ & \sum_{i=1}^m x_i \leq 50 \\ & x_i \in \mathbb{N} \end{aligned}$$

and

$$\begin{aligned} \min \quad & \sum_{i=1}^m w_i y_i \\ \text{s. t.} \quad & \sum_{i=1}^m c_i y_i = z - K \\ & y_i \in \mathbb{N} \end{aligned}$$

The second of this pair returns change for the amount that is K less than the value of the first problem in the *lightest* possible way. The first of this pair is bounded only by the availability of coins and will return the solution $x_i = b_i$. In a blatant abdication of responsibility we are, in effect, saying to the shopkeeper, “here’s what’s in my pocket, keep K for yourself and give me back the remainder as lightly as is possible.” Irresponsible though it may be, from an algorithmic point of view it is a useful strategy for, while the bounds on x_i are rigid, it is very reasonable to assume that the shopkeeper has an *unbounded* supply of each coin. We call this – the second of the above pair – the CHANGE TAKING problem. We can immediately see that it has as a special case the CHANGE MAKING problem where $w_i = 1, 1 \leq i \leq m$.

4 The Change Taking Problem

From the preceding, the CHANGE TAKING problem is quite similar to the classic KNAPSACK problem and so dynamic programming methods are likely to work; a straightforward implementation [2] runs in pseudo-polynomial time $O(nC)$, where in this immediate case n is the number of coins we present to the shopkeeper and C is the value of those coins, z , less the amount to pay, K . Pisinger proposes a *balancing* extension to the dynamic programming algorithm that, for bounded coin values, runs in time linear in the number of coins and the magnitude of the largest coin, $O(nc_m)$. This reduces by a factor of n the running time of the straightforward algorithm. With a more sophisticated algorithm, memory requirements, which could likely be a problem on a small computing device, is addressed also by Pisinger, where the space requirement is reduced by an $O(\log m/m)$ factor, m being the larger of C and the value of the optimal solution, z [10].

Can we do better than this? It turns out that, as with CHANGE MAKING, there are certain circumstances when the greedy solution is optimal. Recall that Pearson showed that it is possible to test if a coin system is canonical in polynomial time [9]. We conjecture that it is also true for a *weighted* set of coins; we stop just short of it by exhibiting an $O(mc_m)$ -time bound analogous to Kozen and Zaks [7] using an analysis similar to theirs. The following definition specifies the weighted analogue of $M(x)$ in the CHANGE MAKING problem without making any assumptions about coins’ weights with respect to each other.

Definition 2. Given a coin system, \mathcal{C} , comprising coins $1 = c_1 < c_2 < \dots < c_m$, of weight w_i respectively, we denote by $M_w(x)$ the minimum weight (lightest) representation of x . $G_w(x)$ denotes the weight of the representation of x found by the greedy strategy. \mathcal{C} is weight-canonical if $M_w(x) = G_w(x)$.

Whereas $M(x)$ and $G(x)$ denote the number of coins in a representation of x ; $M_w(x)$ and $G_w(x)$ denote the weights of those representations, respectively.

When considering weighted greedy representations we must firstly ask if each coin itself is weight-canonical. That is, is the coin c_i itself the lightest representation of $x = c_i$? Clearly this is a necessary condition for a weight-canonical coin system. For arbitrary i this may be solved in pseudo-polynomial time. However, by virtue of Theorem 1 and what follows it we need only concern ourselves with a small number of base cases.

Lemma 1. It is possible to determine if c_2 and c_3 are weight-canonical in constant time.

Proof. c_2 is weight-canonical if and only if $w_2 \leq c_2 \times w_1$.

The possible representations of $x = c_3$ is one use of coin c_3 or c_3 uses of c_1 or some combination of coins c_1 and c_2 . For the latter we can represent c_3 using $l = \lfloor \frac{c_3}{c_2} \rfloor$ instances of c_2 and $c_3 - lc_2$ instances of c_1 . Coin c_3 is weight-canonical if and only if $w_3 \leq \min\{c_3 \times w_1, l \times w_2 + (c_3 - lc_2)w_1\}$ (assuming that c_2 is weight-canonical).

All of the computations can be done in constant time. □

We now investigate when a coin system is weight-canonical. First, the following lemma gives an upper bound on the lightest representation of any amount x .

Lemma 2. For all x and coins $c_i \leq x$ in a coin system \mathcal{C} ,

$$M_w(x) \leq M_w(x - c_i) + w_i. \tag{7}$$

Further, equality holds if and only if there is an optimal representation of x that uses coin c_i .

Proof. The lightest representation of x cannot be greater than the sum of the lightest representation of $x - c_i$ and the weight of c_i so the inequality must hold. If equality holds in equation (7) then there is an optimal representation of x that derives from taking the optimal representation of $x - c_i$ with the additional w_i covered by the coin c_i . If there is an optimal representation of x that uses c_i then by the inequality above $M_w(x) - w_i \leq M_w(x - c_i)$. By removing c_i there is a representation of $x - c_i$ of weight $M_w(x) - w_i$. Thus equality holds.

If \mathcal{C} is not weight-canonical then the following theorem establishes bounds on where a counterexample must exist.

Theorem 1. Let \mathcal{C} be a coin system. If there exists an x such that $M_w(x) < G_w(x)$, then the smallest such x must lie in the interval

$$c_3 + 1 < x < c_m + c_{m-1}.$$

Proof. Since there is only one representation of $c_1 < x < c_2$ then $G_w(x) = M_w(x)$ must hold. By virtue of Lemma 1 $G_w(x) = M_w(x)$ holds for each of $x = c_1, c_2$ and c_3 . Likewise there is only one representation of $c_3 + 1$.

The upper bound can be proved analogously to Kozen and Zaks [7]. Let $x \geq c_{m-1} + c_m$. Using induction, assume that $G_w(y) = M_w(y)$ for all $y < x$. Suppose c_i is a coin used in some optimal representation of x . If $i = m$ then

$$\begin{aligned} G_w(x) &= G_w(x - c_m) + w_m && \text{by definition of } G_w \\ &= M_w(x - c_m) + w_m && \text{by the inductive hypothesis} \\ &= M_w(x) && \text{by Lemma 2} \end{aligned}$$

On the other hand, if $i < m$ then

$$\begin{aligned} G_w(x) &= G_w(x - c_m) + w_m && \text{by definition of } G_w \\ &= M_w(x - c_m) + w_m && \text{by the inductive hypothesis} \\ &= M_w(x - c_m - c_i) + w_m + w_i && \text{by Lemma 2} \\ &\leq G_w(x - c_m - c_i) + w_m + w_i && \text{by definition of } M_w \\ &= G_w(x - c_i) + w_i && \text{by definition of } G_w \\ &= M_w(x - c_i) + w_i && \text{by the inductive hypothesis} \\ &= M_w(x) && \text{by Lemma 2} \\ &\leq G_w(x) && \text{by definition of } M_w \end{aligned}$$

In either case $G_w(x) = M_w(x)$. □

It is possible to search this interval in pseudo-polynomial time for a counterexample, if it exists. This is based on the idea that we can easily compute recursively $G_w(x) = G_w(x - c_i) + w_i$ where $c_i \leq x < c_{i+1}$, and then check that $G_w(x) \leq G_w(x - c_i) + w_i$ for all $c_i < x$. We do not pursue this here but we reiterate that it may be possible to find a fully polynomial algorithm.

Further Conditions? In the proof of Lemma 1 we said that c_2 was weight-canonical if $w_2 \leq c_2 \times w_1$. That is, $w_2/c_2 \leq w_1/c_1$. Normalised weight appears to play an important role in determining if a currency is weight-canonical.

Definition 3. We call \mathcal{C} well-ordered if, for any two coins, $c_i < c_j, w_i^* = w_i/c_i \geq w_j/c_j = w_j^*$. That is, for increasing coin values in \mathcal{C} the normalised weights $w_i^* = w_i/c_i$ form a monotonically non-increasing sequence.

It is interesting to observe that examples exist where equality can hold amongst the coins' normalised weights. In the U.S. coin system the 5¢, 10¢ and 25¢ coins all have normalised weight of 0.044g/\$. Thus 1kg of any mixture of these coins has exactly the same value.

The well-ordered property is not a necessary condition for being weight-canonical: $\mathcal{C} = \{1, 2, 5, 10\}$ with respective weights $\{1, 1, 2.6, 5.0\}$ is not well-ordered since $w_3^* = w_3/c_3 = 0.52 > 0.5 = w_2/c_2 = w_2^*$ yet some quick calculations in the interval given by Theorem 1, [7, 14], show that it is weight-canonical. On

the other hand, if w_m^* is larger than some w_i^* then clearly $G_w(x)|_{x=c_m c_i} > M_w(x)$. Similarly c_1 's normalised weight must be largest.

4.1 The Story So Far...

The situation now is that when localizing **ChangeManager** for a new currency, the boys back in the lab determine if the currency is canonical using Lemma [□](#) and c_1 's and c_m 's normalised weights to check the preconditions, followed by the search for a counterexample in the range given by Theorem [□](#). The linear-time greedy algorithm will determine the correct change if the currency is canonical; a dynamic programming implementation will be required otherwise.

The user consults **ChangeManager**, which already knows the contents of their pocket, and tells it the amount required to pay. They then dump the contents of our pocket on the counter and gently coax the attendant to give them the change as advised by **ChangeManager**. Practicalities now arise. The attendant may not wish to deal with 22 pennies, 4 two-cent coins, 3 5-cent coins, and a 2€ coin for a 1.95€ purchase no matter how reassuring we are that a 50-cent coin is the answer.

This greedy approach does not accommodate side constraints such as an upper bound on the number of coins tendered, also. It appears that it is necessary to revert to the original goal where one tries to *maximise* the outflow (weight) all the time subject to bounds on the number of instances of any coin. To solve this we must again revert to a dynamic programming approach where it is straightforward to limit solutions to upper bounds on each coin type.

5 User-Interface Issues

The success of a tool like this will surely live or die on its ease of use. It is not practical, every time you wish to use it, to have to tell it what is in your pocket, in addition to the amount you wish to pay. Perhaps it might be acceptable to reset the “bank” each morning and, throughout the day, either explicitly pay through **ChangeManager** or update it for miscellaneous payments, such as, a tip at the end of meal. Mornings can be hard for the best of people – especially after the long night that added much to your pocket’s weight – and counting the remains of one’s pockets can be hard.

We propose an aid to the job of counting the contents of the user’s pocket. Though our solution is not quite at the stage of a robotic camera crawling in to the user’s pocket and counting *in situ* we believe our solution has merit. By laying out the coins in the user’s pocket on a flat surface we can use our smartphone / PDA to take a picture of the contents. Figure [□](#) below is a typical picture and illustrates some of the challenges that even this method faces – although a colour representation may help.

With the resulting photograph we run an edge-detector to determine the outlines of the coins. (As evidenced by face recognition software or the image recognition service kooaba, <http://www.kooaba.com/>, edge-detection software

propose here. In our case we can ask the user to disambiguate amongst possible matches by shading all coins that, within appropriate error bounds, are the same, by asking the user to identify this coin class. In effect, rather than sidelining the user, use the user for what they are really good at.

6 Conclusion

In this paper we have motivated whimsically the development of an application that would run on one of today's smartphones / PDAs. More usefully, we posed the CHANGE TAKING problem which is the weighted generalization of the CHANGE MAKING problem. We identified necessary preconditions, as well as a pseudo-polynomial-time algorithm, for recognising when a greedy solution to the problem of finding the minimum-weight decomposition of a value is optimal.

References

1. Adamaszek, A., Adamaszek, M.: Combinatorics of the change-making problem. *Eur. J. Comb.* 31(1), 47–63 (2010)
2. Bellman, R.E.: *Dynamic Programming*. Princeton Univ. Press, Princeton (1957)
3. Cai, X.: Canonical coin systems for change-making problems. In: *International Conference on Hybrid Intelligent Systems*, vol. 1, pp. 499–504 (2009)
4. Chang, S.K., Gill, A.: Algorithmic solution of the change-making problem. *J. ACM* 17(1), 113–122 (1970)
5. Dantzig, G.B.: Discrete-variable extremum problems. *Operations Research* 5(2), 266–277 (1957)
6. Dorn, J.A. (ed.): *The Future of Money in the Information Age*. Cato Institute (1997)
7. Kozen, D., Zaks, S.: Optimal bounds for the change-making problem. *Theor. Comput. Sci.* 123(2), 377–388 (1994)
8. Noelle, M., Penz, H., Rubik, M., Mayer, K.J., Holländer, I., Granec, R.: Dagobert - a new coin recognition and sorting system. In: Sun, C., Talbot, H., Ourselin, S., Adriaansen, T. (eds.) *DICTA*, pp. 329–338. CSIRO Publishing (2003)
9. Pearson, D.: A polynomial-time algorithm for the change-making problem. *Operations Research Letters* 33(3), 231–234 (2005)
10. Pisinger, D.: Dynamic programming on the word ram. *Algorithmica* 35(2), 128–145 (2008)
11. Rahn, R.W.: *The End of Money*. Discovery Institute (1999)
12. <http://sourceforge.net/projects/opencvlibrary/>
13. Wikipedia. http://en.wikipedia.org/wiki/Legal_tender#In_the_Republic_of_Ireland

The Computational Complexity of RACETRACK

Markus Holzer^{1,*} and Pierre McKenzie²

¹ Institut für Informatik, Universität Giessen,
Arndtstraße 2, D-35392 Giessen, Germany
holzer@informatik.uni-giessen.de

² Département d'I.R.O., Université de Montréal, C.P. 6128,
succ. Centre-Ville, Montréal (Québec), H3C 3J7 Canada
mckenzie@iro.umontreal.ca

Abstract. Martin Gardner in the early 1970's described the game of RACETRACK [M. Gardner, Mathematical games—Sim, Chomp and Race Track: new games for the intellect (and not for Lady Luck), *Scientific American*, 228(1):108–115, Jan. 1973]. Here we study the complexity of deciding whether a RACETRACK player has a winning strategy. We first prove that the complexity of RACETRACK reachability, i.e., whether the finish line can be reached or not, crucially depends on whether the car can touch the edge of the carriageway (racetrack): the non-touching variant is NL-complete while the touching variant is equivalent to the undirected grid graph reachability problem, a problem in L but not known to be L-hard. Then we show that single-player RACETRACK is NL-complete, regardless of whether driving on the track boundary is allowed or not, and that deciding the existence of a winning strategy in Gardner's original two-player game is P-complete. Hence RACETRACK is an example of a game that is interesting to play despite the fact that deciding the existence of a winning strategy is most likely not NP-hard.

1 Introduction

RACETRACK is a popular multi-player simulation pencil-paper game of car racing. The origin of the game is not clear, but most people remember this game from high school, where great effort and time was spent to become the RACETRACK champion, even at the expense of the said champion's school results. Variants of the game appeared all over the world under various names like, e.g., *Le Zip* in France or *Vektorrennen* in Germany. Here is the game description, literally taken from Gardner [4]: The game is played on math-paper where a racetrack is drawn. Then the cars are lined up at a grid position at the start line, and at each turn a player moves his car along the track to a new grid position subject to the following rules:

* Part of the work was done while the author was at the Département d'I.R.O., Université de Montréal, C.P. 6128, succ. Centre-Ville, Montréal (Québec), H3C 3J7 Canada, and the Institut für Informatik, Technische Universität München, Boltzmannstraße 3, D-85748 Garching bei München, Germany.

1. The new grid point and the straight line segment joining it to the preceding grid point must lie entirely within the track.
2. No two cars may simultaneously occupy the same grid point, i.e., no collisions are allowed.
3. Acceleration and deceleration are simulated as follows: a car maintains its speed in either direction or it can change its speed by only one distance unit per move—see Figure 1 for illustration. The first move following this rule is one unit horizontally or vertically, or both.

The first car to cross the finish line (point) wins. A car that collides with another car or leaves the track is out of the race.

Here we investigate the complexity of RACETRACK when played as a 1-player or 2-player game. Before describing our results, we note that the shape of the racetrack border is *a priori* arbitrary. Thus, in some far-fetched settings, merely verifying whether a move is valid, i.e., merely checking whether the new grid point and the straight line segment joining it to the preceding grid point lies entirely within the track, could be undecidable. To avoid such complications, we stick to a discrete version of the racetrack border, where the track is drawn along grid lines. This is a reasonable restriction, which does not change the practical appeal of the game. Figure 1 shows the discretization of an arbitrarily shaped track.

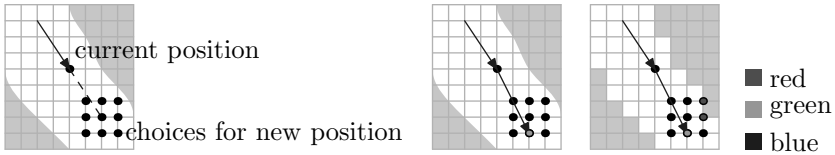


Fig. 1. (Left:) The arrow depicts the last move of the car—two squares east and three squares south—on the racetrack (drawn in white); the gray shaded area is outside of the racetrack. If the car maintains its speed it will follow the dashed line and go two squares east and three squares south again, but it can also reach one extra square north, south, east, or west of this point by changing speed. These extra points are marked by black dots. (Middle:) One out of nine legal moves are shown. (Right:) Discrete version of the RACETRACK game. Observe that certain movements of the car are not possible anymore if the border is *not* allowed for driving; here 7 out of 9 movements remain.

Solitaire RACETRACK will refer to the problem of deciding whether a single player can reach the finish line within an input-specified number of legal moves. By RACETRACK *reachability*, we will mean the simpler problem of deciding whether the single player can reach the finish line at all. In the first part of this paper, we reduce the *touching* variant of RACETRACK reachability (i.e., with the track border considered part of the driving area) to the undirected grid graph reachability problem, and deduce from [1] that this touching variant is NC^1 -hard and can be solved in deterministic logarithmic space. By contrast, and to our

initial surprise, we then show that the *non-touching* reachability variant is NL-complete, hence that the complexity of the RACETRACK reachability problem crucially depends on whether the car is allowed to touch the racetrack border or not. Finally, we settle that solitaire RACETRACK is NL-complete, too, regardless of whether driving on the track boundary is allowed or not.

In the second part of the paper, we turn to the 2-player game and prove that checking whether the first player has a winning strategy is P-complete. In particular, the 2-player game is efficiently solvable (in polynomial time). Consider the following “popular conjecture:”

Conjecture 1. All “fun and interesting” (2-player) games are NP-hard.

This “conjecture” attempts to capture when a “game” makes a game and it is wildly accepted in the algorithmic game theory community: in order to be interesting, a game purportedly needs enough complexity to be able to encode interesting (NP-hard) computational problems. And a game in P supposedly becomes boring because a player can quickly learn “the trick” to perfect play. The 2-player RACETRACK game, being fun and interesting to play, yet polynomial time solvable, is a rare example of a game that violates the implication of this conjecture.

2 Preliminaries

We assume familiarity with the basic concepts of complexity theory [7] such as the inclusion chain $AC^0 \subset NC^1 \subseteq L = SL \subseteq NL \subseteq AL = P$. Here AC^0 and NC^1 refer to the sets of problems accepted by polynomial size uniform families of Boolean {AND, OR, NOT}-circuits having, respectively, unbounded fan-in and constant depth, and, bounded fan-in and logarithmic depth. L is the set of problems accepted by deterministic logarithmic-space bounded Turing machines. SL and NL can be taken to be the sets of problems logspace-reducible to the undirected graph reachability (UGR) and to the directed graph reachability (GR) problems respectively. AL is the set of problems accepted by alternating logspace bounded Turing machines and P is the set of problems accepted by deterministic polynomial time bounded Turing machines. All the relationships depicted in the inclusion chain have been known for a quarter of a century, except for $L = SL$, shown by Reingold [8] in 2008.

Another particularly relevant reachability problem is undirected grid graph reachability (UGGR): given an $n \times n$ grid of nodes such that an edge only connects immediate vertical or horizontal neighbors, is there a path from node s to node t , where s and t are designated nodes from the grid? UGGR is NC^1 -hard under AC^0 reducibility, it belongs to L , yet it is not known to be L -hard [1]. Finally, we recall the GEN problem (generability problem), known to be P-complete [6] and defined as follows: given a finite set T , a binary operation \circ on T presented as a table, a subset S of T , and an element g in T , determine whether g is contained in the smallest subset of T that contains S and is closed under the \circ -operation.

The racetrack in a RACETRACK instance is encoded as a 2-dimensional array $R[i, j]$ indicating whether the grid point (i, j) is on the racetrack, on the racetrack border, on the start line or on the finish line.¹ When the grid point (i, j) happens to be on the racetrack border, 8 further bits of information are required, as follows: Fix a positive distance $\delta < 1$. For each of the non-grid points $(i + \delta, j)$, $(i - \delta, j)$, $(i, j + \delta)$, and $(i, j - \delta)$ we specify whether that point lies within the racetrack or not, and whether that point lies on the racetrack border or not.

3 Complexity of Solitaire RACETRACK

The single-player variant of RACETRACK naturally relates to graph reachability. Here we first analyse the *touching* and the *non-touching* variants of RACETRACK reachability. Observe the following: (i) If a single car moves north, south, east, or west by precisely one square, then the next move of the car can be a complete stop, followed by another move north, south, east or west by precisely one square (this is *stop-and-go* mode). (ii) Since the car starts the game by moving horizontally or vertically by a single square, a car in stop-and-go mode can explore every portion of its track, including its borders, without leaving the game grid.

Theorem 2. *RACETRACK reachability, where the track boundary can be used for driving, is equivalent to UGGR under AC^0 reducibility.* \square

Proof. The RACETRACK reachability problem reduces to UGGR by the above observations on the stop-and-go car operation mode. Conversely, consider a UGGR instance G . We first construct an equivalent UGGR instance G' , to be easily transformed into a RACETRACK instance later. Assume line-column coordinates for the vertices in the UGGR instance. For each vertex (i, j) in G we add to G' four vertices $(2i, 2j)$, $(2i, 2j + 1)$, $(2i + 1, 2j)$, $(2i + 1, 2j + 1)$ and the four edges to form a square. Then a horizontal edge $((i, j), (i, j + 1))$ in G gives rise in G' to the two horizontal edges $((2i + 1, 2j + 1), (2i + 1, 2j + 2))$ and $((2i, 2j + 1), (2i, 2j + 2))$. A vertical edge $((i, j), (i + 1, j))$ in G gives rise in G' to the two vertical edges $((2i + 1, 2j), (2i + 2, 2j))$ and $((2i + 1, 2j + 1), (2i + 2, 2j + 1))$. The start and target vertices in G' are set accordingly. Finally, the RACETRACK instance is built from G' by taking *all* vertices (grid points) in G' .

It remains to identify the grid points sitting on the track border and to provide for those points the extra 8 bits of information required to locate the track relative to the track border. To this end we consider for each grid point its Moore neighborhood consisting of all grid points that are immediate vertical and horizontal neighbors. This forms a 3×3 subgraph of G' . By construction, each such 3×3 subgraph contains at least one square-shaped subgraph arising

¹ An alternative encoding of RACETRACK instances based on polygonal chains of vertices representing the racetrack border is discussed in [3] and [9]. Because the polygonal representation uses binary notation, the number of accessible grid points can be exponential in the input length. Thus, the complexity results for the polygonal encoding may vary significantly from the results presented here.

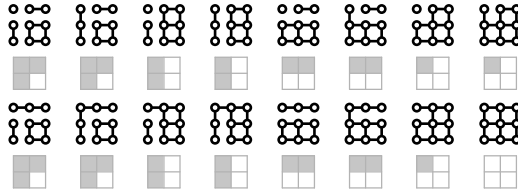


Fig. 2. All possible 3×3 subgraphs, up to mirroring and rotation, that can occur during the construction of G' and their local RACETRACK situations shown underneath

from the four vertices in G' corresponding to each vertex in G . Without loss of generality assume that the square-shaped subgraph resides in the lower right corner of the 3×3 subgraph. Notice that the pair of vertices above this square and the pair of vertices to the left of this square are each connected by an edge since each such pair belongs to a square-shaped subgraph that is adjacent to the 3×3 area. Then we have to consider all further possible edge connections of the nodes that are compatible with the construction of G' . Recall that a vertical (horizontal, respectively) point in G induces *two* vertical (horizontal, respectively) edges in G' . Hence we end up with $2^4 = 16$ possible 3×3 subgraphs that are compatible with the construction of the grid graph G' , since we may introduce two independent single edges and two independent double edges. The upshot is that for each of these subgraphs one can easily determine whether the grid point at the center of the subgraph is on the border or not, and where the non-racetrack part resides. These 3×3 subgraphs and their local RACETRACK situation are shown in Figure 2.

Moreover, the start and finish points in the racetrack are the grid points associated with the start and the target vertex in G' . The two reductions are illustrated in Figure 3. □

The UGGR problem and many related problems were studied at length: UGGR is solvable in deterministic logspace by Blum and Kozen [2], which was known long before Reingold [8] showed that general undirected graph reachability (UGR)

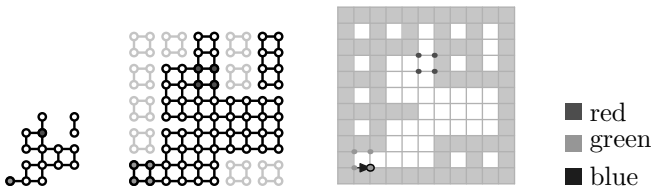


Fig. 3. (Left:) Undirected grid graph with source and target node marked green and red, respectively. Isolated vertices are not shown. (Middle:) Equivalent undirected grid graph where the RACETRACK instance can be easily read off. The square 1×1 subgraphs induced by construction from isolated vertices are drawn in light black. (Right:) RACETRACK instance with start and finish line (point).

is in L . In contrast to UGR, which is L -complete [8], UGGR is only known to be NC^1 -hard [1] and thus seems to be of lower complexity, because even general grid-graph reachability (GGR) is not known to be hard for L under AC^0 reductions.

Corollary 3. *RACETRACK reachability, where the track boundary can be used for driving, is NC^1 -hard under AC^0 reductions and belongs to L . \square*

So why would disallowing the track borders (that is, causing a fatal crash and ending the game when the car hits the track border) make any difference on the complexity of RACETRACK reachability? The surprising answer is that stop-and-go mode then no longer suffices to handle every situation: the car may now need to balance its horizontal and vertical speeds in order to squeeze its way through narrow track portions. This is illustrated by Figure 4.

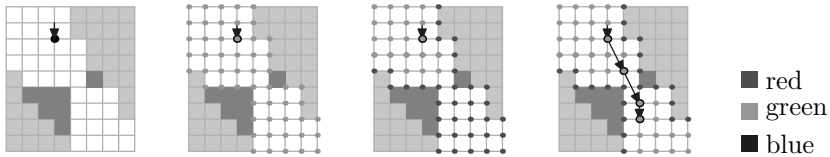


Fig. 4. (Left:) RACETRACK game situation. (Middle-left:) Reachable positions (green) when restricted to stop-and-go-mode, but with drivable track borders. (Middle-right:) Reachable (green) and unreachable positions (red) when restricted to stop-and-go-mode and forbidding driving on the track borders. (Right:) Extending the reachable positions (green) by using speed to traverse the narrow corridor diagonally.

In contrast to Corollary 3, a consequence of the following theorem is that under the usual conjecture that $L \neq NL$, the non-touching variant of RACETRACK reachability is provably harder than its touching variant:

Theorem 4. *RACETRACK reachability and solitaire RACETRACK, where the track boundary cannot be used for driving, are NL -complete.*

Proof. We only give the proof for RACETRACK reachability; handling the solitaire RACETRACK time bound adds no significant complication.

To prove that non-touching RACETRACK reachability is in NL , we reduce it to GR. From a game instance, we build a directed graph whose nodes represent all valid grid-position-speed-vector pairs and whose edges represent legal moves. Thus, a vertex $((i, j), \mathbf{v})$ is connected to $((i, j) + \mathbf{v} + \Delta, \mathbf{v} + \Delta)$, where $\Delta \in \{-1, 0, 1\}^2$, if and only if the line segment starting at (i, j) and ending at $(i, j) + \mathbf{v} + \Delta$ is entirely on the racetrack excluding the boundary points. We further add a source vertex s and an edge to the initial grid-position-speed-vector pair, a target vertex t , and edges for every grid-point-speed-vector for every grid-point on the finish line and any speed vector to vertex t . Then the car starting at the initial position can drive to the finish line not touching the boundary points if and only

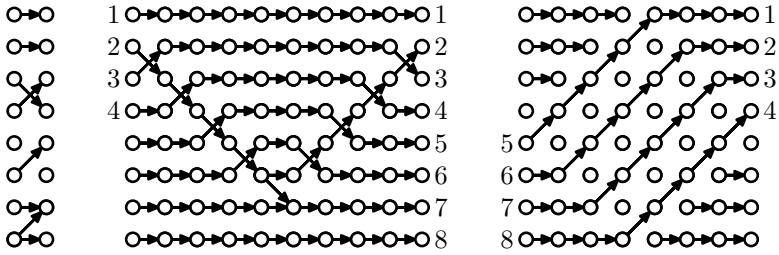


Fig. 5. (Left:) From top to bottom: straight edges, X-crossing edges, up-route edges (with dead ends), and split-join edges. (Middle:) Subgraph induced by an edge (2, 3) of a directed graph G on four vertices. (Right:) Termination subgraph (slightly optimized in length to fit the line) after all edges of G were considered.

if there is a path from s to t in the constructed graph. The construction can be done in logspace since the grid positions and the speed vectors are polynomially bounded in the size of the RACETRACK instance.

To prove that non-touching RACETRACK reachability is NL-hard, we reduce GR to it. Let (G, s, t) be an instance of the GR problem, where $G = (V, E)$ is a directed graph with vertices V and edges $E \subseteq V \times V$, and s and t are the source and the target vertices, respectively. Without loss of generality we may assume that $V = \{1, 2, \dots, n\}$, that $s = 1$ and $t = n$, and that node n has a self-loop, i.e., edge (n, n) is in E . We will proceed in two stages. In stage I, we reduce testing reachability in G to testing reachability in a layered directed graph (LGR) constructed from four types of edge gadgets, namely *straight* edges, *X-crossing* edges, *up-route* edges, and *split-join* edges. See Figure 5 for a drawing of these four gadget types. The semantics of the X-crossing gadget on Figure 5 is that crossing edges do not touch. In stage II, we will later implement the effect of these connection gadgets in a non-touching RACETRACK game.

STAGE I. The layered graph constructed will consist of a $2n \times O(n^4)$ rectangular grid of nodes whose lines (“rows”) are numbered $1, 2, \dots, 2n$ from top to bottom. This grid is divided up into n identical blocks of size $2n \times m$, where $m \in O(n^3)$. The construction will maintain the property that a path of length k with $k > 0$ exists from node i to node j in G if and only if a path of length at most k exists from node $(i, 1)$ to node $(j, m \cdot k)$ in the rectangular grid.

A block is itself the concatenation from left to right of $O(n^2)$ edge layers, followed by a single termination layer. The edge layers are obtained from left to right by considering every edge in the graph G (in any order). The layer corresponding to edge (i, j) in G is constructed by first using a sequence of X-crossing gadgets to “bend line i downwards” across the lines below it. As the (bent) line i crosses the line $n + j - 1$, a split-join gadget is inserted to create a path from line i to line $n + j$. Using further X-crossing gadgets, line i is then bent back upwards and made to return to its original vertical position. The final (termination) layer in the block uses up-route gadgets to safely create paths from line $n + \ell$ to line ℓ , for

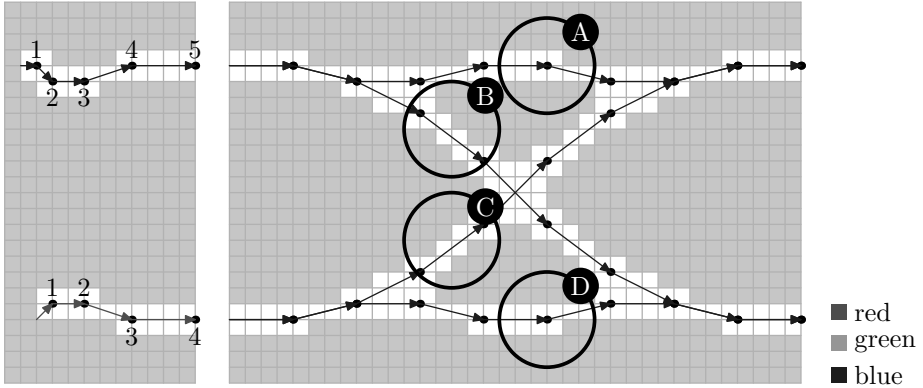


Fig. 6. (Left:) Acceleration tunnels for the 2-player RACETRACK game later used in the proof of Theorem 7—the acceleration tunnel for the 1-player RACETRACK is the top tunnel. (Right:) Blueprint of a 1-player RACETRACK subgame for implementing the four types of gadgets by blocking the racetrack tunnel passages at A, B, C, and/or D, respectively, with obstacles—see the description in the text.

$1 \leq \ell \leq n$. We illustrate the constructions of an edge layer and of a termination layer in Figure 5.

The upshot of concatenating n identical blocks is that node n is reachable from node 1 in G if and only if the rightmost node on line n is reachable from the leftmost node on line 1 in the layered graph. Clearly this construction can be done in logspace.

STAGE II. Here we must simulate the edge gadgets depicted in Figure 5 with RACETRACK (sub)games. The idea is to build a tunnel, of small width to prevent U-turns, which can only be traversed if the car speed (after an appropriate acceleration phase) is within a certain interval. A blueprint of a RACETRACK subgame which can be used to realize all edge gadgets is depicted in Figure 6; the verification that there is no other way through the tunnels is left to the reader. For the actual realization of each edge gadget one has to block some tunnel passages by obstacles appropriately as follows:

- Straight edges: Block the tunnel passage at both B and C.
- X-crossing edges: Block the tunnel passage at both A and D.
- Up-route edges: Block the tunnel passages at A, B, and D.
- Split-join edges: Block only the tunnel passage at C.

Thus one builds in logspace a RACETRACK game for the above constructed graph—the only missing part is an acceleration tunnel that is connected to the game portion induced by the source node of the graph. The construction of an acceleration tunnel is drawn in Figure 6. □

The idea in the proof of Theorem 4 can be made to work for the touching variant of solitaire RACETRACK, proving NL-completeness for that variant also. Indeed

it suffices to slightly modify the gadgets in order to enforce maintaining the driving speed (mostly by shrinking the tunnels as much as possible); deviating from the prescribed speed results in the car not reaching the finish line in time. We must leave the details to the reader. Complementing Theorem 4, we thus have the following:

Theorem 5. *Solitaire RACETRACK, where the track boundary can be used for driving, is NL-complete.* □

4 Complexity of the Usual RACETRACK Game

In this section we consider the 2-player game. We first show the following (the proof uses the AL characterization of P and is omitted):

Theorem 6. *Deciding if the first RACETRACK player has a winning strategy can be done in polynomial time, regardless of whether driving on the track boundary is allowed or not.* □

Next we prove that the non-touching variant is in fact P-complete, thus capturing the precise complexity of the game.

Theorem 7. *Deciding if the first player has a winning strategy in the 2-player RACETRACK game, when the track boundary cannot be used for driving, is P-complete.*

Proof. By Theorem 6, it suffices here to show P-hardness. We will reduce the GEN problem to RACETRACK by adapting the reduction from GEN to GAME (two-player game) mentioned by Greenlaw *et al.* in [5], page 208, A.11.1]. In GAME, Blue attempts to prove that an element t is generated by S . Blue does this by exhibiting two elements r and s , also claimed to be generated by S , such that $t = r \circ s$, while Red attempts to exhibit an element of the pair that is not generated by S . The behaviours of Blue and Red will be simulated by (drum roll!) a RACETRACK game.

Let $T = \{1, 2, \dots, n\}$, $S \subseteq T$, and $g \in T$ be the GEN goal. Notice that g is generated by S if and only if g would appear in a set X initialized to S after at most n iterations of the informal operation $X \leftarrow X \cup (X \circ X)$. The racetrack will be a $(n + n^2 + 2) \times O(n^5)$ rectangular grid of nodes whose lines (“rows”) are named $w, 1, 2, \dots, n, (1, 1), (1, 2), \dots, (n, n), c$ from top to bottom, where w is mnemonic for the “winning” (blue) line and c is mnemonic for the “continuing” (red) line.

At the meta-level, imagine the virtual edges $(t, (r, s)), ((r, s), r)$ and $((r, s), s)$, for $r, s, t \in T$ such that $t = r \circ s$. Initially, Blue’s goal is to prove that $t = g$ is generated (suppose that $g \notin S$). The racetrack will consist of n identical blocks. Each block will implement the following. Blue will be forced to traverse one of the $O(n^2)$ virtual $(t, (r, s))$ edges. This will pin Blue to a choice of a pair (r, s) such that $t = r \circ s$. Red will come along and be forced to traverse either the virtual $((r, s), r)$ edge or the virtual $((r, s), s)$ edge. This will pin Red to a choice

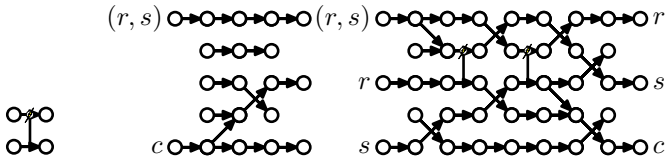


Fig. 7. (Left:) Killer edge gadget or killer edges, for short—Blue is driving at the top and Red at the bottom. (Middle and Right:) Parting gadget with selector gadget on the left. Again Blue drives on the top and Red on the bottom.

of a challenge $\sigma \in \{r, s\}$. At this point, a gadget will force Blue to meet the challenge, that is, to proceed on to the next block with $t = \sigma$ ²

We now give the details. The behaviour of Blue choosing a virtual $(t, (r, s))$ edge can already be simulated with the edge gadgets depicted in the middle of Figure 5, now viewing these gadgets as operating on virtual edges $(t, (r, s))$. To implement Red’s behaviour, assume temporarily that we have at our disposal a RACETRACK subgame implementing the following *killer* gadget: the gadget has two entry points and two exit points, and whenever both cars enter, only Red makes it through while Blue experiences the inconvenience of a fatal accident; when a *single* car of either colour enters, that car makes it through safely. With the help of such a killer gadget, we can construct a controlled *parting* gadget with three input tunnels named (r, s) , r , and s and with three output tunnels named r , s , and c satisfying the following property:

- If Blue enters at (r, s) and Red enters at r (s , respectively), then Blue can only exit at r (s , respectively) and Red exits at c .

See Figure 7 for the killer gadget icon and for the parting gadget. Assuming proper operation of the killer gadget, the correct operation of the parting gadget should be clear.

But the parting gadget is exactly the device required to implement Red’s challenge of an element of the pair (r, s) , in response to Blue’s choice of $(t, (r, s))$, followed by the handing over of that challenge to Blue at the next move! In other words, the parting gadget implements Red’s meta-level choice of a virtual edge $((r, s), r)$ or $((r, s), s)$: Blue can only proceed towards the line $(r$ or $s)$ selected by Red.

Constructing a block in the reduction from GEN to RACETRACK therefore consists, first, in aligning the devices for the virtual edges $(t, (r, s))$ in sequence, describing possible driveways for the blue car, and additionally installing a parallel street c for the red car. Next, for all pairs (r, s) , the devices for the virtual edges $((r, s), r)$ and $((r, s), s)$ are aligned in sequence and the outputs r and s are redirected to their respective initial lines. Finally, for all elements $t \in S$, a branch-off to the line w is installed.

² Technically, the gadget will connect Blue to the σ line and will connect Red to the c line; this will be followed by an exit gadget connecting every ℓ line with $\ell \in S$ to the w line, thus allowing Blue to speed off to a win when $\sigma \in S$.

Once the n identical blocks are assembled, final details involve taking care of the trivial case in which g happens to belong to S from the start (this requires inserting an exit gadget before the first block), connecting appropriate acceleration tunnels for both players and joining the c line to the finish line (so that Red can make it to the finish line one step ahead of Blue if Blue after n steps has not proven his point that the initial GEN goal is generated).

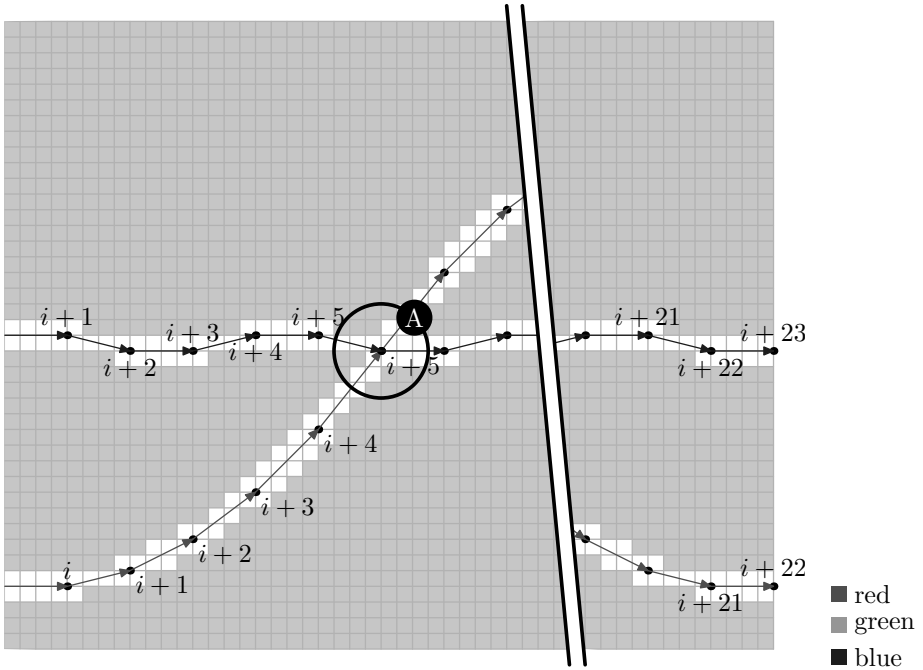


Fig. 8. RACETRACK subgame for the killer gadget—the blue car (player 1) is blocked by the red car (player 2) in the $(i + 5)$ th turn at A. In the middle part of the gadget, which is not shown, the tunnel for the blue car continues snake like and the tunnel for the red car bends in an S-like fashion first slightly up, then down intersecting with the snake like tunnel of blue, and finally towards the exit at the bottom of the subgame.

It remains to describe the implementation of the killer gadgets. We make use of tunnels of small width and appeal to the RACETRACK rule that no two cars may simultaneously occupy the same grid point. Crossing tunnels are of course required. Note that we have used crossing tunnels before, namely when implementing gadgets requiring the RACETRACK blueprint shown on Figure 6. Collisions were not an issue when the blueprint was used in the single-player context, but here we need the same blueprint to implement the crossings implicit in Blue’s behavior, as explained above: Figure 6 shows that for such a purpose, collisions can be avoided by construction. A killer gadget, on the other hand, requires a crossing tunnel that allows collisions, as shown in Figure 8. To obtain

the desired killer gadget effect, the red car must enter the gadget one step ahead of the blue car: this property can be maintained throughout the racetrack by fine-tuning the acceleration tunnels at the beginning of the game—see the drawing in Figure 6. This completes the description of the (logspace-computable) reduction from GEN to non-touching RACETRACK. \square

5 Conclusion

We investigated the complexity of RACETRACK and obtained precise complexity characterizations in terms of completeness results for solitaire RACETRACK and for (the non-touching variant of) 2-player RACETRACK. As to (1-player) RACETRACK reachability, it turned out that having or not having access to the boundary of the racetrack for driving affects the complexity of the problem. More generally, we observed that 2-player RACETRACK is a polynomial time solvable game (hence unlikely to be NP-complete), yet interesting and fun to play.

We have not settled the P-hardness of the touching variant of 2-player RACETRACK. We leave the latter question, as well as the study of the wealth of game variants introduced by the gaming community over the years, as open problems for further research.

Acknowledgements. We thank Martin Beaudry and Martin Kutrib for useful discussions, and the anonymous referees for raising interesting issues, such as the need to distinguish solitaire RACETRACK from RACETRACK reachability.

References

1. Allender, E., Mix Barrington, D.A., Chakraborty, T., Datta, S., Roy, S.: Planar and grid graph reachability problems. *Theory of Computing Systems* 45(4), 675–723 (2009)
2. Blum, M., Kozen, D.: On the power of the compass (or, why mazes are easier to search than graphs). In: *Proceedings of the 19th Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA*, pp. 132–142. IEEE Computer Society, Los Alamitos (1978)
3. Erickson, J.: How hard is optimal racing? (2009), <http://3dpancakes.typepad.com/ernie/2009/06/how-hard-is-optimal-racing.html>
4. Gardner, M.: Mathematical games—Sim, Chomp and Race Track: new games for the intellect (and not for Lady Luck). *Scientific American* 228(1), 108–115 (1973)
5. Greenlaw, R., Hoover, H.J., Ruzzo, W.L.: *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press, Oxford (1995)
6. Jones, N.D., Laaser, W.T.: Complete problems for deterministic polynomial time. *Theoretical Computer Science* 3, 105–117 (1977)
7. Papadimitriou, C.H.: *Computational Complexity*. Addison-Wesley, Reading (1994)
8. Reingold, O.: Undirected connectivity in log-space. *Journal of the ACM* Article 17, 55(4), 24 (2008)
9. Schmid, J.: VectorRace (2005), <http://schmid.dk/articles/vectorRace.pdf>

Simple Wriggling Is Hard Unless You Are a Fat Hippo

Irina Kostitsyna¹ and Valentin Polishchuk²

¹ Computer Science department, Stony Brook University, Stony Brook,
NY 11794, USA

ikost@cs.sunysb.edu

² Helsinki Institute for Information Technology, Computer Science department,
University of Helsinki, FI-00014, Finland

polishch@hiit.fi

Abstract. We prove that it is NP-hard to decide whether two points in a polygonal domain with holes can be connected by a wire. This implies that finding any approximation to the shortest path for a long snake amidst polygonal obstacles is NP-hard. On the positive side, we show that snake's problem is "length-tractable": if the snake is "fat", i.e., its length/width ratio is small, the shortest path can be computed in polynomial time.

1 Introduction

The most basic problem in VLSI and printed circuit board design is to connect two given points, s and t , by a shortest "thick" path avoiding a set of polygonal obstacles in the plane. The quarter-of-a-century-old approach to the problem is to inflate the obstacles by half the path width, and search for the shortest s - t path amidst the inflated obstacles [9]. The found path, when inflated, is the shortest thick s - t path.

It went almost unnoticed that the thick path built by the above procedure may self-overlap (Fig. 1): apart from our recent work on thick paths [4], we only found one mention of the possibility of the overlap — Fig. 4 in [13] (in a different context, Bereg and Kirkpatrick [8, Fig. 2] also noted that Minkowski sum of a disk and a path may be not simply-connected). When the path represents a thick *wire* connecting terminals on a VLSI chip or on a circuit board, self-overlap is undesirable as the wire must retain its width throughout. Thus, the objective in the basic wire routing problem should be to find the shortest *non-selfoverlapping* thick path.

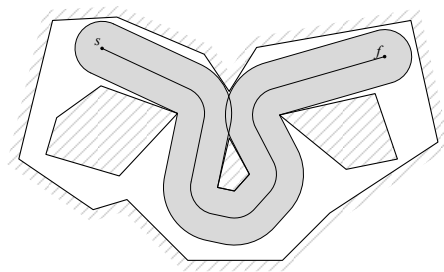


Fig. 1. Only a self-overlapping thick path exists

The problem shows up in other places as well. For instance, one may be interested in the optimal conveyor belt design: the belt is a non-selfoverlapping thick path. Our particular motivation comes from air traffic management where thick paths represent lanes for air traffic. Lane thickness equals to the minimum lateral separation standard, so that aircraft following different lanes stay sufficiently far apart to allow for errors in positioning and navigation. If an airplane self-overlaps, two aircraft following the lane may come too close to each other; thus it is desirable to find lanes without self-overlaps.

1.1 Our Contributions

We prove a surprisingly strong *negative result* (Section 4): it is NP-hard even to *decide* whether there exists (possibly, arbitrarily long) s - t wire; this implies that no approximation to the shortest wire can be found in polynomial time (unless $P=NP$).

Short Snakes. Our intractability result means that in general it is NP-hard for a snake to wriggle its way amidst polygonal obstacles (assuming the snake is uncomfortable with squeezing itself). The good news for snakes is that in our hardness proof the sought wire is considerably long; i.e., the hardness of path finding applies only to *long* snakes. Our *positive result* (Section 3) is that for a bounded-length snake, the shortest path can be found in polynomial time (assuming real RAM and the ability to solve constant-size differential equations in constant time) by a Dijkstra-like traversal of the domain.

1.2 Related Work

In VLSI numerous extensions and generalizations of the basic problem were considered. These include routing multiple paths, on several levels, and with different constraints and objectives.

In robotics thick paths were studied as routes for a circular robot. In this context, path self-overlap poses no problem as even a self-overlapping path may be traversed by the robot; that is, in contrast to VLSI, robotics research should not care about finding non-selfoverlapping paths. In [9], Chew gave an efficient algorithm for finding a shortest thick path in a polygonal domain. In a sense, our algorithm for shortest path for a short snake (Section 3) may be viewed as an extension of Chew's.

Our bounded-length snake problem is reminiscent of path planning for a *segment* (rod) [5,14,6,17]. Short snakes are also relevant to more recent applications of motor protein motion [10,18].

2 Snake Anatomy and Physiology

In this section we introduce the notation and formulate our problem.

Let P be an n -vertex polygonal domain with obstacles. For a planar set \mathcal{S} let $\text{bd}\mathcal{S}$ denote the boundary of \mathcal{S} , and for $r > 0$ let $\langle \mathcal{S} \rangle^{(r)}$ denote the Minkowski

sum of \mathcal{S} with the radius- r open disk centered at the origin. Let $P^r = P \setminus \langle \text{bd}P \rangle^{(r)}$ be P offset by r inside. The boundary of P^r consists of straight-line segments and arcs of circles of radius r centered on vertices of P . We call such (maximal) arcs r -slides.

Let π be a path within P^1 ; let $|\pi|$ denote its length. A *thick path* Π is the Minkowski sum $\Pi = \langle \pi \rangle^{(1)}$. The path π is called the *reference path* of Π ; the *length* of Π is $|\pi|$.

A *snake* is a non-selfoverlapping thick path, i.e., a path which is a simply-connected region of the plane. The reference path of the snake is its *spine* (Fig. 2). One of the endpoints of the spine is the snake’s *mouth* m . The snake is a “rope” that “pulls itself by the head”: imagine that there are little legs (or a wheel, for a toy snake) located at m , by means of which the snake moves. The friction between the snake’s body and the ground is high: any point p of the spine will move only when the path from the mouth to p is a “pulled-taut string”, i.e., is a *locally shortest* path. That is, the snake always stays pulled taut against the obstacles (or itself).

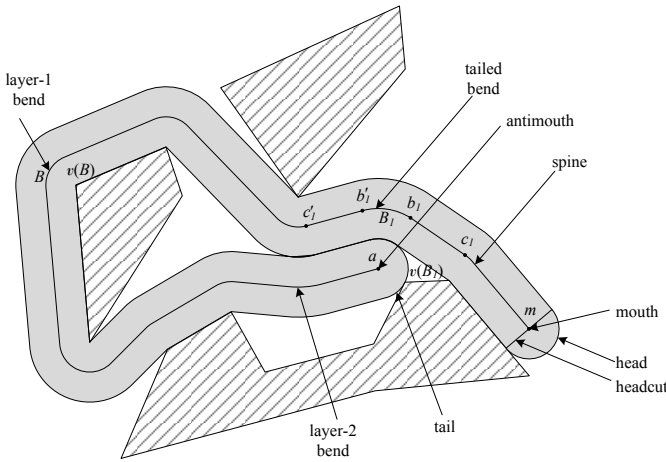


Fig. 2. Spine, mouth, head, headcut, antimouth, tail, bends. A layer-2 tailed bend B_1 does not have a point connected to $v(B_1)$ with a length-2 segment fully lying within the snake.

The input to our problem is the domain P , two points $s, f \in P^1$ – the “start” and the “food”, a number $L > 0$ – the length of the snake, and the initial direction of the snake at s . (Assume w.l.o.g. that s and f are at distance 1 from some vertex of P .) The goal is to find a path for the snake such that the snake’s mouth starts at s and ends at f ; the constraints are that the snake remains pulled taut and non-selfoverlapping throughout the motion. The objective is to minimize the distance traveled by the mouth, or equivalently, assuming constant speed of motion, the time until the snake reaches the food.

Remark. Whoever guesses that the above model of a snake was developed just for FUN, is right. Nevertheless, the proposed problem formulation may be relevant also in more serious circumstances. It models, e.g., the path of a rope being pulled by its frontpoint through a polygonal domain. If it is a fire hose or a tube delivering life-saving medicine [3, 11], minimizing the time to reach a certain point seems like a natural objective (more important than, say, the work spent on pulling the tube). For another application, consider a chain of robots moving amidst obstacles. Each robot, except for the leader of the chain, has very simple program of following its predecessor – just keeping the distance to it. Then the robots form (approximately) a pulled taut thick string.

3 Shortest Path for a Fat Hippo

When the snake is relaxed and its spine is a straight-line segment, the snake is the Minkowski sum of the length- L segment and the unit disk. Such sums are known as *hippodromes* [16, 12, 7, 2, 1], or *hippos* for short. We say that a hippo is *fat* if its length is constant: $L = O(1)$. In this section we show that for a fat hippo our problem can be solved in polynomial time.

Overview of the approach. Our algorithm is a Dijkstra-style search in an implicitly defined graph \mathcal{G} (Fig. 3): neither the nodes nor the edges of the graph are known in advance. Instead, \mathcal{G} is built incrementally, by propagating the labels from the node v with the smallest temporary label (as in standard Dijkstra). The labels are propagated to nodes in the “ L -visibility” region of v , which is what the snake “sees” while it slithers for distance L starting from v . In order to discover the nodes in the region, we pull the snake from v for length L along every possible combinatorial type of path; by a packing argument, there is only a small number of types. The edges of \mathcal{G} correspond to bitangents between the paths and 1-slides. We prove that the algorithm is polynomial-time by observing that the snake must travel for at least $2\pi - 2$ before it “touches” itself with its head; this implies that the snake never “covers” any point of $\text{bd}P$ with more than $O(L/\pi) = O(1)$ “layers”, and hence there is a polynomial number of relevant bitangents. In the remainder of the section we elaborate on the algorithm’s details.

3.1 A Bit More Anatomy

Consider the unit disk $\langle m \rangle^{(1)}$. The part of the boundary of $\langle m \rangle^{(1)}$ that is also the boundary of the snake is the unit semicircle whose diameter is perpendicular to the spine at m ; we call the part the *head* and the diameter the *headcut*. The endpoint of the spine that is not the mouth is called the *antimouth* a . The part of the boundary of $\langle a \rangle^{(1)}$ that is also the boundary of the snake is called the *tail*. Refer to Fig. 2.

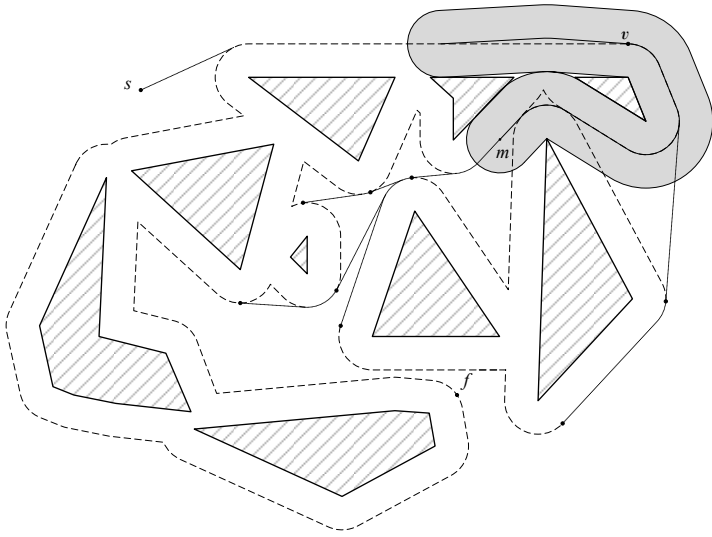


Fig. 3. v is a node of \mathcal{G} ; pull the snake from v for distance L along “every possible” path. Endpoints of visibility edges (some of them are shown with solid circles) become nodes of \mathcal{G} .

3.2 Freeze!

Let us have a closer look at the structure of the pulled-taut snake at any moment of time. Some pieces of the spine are straight-line segments. The segments are bitangents between the other, non-segment pieces supported by vertices of P , possibly via several “layers” of the snake. We call each such (maximal) piece B a *bend*; we denote the vertex that supports B by $v(B)$, and say that v is *responsible* for B .

Snake layers. We show that each vertex v is responsible for $O(1)$ bends. Say that a bend B belongs to *layer 1* if there exists a point $b \in B$ such that $|bv(B)| = 1$. Recursively, B is *layer- $(k + 1)$* bend if it is not layer- k and there exists a point $b \in B$ such that for some point b' at layer k , we have $|b'b| = 2$ and $b'b$ fully lies within the (closure of the) snake (Fig. 2).

Let K be the maximum index of a layer. Because the snake has to wrap around at least one obstacle before touching itself, we have:

Lemma 1. $K \leq \lceil L/(2\pi - 2) \rceil$.

As a corollary, life is very simple for a fat enough hippo: for $\alpha \geq 2\pi - 2$, an α -fat hippo can follow a shortest thick path without self-overlap.

Tailed and headed bends. One may wonder why we did not opt for a simpler definition of layer- k bend B as one having a point b such that $|bv(B)| = 2k - 1$ and $bv(B)$ lies fully within the (closure of the) snake. The reason are two special kinds of bends which make the snake’s portrait more complicated than in the case of an infinite-length snake. Specifically, we say that a bend B at layer k is

tailed (resp. *headed*) if $\langle B \rangle^{(2k-3)}$ touches the tail (resp. head). The simpler definition may not work for such bends (Fig. 2).

Any bend that is not tailed or headed is an arc of a circle (actually, it is part of a slide). A tailed or headed bend B is a different shape – string pulled taut against a ball touching $v(B)$.

Snake configuration. The snake can be reconstructed in linear time as soon as the following is specified: (1) list of the vertices responsible for the bends; (2) for each bend, its layer; (3) for a headed bend – the vertex or the edge of P in contact with the head, and the slope of the headcut; (4) similar information for a tailed bend; (5) positions of the mouth m and antimouth a . We will call (1)–(5) the *configuration* of the snake.

Because each vertex is responsible for at most K bends, we have:

Lemma 2. *The list (1) contains $O(n)$ vertices.*

3.3 Move!

Suppose that we are given the configuration \mathcal{C} of the snake at some time; let π be the spine when the snake is in \mathcal{C} and let $m_{\mathcal{C}}$ be the position of the mouth in \mathcal{C} . Suppose we are also given a path γ for the mouth starting at $m_{\mathcal{C}}$. Assume γ has the following properties: (1) it is consistent with \mathcal{C} in that the tangent to γ at $m_{\mathcal{C}}$ coincides with the tangent to π at $m_{\mathcal{C}}$; (2) it has a polynomial number of pieces, each of constant description complexity; (3) $|\gamma| \leq L$. In what follows we assume that every path γ has these properties. We claim that in polynomial time we can check whether γ is a feasible path for the mouth, i.e., whether the snake stays obstacle-free when m is pulled along γ (say, at unit speed).

First we note that it is enough to check only whether the mouth moves feasibly. Indeed, the first time that the snake (possibly) becomes infeasible as m follows γ , the mouth is necessarily a part of “certificate of infeasibility”. This is so because the only way that the snake experiences “side pressure” is due to appearance of a headed bend. Other than that, any piece \mathcal{P} of the snake is merely pulled by the preceding piece, \mathcal{P}' : either \mathcal{P} exactly follows the same path that just was feasible for \mathcal{P}' , or \mathcal{P} is a tailed bend B . Of course, the bend morphs as the time passes, but only becomes “more feasible” with time, i.e., the free space around B increases – B is pushed only by the tail, and the tail “moves away” with time.

Let now γ' be a piece of γ . If we know how each piece \mathcal{P} of the snake changes with time, we can test whether m stays feasible while following γ' , by checking the feasibility against each \mathcal{P} in turn. That is, while neither the configuration of the snake nor the piece γ' of γ changes, the feasibility test can be done piece-versus-piece in constant time (assuming real RAM). Observe that overall there is only a polynomial number of configuration changes. Indeed, the configuration may change only due to one of the following *events*: (1) the tail starts to follow another feature of P , (2) a headed bend appears or changes its combinatorial structure, (3) a tailed bend disappears or changes its combinatorial structure. But each of the events (1)–(3) may happen only once per vertex-bend-layer triple; thus, by Lemmas 1 and 2 there is only a polynomial number of events.

Tracking the configuration changes is easy *given* the way each piece changes with time. For event (1), we only have to know how the tail speed changes with the time between consecutive events (the tail speed is not necessarily constant, we elaborate on it in the next paragraphs). For event (2) we check what is the first time that the head collides with a piece or when a headed bend hits a vertex; all this can be done in polynomial time as there is only a linear number of candidate collisions. Event (3) is similar. The next event time is then the minimum of the event times over all the pieces.

It remains to show how to determine the way each piece changes. Here the crucial role is played by the headed and tailed bends; again, it is the finiteness of the snake length that makes things involved. Let $B_1 = b_1b'_1$ be the first such bend counting from m (Fig. 2). Assume that B_1 is a layer- k tailed bend; the situation with a headed bend is actually simpler (because the mouth speed is constant). Let c_1b_1 and $b'_1c'_1$ be the pieces adjacent to B_1 – both are bitangents (straight-line segments, possibly of 0 length) to B_1 and adjacent bends. Every point of the spine between m and c_1 moves at speed 1, and none of the bends before B_1 changes with time.

To figure out what happens after c_1 , we have to solve a constant-size differential equation that describes the "propagation" of speed of motion of the spine. Specifically, let $u(\tau)$ denote the speed at which the antimouth moves at time τ . Knowing $u(\tau)$ and knowing the initial position of the antimouth, we can write the antimouth position as a function of time, and hence we know $\langle a \rangle^{(2k-2)}$ as a function of time. The points b_1 and b'_1 are points of tangency to $\langle a \rangle^{(2k-2)}$ from c_1 and c'_1 ; thus knowing $\langle a \rangle^{(2k-2)}$ we know how the length $|c_1b_1| + |\pi(b_1b'_1)| + |b'_1c'_1|$ changes with time. Knowing that, and recalling that at c_1 the spine moves with unit speed, we can write what the spine speed at c'_1 is as a function of τ .

Now, the spine speed does not change between c'_1 and the next point, c_2 , that is the start of the tangent to the next headed or tailed bend, B_2 . We perform at B_2 the same operations as above, and get the speed of motion of the spine past the bend B_2 . Continuing in this fashion, in the end, after going through all bends, we obtain some expression, $\mathcal{E}(u(\tau))$ for the spine speed after the last bend.

Finally, to close the loop, we solve the equation

$$u(\tau) = \mathcal{E}(u(\tau)) \tag{1}$$

Since there is only a constant number of layers (Lemma 1), there is only a constant number of the headed and tailed bends, and hence the expression \mathcal{E} is a sum of a constant number of terms (each containing $u(\tau)$ and $\int u(\tau)d\tau$), each of constant description complexity. In our computation model, we can solve the equation for $u(\tau)$ in constant time. Substituting $u(\tau)$ back into the formulae for the different bends, we obtain the spine as a function of time, as desired.

3.4 See!

Assume that at some point the snake is in configuration \mathcal{C} . What happens next, as the snake follows the optimal path to f ? Local optimality conditions imply that

it will “wiggle around the obstacles” for some time and then “shoot” towards a vertex of P . We formalize this below.

Final piece of anatomy. The *eye* of the snake is collocated with the mouth. The snake can *see* a point $x \in P^1$ if the segment mx lies fully within P^1 and is tangent to the spine at m . Recall that $P^1 = P \setminus \langle \text{bd}P \rangle^{(1)}$ where $\langle \text{bd}P \rangle^{(1)}$ is the Minkowski sum of $\text{bd}P$ with *open* unit disk; hence mx can go along the boundary of P^1 (with x being, e.g., an endpoint of a slide; see Fig. 3). The snake itself is transparent for its eye: we do not forbid mx to intersect the snake.

Definition 1. A point p on $\text{bd}P^1$ is L -visible from m_C if there exists a path γ for the mouth ending in a point m^* such that m^* sees p and m^*p is tangent to $\text{bd}P^1$ at p . We say that m_Cp is an L -visibility edge, or an L -edge for short. We say that γ is the wiggle segment of m_Cp , and that m^*p is the visibility segment of the edge. (We remind that we assume γ enjoys the properties listed in the beginning of Section 3.2: tangent at m_C consistent with \mathcal{C} , polynomial-size description, length $\leq L$.)

To be on a (locally) optimal path, the mouth would like to follow an L -edge also past m^* . This may not be feasible due to a conflict with the snake itself. However, such conflicts can be discovered “on-the-fly”, as the mouth attempts to move along m^*p . Specifically, try to move the mouth along m^*p , as described in Section 3.3. If during the motion, the head collides with the snake, note what kind of bend the head collides with. If this bend B is not tailed, adjust the path for the mouth so that it is tangent to $\langle B \rangle^{(2)}$, and follow the adjusted path. If by the time the mouth reaches $\langle B \rangle^{(2)}$, the bend B is “gone”, i.e., the head “misses” the snake, we know that we are dealing with a tailed bend (or with the tail itself). We identify the time and place of contact with the bend by solving a differential equation similar to (II): assuming the speed of the tail is $u(\tau)$, we know how the bitangent between m and the corresponding ball centered at a changes; in particular, we know its length $l(\tau)$ as a function of time. The time τ^* when the head hits the tail is then the solution to the equation $\tau^* = l(\tau^*)$. After solving the equation we know the configuration of the snake at τ^* , and continue moving the mouth to p around the tail.

The above procedure essentially “develops” the path γ piece-by-piece. This is consistent with Section 3.3 where we described how to pull the snake along a *given* path γ for the mouth: the pulling was done piece-by-piece, which means that it can be performed even if γ is not given in advance but instead is revealed piece after piece. More importantly, using the procedure, we can develop *all* L -edges incident to m_C , piece-by-piece, in a BFS manner. We describe this below.

To initiate the developing, look at the visibility segment $m'm_C$ of the L -edge that has led the mouth to m_C . The segment is tangent to a slide $S \ni m_C$. The slide S down from m_C (i.e., in the direction consistent with $m'm_C$) is the first (potential) piece of a new L -edge. We extend bitangents from the piece to all other 1-slides and to all pieces of the spine inflated by 2. These bitangents become the next potential pieces for the L -edges. After the bitangents, the next potential pieces are the slides and the spine pieces at which the bitangents end. We continue in this way (possibly adjusting the pieces of a particular edge to

account for snake self-interaction) until for each edge its wiggle segment reaches length L .

We now bound the time spent on developing all L -edges from m_C . Each edge has linear complexity; this can be proved identically to Lemma 2. Thus all edges can be grown in polynomial time if the number of edges is polynomial. This is what we prove next:

Lemma 3. *Let \mathbb{E} be the set of all L -edges incident to m_C . $|\mathbb{E}| = O(n^2)$.*

Proof. By definition, every edge starts from a path γ for the mouth of length at most L . Say that paths γ_1, γ_2 are the same *combinatorial type* if the sequence of vertices visited by γ_1 is a subsequence of that for γ_2 (or vice versa). Being of the same type is an equivalence relation that splits \mathbb{E} into classes. For each class, keep only the path with the longest sequence of visited vertices, and identify the class with the path. Let \mathbb{E}^* be the obtained collection of classes.

Let $\gamma^* \in \mathbb{E}^*$. Any L -edge that has a path $\gamma \in \gamma^*$ as its wiggle segment is obtained by extending a bitangent from γ (or equivalently from γ^*) to some slide. Thus the total number of L -edges having a path in γ^* as the wiggle segment is at most the number of bitangents from γ^* to the slides, which is $O(n^2)$ since γ^* is $O(n)$ -complexity.

It remains to prove that $|\mathbb{E}^*| = O(1)$. It is easy to see that $|\mathbb{E}^*|$ depends only on the number H of the holes in P^1 reachable by length- L paths from m_C (not on the number of vertices). By a packing argument, $H = O(1)$.

3.5 Label!

We are ready now to traverse the " L -visibility" graph \mathcal{G} of P , searching only the relevant part of \mathcal{G} and not building the whole graph explicitly. The label of each node in \mathcal{G} consists of two parts: the *distance label* (storing the distance from s) and the *configuration label* (storing the snake configuration at which the node was reached). That is, a node may have several labels – one per configuration. However, since there is only a constant number of different configurations of pulled-taut snakes that may reach the node, the total number of labels of any node is constant.

Start from $m_C = s$, and assign distance label 0 to s . The algorithm grows the graph \mathcal{G} whose edges are the discovered L -edges and whose nodes are the endpoints of the L -edges. Note that by the definition of L -visibility (Definition 1), all nodes of \mathcal{G} reside on slides. At a generic step, take the node with the smallest (temporary) distance label, make the label permanent, and construct L -edges from the node. The endpoints of the edges join \mathcal{G} and get their (temporary) distance labels and configuration labels. In addition, each already existing node over which the mouth passes, gets its distance label updated if the distance label carried with the mouth is smaller than the node's current label *and* the configuration label carried with the mouth is the same as the node's configuration label. The search stops when f is reached.

We now prove that the algorithm terminates in a polynomial number of steps. The visibility segments are bitangents between 1-slides and pieces of the wiggle

segments. These latter pieces are of two types: (1) slides up to layer K and (2) curves that are obtained as the head rolls over the tail, possibly padded by up to K layers of the snake. There is in principle an uncountable number of possible pieces of the second type. Nevertheless, every step of the algorithm discovers at least one new visibility segment tangent to a piece of the first type. Since the number of slides up to layer K is $O(n)$, there is $O(n^2)$ of such visibility segments. Each such segment may be discovered only a constant number of times — once per homotopy type of the snake reaching the endpoint of the segment. Thus overall there is $O(n^2)$ steps.

Theorem 1. *Shortest path for a fat hippo can be computed in polynomial time.*

4 Being a Long Snake Is Hard

In this section we show that if the snake length is not bounded, deciding existence of a path for the snake is NP-hard. Specifically, our problem is: Given polygonal domain P and points s, f , find a thick non-selfoverlapping s - f path.

To show the hardness, start from an instance I of planar 3SAT [15] embedded so that all variables are aligned along a horizontal line. We identify the instance with its (planar) graph, and the variables and clauses with the points in the plane into which they are embedded.

We say that a clause C' is a *child* of a clause C if C' can be connected to C without intersecting any variable-clause edge. Add to I all parent-child edges; if a parent has only one child, connect them by 2 parallel edges (parallel in the graph-theoretic sense, in the embedding they are not parallel). Add also edges between siblings (Fig. 4). A clause that is noone's child is an *orphan*. Assume all orphans are aligned along a horizontal line, and connect consecutive orphans.

Add nodes s and f to I ; add edges from s to x_1 and from the leftmost orphan to f . Add a parallel edge for each clause-variable edge. Add an extra connection

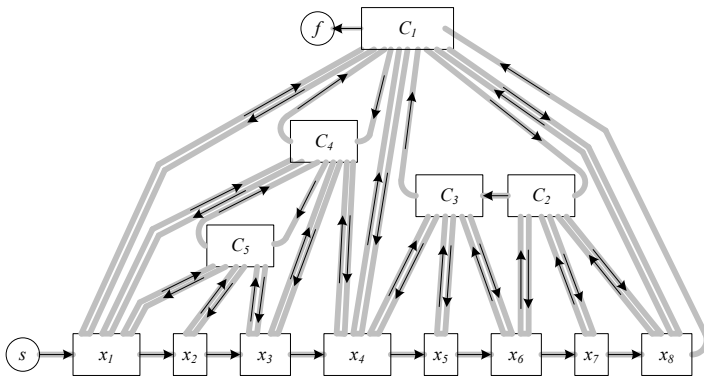


Fig. 4. I is augmented with parent-child edges, sibling edges and an edge between the last variable and the rightmost orphan. The walk is indicated with the arrows.

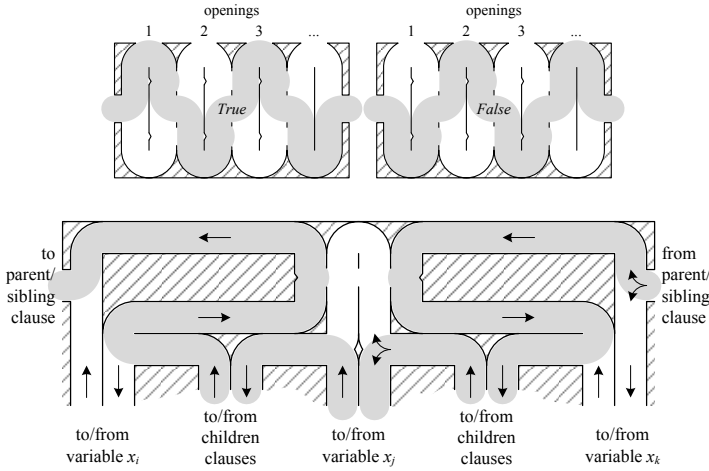


Fig. 5. Top: Traversing a variable gadget sets the truth assignment. Bottom: When a clause gadget is traversed, from right to left, one of the channels leading to variables must be used. Otherwise, 3 subpaths go through the top of the gadget leading to a self-overlap.

from the last variable to the rightmost orphan. Now I has an s - f walk that traverses the variables and then the clauses, recursing to children in the DFS manner, and revisiting variables from their clauses; refer to Fig. 4.

Next, convert I to an instance of finding a thick wire. The variable and clause gadgets are shown in Fig. 5. The connections (channels) between variables and clauses (Fig. 6) ensure that whenever a channel from a clause to a variable is used by the wire, the variable satisfies the clause. The only s - f wire in the instance is one that follows the walk in I ; the wire exists if and only if I is satisfiable.

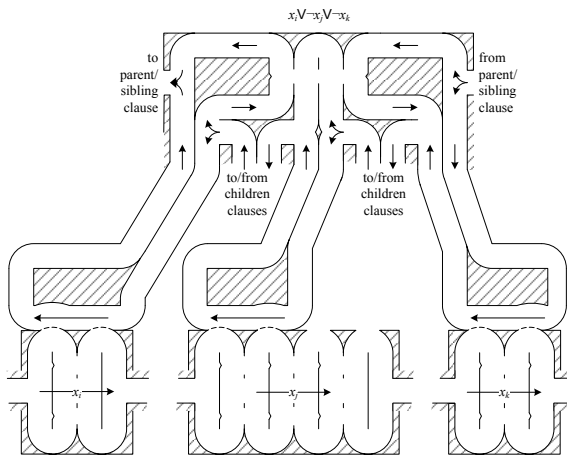


Fig. 6. If a clause-variable channel is used, the variable must satisfy the clause

Acknowledgements. We thank Estie Arkin, David Kirkpatrick, Joe Mitchell and Jukka Suomela for discussions, and the referees for their comments. This work is partially supported by the Academy of Finland grant 118653 (ALGODAN).

References

1. Agarwal, P.K., Efrat, A., Sharir, M.: Vertical decomposition of shallow levels in 3-dimensional arrangements and its applications. *SIJCOMP* 29(3), 912–953 (2000)
2. Agarwal, P.K., Sharir, M.: Efficient algorithms for geometric optimization. *ACM Computing Surveys* 30, 412–458 (1998)
3. Alterovitz, R., Branicky, M.S., Goldberg, K.Y.: Motion planning under uncertainty for image-guided medical needle steering. *Int. J. Rob. Res.* 27(11-12), 1361–1374 (2008)
4. Arkin, E.M., Mitchell, J.S.B., Polishchuk, V.: Maximum thick paths in static and dynamic environments. *Comp. Geom. Theory and Appl.* 43(3), 279–294 (2010)
5. Asano, T., Kirkpatrick, D., Yap, C.K.: d_1 -optimal motion for a rod. In: *SoCG 1996* (1996)
6. Asano, T., Kirkpatrick, D., Yap, C.K.: Minimizing the trace length of a rod endpoint in the presence of polygonal obstacles is NP-hard. In: *CCCG 2003* (2003)
7. Barcia, J., Diaz-Banez, J., Gomez, F., Ventura, I.: The anchored Voronoi diagram: static and dynamic versions and applications. In: *EuroCG 2003* (2003)
8. Bereg, S., Kirkpatrick, D.: Curvature-bounded traversals of narrow corridors. In: *SoCG 2005* (2005)
9. Chew, L.P.: Planning the shortest path for a disc in $O(n^2 \log n)$ time. In: *SoCG 1985* (1985)
10. Chowdhury, D.: Molecular motors: Design, mechanism, and control. *Computing in Science and Engineering* 10, 70–77 (2008)
11. Cook IV, A.F., Wenk, C., Daescu, O., Bitner, S., Cheung, Y.K., Kurdia, A.: Visiting a sequence of points with a bevel-tip needle. In: *LATIN 2010* (2010)
12. Efrat, A., Sharir, M.: A near-linear algorithm for the planar segment center problem. *Discrete & Computational Geometry* 16, 239–257 (1996)
13. Hu, T., Kahng, A., Robins, G.: Optimal robust path planning in general environments. *IEEE Transactions on Robotics and Automation* 9, 775–784 (1993)
14. Lee, J.Y., Choset, H.: Sensor-based planning for a rod-shaped robot in three dimensions: Piecewise retracts of $R^3 \times S^2$. *Int. J. Rob. Res.* 24(5), 343–383 (2005)
15. Lichtenstein, D.: Planar formulae and their uses. *SIJCOMP* 11(2), 329–343 (1982)
16. Pach, J., Tardos, G.: Forbidden patterns and unit distances. In: *SoCG 2005* (2005)
17. Sifrony, S., Sharir, M.: A new efficient motion-planning algorithm for a rod in two-dimensional polygonal space. *Algorithmica* 2, 367–402 (1987)
18. Veigel, C., Coluccio, L.M., Jontes, J.D., Sparrow, J.C., Milligan, R.A., Molloy, J.: The motor protein myosin-I produces its working stroke in two steps. *Nature* 398, 530–533 (1999)

The Urinal Problem

Evangelos Kranakis¹ and Danny Krizanc²

¹ School of Computer Science, Carleton University
K1S 5B6, Ottawa, Ontario, Canada

² Department of Mathematics and Computer Science
Wesleyan University, Middletown CT 06459, USA

Abstract. A man walks into a men’s room and observes n empty urinals. Which urinal should he pick so as to maximize his chances of maintaining privacy, i.e., minimize the chance that someone will occupy a urinal beside him? In this paper, we attempt to answer this question under a variety of models for standard men’s room behavior. Our results suggest that for the most part one should probably choose the urinal furthest from the door (with some interesting exceptions). We also suggest a number of variations on the problem that lead to many open problems.

1 Introduction

The question of digital privacy and how to protect it has a long history of study in computer science and it raises a number of interesting algorithmic (as well as other) research problems [7]. But it is also the case that algorithmic issues may arise when considering physical privacy. A particular instance of this occurs when one considers the use of a public men’s room. The standard design of a men’s room contains a number of urinals which are generally located along one wall with neighboring stations in full view of each other (Figure 1). (Although recently there has been a trend to place dividers between adjoining positions [1]. For a history of privacy concerns in restroom design see [15].) In order to obtain some amount of privacy while vacating one’s bladder it is desirable to have a urinal such that its neighboring positions are unoccupied [4].

This leads to the following algorithmic question: A man walks into a men’s room and observes n empty urinals. Which urinal should he pick so as to maximize his chances of maintaining privacy, i.e., minimize the chance that someone will occupy a urinal beside him? One’s intuition might suggest that choosing one of the end urinals is the best choice. It turns out the answer depends on many

¹ We hope that by focusing on the men’s room version of the problem we are not leaving out our female readers. We chose this version since (1) we understand (not from personal experience) that most women’s rooms contain only water closets which already provide some amount of privacy beyond that afforded by urinals and (2) we are more familiar with the behavior of men in a public restroom situation which we shall see is an important aspect of our study. It has been pointed out to us that women may also use urinals [2] and should this become a more prevalent behavior our results may directly interest our female readers.



Fig. 1. Standard men’s room urinal arrangement

things but in particular on how one models the behavior of the men who enter later. In this paper, we consider a variety of models for men’s room behavior and attempt to develop strategies for maintaining privacy under each. The results of this preliminary study suggest that for the most part one’s intuition is correct (the ends are best) with some interesting exceptions. We also examine a number of variations on the problem that might lead to a better understanding of physical privacy concerns and which suggest many interesting open questions.

2 Related Work

As far as we know the problem we consider has not been studied before. If one is interested in finding the most private position to place an object in the sense of it being furthest from all other occupied positions in some space this can be formulated as a version of the well-studied *Obnoxious Facility Location Problem*. (See [13] for a survey of results in this area.) But in our case this question is trivially solved by taking the position at the mid-point of the largest gap of urinals (i.e., contiguous sequence of empty urinals) and it does not really answer our question. The question of privacy in querying sensors in a sensor network also shares some aspects of our question but again the concerns turn out to be different [16].

Most closely related to our question (at least for the case where the men behave randomly) seems to be the following “unfriendly seating arrangement” problem posed by Freedman and Shepp [4]: There are n seats in a row at a

luncheonette and people sit down one at a time at random. They are unfriendly and so never sit next to one another (no moving over). What is the expected number of persons to sit down? Solutions to this problem were provided by Friedman, Rothman and MacKenzie [5,12] who show that as n tends to infinity the expected fraction of the seats that are occupied goes to $\frac{1}{2} - \frac{1}{2e^2}$. (For a nice exposition of this and related problems see [3].) Georgiou et al. [6] consider the following generalization of this problem (the unfriendly theatre seating arrangement problem): People arrive one at a time to a theatre consisting of m rows of length n . Being unfriendly they choose seats at random so that no one is in front of them, behind them or to either side. What is the expected number of people in the theatre when it becomes full, i.e., it can not accommodate any more unfriendly people? They give bounds on the fraction of the theatre that is full (in the limit) for all m and show that for $m = 2$ this limit is $\frac{1}{2} - \frac{1}{4e}$.

While perhaps related to our problem in name only, our study was at least partially inspired by Knuth’s Toilet Paper problem [11]. Don Knuth relates that the toilet paper dispensers in the Stanford Computer Science public restrooms are designed to hold two rolls of tissues either of which is available for use. This led him to consider the following problem: A toilet stall contains two rolls of toilet paper of n sheets each. The stall is used by people of two types: *big choosers* and *little choosers*. They arrive to use the toilet randomly and independently, the former with probability p and the latter with probability $1 - p$. Big (respectively, little) choosers select exactly one sheet of paper from the roll with the most (respectively, least) number of sheets. What is the expected number of toilet sheets remaining just after one of the two rolls has emptied, defined to be the residue $R_n(p)$?

Knuth used combinatorial techniques to prove that for fixed p and r , which satisfy the condition $4p(1 - p) < r < 1$, we have that

$$E[R_n(p)] = \begin{cases} \frac{p}{2p-1} + O(r^n) & \text{if } p > 1/2 \\ 2\sqrt{\frac{p}{\pi}} - \frac{1}{4} \frac{1}{\sqrt{n\pi}} + O(n^{-3/2}) & \text{if } p = 1/2 \\ \frac{1-2p}{1-p}n + \frac{p}{1-2p} + O(r^n) & \text{if } p < 1/2 \end{cases}$$

as $n \rightarrow \infty$ where the constants implied by the O notation depend on p, r but not on n . A high probability version of this result was recently used to prove bounds on the time required for routing on a Markovian grid [9].

3 The Urinal Problem

For our initial study we assume that you are the first man to enter and all of the urinals are available. We will discuss the case where some urinals are occupied below. Further we assume that men enter one at a time and depending on one of the strategies outlined below they make a choice of a urinal that provides privacy (i.e., is unoccupied on either side) if possible. If this is not possible

then we assume the *random* strategy for choosing a urinal where one chooses uniformly at random among all unoccupied urinals. We note that other models of how the remaining urinals are filled are possible and make some remarks on some of them below. Finally, we assume that men enter at a constant rate of one man every time unit and stay at their position until you have finished.

For each of the scenarios we consider, we are interested in maximizing the expected time until your privacy will be violated, i.e., a man occupies one of the urinals beside yours. We call a set of the occupied urinals a *configuration*. We call a configuration *saturated* if the next man entering is forced to violate at least one person's privacy. It is fairly clear that, under the assumptions above, the time until your privacy is violated can be divided into two phases: the time until saturation is reached and time until a man violates your privacy once the configuration is saturated. As such we divide our analysis for each of the cases below into two parts: the time until saturation and the time until privacy violation once saturation has been achieved.

3.1 Lazy Behavior

In this model one chooses the lowest number unoccupied urinal that provides privacy, i.e., we assume that a urinal's distance from the door of the men's room is directly proportional to its number and that men, being naturally lazy, will always choose the first empty private spot. The analysis of the first phase divides into two cases depending upon whether n is even or odd.

If n is even and you choose an odd numbered position then clearly the saturated configuration will consist of all of the odd positions and will contain $n/2$ occupied urinals. If you choose an even numbered position, $2k$, then the saturated configuration consists of the $k - 1$ odd positions before you (excluding $2k - 1$) and the $n/2 - k$ even positions after you. Again there are $n/2$ occupied urinals at saturation. In the odd case, you are better off choosing an odd position as in that case the saturated configuration consists of all of the odd positions and contains $(n + 1)/2$ men versus $(n - 1)/2$ men in case you choose an even position.

In either case, choosing any odd position yields $\lceil n/2 \rceil$ men in the saturated configuration. At this point, if the remaining positions are filled randomly, there is a distinct advantage to picking one of the positions at the end. The number of men that enter before picking a urinal beside you follows a negative hypergeometric distribution with parameters $N = n - \lceil n/2 \rceil$, a where a is the number of positions available beside you, i.e., $a = 1$ if you occupy position 1 or n and $a = 2$ otherwise. The expectation of such a random variable is given by $(N + 1)/(a + 1)$ so we get the expected time until your privacy is violated is

$$\lceil n/2 \rceil + \frac{\lfloor n/2 \rfloor}{2}$$

if you choose positions 1 or n and

$$\lceil n/2 \rceil + \frac{\lfloor n/2 \rfloor}{3}$$

otherwise. Clearly, choosing an end position is to your advantage for all n . (We discuss the case where urinals are filled in a lazy fashion after saturation is reached below.)

3.2 Cooperative Behavior

In this model, all men cooperate to ensure that the configuration becomes saturated at the last possible moment, by each choosing a position that guarantees the maximum number of men have full privacy for as long as possible. It is easy to establish that this case is very similar to the above in that as long as you choose an odd position the saturated configuration will contain $\lceil n/2 \rceil$ men and the best overall choice turns out to be either end.

3.3 Maximize Your Distance Behavior

In this model we assume that a new entrant to the washroom will choose the urinal which maximizes his distance to any of the current occupants of urinals. If more than one such urinal exists then a random one among them is chosen. If no urinal with privacy exists then a random urinal is chosen.

First we observe that by symmetry the number and positions of the men when saturation is first reached are independent of the random choices made among equidistant positions and only depends on the choice that you make as the first man. Let $A(n, i)$ be number of men in the saturated state if the first man chooses position $i, i = 1, \dots, n$. Let $B(n)$ be the number of men in such a saturated state assuming positions 1 and n are filled (not including the men at urinals 1 and n).

Assume $n > 3$. (The cases $1 \leq n \leq 3$ are straightforward to analyze.) Observe that if the first man chooses position 1 (respectively, n) then the second one will choose n (respectively, 1) and the saturated configuration will contain $2 + B(n)$ men. If the first man chooses 2 (respectively, $n - 1$) the second will choose n (respectively, 1). If the first man chooses some position i between 3 and $n - 2$ then the second man will choose the further of 1 and n and eventually someone will choose the other. This yields the following equation for $A(n, i)$:

$$A(n, i) = \begin{cases} 2 + B(n) & i = 1, n \\ 2 + B(n - 1) & i = 2, n - 1 \\ 3 + B(i) + B(n - i + 1) & 3 \leq i \leq n - 2. \end{cases}$$

Further observe that in any gap of $k > 2$ unoccupied urinals with the urinals at either end occupied, the first of the unoccupied positions to be occupied will be the middle one (or one of the two middle ones if k is even). This yields:

$$B(n) = \begin{cases} 0 & n \leq 4 \\ 1 + B(\lfloor \frac{i+1}{2} \rfloor) + B(\lceil \frac{i+1}{2} \rceil) & n > 4. \end{cases}$$

It is relatively straightforward to establish the following closed form solution for $B(n)$:

$$B(n) = 2^{\lfloor \log(n-1) \rfloor - 1} + (\lfloor \frac{n-1}{2^{\lfloor \log(n-1) \rfloor - 1}} \rfloor - 2)(n - 3 \cdot 2^{\lfloor \log(n-1) \rfloor - 1} - 1) - 1$$

which in turn allows us to compute $A(n, i)$ for any n, i . Unfortunately, we were not able to establish a closed form expression for $\max_i A(n, i)$ or show that it can be computed any faster than evaluating $A(n, i)$ for all i . Note that this is a pseudo-polynomial time algorithm as the input is just n . (See the discussion below concerning different initial configurations for more on the analysis of this algorithm.) While the function is reasonably well-behaved it does vary widely for some values of n between the extremes of $\lceil n/3 \rceil$ and $\lceil n/2 \rceil$.

If one was just concerned about maximizing the time until saturation occurs it turns out that end positions are not optimal. In fact, for some values of n position 1 is the worst choice in this regard. For example, for n of the form $3 \cdot 2^k + 1$ for some $k > 0$, $A(n, 1) = \lceil n/3 \rceil$ while $A(n, \lceil n/3 \rceil) = \lceil n/2 \rceil$.

Fortunately, one can show that once again the end positions have an advantage when we assume that the remaining positions are filled in random order. It is fairly easy to show that $A(n, i) \geq \lceil n/3 \rceil$ for all n (at least one of every three urinals must be occupied) and that $A(n, i) \leq \lceil n/2 \rceil$ (at most one of every pair may be occupied) for all n and i and since

$$\lceil n/3 \rceil + \frac{n - \lceil n/3 \rceil}{2} > \lceil n/2 \rceil + \frac{\lfloor n/2 \rfloor}{3}$$

we see that urinal number 1 (and n) is always (by a very slight margin in some cases, e.g., for n of the form $3 \cdot 2^k + 1$ for some $k > 0$ the difference is $1/6$) the optimal choice.

3.4 Random Behavior

In this model we assume that when a man enters if the configuration is not saturated he chooses uniformly at random among all positions that provide him with privacy on both sides and that once saturation is reached he chooses uniformly among all available positions. If the first man chooses randomly like everyone else then the question of how many men there are at the time of saturation corresponds to the unfriendly seating arrangement problem discussed above. In our case we are interested in computing whether it makes a difference which position you choose as the first man. And indeed it does. Consider the case $n = 5$. If position 1 is chosen it is easy to calculate that the expected number of men at saturation is 2.67 but if you choose position 3 you are guaranteed 3 men at saturation so it would appear that position 3 is better. But as above it is during the filling phase that the advantage of an end takes over. In this example, position 1 expects 4.33 men (including himself) before his privacy is violated versus the situation for position 3 where the fourth man will certainly violate his privacy.

Let $E(n, i)$ be the expected number of men entering before your privacy is violated assuming you choose position i . Let $p_{i,k}$ be the probability that if you choose position i the resulting saturated configuration has k men in it. Let $q_{j,k,\ell}$ be the probability that, if there are k spots available after saturation and $\ell = 1, 2$

depending on whether you are at an end or not (indicating the number of positions you have available beside you), your privacy is violated by the j th man ($j = 1, \dots, n - k$) entering after saturation. As above we note that a saturated configuration has between $\lceil n/3 \rceil$ and $\lceil n/2 \rceil$ positions taken and so we can calculate:

$$\begin{aligned}
 E(n, i) &= \sum_{s=\lceil n/3 \rceil+1}^n s \sum_{j+k=s, j>0, \lceil n/3 \rceil \leq k \leq \lceil n/2 \rceil} q_{j,n-k,\ell} p_{i,k} \\
 &= \sum_{k=\lceil n/3 \rceil}^{\lceil n/2 \rceil} (k + \sum_{j=1}^{n-k} j \cdot q_{j,n-k,\ell}) p_{i,k}.
 \end{aligned}$$

Using the fact that that $q_{j,k,\ell}$ is negative hypergeometrically distributed we get:

$$E(n, i) = \begin{cases} \frac{n+1}{2} + (1/2) \sum_{k=\lceil n/3 \rceil}^{\lceil n/2 \rceil} k p_{i,k} & i = 1, n \\ \frac{n+1}{3} + (2/3) \sum_{k=\lceil n/3 \rceil}^{\lceil n/2 \rceil} k p_{i,k} & i = 2, \dots, n - 1. \end{cases}$$

Let $F(n, i) = \sum_{k=\lceil n/3 \rceil}^{\lceil n/2 \rceil} k p_{i,k}$. Note that $F(n, i)$ is the expected number of urinals occupied at saturation if position i is chosen first. But it is easy to see that

$$F(n, i) = 1 + \begin{cases} F(n - 2) & i = 1, n \\ F(n - 3) & i = 2, n - 1 \\ F(i - 2) + F(n - i - 1) & n = 3, \dots, n - 2 \end{cases}$$

where $F(n) = \sum_{i=0}^{n-1} \frac{(-2)^i (n-1)}{(i+1)!}$ is the expected number of urinals at saturation if all choices are random established in [5][12].

While $F(n)$ is monotonic as we saw above for the case $n = 5$ it is not the case that $F(n, 1) > F(n, i)$ for all i . On the other hand, it is fairly straightforward to show that $F(n, 1) + 1 > F(n, i)$ and $F(n, i) \leq \lceil n/2 \rceil$ for all i and n and from this conclude (from the formula for $E(n, i)$ above) that once again positions 1 (and n) are the optimal choice.

4 Variations on Our Theme

4.1 Different Filling Strategies

The above analysis assumes once saturation is reached the remaining positions are filled uniformly at random. This may not be realistic as one could imagine other approaches.

Consider, for example, a *lazy* filling strategy whereby a new entrant, finding the configuration saturated, chooses the first available urinal. We say a position is *fully-private* if the urinals on either side of it (if they exist) are unoccupied. A non-end position is *semi-private* if only one of its neighbors is occupied. A position is *non-private* if it is not fully private or semi-private. We say a man's

behavior is *fully-private first* if a man always chooses a fully-private position if available. All of the behaviors we have been considering are fully-private first.

It is fairly straightforward to show that for any behavior that follows a fully-private first initial strategy and a lazy filling strategy, the optimal choice is always position n as in this case all urinals except $n - 1$ must be used before your privacy is violated.

An interesting situation arises in the case where semi-private positions are preferred to non-private positions when filling. The expected time until your privacy is violated now depends on how many semi-private positions versus non-private positions exist at saturation and what types of gaps exist beside you (one or two unoccupied positions). Given this information the expected time of your privacy can be calculated using the appropriate negative hypergeometric distribution.

Consider what happens in the case of the distance maximizing behavior above if (1) in the initial phase when choosing between two equi-distant alternatives one always chooses the lower numbered urinal (i.e., the lazy choice) and (2) after reaching saturation the men choose randomly first among semi-private positions and only after they are filled they choose randomly among non-private positions. In this case, the advantage held by the end positions disappears! If $n = 8$ the expected time until privacy is violated for position 1 is 5 versus 6.33 if you choose position 3. If $n = 21$ the expected length of your privacy for position 21 is 14.5 versus 15 if you choose position 3. In fact, one can construct infinitely many n for which position 1 (or position n) is not the best choice. ($n = 8$ is the smallest case for position 1 and $n = 6$ the smallest for the last position.) But in each of these cases, we find that if the first urinal is not optimal, the last is, and vice versa. We conjecture that under the behavior above it will always be the case that either position 1 or position n will be optimal. Further we conjecture that if instead of making a lazy choice for equidistant positions in the first phase but make a random choice, the advantage for both position 1 and position n remains. If your filling strategy is to always fill a randomly chosen gap of size 2 in order to get a semi-private position (if one exists) in a lazy manner, then one can show that position 1 is no longer optimal (already it fails for $n = 4$) but position n is always optimal. For the random behavior and for any of the filling strategies considered above one can show that position n remains optimal.

4.2 Non-empty Initial Configuration

More often than not when you enter a public men's room in a busy area the configuration you observe is not the empty one. What should one do in this case?

A rule of thumb that follows from the discussion above is it likely the case that if one of the end positions is open and its single neighbor is unoccupied, you should choose that position. Note that this is not always the case! If we are in a situation where one of the ends is available but choosing it will force a gap of size 2 to form beside it, it may be the case that, if semi-private positions are preferred after saturation, choosing the end is not a good idea.

In general, the only way we know of to be certain you make the best choice is to compute for each position the expected length of privacy using the formulas provided above applying them to the subproblems created by the configuration. For example, if one is analyzing the case where maximizing the distance behavior is assumed, then one can compute the time to saturation by adding the times resulting from each of the gaps in the configuration using the formula $B(n)$ (or an appropriate modification in the case where one end of a gap is position 1 or n). After saturation, the filling time can be computed using the appropriate negative hypergeometric distribution. If the configuration is presented as $\{0, 1\}$ vector then this results in a $O(n^2)$ time algorithm for finding the best urinal to choose. If the configuration is given as a list of positions then in general the algorithm is only pseudo-polynomial. We conjecture there is a polynomial time algorithm for each of the behaviors and filling strategies discussed above but have not been able to determine one for all cases.

4.3 Dynamic Situation

Perhaps the least realistic assumptions we have made above are (1) the arrival rate of men is constant and (2) the men stay indefinitely once they arrive rather than leaving once they have completed their business. It is fairly evident that both the arrivals and service times of the men are complex random processes. As a first attempt, the use of queueing theory to model this aspect of the problem seems warranted.

The simplest assumption in this regard would be that the arrival times form a Poisson process and the time required to relieve oneself is exponential distributed. As there are n urinals, this leads to what is termed an M/M/ n queue in the standard queueing theory terminology [10,14]. Given $1/\lambda$, the mean time between arrivals, and $1/\mu$, the mean service time, it is straightforward to calculate such quantities as the expected number of “customers” in the system (either being serviced or waiting for a urinal to open up in case all are full), the probability that k urinals are free, etc. and use this information to help devise a strategy for picking a urinal.

For example, if $\frac{\lambda}{\mu} < n/2$ then you are in a region where one expects that saturation will not be reached during your visit (or even if it is the number of filled positions after saturation will be small.) In this case, if one assumes the maximizing distance behavior above then the advantage of the ends disappears. One should use the algorithm above to estimate the position that prolongs the time until saturation the longest. Generally, this will be somewhere in the largest gap of urinals and not necessarily at the end. (Again, if we assume a configuration is represented by a $\{0, 1\}$ vector this will be a polynomial time algorithm.) On the other hand, if one assumes random behavior then for most configurations (ones with a sufficiently large gap) a search using a Python program suggests the best choice is one that leaves three empty urinals between you and either the beginning or the end of the largest gap. This appears to insure that the time to saturation in that gap will be maximized.

The above analysis assumes you have an good estimate of λ and μ which seems unlikely². Without this information one can make a maximum likelihood estimate of $\frac{\lambda}{\mu}$ using the number of men in the configuration upon your arrival. If the current configuration is not saturated you may as well assume that this is the steady-state and follow the guidelines described in the last paragraph for the case $\frac{\lambda}{\mu} < n/2$. On the other hand if the configuration is saturated then you have no choice but to pick a semi-private or even non-private position. It might be interesting to investigate whether one such spot is better than another for increasing your privacy. This would seem to depend on having an estimate of the time remaining to service individuals currently in the configuration but perhaps not.

4.4 Game-Theoretic Formulation

Another objection to the approach taken above is that those arriving after the first arrival are limited to following a particular behavior. In reality they are also interested in maximizing their private time. How might considerations of more general strategies effect your choice? If the second man to enter is going to maximize his privacy based upon your choice how does that effect your choice as first. If k men enter simultaneously, is there a mechanism by which they might be led to choose a configuration that maximizes the average or maxmin privacy?

These questions suggest that our problem is in reality some sort of game played by multiple players in rounds and the tools of game theory might be applied to establish better strategies. We leave it up to experts in algorithmic game theory to formulate the right questions that might shed some light on this problem. We are hopeful this might lead to some interesting lines of research.

4.5 Metric Space Generalizations

The urinal problem is just one of many situations where physical privacy might be desirable, in a place where people are entering (and leaving) over time. The unfriendly theatre seating arrangement problem described above is one such example (as is any situation with open seating). Another might be called the beach blanket problem: Frankie and Annette arrive at the beach early in the morning and want to decide where to place their beach blanket to allow for the most privacy throughout the day in order to engage in whatever activities they have planned for the day. Under various assumptions about the behavior of later arrivals, what place should they choose? We note that some old fashioned urinals consist of one contiguous urinal with a single central drain. This would correspond to the continuous version of our original urinal problem.

² The designers of men's rooms do make estimates of such quantities in order to decide how many urinals are required for a given site. As one might expect this will depend on the size of a building, what it is used for (e.g., office building, cinema, shopping mall, etc.) and the expected mix of men versus women versus children. For tables of the recommended number of fixtures required see [8].

A natural generalization of the urinal problem that captures all of these might be termed the *metric space privacy problem*. You are given a metric space (M, d) , an ϵ (the *radius of privacy*), and a deterministic or randomized behavior describing the choices of points in the space of later arrivals. Given a configuration (a subset of points in M) choose a point that maximizes the (expected) time until your privacy radius is violated, i.e., a later arrival chooses a point within distance ϵ of your spot. In the theatre seating problem we take M to be the vertices of an $m \times n$ grid with edge weights equal to 1 and $\epsilon = 1$. In the beach blanket problem M may be modelled by a simple polygon (perhaps even a rectangle) with the Euclidean distance measure and ϵ may depend upon how much privacy you need. In the continuous urinal problem M is a unit interval and ϵ is a function of a man's width and privacy needs.

One suspects that some instances of these problems will turn out to be *NP*-hard as it would appear that in some cases you would be forced to solve a version of the obnoxious facility location problem that in some cases is known to be *NP*-hard. Consideration of a given metric space can be combined with dynamics and/or game theory to yield more problems. We feel there is the potential for many interesting open questions in this area.

5 Conclusions

Our main conclusion is that when faced with the decision of what urinal to choose upon entering the men's room, in order to maximize your privacy, you should probably choose the one furthest from the door if it is available and the one next to it is unoccupied. For a vast majority of the (what we consider) natural behaviors that men choosing urinals might follow, this choice is optimal (in the sense defined above). Beyond this observation, we feel that this problem leads to many interesting variations that are worthy of investigating further and we encourage everyone to do more of their thinking while using public restrooms.

Acknowledgments

We thank Pat Morin for pointing out reference [2] and the anonymous referees for many helpful and amusing suggestions. This research was supported in part by Natural Sciences and Engineering Research Council of Canada (NSERC) and Mathematics of Information Technology and Complex Systems (MITACS).

References

1. American Restroom Association, Public Restroom Design Issues, <http://www.americanrestroom.org/design/index.htm>
2. Decker, D.: A Woman's Guide on How to Pee Standing Up, <http://web.archive.org/web/20031202020806/http://www.restrooms.org/standing.html>

3. Flajolet, P.: A Seating Arrangement Problem, <http://algo.inria.fr/libraries/autocomb/fatmen-html/fatmen1.html>
4. Freedman, D., Shepp, L.: An Unfriendly Seating Arrangement Problem. *SIAM Review* 4, 150 (1962)
5. Friedman, H.D., Rothman, D.: Solution to An Unfriendly Seating Arrangement Problem. *SIAM Review* 6, 180–182 (1964)
6. Georgiou, K., Kranakis, E., Krizanc, D.: Random Maximal Independent Sets and the Unfriendly Theater Arrangement Problem. *Discrete Mathematics* 309, 5120–5129 (2009)
7. Hoffman, L.: Computers and Privacy: A Survey. *Computing Surveys* 1, 85–103 (1969)
8. International Code Council, *International Plumbing Code* (2009)
9. Keane, M., Kranakis, E., Krizanc, D., Narayanan, L.: Routing on Delay Tolerant Sensor Networks. In: *Algosensors 2009*, pp. 155–166 (2009)
10. Kleinrock, L.: *Queueing Systems, Volume 1: Theory*. Wiley, Hoboken (1975)
11. Knuth, D.: The Toilet Paper Problem. *American Math. Monthly* 91, 365–370 (1984)
12. MacKenzie, J.K.: Sequential Filling of a Line by Intervals Placed at Random and Its Application to Linear Absorption. *Journal of Chemical Physics* 37, 723–728 (1962)
13. Paola, C.: *A Survey of Obnoxious Facility Location Problems*. TR-99-11, University of Pisa, Dept. of Informatics (1999)
14. Ross, S.M.: *Stochastic Processes*, 2nd edn. John Wiley & Sons, Inc., Chichester (1996)
15. Soifer, S.: *The Evolution of the Bathroom and the Implications for Paruresis*, <http://paruresis.org/evolution.pdf>
16. Vogt, R., Nascimento, M.A., Harns, J.: On the Tradeoff between User-Location Privacy and Queried-Location Privacy in Wireless Sensor Networks. In: *AdHoc Now 2009*, pp. 241–254 (2009)

Fighting Censorship with Algorithms

Mohammad Mahdian

Yahoo! Research, Santa Clara, CA, USA

mahdian@alum.mit.edu

<http://www.mahdian.org>

Abstract. In countries such as China or Iran where Internet censorship is prevalent, users usually rely on proxies or anonymizers to freely access the web. The obvious difficulty with this approach is that once the address of a proxy or an anonymizer is announced for use to the public, the authorities can easily filter all traffic to that address. This poses a challenge as to how proxy addresses can be announced to users without leaking too much information to the censorship authorities. In this paper, we formulate this question as an interesting algorithmic problem. We study this problem in a static and a dynamic model, and give almost tight bounds on the number of proxy servers required to give access to n people k of whom are adversaries. We will also discuss how trust networks can be used in this context.

1 Introduction

Today, Internet is playing an ever-increasing role in social and political movements around the globe. Activists connect and organize online and inform ordinary citizen (and the rest of the world) of the news of arrests and crackdowns and other news that the political powers do not want to be spread. In particular, Web 2.0 has broken the media monopoly and has given voice to dissidents and citizen journalists with no access to traditional media outlets. The role that Twitter, Facebook, YouTube, CNN's iReport and many other websites and blogs have played in the recent events in Iran is a great example of this [9,10].

Threatened by this paradigm, repressive regimes have tried hard to control and monitor their citizen's access to the web. Sophisticated filtering and surveillance technologies are developed or purchased by governments such as China or Iran to disallow access to certain blacklisted websites (See Figure 1) or monitor the activities of users [19]. Blacklisted websites include news websites such as BBC or CNN, Web 2.0 websites, many blogs, and even Wikipedia. Internet censorship activities are documented in great detail by organizations such as Reporters Without Borders [18] or the OpenNet Initiative [16,5].

At the same time, users in such countries and supporters of free online speech have looked for *circumvention technologies* to get around Internet censorship. These range from secure peer-to-peer networks like Freenet [11,2] to anonymous traffic routing softwares like The Onion Router (Tor) [6] to web proxy systems such as Psiphon [17]. These tools sometimes use cryptographic protocols to provide security, but to elude detection and filtering, the common way is to route



Fig. 1. A typical Internet browsing session in Iran involves multiple encounters with pages like the above. The text in Persian reads: “According to the laws of the Islamic Republic of Iran and directives from the judiciary access to this website is forbidden.”

traffic through intermediaries not known to the censorship authorities. For example, when a user of Psiphon requests a webpage, this request is routed through a proxy. The proxy is a computer outside the censored country, and therefore can access the requested webpage and send it back to the user¹. This can be any computer on the net whose owner is willing to contribute to the anti-censorship network by installing the Psiphon software. Usually, this is done by people outside the censored country who have friends or relatives inside the censored country. They simply install the software on their machines (with full-time Internet connectivity), and send an invitation (which is basically a link through which the recipient can access the Internet) to their friends and family².

This model works well for users who have friends outside the censored territory. The challenge is to provide censorship-free Internet access to those who have no direct connection to the outside world. In the case of Iran, sometimes US-sponsored radio stations such as Voice of America or Radio Farda broadcast URLs of anonymizers on their program. The obvious problem with this approach

¹ This is a simplification of how Psiphon operates. In reality (starting from version 2.0), Psiphon is a 2-hop proxy system, where the first hop (the in-proxy) can be any computer outside the censored country that runs the Psiphon software and simply forwards the requests, and the second hop (the managed Psiphon server) is a dedicated computer that serves and manages Psiphon requests. For the purpose of our discussions, the simplified view is enough.

² Freenet’s “darknet” mode operates similarly.

is that the censorship authorities also listen to these radio stations and quickly add the announced URL to their blacklist.

This motivates the main problem studied in this paper: how can a set of proxies be distributed among n users, k of whom adversaries (agents of censorship), in such a way that all legitimate users can have access to a proxy that is not filtered. We give a more precise definition of our model in the next section, and define a static (one-shot) version of this problem, as well as a (more realistic) dynamic problem. As we observe in Section 3, the static problem is equivalent to a previously studied network design problem. The main contribution of this paper is our solution for the dynamic model, which will be presented in Section 4. In Section 5, we discuss the role of trust networks in building a user base, and how this affects the algorithmic problem studied in this paper. We conclude with a number of open problems and conjectures.

Related work. To the best of our knowledge, this is the first systematic study of methods for proxy distribution. As we observe in Section 3, the static version of this problem is essentially equivalent to finding union-free families of sets [13]. In coding theory, these objects are known as *superimposed codes* [12, 8], and have a wide range of applications (see, for example, [3, 4, 11]). The static case is also closely related to the combinatorial group testing literature [7]. The dynamic problem, however, does not fit within the group testing framework, as in our problem the adversary can strategically delay compromising a proxy.

2 The Model

Consider a population of n users, k of whom are adversaries, and the remaining $n - k$ are legitimate users. We do not know the identities of the adversaries, but throughout the paper we assume that $k \ll n$. We have a set of at most m keys (representing proxies or anonymizers) that we can distribute among these users in any way we want. If a key is given to an adversarial user, she can *compromise* the key, meaning that from that point on, the key will be unusable. Our goal is to distribute the keys in such a way that at the end, every legitimate user has at least one uncompromised key.

This problem can be studied in a static (one shot) model, or in a dynamic (adaptive) model. The static model is a one-round problem: we give each user a set of keys at once, and then the adversarial users compromise the keys they have received. In the dynamic model, there are multiple rounds. In each round we distribute a number of keys to users who have no uncompromised key. The next round starts when one or more of the keys given to an adversarial user is compromised. Note that in this model the adversarial users do not have to compromise a key as soon as they see them; instead, they can strategically decide when to compromise a key. Also, observe that in this model we can assume without loss of generality that in any round each user who has no uncompromised key receives only one key (i.e., there is no point in giving more than one uncompromised key to a user at any time).

In both of the above models, if the number m of available keys is at least n , the problem is trivially solvable: simply give each user her personal key. Our objective is to find the smallest value of m for which the problem is solvable. As we will see in the next sections, it is indeed possible to solve the problem with only a sublogarithmic number of keys.

Variants of the problem. Several variants of our model can also be considered. For example, what if instead of requiring all legitimate users to have access at the end of the algorithm, we only require a $1 - \epsilon$ fraction? Also, the dynamic model can be defined not as an adversarial model but as a stochastic one: instead of having a bound k on the number of adversarial nodes, we can assume that each node is adversarial with probability p . As we will remark at the end of Section 4, both of these variants are easily solvable using the same algorithms that we propose for the main problem (with $\log(1/\epsilon)$ replacing $\log n$ for the first variant, and pn replacing k for the second).

A more challenging variant of the problem concerns situations where the nodes can invite other users to join, forming a trust network structure. This gives rise to a challenging algorithmic problem that will be discussed in Section 5.

3 Static Key Distribution

The static key distribution problem is equivalent to designing a bipartite graph between the set of n users and the set of m available keys. An edge between a user and a key means that the key is given to that user. k of the user nodes in this graph are adversaries, and all of the neighbors of these nodes will be compromised. After that, a user is *blocked* if all its adjacent keys are compromised.

This is precisely equivalent to a secure overlay network design problem studied by Li et al. [14], although the motivating application and therefore the terminology in that problem are different from ours. Users in our problem are equivalent to Access Points (APs) there, and keys are equivalent to servelets.

The result in [14] that is relevant to our case is Theorem 6 (the adversarial model). That theorem exploits the connection between the solutions of our problem to k -union-free families of sets [13], and proves the following (restated using our terminology):

Theorem 1. (Restatement of Theorem 6 in [14]) *For every m and k , the maximum value of n for which the key distribution problem has a solution is at least $(1 - \frac{k^k}{(k+1)^{k+1}})^{-m/(k+1)}$ and at most $O(k2^{m/k}m^{-1/(2k)})$.*

The lower bound in this theorem is proved using the probabilistic method (giving each key to each user independently with probability $1/(k+1)$), and the upper bound is proved using Sperner's theorem in extremal set theory. A simple calculation from the above theorem gives the following.

Theorem 2. *There is a solution for the static key distribution problem with at most $O(k^2 \log n)$ keys. Furthermore, no key distribution scheme with fewer than $\Omega(k \log(n/k))$ keys can guarantee access to all legitimate users.*

4 Dynamic Key Distribution

In this section, we study the key distribution problem in the dynamic model, i.e., when the algorithm can respond to a compromised key by providing new keys for affected users. The quantity of interest here is the expected number of keys required before until *every* legitimate user gets access. The following theorem shows that $O(k \log(n/k))$ keys suffice.

Theorem 3. *The dynamic key distribution problem has a solution with at most $k(1 + \lceil \log_2(n/k) \rceil)$ keys.*

Proof. Consider the following algorithm: In the first round, all n users are divided into k groups of almost equal size, and each group is given a distinct key. After this, any time a key is compromised, the users that have that key (if there is more than one such user) are divided into two groups of almost equal size, and each group is given a new key (i.e., a key that has not been used before). This completes the description of the algorithm.

One can visualize the above algorithm as a tree, where the root corresponds to the set of all n users, and each node corresponds to a subset of users. The root has k children and other nodes have 2. The sets corresponding to the children of a node comprise an almost balanced partition of the corresponding set. In this view, any time the key corresponding to a node is compromised, we move one level down in the tree.

We now show that this procedure uses at most $k(1 + \lceil \log_2(n/k) \rceil)$ keys. Consider the k adversarial users. At any point in time, each such node is contained in one group. At the end of the first round, the size of this group is n/k , and after that, every time this adversary compromises a key, the size of this group is divided in half. Therefore, each of the k adversaries can compromise at most $\lceil \log_2(n/k) \rceil$ keys. Putting this together with the fact that we started with k keys in the first round gives us the result.

It is not hard to show that in the above algorithm, dividing the users initially into k groups and then into 2 groups in subsequent rounds is essentially the best possible for this style of algorithms. That is, we cannot asymptotically improve the bound given in the above theorem by devising a different partitioning scheme. This might lead one to believe that the bound in the above theorem is asymptotically optimal. However, this is not true. As the following theorem shows for the case of $k = 1$, it is possible to solve the problem with a sublogarithmic ($O(\log n / \log \log n)$) keys. The idea is to “reuse” keys that are already given to people in other branches. The following theorem will later guide us (and will serve as a basis of induction) to a solution for general k with sublogarithmic dependence on n .

Theorem 4. *For $k = 1$, the dynamic key distribution problem has a solution with $O(\frac{\log n}{\log \log n})$ keys.*

Proof. Let $\ell = \lceil \frac{\log n}{\log \log n} \rceil$. As in the proof of the previous theorem, the algorithm proceeds in a tree-like fashion. At first, the set of all users is divided into ℓ

groups of almost equal size, and a distinct key is given to each set. Once a key is compromised, we would know that the single adversary is among the users in the group that has that key. We call this group the *suspicious group*. Since $k = 1$, we know that all users in remaining groups are legitimate users; we call these users *trusted* users. We divide the suspicious group into ℓ subgroups of almost equal size. But instead of giving each subgroup its distinct new key (as was the case in the proof of Theorem 3), we give one subgroup a new key, and give the remaining $\ell - 1$ subgroups the other $\ell - 1$ keys that are already in use by trusted users.

Similarly, when another one of the keys fail, we would know which subgroup the adversary belongs to; so now only users in that subgroup are suspicious, and the remaining $\ell - 1$ subgroups become trusted. Again, we divide the suspicious subgroup into ℓ almost equal-sized subsubgroups. One of these subsubgroups is given a new key, and the remaining $\ell - 1$ are given the keys already in use. We also need to give keys to trusted users whose key is compromised (since there could be trusted users that use the same key as the users in the suspicious subgroup). We give these users an arbitrary uncompromised key already in the system. This process continues until the size of the suspicious group reaches 1, at which point we stop.

Since in each round, the algorithm divides the size of the suspicious group by ℓ , and in each round exactly one key is compromised, the total number of compromised key in this algorithm is at most $\log_\ell n$. This, together with the fact that there are precisely $\ell - 1$ keys that remain uncompromised, shows that the total number of keys used by our algorithm is at most

$$\ell - 1 + \frac{\log n}{\log \ell} \leq \frac{\log n}{\log \log n} + \frac{\log n}{\log \lceil \log n / \log \log n \rceil} = O\left(\frac{\log n}{\log \log n}\right).$$

This completes the proof.

We are now ready to give a sublogarithmic bound for the number of required keys for general k .

Theorem 5. *For any k , there is a solution for the dynamic key distribution problem that uses $O(k^2 \log n / \log \log n)$ keys in expectation.*

Proof. First, note that we can assume without loss of generality that $k < \log \log n$, since for larger values of k , Theorem 3 already implies the desired bound. We use induction on k . The $k = 1$ case is already solved in Theorem 4. For $k > 1$, we proceed as follows. The structure of the first stage of our algorithm is similar to the algorithm in the proof of Theorem 4: we fix $\ell = \lceil \frac{\log n}{\log \log n} \rceil$, and start by dividing the users into ℓ groups of almost equal size (to do this, we can put each user in one of the ℓ groups uniformly at random). At any point in this stage, there are precisely ℓ uncompromised keys in the system. Once a key is compromised, we replace it by a new key, split *all* groups that have the compromised key into ℓ groups and give each group one of the ℓ available keys. In other words, once a key is compromised, it is replaced by a new one and each

user who was using that key receives a *random* key among the ℓ available keys. This process is repeated for R rounds, where R will be fixed later.

The sketch of the rest of the proof is as follows: for any legitimate user, in each round the probability that the user receives a key shared by an adversary is at most k/ℓ (since there are k adversaries and ℓ keys available, and key assignments are random). Therefore, in expectation, after R rounds, this user has had at most Rk/ℓ compromised keys. However, at least one of the k adversaries has had at least R/k compromised keys in this stage (since each compromised key is given to an adversary). By setting the right value for R and using the Chernoff bound, we will be able to show that the set of users that have had at least R/k compromised keys contains at least one adversary and almost surely no legitimate user. Given this, we can give each user in this set her personal key, and solve the problem recursively (with at least one fewer adversary) for the remaining set of users.

We now formalize the proof. Consider an arbitrary legitimate user u , and define a sequence of random events that determine the number of times this user’s key is compromised. Every time u is given a new key, there is probability of at most k/ℓ (independent of all the previous events) that she receives a key that one of the adversaries in the system already has. If this event occurs, we wait until this key is compromised (if at all) and a new key is given to u ; otherwise, the next event corresponds to the next time one of the adversaries compromises a key. At that time, one or more adversaries will receive other random keys. The next event, whose probability can also be bounded by k/ℓ is that one of the keys given to the adversaries is the one u already has. We proceed until the end of the R rounds. This gives us a sequence of events at most R , each with probability k/ℓ of occurring independent of the previous events, and the number of time u ’s key is compromised can be bounded by the number of events that occur in this sequence. Therefore, this number can be bounded by the sum of R Bernoulli random variables, each with probability k/ℓ of being one. Let this sum be denoted by X . We have $\mu := \text{Exp}[X] = Rk/\ell$. By the Chernoff bound (Theorem 4.1 in [15]) with $1 + \delta = \ell/k^2$ we have:

$$\Pr[X > R/k] = \Pr[X > (1 + \delta)\mu] < \left[\frac{e^\delta}{(1 + \delta)^{1+\delta}} \right]^\mu < \left(\frac{ek^2}{\ell} \right)^{R/k}.$$

Setting $R = ck \log n / \log \log n$ for a large constant c , the above probability can be bounded by:

$$\exp \left(\frac{c \log n}{\log \log n} \log \left(\frac{ek^2 \log \log n}{\log n} \right) \right) < \exp \left(-\frac{c}{2} \log n \right) = n^{-c/2},$$

where the inequality follows from the fact that $k < \log \log n$ and therefore asymptotically, $\log \left(\frac{ek^2 \log \log n}{\log n} \right) < -\frac{1}{2} \log \log n$. By the above inequality, we know that the probability that the set S of users whose key is compromised at least R/k times contains u is at most $n^{-c/2}$. Therefore, the expected cardinality of this set is at most $k + n^{1-c/2}$, and hence we can afford to give each user in this set her personal proxy (just in the case of the unlikely event that there is a legitimate

user in this set). On the other hand, with probability 1 there is at least one adversary in S . Hence, by removing S , our problem reduces to one with fewer adversaries. We use induction to solve the remaining problem.

The expected total number of keys required by the above procedure is at most ℓ (the initial keys) plus R (the number of keys that replace a compromised key) plus $k + n^{1-c/2}$ (for the set S), plus keys that are needed for the recursive procedure. It is easy to see that this number is $O(k^2 \log n / \log \log n)$.

Our last result is a lower bound that shows that the upper bound in the above theorem is tight up to a factor of k , i.e., as a function of n , the optimal solution to the dynamic key distribution problem grows precisely as $\log n / \log \log n$. This theorem is proved using a simple entropy argument.

Theorem 6. *Any solution to the dynamic key distribution problem requires at least $\Omega\left(\frac{k \log(n/k)}{\log k + \log \log n}\right)$ keys.*

Proof. Consider an oblivious adversary: initially k random users are adversarial, and in each stage a random adversarial user compromises her key. Let ℓ denote the number of keys used by the algorithm. Since the algorithm eventually proves access to all legitimate users, it must be able to identify the set of all adversarial users. There are $\binom{n}{k}$ such sets, and they are all equally likely. The information that the algorithm receives in each round is the index of the key that is compromised. This index has one of the ℓ values, and therefore can be written in $\log \ell$ bits. Since the total number of rounds is at most ℓ , all the information that the algorithm receives can be written as an $\ell \log \ell$ bit binary sequence. Since this information is enough to find the k adversarial nodes, we must have

$$\ell \log \ell \geq \log \binom{n}{k} = \Omega(k \log(n/k)).$$

A simple calculation from the above inequality implies the lower bound.

Variants of the problem. It is not hard to see that if we only require access for a $1 - \epsilon$ fraction of the legitimate users, the upper bounds in the above theorems can be improved significantly. Namely, Theorem 3 can be adapted to solve the problem with only $O(k \log(1/\epsilon))$ keys.

Also, the upper bounds in this section do not require a precise knowledge of the value of k . Therefore, for the stochastic version of the problem, the same upper bounds hold with k replaced by pn . Furthermore, since the lower bound proof only uses a simple randomized adversary, a similar lower bound holds for the stochastic model. The details of these proofs are left to the final version of the paper.

5 Trust Networks

A common way to build a user base in a country under censorship is through personal trust relationships: the system starts with a few trusted users, and

then each trusted user will be allowed to invite new users whom she trusts to the system. In this way, the set of users grows like a *trust network*, a rooted tree on the set of users with the initial trusted users as the children of the root, and edges indicating trust relationships. In fact, newer versions of the Psiphon system rely on an invitation-based method very similar to this to build a user base and a tree structure of trusts among them [17].

Using trust networks to build the user base poses an additional risk to the system: if an adversary infiltrates the network, she can invite new adversarial users (perhaps even fake accounts controlled by herself) to the network, increasing the value of k for the algorithms in the previous section to an unacceptable level. In this section, we formulate this problem theoretically. We will leave it as an open problem for the most part; but for $k = 1$, we give non-trivial solutions, exhibiting that the general problem is interesting and possibly tractable.

The model. We have a population of n users that are nodes of a rooted tree (called the trust network) T . We assume that the depth of the tree T is small (e.g., at most $O(\log n)$)³. The adversary controls at most k nodes in this tree and all nodes descending from them. Our objective is to design a dynamic key distribution scheme that eventually guarantees that each legitimate user receives an uncompromised key. The challenge is to do this with the minimum number of keys.

In the following theorem, we show that for $k = 1$, it is possible to solve this problem with $O(\log n)$ keys. We leave the problem for $k > 1$ as an open question.

Theorem 7. *For $k = 1$, there is a solution for the dynamic key distribution problem on a trust network that uses at most $O(\log n)$ keys.*

Proof (Proof Sketch). We use a binary division method similar to the one used in Theorem 3, except here we keep the divisions consistent with the trust tree T . Recall that the algorithm in the proof of Theorem 3 maintains a *suspicious* group of users, and every time a new key is compromised, it divides the suspicious group into two subgroups of almost equal size, giving a distinct key to each subgroup. We do the same, except we maintain the following invariant: at any point, the *suspicious group* consists of some children u_1, \dots, u_r ($r \geq 2$) of a node u (we call u the root of the suspicious group), and all descendants of u_i 's. Of course we cannot be sure that u and its ancestors are legitimate, but if any of them is an adversary, we have already achieved the goal of giving access to all legitimate users. Therefore, we treat them as if they are legitimate. However, we need to be careful about the possibility that they are adversarial, and will try to get our algorithm to use too many keys *after* all legitimate nodes already get access. This can occur if we see the key that is given to nodes that our algorithm already assumes to be legitimate is compromised. In this case, we simply give

³ This assumption is necessary. For example, if the trust structure is a long path, it is easy to see that the problem has no non-trivial solution, even for $k = 1$. Furthermore, this is a realistic assumption since trust networks are generally small-world networks and have small diameter.

new personal keys to all the nodes on the path from the root of the tree to the root of the suspicious subtree, and stop the algorithm.

It is not hard to show that since the number of nodes in the suspicious group reduces by a constant factor in each iteration, and the depth of T is at most logarithmic, the total number of keys used by the above algorithm is at most logarithmic in n .

6 Conclusion and Open Problems

In this paper we studied a key distribution problem motivated by applications in censorship circumvention technologies. The algorithms we give for the problem are simple and intuitive, and therefore have a chance at being useful in practice. From a theoretical point of view, still a few questions remain open:

- In the dynamic model, what is the optimal dependence of the number of required keys on k ? There is a gap of roughly $O(k)$ between our best lower and upper bounds. Our conjecture is that (at least for k up to $O(\log n)$), $O(k \log n / \log \log n)$ is the right answer for this problem. A similar question can be asked for the static problem, but given the connection between this problem and a long-standing open problem in extremal set theory on k -union-free families of sets [13], this problem is unlikely to have a simple solution.
- Our upper bound in Theorem 5 is on the *expected number* of keys used, whereas the weaker bound in Theorem 3 holds with probability 1. Is it possible to prove a bound similar to Theorem 5 with probability 1? This is mainly a theoretical curiosity, since the bound in Theorem 5 holds not only in expectation but also with high probability.
- The key distribution problem with trust networks for $k > 1$.

Acknowledgments. I would like to thank Olgica Milenkovic for pointing me to the literature on superimposed codes. Also, I am thankful to Nicole Immorlica and Abie Flaxman for discussions on an earlier version of this problem.

References

1. Clarke, I., Sandberg, O., Wiley, B., Hong, T.W.: Freenet: A Distributed Anonymous Information Storage and Retrieval System. In: Federrath, H. (ed.) *Designing Privacy Enhancing Technologies*. LNCS, vol. 2009, pp. 46–66. Springer, Heidelberg (2001)
2. Clarke, I., Miller, S.G., Hong, T.W., Sandberg, O., Wiley, B.: Protecting Free Expression Online with Freenet. In: *IEEE Internet Computing*, pp. 40–49 (January/February 2002)
3. Clementi, A.E., Monti, A., Silvestri, R.: Selective families, superimposed codes, and broadcasting on unknown radio networks. In: *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 709–718 (2001)

4. Dai, W., Milenkovic, O.: Weighted Superimposed Codes and Integer Compressive Sensing. *IEEE Trans. on Inform. Theory* (June 2008) (submitted)
5. Deibert, R.J., Palfrey, J.G., Rohozinski, R., Zittrain, J. (eds.): *ACCESS DENIED: The Practice and Politics of Internet Filtering*. The MIT Press, Cambridge (2008)
6. Dingledine, R., Mathewson, N., Syverson, P.: Tor: the second-generation onion router. In: *Proceedings of the 13th Conference on USENIX Security Symposium*, pp. 21–21 (2004)
7. Du, D.-Z., Hwang, F.K.: *Combinatorial group testing and its applications*. World Scientific Publishing Co., Singapore (2000)
8. D'yachkov, A., Lebedev, V., Vilenkin, P., Yekhanin, S.: Cover-free families and superimposed codes: constructions, bounds and applications to cryptography and group testing. In: *Proceedings of International Symposium on Information Theory (ISIT)*, p. 117 (2001)
9. Fay, J.: Iran's revolution will not be televised, but could be tweeted, *The Register*, June 16 (2009), http://www.theregister.co.uk/2009/06/16/iran_twitter/
10. Grossman, L.: Iran Protests: Twitter, the Medium of the Movement, *Time*, June 17 (2009), <http://www.time.com/time/world/article/0,8599,1905125,00.html>
11. Indyk, P.: Deterministic Superimposed Coding with Applications to Pattern Matching. In: *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, p. 127 (1997)
12. Kautz, W., Singleton, R.: Nonrandom binary superimposed codes. *IEEE Transactions on Information Theory* 10(4), 363–377 (1964)
13. Kleitman, D., Spencer, J.: Families of k-independent sets. *Discrete Mathematics* 6, 255–262 (1973)
14. Li, L., Mahdian, M., Mirrokni, V.: Secure Overlay Network Design. *Algorithmica* 57(1), 82–96 (2010)
15. Motwani, R., Raghavan, P.: *Randomized Algorithms*. Cambridge University Press, Cambridge (1995)
16. OpenNet Initiative, ONI reports and articles, <http://opennet.net/reports>
17. Psiphon, Developed at the Citizen Lab at the University of Toronto, <http://psiphon.ca>
18. Reporters Without Borders, *Internet Enemies*, 2009th edn., March 12 (2009), http://www.rsf.org/IMG/pdf/Internet_enemies_2009_2_.pdf
19. Rhoads, C., Chao, L.: Iran's Web Spying Aided By Western Technology. *The Wall Street Journal*, A1 (June 22, 2009)

The Complexity of Flood Filling Games

David Arthur, Raphaël Clifford, Markus Jalsenius,
Ashley Montanaro, and Benjamin Sach

Department of Computer Science, University of Bristol, UK
dave@localstorm.co.uk, {clifford,markus,montanar,ben}@cs.bris.ac.uk

Abstract. We study the complexity of the popular one player combinatorial game known as Flood-It. In this game the player is given an $n \times n$ board of tiles, each of which is allocated one of c colours. The goal is to fill the whole board with the same colour via the shortest possible sequence of flood filling operations from the top left. We show that Flood-It is **NP-hard** for $c \geq 3$, as is a variant where the player can flood fill from any position on the board. We present deterministic $(c-1)$ and randomised $2c/3$ approximation algorithms and show that no polynomial time constant factor approximation algorithm exists unless **P=NP**. We then demonstrate that the number of moves required for the ‘most difficult’ boards grows like $\Theta(\sqrt{cn})$. Finally, we prove that for random boards with $c \geq 3$, the number of moves required to flood the whole board is $\Omega(n)$ with high probability.

1 Introduction

In the popular one player combinatorial game known as Flood-It, each tile of an $n \times n$ board is allocated one of c colours, where c is a parameter of the game. Two left/right/up/down adjacent tiles are said to be connected if they have the same colour and a (connected) region of the board is defined to be any maximal connected component. The standard version of the game starts with the player ‘flooding’ the region that contains the top left tile. The flooding operation simply involves changing the colour of all the tiles in the region to be some new colour. However, this also has the effect of connecting the newly flooded region to all neighbouring regions of this colour. The overall aim is to flood the entire board, that is connect all regions, in as few flooding operations as possible. Figure 1 gives an example of the first few moves of a game. The border shows the outline of the region which has so far been flooded.

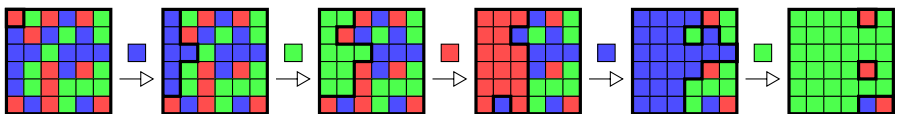


Fig. 1. A sequence of five moves on a 6×6 Flood-It board with 3 colours

We show that not only are natural greedy approaches bad, but in fact finding an optimal solution for Flood-It is **NP**-hard for $c \geq 3$ and that this also holds for a variant of the game we call Free-Flood-It where the player can perform flooding operations at any position on the board. Next we show how a $(c - 1)$ approximation and a randomised $2c/3$ approximation algorithm can be derived. However, no polynomial time constant factor, independent of c , approximation algorithm exists unless **P=NP**. We then consider how many moves are required for the most difficult boards and show that the number grows as fast as $\Theta(\sqrt{cn})$. Finally we investigate boards where the colours of the tiles are chosen at random and show that for $c \geq 3$, the number of moves required to flood the whole board is $\Omega(n)$ with high probability.

Publicly available versions, including our own implementation, can be found linked from <http://floodit.cs.bris.ac.uk>. Our implementation provides two novelties relevant to the reader. First, we have included playable versions of both the **NP**-completeness embeddings described in the paper. Second, the reader can watch the various algorithms discussed play Flood-It.

History and related work: Perhaps the most famous recent hardness result involving a popular game is the **NP**-completeness of Tetris [3]. Flood-It seems to be a somewhat newer game than Tetris, first making its appearance online in early 2006 courtesy of a company called Lab Pixies. Since then numerous versions have become available for almost every conceivable platform. We have very recently become aware of a sketch proof by Elad Verbin posted on a blog of the **NP**-hardness of Flood-It with 6 colours [11]. Although our work was completed independently, it is interesting to note that there is some similarity to the techniques used in our **NP**-hardness proof for $c \geq 3$ colours. The most closely related game whose complexity has been studied in detail is known as Clickomania [2]. A rectangular board is initialised in the same way as in Flood-It. On each move, the player chooses a connected monochromatic component of at least two tiles to remove after which any tiles above it fall down as far as they can. **NP**-hardness results were given for particular board shapes and numbers of colours. Flood-It can also be thought of as a model for a number of different (possibly not entirely) real world applications. For example, our results supplement that of recent work on zombie infestation [9] if one regards the flooding operation as one where the minds of neighbouring non-zombies are infected by those who have already been turned into zombies. A separate but no less significant line of research considers the complexity of tools commonly provided with Microsoft Windows. Previous work has shown that aspects of Excel [4] and even Minesweeper [6] are **NP**-complete. Our work extends this line of research by showing that flood filling in Microsoft's Paint application is also **NP**-hard.

1.1 Notation and Definitions

Let $B_{n,c}$ be the set of all $n \times n$ boards with at most c colours. We write $m(B)$ for the minimum number of moves required to flood a board $B \in B_{n,c}$. We will refer to rows and columns in a board in the usual manner. We further denote

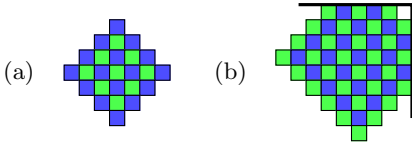


Fig. 2. (a) An alternating 4-diamond and (b) a cropped 6-diamond

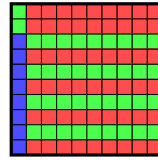


Fig. 3. A 10×10 board where a greedy approach is bad

the colour of the tile in row i and column j as $B[i, j]$; colours are represented by integers between 1 and c . Throughout we assume that $2 \leq c \leq n^2$.

We define a *diamond* to be a diamond-shaped subset of the board (see Figure 2a). These structures are used throughout the paper. The centre of the diamond is a single tile and the *radius* is the number of tiles from its centre to its leftmost tile. We write r -diamond to denote a diamond of radius r . A single tile is therefore a 1-diamond. For $i \in \{1, \dots, r\}$, the i th *layer* of an r -diamond is the set of tiles at board distance $i - 1$ from its centre. We will also consider diamonds which are cropped by intersection with the board edges as in Figure 2b.

2 A Greedy Approach Is Bad

An obvious strategy for playing the Flood-It game is the greedy approach. There are two natural greedy algorithms: (1) we pick the colour that results in the largest gain (number of acquired tiles), or (2) we choose the colour dominating the perimeter of the currently flooded region. It turns out that both these approaches can be surprisingly bad. To see this, let B be the 10×10 board on three colours illustrated in Figure 3. The number of moves required to flood B is three. However, either greedy approach given would first pick the colours appearing on the horizontal lines before finally choosing to flood the left-hand vertical column. In both cases, this requires 10 moves to fill the board. It should be clear how this example can easily be extended to arbitrarily large $n \times n$ boards.

3 The Complexity of Flood-It

Let c -FLOOD-IT denote the problem which takes as input an $n \times n$ board B of c colours and outputs the minimum number of moves $m(B)$ in a Flood-It game that are required to flood B . Similarly, let c -FREE-FLOOD-IT denote the generalised version of c -FLOOD-IT in which we are free to flood fill from an arbitrary tile in each move. Although we have seen that a straightforward greedy algorithm fails, it is not too far-fetched to think that a dynamic programming approach would solve these problems efficiently, but the longer one ponders over it, the more inconceivable it seems. To aid frustrated Flood-It enthusiasts, we prove in this section that both c -FLOOD-IT and c -FREE-FLOOD-IT are indeed NP-hard, even when the number of colours is as small as three.

To show **NP**-hardness, we reduce from the *shortest common supersequence* problem, denoted SCS, which is defined as follows. The input is a set S of k strings over an alphabet Σ . A *common supersequence* s of the strings in S is a string such that every string in S is a subsequence of s . The output is the length of a shortest common supersequence of the strings in S . The decision version of SCS takes an additional integer ℓ and outputs yes if the shortest common supersequence has length at most ℓ , otherwise it outputs no.

Maier [8] showed in 1978 that the decision version of SCS is **NP**-complete if the alphabet size $|\Sigma| \geq 5$. A couple of years later, R ah a and Ukkonen [10] extended this result to hold for $|\Sigma| \geq 2$. For a long time, various groups of people tried to approximate SCS but no polynomial-time algorithm with guaranteed approximation bound was to be found. It was not until 1995 that Jiang and Li [5] settled this open problem by proving that no polynomial-time algorithm can achieve a constant approximation ratio for SCS, unless $\mathbf{P} = \mathbf{NP}$. The following lemma proves the **NP**-hardness of both c -FLOOD-IT and c -FREE-FLOOD-IT when the number of colours is at least four. The inapproximability of both problems also follows immediately from the approximation preserving nature of the reduction. We will need a more specialised reduction for the case $c = 3$, which is given in Lemma 2.

Lemma 1. *For $c \geq 4$, c -FLOOD-IT and c -FREE-FLOOD-IT are **NP**-hard (and the decision versions are **NP**-complete). Further, for an unbounded number of colours c , there is no polynomial-time constant factor approximation algorithm, unless $\mathbf{P} = \mathbf{NP}$.*

Proof. The proof is split into two parts; first we prove the lemma for c -FLOOD-IT in which we flood fill from the top left tile in each move, and in the second part we generalise the proof to c -FREE-FLOOD-IT in which we can flood fill from any tile in each move.

We reduce from an instance of SCS that contains k strings s_1, \dots, s_k each of length at most w over the alphabet Σ . Suppose that $\Sigma = \{a_1, \dots, a_r\}$ contains $r \geq 2$ letters and let $\Sigma' = \{b_1, \dots, b_r\}$ be an alphabet with r new letters. For $i \in \{1, \dots, k\}$, let s'_i be the string obtained from s_i by inserting the character b_j after each a_j and inserting the character b_1 at the very front. For example, from the string $a_3a_1a_4a_3$ we get $b_1a_3b_3a_1b_1a_4b_4a_3b_3$.

Let $\Sigma \cup \Sigma'$ represent the set of $2r$ colours that we will use to construct a board B . First, for $i \in \{1, \dots, k\}$, we define the $|s'_i|$ -diamond D_i such that the j th layer will contain only one colour which will be the j th character from the right-hand end of s'_i . Thus, the colour of the outermost layer of D_i is the first character of s'_i (which is b_1 for all strings) and the centre of D_i is the last character of s'_i . The reason why we intersperse the strings with letters from the auxiliary alphabet Σ' is to ensure that no two adjacent layers of a diamond have the same colour. This property is crucial in our proof. Let B be a sufficiently large $n \times n$ board constructed by first colouring the whole board with the colour b_1 and then placing the k diamonds D_i on B such that no two diamonds overlap. Since each of the k diamonds has a radius of at most $2w + 1$, we can be assured that n never has to be greater than $k(4w + 1)$.

Suppose that s is a shortest common supersequence of s_1, \dots, s_k and suppose its length is ℓ . We will now argue that the minimum number of moves to flood B is exactly 2ℓ , first showing that 2ℓ moves are sufficient. Let s' be the 2ℓ -long string obtained from s by inserting the character b_j after each a_j . We make 2ℓ moves by choosing the colours in the same order as they appear in s' . Note that we flood fill from the top left tile in each move. From the construction of the diamonds D_i it follows that all diamonds, and hence the whole board, are flooded after the last character of s' has been processed.

It remains to be shown that at least 2ℓ moves are necessary to flood B . Let s'' be a string over the alphabet $\Sigma \cup \Sigma'$ that specifies a shortest sequence of moves that would flood the whole board B . From the construction of the diamonds D_i it follows that the string obtained from s'' by removing every character in Σ' is a common supersequence of s_1, \dots, s_k and therefore has length at least ℓ . By symmetry (replace every a_j with b_j in the strings s_1, \dots, s_k), the string obtained from s'' by removing every character in Σ has length at least ℓ as well. Thus, the length of s'' is at least 2ℓ .

Since the decision version of SCS is **NP**-complete even for a binary alphabet Σ , it follows that c -FLOOD-IT is **NP**-hard for $c \geq 4$, and the decision version is **NP**-complete. The inapproximability result in the statement of the lemma follows immediately from the reduction.

Now we show how to extend these results to c -FREE-FLOOD-IT. The reduction from SCS is similar to the previously presented reduction. However, instead of constructing only one board B , we construct $2kw + 1$ copies of B and put them together to one large $n' \times n'$ board B' . If necessary in order to make B' a square, we add sufficiently many $n \times n$ boards that are filled only with the colour b_1 . Note that $(2kw + 1)n$ and hence $(2kw + 1)k(4w + 1)$ is a generous upper bound on n' .

From the construction of B' it follows that exactly 2ℓ moves are required to flood B' if we flood fill from the top left tile in each move; all copies of B will be flooded simultaneously. The question is whether we can do better by flood filling from tiles other than the top left one (or any tile in its connected component). That is, can we do better by picking a tile inside one of the diamonds? We will argue that the answer is no. First note that $2\ell \leq 2kw$. Suppose that we do flood fill from a tile inside some diamond D for some move. This move will clearly not affect any of the other diamonds on B' . Suppose that this move would miraculously flood the whole of D in one go so that we can disregard it in the subsequent moves. However, there were originally $2kw + 1$ copies of D , which is one more than the absolute maximum number of moves required to flood B' , hence we can use a recursive argument to conclude that flood filling from a tile inside a diamond will do us no good and would only result in more moves than if we choose to flood fill from the top left tile in each move. \square

The reduction in the previous proof is approximation preserving, which allowed us to prove that there is no efficient constant factor approximation algorithm. We reduced from an instance of SCS by doubling the alphabet size, resulting in instances of c -FLOOD-IT and c -FREE-FLOOD-IT with $c \geq 4$ colours. To establish

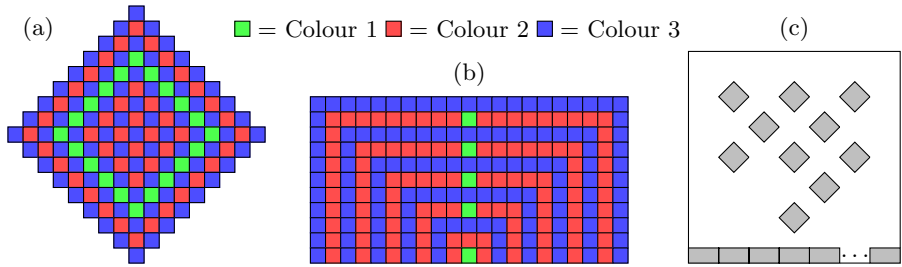


Fig. 4. An example of (a) a diamond, (b) a rectangle and (c) a board constructed in the proof of Lemma 2

NP-hardness for $c = 3$ colours, we need to consider a different reduction. We do this in the lemma below by reducing from the decision version of SCS over a binary alphabet to the decision versions of 3-FLOOD-IT and 3-FREE-FLOOD-IT. Note that this reduction is not approximation preserving.

Lemma 2. *3-FLOOD-IT and 3-FREE-FLOOD-IT are NP-hard (and the decision versions are NP-complete).*

Proof. We reduce from an instance of the decision version of SCS on k strings s_1, \dots, s_k of length at most w over the binary alphabet $\{1, 2\}$ and an integer ℓ . The yes/no question is whether there exists a common supersequence of length at most ℓ .

For $i \in \{1, \dots, k\}$, let s'_i be the string obtained from s_i by inserting the new character 3 at the front of s_i and after each character of s_i . Let the set $\{1, 2, 3\}$ represent the colours that we will use to construct a board B . First, for each of the k strings s'_i we define the diamond D_i exactly as in the proof of Lemma 1 (see Figure 4a). We define R to be the following rectangular area of the board of width $4\ell + 5$ and height $2\ell + 3$. Let x be the middle tile at the bottom of R . Around x we have layers of concentric half rectangles (see Figure 4b). We refer to these layers as *arches*, with the first arch being x itself. As demonstrated in the figure, the first arch has the colour 1 and the second arch has the colour 2. All the remaining odd arches have the colour 3, and all the remaining even arches are coloured 2 everywhere except for the tile above x which has the colour 1. As described in detail below, the purpose of these arches is to control which minimal sequences of moves would flood B .

Let B be a sufficiently large $n \times n$ board constructed as follows. First colour the whole board with the colour 3. Then, at the bottom of B starting from the left, place $2\ell + 3$ copies of R one after another without any overlaps. Finally place the k diamonds D_i on B such that no two diamonds overlap and no diamond overlaps any copy of R . Figure 4c illustrates a board B . Since a diamond has a radius of at most $2w + 1$ and $\ell \leq kw$, $k(4w + 1) + (2kw + 3)(4kw + 5)$ is an upper bound on n .

The reason why we place copies of R on the board B is to make sure that at least $2\ell + 2$ moves are required to flood B , even in the absence of diamonds.

To see this, suppose first that we flood fill from the top left square in each move. From the definition of the arches of R , disregarding the diamonds on B , a minimal sequence of moves will consist of ℓ 1s or 2s interspersed with a total of $\ell - 1$ 3s, followed by the three moves 3, 2 and 1, respectively. Note that only one copy of R on B would be enough to achieve this. However, having several copies of R on B does not affect the minimum number of moves as all copies will get flooded simultaneously. The idea with the $2\ell + 3$ copies of R is to make sure that at least $2\ell + 2$ moves are required to flood B even when we are allowed to choose which tile to flood fill from in each move. To see this, suppose that we choose to flood fill from a tile inside one of the copies of R . Since there are $2\ell + 3$ copies, similar reasoning to the end of the proof of Lemma 1 tells us that we will do worse than $2\ell + 2$ moves.

We will now argue that the number of moves required to flood B is $2\ell + 2$ if and only if there is a common supersequence of s_1, \dots, s_k of length at most ℓ . We choose to flood fill from the top left tile in each move.

Suppose first that there is a common supersequence s of length $\ell' \leq \ell$. Let s' be the string s followed by $\ell - \ell'$ 1s. Let s'' be the $(2\ell + 2)$ -long string obtained from s' by inserting a 3 after each character of s' and adding the two additional characters 2 and 1 to the end. We make $2\ell + 2$ moves by choosing the colours in the same order as they appear in s'' . Note that all diamonds are flooded after $2\ell'$ moves, and by the last move we have also flooded every copy of R , and hence the whole board B .

Suppose second that B can be flooded in $2\ell + 2$ moves. The centre of each diamond has the colour 3 and therefore the first 2ℓ moves flood the diamonds. The subsequence of these first 2ℓ moves induced by the the colours 1 and 2 is an ℓ -long common supersequence of s_1, \dots, s_k . □

We can now summarise Lemmas 1 and 2 in the following theorem.

Theorem 3. *For $c \geq 3$, c -FLOOD-IT and c -FREE-FLOOD-IT are NP-hard (and the decision versions are NP-complete). Further, for an unbounded number of colours c , there is no polynomial-time constant factor approximation algorithm, unless $P = NP$.*

4 Approximating the Number of Moves

As we have seen, c -FLOOD-IT and c -FREE-FLOOD-IT are not efficiently approximable to within a constant factor for an unbounded number of colours c . However, a $(c - 1)$ -approximation for c -FLOOD-IT, $c \geq 3$, can easily be obtained as follows. Suppose that B is a board on the colours $1, \dots, c$. Clearly, if we repeatedly cycle through the sequence of colours $1, \dots, c$ then B will be flooded after at most $c \times m(B)$ moves. We can do a little better by first cycling through the ordered sequence of colours $1, \dots, c$ and then repeatedly alternating between a cycle of the sequence $(c - 1), \dots, 1$ and a cycle of $2, \dots, c$ until there are only two distinct colours left on the board, after which we alternate between the two remaining colours. Note that there are always exactly two distinct colours left

before the final move. The board B is guaranteed to be flooded after at most $c + (c - 1)(m(B) - 2) + 1 \leq (c - 1)m(B)$ moves, which gives us a $(c - 1)$ -approximation algorithm.

A randomised approach with an expected number of moves of approximately $2c/3 \times m(B)$ is obtained as follows. Suppose that s is a minimal sequence of colours that floods B (flood filling from the top left square in each move). We shuffle the c colours and process them one by one. If B is not flooded then we shuffle again and repeat. Thus, if $m(B) = 1$ then the algorithm takes c moves. If $m(B) = 2$ then it takes $c + \frac{1}{2}c = 3c/2$ expected number of moves; with probability a half, both moves in s appear (in correct order) during the first c moves, otherwise we need c additional moves for the last move in s . We generalise this as follows. Let $T(m)$ be (an upper bound on) the expected number of moves it takes to produce a fixed sequence of m moves. We have $T(m) = c + \frac{1}{2}T(m - 1) + \frac{1}{2}T(m - 2)$. Solving the recurrence with the values of $T(1)$ and $T(2)$ above gives us a solution in which $T(m)$ is asymptotically $(2c/3)m$ for a fixed c .

5 General Bounds on the Number of Moves

Recall that we denote the minimum number of moves which flood some board B as $m(B)$. In this section we investigate bounds on the maximum $m(B)$ over all boards in $B_{n,c}$ which we denote $\max\{m(B) \mid B \in B_{n,c}\}$. Intuitively, this can be seen as the minimum number of moves to flood the ‘worst’ board in $B_{n,c}$.

For motivation, consider an $n \times n$ checker board of two colours as shown in Figure 5. First observe that as the board has only two colours, the player has no choice in their next move. Consider a diagonal of tiles in the direction top-right to bottom-left where the 0th diagonal is the top-left corner. Further observe that move k floods exactly the k th diagonal, so the total number of moves is $2(n - 1)$. Thus we have shown that $\max\{m(B) \mid B \in B_{n,c}\} \geq 2(n - 1)$.

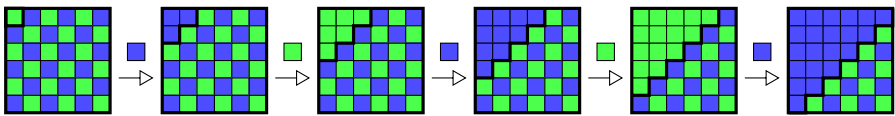


Fig. 5. Progression of a 6×6 checker board

We now give an overview of a simple algorithm which floods any board in $B_{n,c}$ in at most $c(n - 1)$ moves. The algorithm performs n stages. The purpose of the i th stage is to flood the i th row. Stage i repeatedly picks the colour of the leftmost tile in row i which is not in the flooded region, until row i is flooded.

First observe that Stage 1 performs at most $n - 1$ moves to flood row i (we can flood at least one tile of row 1 per move). When the algorithm begins Stage $i \geq 2$, observe that row $i - 1$ is entirely flooded as well as any tiles in row i

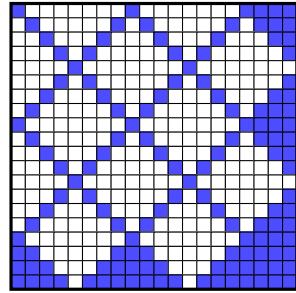


Fig. 6. The board decomposition used in the proof of Theorem 4

Fig. 7. 4-diamonds packed into a 20x20 board

which match the colour of row $i - 1$. Therefore when a new colour is selected, all tiles in row i of this colour become flooded. Hence at most $c - 1$ moves are performed by Stage i . Summing over all rows, this gives the desired bound that $\max\{m(B) \mid B \in B_{n,c}\} \leq c(n - 1)$. Observe that from the previous example with the checker board on $c = 2$ colours, the bound $c(n - 1)$ is tight. Thus, the checker board is the ‘worst’ board in $B_{n,2}$.

As motivation, we have given weak bounds on $\max\{m(B) \mid B \in B_{n,c}\}$. We now tighten these bounds for large c by providing a better algorithm for flooding an arbitrary board. We will also give a description of ‘bad’ boards which require many moves to be flooded. It will turn out that $\max\{m(B) \mid B \in B_{n,c}\}$ is asymptotically $\Theta(\sqrt{cn})$ for increasing n and c .

Theorem 4. *There exists a polynomial time algorithm for FLOOD-IT which can flood any $n \times n$ board with c colours in at most $2n + (\sqrt{2c})n + c$ moves.*

Proof. For a given integer ℓ (to be determined later), we partition the board horizontally into $\ell + 1$ contiguous sections, denoted S_0, \dots, S_ℓ from top to bottom, as follows. Let $q = \lfloor n/\ell \rfloor$ and $r = n \bmod \ell$. Section S_0 consists of the first $\lceil q/2 \rceil$ rows, S_1, \dots, S_r contain $(q + 1)$ rows each (if $r > 0$), and $S_{r+1}, \dots, S_{\ell-1}$ contain q rows each (if $r < \ell - 1$). Section S_ℓ contains $\lfloor q/2 \rfloor$ rows. See Figure 6 for an illustration. We let $y(i)$ denote the final row of S_i .

The algorithm performs the following three stages. Stage 1: Flood the first column. Stage 2: Flood row $y(x)$ for all $0 \leq x < \ell$. Stage 3: Cycle through the c colours until the board is flooded.

The correctness of our algorithm is immediate as Stage 3 ensures that the board is flooded by cycling colours. Stage 1 can be implemented to perform at most $n - 1$ moves as argued for the simple algorithm above. Similarly, Stage 2 can be completed in $\ell(n - 1)$ moves. We now analyse Stage 3.

First consider S_0 . At the start of Stage 3, row $y(0)$ is entirely in the top-left region, so a single cycle of the c colours suffices to expand the region to include row $y(0) - 1$. Each subsequent cycle of c colours expands the region to include an additional row. Therefore, after $c(\lceil q/2 \rceil - 1) \leq cq/2$ moves of Stage 3, all

rows above $y(0)$ are included in the top left region. Similarly, the section S_ℓ will be included in the top-left region as it contains $\lfloor q/2 \rfloor \leq q/2$ rows.

Now consider section S_i for some $0 < i < \ell$. Observe that there are at most q rows in S_i which are not already completely in the top-left section (after stage 2). Further observe that any cycle of colours expands the region to include *two* more of these rows. One row is gained from the region bordering the top of the section (which is in the top-left region from stage 2). The second is gained from the region bordering the bottom of the section (which is again in the top-left region). Therefore after at most $c\lceil q/2 \rceil$ moves of Stage 3 the board is flooded.

Over all three stages this gives a total of at most $n + \ell n + c\lceil q/2 \rceil$ moves. We pick $\ell = \lceil \sqrt{c/2} \rceil$ to minimise this number of moves. By recalling that $q = \lfloor n/\ell \rfloor$ and simplifying we have that this total is less than $2n + \sqrt{2c}n + c$ moves. \square

Theorem 5. *For $2 \leq c \leq n^2$, there exists an $n \times n$ board with (up to) c colours which requires at least $\sqrt{c-1}n/2 - c/2$ moves to flood.*

Proof. Suppose first that c is even. For a given integer $r \geq 1$, let $D_{(x,y)}$ be an r -diamond where odd layers are coloured x and even layers are coloured y . Any board containing $D_{(x,y)}$ requires at least r moves of colours x and y . Further, observe that as long as the centre of $D_{(x,y)}$ is in the board, even if it is cropped by at most two edges of the board, at least r moves of colours x and y are still required (see Figure 2b). We refer to such an r -diamond as *good*. The central idea is to populate the board with good r -diamonds, $D_{(1,2)}, D_{(3,4)}, \dots, D_{(c-1,c)}$. As each r -diamond uses two colours (or one colour if $r = 1$) which do not occur in any other diamond, the board must take at least $rc/2$ moves to flood.

It is not difficult to show that at least $(n^2 - r^2)/(2r^2)$ good r -diamonds can be embedded in an $n \times n$ board. An example of such a packing for a 20×20 board is given in Figure 7 (which shows only the edges of diamonds and not their colouring). This scheme generalises well to an $n \times n$ board but the details are omitted in the interest of brevity.

We now take $r = \lfloor n/\sqrt{c} \rfloor < n/2$ and note that $r \geq 1$. As $r < n/2$, the r -diamonds are cropped by at most two board edges as required. Therefore we have at least $(n^2 - r^2)/(2r^2) \geq c/2 - 1/2$ good r -diamonds in our board. However, as the number of good r -diamonds is an integer, this is at least $c/2$ as required. Therefore, at least $rc/2 > n\sqrt{c}/2 - c/2$ moves are required to flood this board.

Finally, in the case that c is odd we proceed as above using $c-1$ of the colours to give the stated result. \square

Corollary 6. $(\sqrt{c-1}n - c)/2 \leq \max\{m(B) \mid B \in B_{n,c}\} \leq 2n + \sqrt{2c}n + c.$

6 Random Boards

In this section, we try to understand the complexity of a random Flood-It board – that is, a board where each tile is coloured uniformly at random. Intuitively, such boards should usually require a large number of moves to flood. We will see that this intuition is indeed correct, for boards of three or more colours: in fact, almost all such boards need $\Omega(n)$ moves, as formalised in the following theorem.

Theorem 7. *Let $B \in B_{n,c}$ be a board where the colour of each tile is chosen uniformly at random from $\{1, \dots, c\}$. Then, for $c \geq 4$, $\Pr[m(B) \leq 2(3/10 - 1/c)(n - 1)] < e^{-\Omega(n)}$. For $c = 3$, $\Pr[m(B) \leq (n - 1)/22] < e^{-\Omega(n)}$.*

In order to prove this theorem, we will use two lemmas concerning paths in Flood-It boards. Let P be a simple path in a Flood-It board, i.e. a simple path on the underlying square lattice¹, where tiles are vertices on the path. Note that a path of length k includes $k + 1$ tiles. We say that a simple path P is *non-touching* if every tile in P is adjacent to at most two tiles that are also in P . Define the *cost* of P , $\text{cost}(P)$, to be the number of monochromatic connected components of the path, minus one (so a monochromatic path has cost 0). For proofs of the following two lemmas, see [1].

Lemma 8. *For any $B \in B_{n,c}$, there is a non-touching path from $(1, 1)$ to (n, n) with cost at most $m(B)$.*

Lemma 9. *For any integer $\ell \geq 3$, there are at most $4 \cdot 7^{(\ell-1)/2} < 2 \cdot (\sqrt{7})^\ell$ non-touching paths of length ℓ from any given tile.*

Before proving Theorem 7, the last result we will need is the following standard Chernoff bound.

Fact 10. *Let X_i , $1 \leq i \leq m$, be independent 0/1-valued random variables with $\Pr[X_i = 1] = p$. Then $\Pr[\frac{1}{m} \sum_{i=1}^m X_i \geq p + \epsilon] \leq e^{-D(p+\epsilon||p)m} \leq e^{-2\epsilon^2 m}$, where $D(x||y) = x \ln(x/y) + (1-x) \ln((1-x)/(1-y))$ is the Kullback-Leibler divergence.*

Proof (of Theorem 7). For any $k \geq 0$, and for any board B such that $m(B) \leq k$, by Lemma 8 there exists a non-touching path from $(1, 1)$ to (n, n) with cost at most k . So consider an arbitrary non-touching path P in B of length ℓ between these two tiles, and let P_i denote the i th tile on the path, for $1 \leq i \leq \ell + 1$. Note that $\ell \geq 2(n - 1)$. Then $\text{cost}(P) = |\{i : P_{i+1} \neq P_i\}|$, or equivalently $\text{cost}(P) = \ell - |\{i : P_{i+1} = P_i\}|$. Define the 0/1-valued random variable X_i by $X_i = 1 \Leftrightarrow P_{i+1} = P_i$. Then, as the colours of tiles are uniformly distributed, $\Pr[X_i = 1] = 1/c$ for all i , and

$$\Pr[\text{cost}(P) \leq k] = \Pr \left[\sum_{i=1}^{\ell} X_i \geq \ell - k \right] \leq e^{-D(1-k/\ell || 1/c)\ell},$$

where we use Fact 10. Thus, using the union bound over all paths of length at least $2(n - 1)$ from $(1, 1)$ to (n, n) , we get that the probability that there exists any path of cost at most k is upper bounded by

$$2 \sum_{\ell=2(n-1)}^{\infty} (\sqrt{7})^\ell e^{-D(1-k/\ell || 1/c)\ell} = 2 \sum_{\ell=2(n-1)}^{\infty} e^{((1/2) \ln 7 - D(1-k/\ell || 1/c))\ell}, \quad (1)$$

¹ Simple paths on square lattices have been intensively studied, and are known as *self-avoiding walks* [7]. There are known upper bounds, which are slightly stronger than Lemma 9, on the number of self-avoiding walks of a given length; however, we avoid these here to keep our presentation elementary.

where we use the estimate for the number of paths which was derived in Lemma 9. In the final part of the proof, we consider the cases $c \geq 4$ and $c = 3$ separately.

First suppose that $c \geq 4$. We take $k = 2(3/10 - 1/c)(n - 1) \leq (3/10 - 1/c)\ell$, as in the statement of the theorem, and use $D(1 - k/\ell || 1/c) \geq 2(1 - k/\ell - 1/c)^2$ (from Fact 10) to obtain the bound

$$2 \sum_{\ell=2(n-1)}^{\infty} e^{((1/2) \ln 7 - 2(1 - k/\ell - 1/c)^2)\ell} \leq 2 \sum_{\ell=2(n-1)}^{\infty} e^{((1/2) \ln 7 - 49/50)\ell}.$$

As $49/50 > (1/2) \ln 7 \approx 0.973$, this sum is exponentially small in n .

Lastly, suppose that $c = 3$. In this case, our choice of k above is negative. Instead we take $k = (n - 1)/22$, which implies $1 - k/\ell \geq 43/44$. In order to obtain a sufficiently tight bound on $D(1 - k/\ell || 1/3)$, we use the explicit formula in Fact 10 to show that $D(43/44 || 1/3) > 0.974 > (1/2) \ln 7$, which implies that there is a bound in Equation (11) which is exponentially small in n . This completes the proof. \square

Acknowledgements

AM was funded by an EPSRC Postdoctoral Research Fellowship. MJ was supported by the EPSRC. We would like to thank Leon Atkins, Aram Harrow, Tom Hinton and Alex Popa for many helpful and encouraging discussions.

References

1. Arthur, D., Clifford, R., Jalsenius, M., Montanaro, A., Sach, B.: The complexity of flood filling games (2010), <http://arxiv.org/abs/1001.4420>
2. Biedl, T.C., Demaine, E.D., Demaine, M.L., Fleischer, R., Jacobsen, L., Munro, J.I.: The complexity of Clickomania. In: More games of no chance. MSRI Publications, vol. 42, pp. 389–404. Cambridge University Press, Cambridge (2002)
3. Demaine, E.D., Hohenberger, S., Liben-Nowell, D.: Tetris is hard, even to approximate. In: Computing and Combinatorics, pp. 351–363 (2003)
4. Iwama, K., Miyano, E., Ono, H.: Drawing Borders Efficiently. *Theory of Computing Systems* 44(2), 230–244 (2009)
5. Jiang, T., Li, M.: On the approximation of shortest common supersequences and longest common subsequences. *SIAM Journal of Computing* 24(5), 1122–1139 (1995)
6. Kaye, R.: Minesweeper is NP-complete. *The Mathematical Intelligencer* 22(2), 9–15 (2000)
7. Madras, N., Slade, G.: *The Self-Avoiding Walk*. Birkhäuser, Basel (1996)
8. Maier, D.: The complexity of some problems on subsequences and supersequences. *Journal of the ACM* 25(2), 322–336 (1978)
9. Munz, P., Hudea, I., Imad, J., Smith, R.J.: When zombies attack!: Mathematical modelling of an outbreak of zombie infection. In: *Infectious Disease Modelling Research Progress*, pp. 133–150. Nova Science, Bombay (2009)
10. Rähä, K.-J., Ukkonen, E.: The shortest common supersequence problem over binary alphabet is NP-complete. *Theoretical Computer Science* 16, 187–198 (1981)
11. Is this game NP-hard? (May 2009), <http://valis.cs.uiuc.edu/blog/?p=2005>

The Computational Complexity of the KAKURO Puzzle, Revisited

Oliver Ruepp¹ and Markus Holzer²

¹ Institut für Informatik, Technische Universität München,
Boltzmannstraße 3, D-85748 Garching bei München, Germany
ruepp@in.tum.de

² Institut für Informatik, Universität Giessen,
Arndtstraße 2, D-35392 Giessen, Germany
holzer@informatik.uni-giessen.de

Abstract. We present a new proof of NP-completeness for the problem of solving instances of the Japanese pencil puzzle KAKURO (also known as Cross-Sum). While the NP-completeness of KAKURO puzzles has been shown before [T. Seto. The complexity of CROSS SUM. *IPSJ SIG Notes*, AL-84:51–58, 2002], there are still two interesting aspects to our proof: we show NP-completeness for a new variant of KAKURO that has not been investigated before and thus improves the aforementioned result. Moreover some parts of the proof have been generated automatically, using an interesting technique involving SAT solvers.

1 Introduction

Pencil puzzles have gained considerable popularity during recent years. The arguably most prominent example is the game of Number Place (jap. *Sudoku*), but there are also many other logic puzzles, which are especially popular in Japan. Here, we are going to take a closer look at the so-called KAKURO puzzle.

KAKURO (also known as Cross-Sum) is a pencil puzzle that could be described as the mathematical version of a crossword puzzle. The difference to the crossword puzzle is that in a KAKURO puzzle, we have to fill numbers into the empty fields instead of letters, and the hints that are used to describe words in a crossword puzzle are also replaced by numbers, namely the sum of the corresponding number sequence.

For a computer scientist, pencil puzzles are especially interesting from the computational complexity point of view. The probably most basic problem is finding a solution for a given puzzle, and in most cases, the corresponding decision problem (“is there a solution?”) turns out to be NP-complete. Here NP denotes the class of problems solvable in polynomial time on a nondeterministic Turing machine. To our knowledge, the first result on pencil puzzles is due to Ueda and Nagao [9], who showed that Nonogram is NP-complete. Since then, a number of other pencil puzzles have been analyzed and also found to be NP-complete, too, e.g., Corral [2], Cross-Sum (jap. *Kakkuro*) [7,8], Fillomino [11], Heyawake [4], Slither Link [11], Sudoku [11], to mention a few.

More formally, a KAKURO puzzle is played on a finite, rectangular grid that contains blank cells and black cells. The objective is to fill numbers into the blank cells, according to the following rules:

1. A sum is associated with every horizontal or vertical sequence of white cells. Only maximal sequences are considered, i.e., sequences that do not have a horizontal/vertical neighboring blank cell at either end.
2. Sequences have a minimum length of 2.
3. Only the numbers 1, 2, . . . , 9 can be placed into the blank cells.
4. Each horizontal respectively vertical sequence has a black cell left of resp. above its first cell, and that black cell contains as hint the sum that is associated with the sequence. Black fields may contain 0, 1 or 2 hints.
5. In each horizontal/vertical sequence of cells, every number may occur at most once.
6. The sum of the numbers of a sequence must equal the number that is denoted in the corresponding hint.

An example of a KAKURO puzzle is shown in Figure 1. The NP-completeness of solving these puzzles has originally been shown in [7,8]. The reduction developed there is rather complicated, and derived from a very special problem. In this work, we will instead show an alternative reduction that is based on the problem of planar 3SAT, which is well-known to be NP-complete [5], and constitutes a slight improvement to the original proof. We are going to prove the following theorem:

Theorem 1. *Deciding whether a KAKURO puzzle has a solution or not is NP-complete, even if the sequence of white cells is at most of length 4.*

We assume the reader to be familiar with the theory of NP-completeness as contained in, e.g., [3].

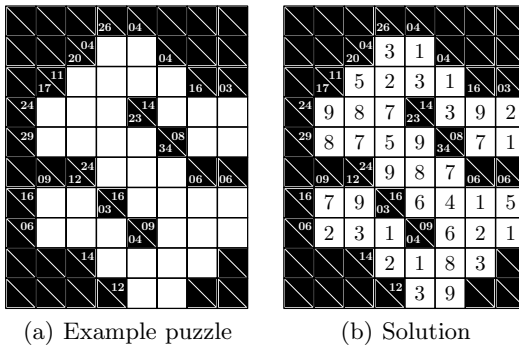


Fig. 1. KAKURO example puzzle with its solution

2 KAKURO Is Intractable

To prove Theorem 1, we have to show that the KAKURO problem is contained in NP, and that it is NP-hard. Containment in NP is immediate, since we can guess an assignment of numbers to empty cells and verify that the result is correct.

To show NP-hardness, we will emulate a planar 3SAT circuit using wires, input nodes, NOT gates, and OR gates, and there will also be some supporting gadgets like the phase shifting gadget, a pattern inverter, and a signal shifter. In the gadget diagrams, we will use a chess-like coordinate system to refer to specific cells. Whenever there is a number that is predetermined because of the KAKURO rules, that number will be preinserted into the diagram.

Figure 2 shows the input node gadget. The actual information generated in the gadget is the position of the 1's and 2's, and all in all, there are only two solutions: the sum 6 in a three cell sequence can only be built by summing up the numbers 1, 2, 3, so we know that the 3 must occur somewhere in the fields $b3, c3, d3$ and somewhere in the fields $c2, c3, c4$. But there can be no 3 at $c2$, because the horizontal sum 3 can only be the result of summing 1 and 2, and the analogous argumentation shows that the 3 is not at $b3$ either. The sum 7 in a sequence of three fields can only be built by summing up the numbers 1, 2, 4, so no 3 is allowed in the corresponding sequences, thus the 3 cannot be placed at $d3$ and $c4$, and the only position that is left for the 3 is then $c3$, as shown. The alternative solution is created by exchanging the 2's and 1's.

The “output” of this gadget is located at the lower right corner, the corresponding *interface fields* are shown as gray shaded boxes. Gadgets will be connected to each other such that the interface fields overlap. All of our gadgets can be rotated arbitrarily in steps of 90 degrees, as well as mirrored diagonally.

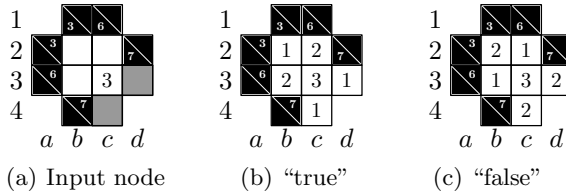


Fig. 2. The input node (a), with its two solutions (b) and (c)

The wire gadget is depicted in Figure 3. It simply transports the alternating pattern of 1's and 2's. Other gadgets will be attached to both ends such that the interface fields overlap, which explains where the number 4, which is apparently neither defined by horizontal nor vertical hints, comes from: we assume that a horizontal respectively vertical sum of value 7 is defined for the corresponding sequences, and all of our gadgets will be designed such that this will be the case. As an example, imagine how this would work out in a combination of an input node gadget and a wire gadget. Information flow is from the top left corner to the lower right corner. Rotated versions of the gadget will obviously work as well.

Figure 4 shows the bending device, which allows us to change the direction of signals by 90 degrees. Note that the output of the gadget is inverse to the input, which is an unwanted side effect, but not really a problem since we can simply attach a negation gadget to restore the original signal. The predefined

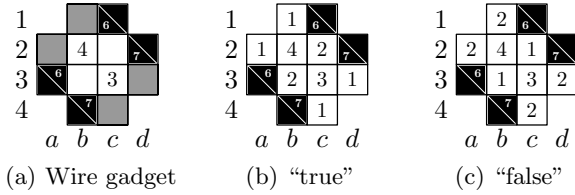


Fig. 3. The wire gadget (a), and its two solutions (b) and (c) corresponding to the two different boolean values, respectively

numbers in the diagram can be explained as follows: the horizontal sum 15 which is defined at $c1$ is the smallest possible sum for 5 cells and can only be the result of summing the numbers from 1 to 5. Thus, the highest possible number in this vertical sequence is a 5. Since the horizontal sum defined at $b4$ is 14, we must place this 5 at $c4$, because otherwise there would be a value larger than 10 in the field at $d4$. This also determines the 9 at $d4$, which means that there can only be the values 1 and 2 at $d3$ and $d5$, which in turn determines the 3 and 4 at $c3$ and $c5$.

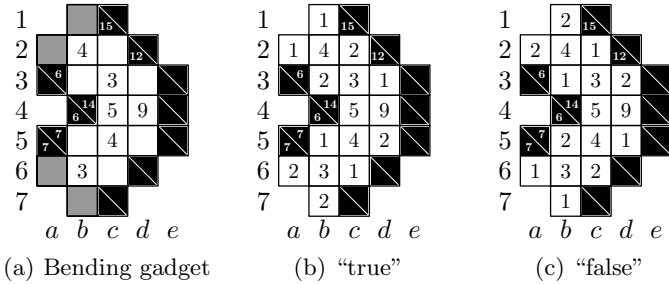


Fig. 4. The bending gadget (a) and its two solutions (b) and (c)

The mentioned negation gadget is shown in Figure 5. There are lots of pre-determined numbers in the diagram, and for a reader without KAKURO experience it might not be clear why this is the case, so we will explain it in detail. The 4's and 3's at $b2$, $c3$ and $h8$ should need no further explaining. The next interesting value is the 7 at $f5$. The reasoning goes as follows: the vertical sum that is defined there is 24, which is the highest value that can be the result of summation over three fields, and thus must be the sum of 9, 8 and 7. But it is not possible to place a 9 or a 8 at $f5$, because then the remaining horizontal sum would be reduced to 4 or 5, and it is not possible to have a sum of such small value in three fields, as the lowest possible sum is $1 + 2 + 3 = 6$. With analogous reasoning, we can determine the numbers at $e6$, $f6$, $g6$ and $f7$. The 3 and 4 at $e5$ and $g7$ are forced because the remaining horizontal and vertical sums are 6 and 7, thus the previous argument about the other predefined 3's and 4's can be applied. Note that even though the values in the interface fields take on the same values, the pattern of 1's and 2's is shifted, which is what we actually wanted to achieve.

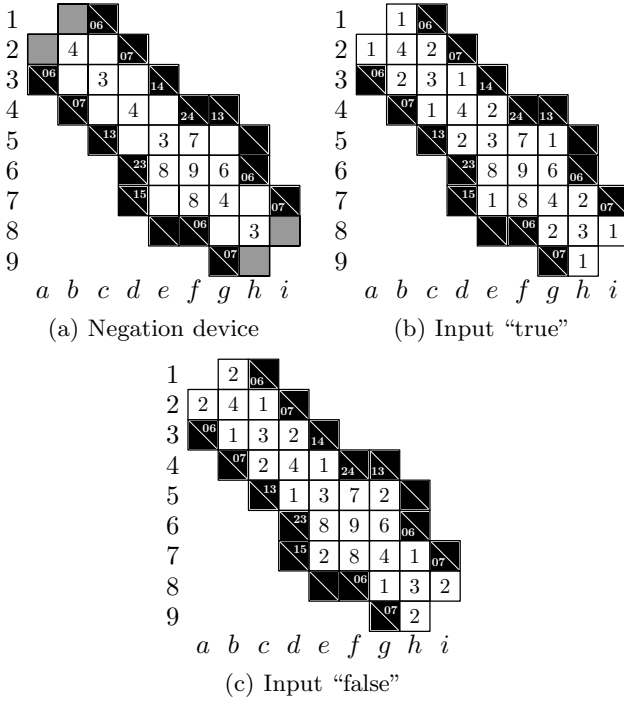


Fig. 5. The negation gadget (a) with its solutions (b) and (c)

There is one problem with the negation gadget: the alternating pattern of 3’s and 4’s that occur in the middle of our gadgets is shifted by 1, and this might be problematic when we try to combine gadgets. What we need here is a phase shifter gadget, and this is shown in Figure 6. The gadget transports the alternating pattern of 1’s and 2’s without modifying it. The 4 at b_2 is enforced because there has to be a 4 at b_1 , b_2 or b_3 (recall that we can assume that a vertical sum of 7 is defined there), but it cannot be at b_2 (a horizontal sum of value $6 = 1 + 2 + 3$ will be defined there) and not at b_4 , so the only possible position is b_3 . The next predefined field is c_3 with a value of 3. That 3 cannot be at d_3 , because then the horizontal sum of 10 of the vertical sequence would be reduced to 7, which is the lowest number that can possibly appear at d_5 because of the horizontal sum $24 = 9 + 8 + 7$ there. But this would mean that we would have to put a 0 at d_5 , which is not allowed. The 7 at d_4 is determined because if we would place a 9 or 8 there, the corresponding vertical sum would be reduced to 2 or 1, and such a small value is obviously not possible as the result of summation in the two remaining fields. The argumentation for the other predefined fields is completely analogous.

One of the most important—and usually also most complicated—gadgets needed in a proof like ours is the splitter gadget, which creates a copy of an input signal. The authors tried quite hard to manually create a puzzle that served this purpose, but did not succeed. However, having a very fast KAKURO

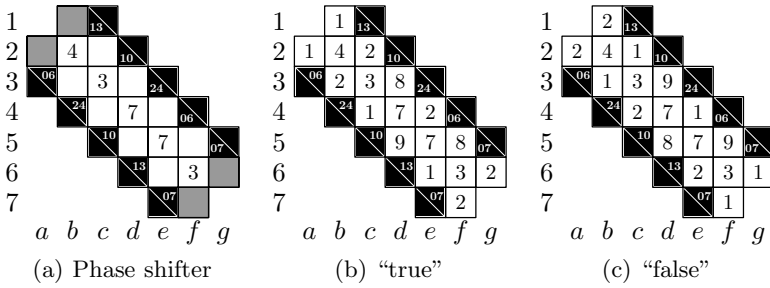


Fig. 6. The phase shifter gadget (a) and the two possible solutions (b) and (c)

solver at disposal (this solver will be described in the latter part of the paper), it was possible to do an exhaustive search for a puzzle that fulfills the desired properties, and this search was indeed successful. The result is shown in Figure 7(a). Note that the output of the puzzle is “inverted,” in that the pattern no longer consist of 1’s and 2’s, but 8’s and 9’s instead. But it is easy to transform this modified pattern back into the standard one, and this will be explained later.

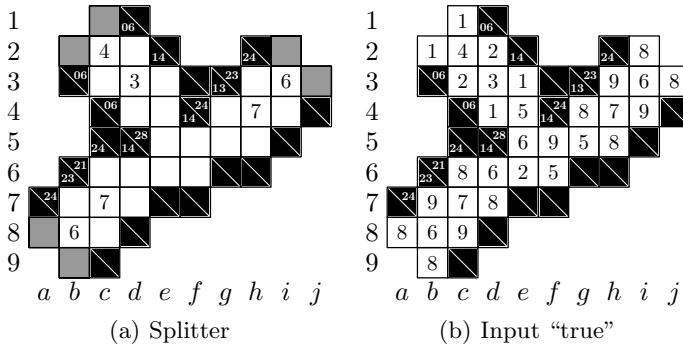


Fig. 7. The splitter gadget (a) duplicates its input. An example is shown in (b).

As usual, we begin by explaining the predetermined numbers in the puzzle. The 4 at $c2$ should be clear, the argumentation is the same as in the previous cases. At $d1$, a vertical sum of value 6 is defined, which means that a 3 must appear either at $d3$ or $d4$. But it cannot be at $d4$, because the vertical sum defined there is 6, which means that another 3 at $e4$ would be enforced, which is not allowed. Analogous argumentation shows that the 7 belonging to the sum 24 that is defined at $a7$ can only be at $c7$. Determining the position for the 7 belonging to the horizontal sum defined at $f4$ is a little bit more complicated: there is no obvious reason why it cannot be at $g4$, so we have to fill out the puzzle until a conflict arises. If we assume the 7 is placed at $g4$, then we would have a 6 at $g5$ and another 7 at $h5$, which already makes a sum of 13, and the remaining value of the horizontal sum there is $28 - 13 = 15$. But 15 can only be the result of $9 + 6$ or $8 + 7$, and both 6 and 7 already occur at $g5$ resp. $h5$, which means that it is not possible to solve the puzzle any more, and so the 7 has to be at $h4$.

By now, we already know a lot about this sub-puzzle: the sum of fields $e3$ and $e4$ will be 6, thus the remaining horizontal sum for fields $e5$ and $e6$ is $14 - 6 = 8$. For the fields $e5$ and $f5$, we have a sum of $28 - 13 = 15$ left, because the numbers at $g5$ and $h5$ will always build the sum 13. With analogous reasoning, we see that the fields $e6$ and $e7$ must contain a sum of $21 - 14 = 7$. All in all, we have a sub-puzzle in the fields $e5, f5, e6, f6$ that corresponds to the puzzle shown in Figure 8.

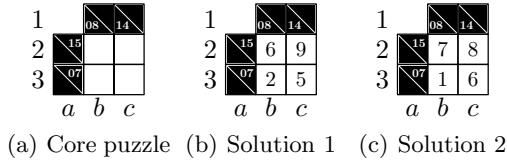


Fig. 8. The center of the splitter gadget (a), and its two solutions (b) and (c)

It is easy to see that this puzzle has exactly two solutions: the vertical sum 14 defined at $c2$ means that the value at $c3$ is at least 5, but the horizontal sum 7 also tells us that the value cannot be bigger than 6. Looking at cell $e3$ in the splitter puzzle, we see that there is either a 1 or a 2, and this obviously determines the solution chosen in the center of the gadget because of Rule 5. This solution in turn determines the numbers chosen at $h5$ and $d6$ (in the original gadget), effectively synchronizing all values. Thus, we have shown that the splitter gadget performs as desired. One example of the full solution is shown in Figure 7(b).

What we still need now is some gadget to transform the 8/9 output pattern into a corresponding 1/2 pattern. In addition to that, we need to shift the pattern vertically or horizontally by one cell, to assure that it can be connected to other gadgets. A gadget that has this effect is shown in Figure 9. The diagram shows a gadget that shifts the signal down vertically by one cell, but by mirroring it on the diagonal we can obtain a gadget that shifts the signal right by one cell. As usual, rotated versions of the device will also work.

The number 7 at $b2$ can be explained with standard reasoning, as well as the 6 at $c3$. The first more difficult number is the 7 at $d4$: because of the horizontal sum of 24 defined there, a 7 has to appear either at $d4$ or $e4$. But it cannot be at $e4$, because then, there would be a 9 or 8 at $d4$, and we already know that there will be a 8 resp. 9 at $d3$, and this means that the sum of fields $d3$ and $d4$ would already be 17, leaving 0 for $d5$, which is of course not allowed. With the 7 fixed at $d4$, we also know that there can only be the values 1 or 2 at $d5$, and this in turn determines the 3 at $e5$. The 7 at $h8$ is determined there because the horizontal sum that is defined there can only be the result of adding 9 and 7, but if the 9 were placed here, the vertical sum would be reduced to 5, which is too small for the three remaining fields. This also forces the 9 at $g8$. The remaining 3's and 4's should need no explicit explanation, since they can all be explained with the standard argumentation.

A puzzle that emulates an OR-gate is shown in Figure 10. Its basic layout is identical to that of the splitter gadget, and it was found using the same

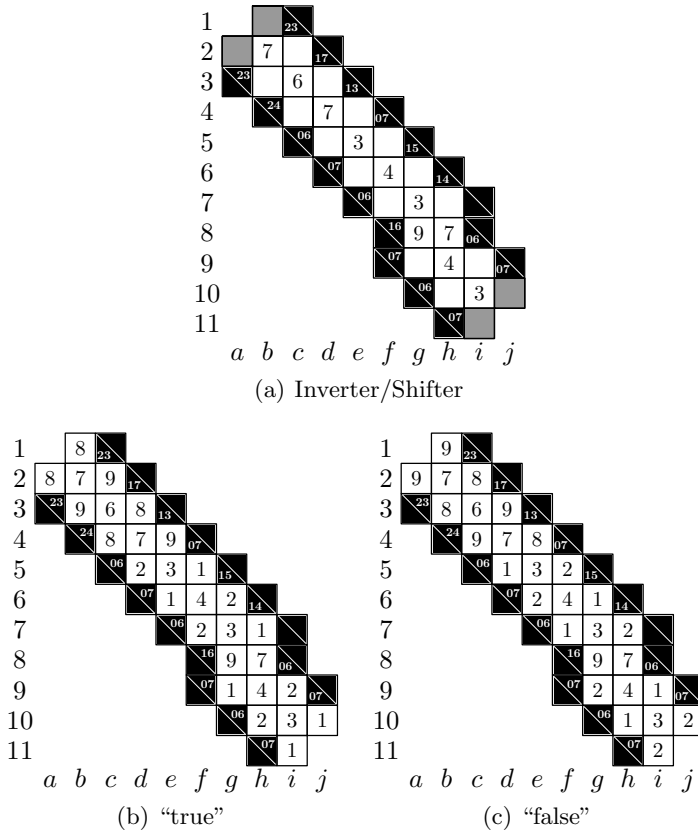


Fig. 9. The inverter/shifter gadget (a) and solutions (b) and (c)

exhaustive search method. The device does not produce an output signal that corresponds to the OR of two input signals, but instead it takes three input signals and enforces that not all of these signals are “false.”

The predefined numbers at *c7* and *h4* should need no further explaining. The placement of the 3 at *d3* is not obvious, it could also be placed at *e3* without a direct resulting conflict. In that case however, a 4 would be enforced at *e4*, and the vertical sum so far is 7, which leaves $12 - 7 = 5$. But 5 can only be partitioned as $5 = 1 + 4$ or $5 = 2 + 3$, and both 3 and 4 already occur in the vertical sequence, so this is where the conflict occurs.

As in the case of the splitter gadget, a sub-puzzle in the fields *e5*, *f5*, *e6*, *f6* emerges, and this sub-puzzle is shown in Figure 11(a). It is easy to derive the three shown solutions to this gadget: there are 4 possibilities to partition the 6 that is defined at *a2*: $1 + 5$, $2 + 4$, $4 + 2$, $5 + 1$. By simply trying all of these, we see that a 1 is impossible at *c2*, which prevents a solution in the last case, while all the other cases lead to a valid solution.

To prove that the gadget works as described we would need to verify that there is no solution if all the input patterns are “false,” and that there is a

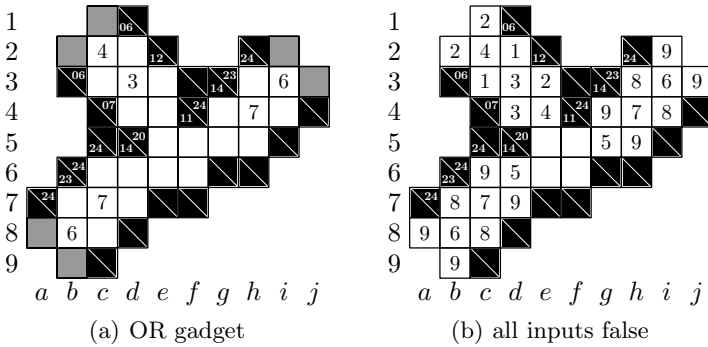


Fig. 10. The OR gadget (a), and the situation when all inputs are false (b)

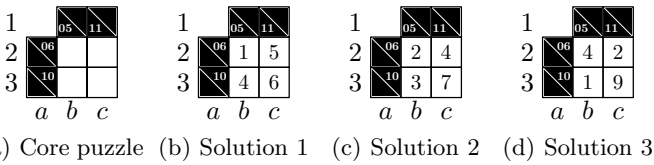


Fig. 11. The core puzzle of the OR gadget (a) has three solutions (b), (c), and (d)

solution in all other cases. This is a cumbersome exercise, so we will show the former property and one example for the latter property.

Now let us assume that all input values are “false,” which means that the number 2 resp. 9 appears in the interface fields. The situation is shown in Figure 10(b). When we fill out the puzzle, there will be a 2 at $e3$, which means that only the solutions 1 and 3 shown in Figure 11 are possible for the core puzzle. But there will also be a 5 at $g5$, which means that solution 1 is also not possible, and solution 3 will also not be possible since there will be a 9 at $c6$.

Next, let us see what happens if we change the upper left input to “true,” in which case a solution should be possible. Then there would be a 1 at $e3$, which means that we have to choose solution 2 for the core puzzle. But with this core puzzle solution, there will be no restrictions on the other two input values, because, e.g., at $g5$ and $h5$ the only possible values are 5 and 9 or 6 and 8, neither of which conflicts with the core puzzle solution. This means that arbitrary input values are allowed at the upper right corner, and with analogous reasoning, we can show that the same holds for the lower left input. Thus, we have already shown that all input combinations where the upper left input is “true” are valid solutions. Showing that all remaining input combinations lead to valid solutions is done completely analogous.

It is clear now that we have developed all the gadgets that are needed to emulate arbitrary planar 3SAT instances, and thus we have proven the first part of Theorem 1. Moreover, the result also constitutes an improvement to the original proof in [7,8], in that the maximum sequence length of white cells needed is 4 instead of 5. This can be achieved by replacing the bending gadget with a

splitter gadget and simply attaching an input node to the superfluous output, effectively discarding the unneeded signal. This finally proves the second part of our main result.

The reduction developed herein is parsimonious, which means that we have also proven a stronger result: Checking a KAKURO puzzle for unique solvability is DP-complete under randomized polynomial time many-one reductions by employing Valiant and Vazirani’s [10] result on unique satisfiability. Here DP denotes the class of differences of any two NP sets [6]—note that DP is equal to the second level of the Boolean Hierarchy over NP.

3 Automatic Gadget Generation

As has been mentioned before, some of the gadgets that have been used for the proof have been generated automatically, using an efficient KAKURO solver. That solver actually translates the KAKURO instance into a SAT instance, and applies a SAT-solver to the result, in our case Minisat [1]. That principle has become a standard approach in recent years, and SAT-solvers as well as constraint-based solvers have been used to tackle many pencil puzzles.

What we have done now to find gadgets that implement certain functionalities is the following: we knew how we would like our signals to be transmitted, i.e., the shape of the wire gadgets was already known. Furthermore, we knew that all of the gadgets we were looking for (splitter, OR) have three wire gadgets connected, so we tried to find a shape that connects three wires and is as small as possible. Figure 12 shows the puzzle shape we came up with.

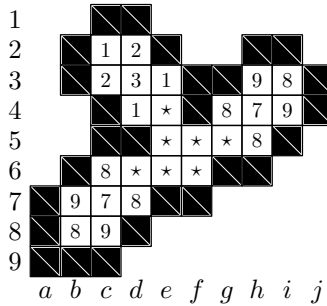


Fig. 12. The search pattern used for the OR gadget

The actual search then worked like this: the fields containing a \star symbol have been used as wildcards, which means that we simply exhaustively generated all possible puzzles by filling in numbers greater or equal to 1 into the fields containing the \star symbol. Note that the input to the puzzle solver consists only of the sums of number sequences, not of the numbers themselves. Thus, even though the search mask shown in Figure 12 looks as though some numbers were fixed, they actually are not, and there might be solutions where the numbers turn out differently.

So after generating all the puzzles by filling in numbers, we ran the KAKURO solver on each of them, and had the solver find alternative solutions. The first step then was filtering out all puzzles that did not have the correct total number of solutions, which is 7 for the OR gate. It is clear that the OR gadget must have exactly 7 solutions, since all possible input value assignments are allowed except for all input values being “false.” After filtering out merely by number of solutions, 2472 potential candidates for the OR gadget were left. For the puzzles that had 7 solutions, we then looked at the values of those solutions at the critical positions $c2$, $i3$ and $b8$, sorting out puzzles that had redundant solutions. Thus, we were able to find not only one, but a handful of puzzles that fulfilled the requirements, and we chose one of those puzzles as our gadget.

Automatically finding gadgets for the splitter device was done analogously. The program source codes used will be made available upon request.

4 Conclusion

In this paper we have developed a new proof showing that the pencil puzzle KAKURO is NP-complete. Though the result was known before, our proof is still interesting, because it is based on a reduction from a better known problem, and also because some parts of the proof have been generated automatically. Furthermore, we were able to show that the KAKURO puzzle remains NP-complete if only sequences of white cells of length at most 4 are used. It is not clear how the complexity of the problem is affected if number sequences have length at most 3, but observe, that Seta has shown in [7] that KAKURO puzzles with sequences that have length at most 2 can be efficiently solved in linear time.

References

1. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
2. Friedman, E.: Corral puzzles are NP-complete. Technical report, Stetson University, DeLand, FL 32723 (2002)
3. Garey, M.R., Johnson, D.S.: Computers and Intractability; A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York (1990)
4. Holzer, M., Ruepp, O.: The troubles of interior design—a complexity analysis of the game heyawake. In: Crescenzi, P., Prencipe, G., Pucci, G. (eds.) FUN 2007. LNCS, vol. 4475, pp. 198–212. Springer, Heidelberg (2007)
5. Lichtenstein, D.: Planar formulae and their uses. *SIAM Journal on Computing* 11(2), 329–343 (1982)
6. Papadimitriou, C.H., Yannakakis, M.: The complexity of facets (and some facets of complexity). *Journal of Computer and System Sciences* 28(2), 244–259 (1984)
7. Seta, T.: The complexities of puzzles, cross sum and their another solution problems (asp). Senior thesis, Univerity of Tokyo, Department of Information Science, Faculty of Science, Hongo 7-3-1, Bunkyo-ku, Tokyo 113, Japan (February 2001)
8. Seta, T.: The complexity of CROSS SUM. *IPJS SIG Notes*, AL-84, 51–58 (2002) (in Japanese)

9. Ueda, N., Nagao, T.: NP-completeness results for NONOGRAM via parsimonious reductions. Technical Report TR96-0008, Department of Computer Science, Tokyo Institut of Technology, Ôokayama 2-12-1 Meguro Tokyo 152-8552, Japan (May 1996)
10. Valiant, L.G., Vazirani, V.V.: NP is as easy as detecting unique solutions. *Theoretical Computer Science* 47(3), 85–93 (1986)
11. Yato, T.: Complexity and completeness of finding another solution and its application to puzzles. Master's thesis, Univerity of Tokyo, Department of Information Science, Faculty of Science, Hongo 7-3-1, Bunkyo-ku, Tokyo 113, Japan (January 2003)

Symmetric Monotone Venn Diagrams with Seven Curves

Tao Cao, Khalegh Mamakani*, and Frank Ruskey**

Dept. of Computer Science, University of Victoria, Canada

Abstract. An n -Venn diagram consists of n curves drawn in the plane in such a way that each of the 2^n possible intersections of the interiors and exteriors of the curves forms a connected non-empty region. A k -region in a diagram is a region that is in the interior of precisely k curves. A n -Venn diagram is *symmetric* if it has a point of rotation about which rotations of the plane by $2\pi/n$ radians leaves the diagram fixed; it is *polar symmetric* if it is symmetric and its stereographic projection about the infinite outer face is isomorphic to the projection about the innermost face. A Venn diagram is *monotone* if every k -region is adjacent to both some $(k-1)$ -region (if $k > 0$) and also to some $k+1$ region (if $k < n$). A Venn diagram is *simple* if at most two curves intersect at any point. We prove that the “Grünbaum” encoding uniquely identifies monotone simple symmetric n -Venn diagrams and describe an algorithm that produces an exhaustive list of all of the monotone simple symmetric n -Venn diagrams. There are exactly 23 simple monotone symmetric 7-Venn diagrams, of which 6 are polar symmetric.

Keywords: Venn diagram, symmetry.

1 Introduction

1.1 Historical Remarks

The familiar three circle Venn diagram is usually drawn with a three-fold rotational symmetry and the question naturally arises as to whether there are other Venn diagrams with rotational symmetry. Grünbaum [5] discovered a rotationally symmetric 5-Venn diagram. Henderson [7] proved that if an n -curve Venn diagram has an n -fold rotational symmetry then n must be prime. Recently, Wagon and Webb [11] cleared up some details of Henderson’s argument. The necessary condition that n be prime was shown to be sufficient by Griggs, Kilian and Savage [4] and an overview of these results was given by Ruskey, Savage, and Wagon [10].

A Venn diagram is *simple* if at most two curves intersect at any point. There is one simple symmetric 3-Venn diagram and one simple symmetric 5-Venn diagram. Edwards wrote a program to exhaustively search for polar symmetric

* Research supported in part by University of Victoria Graduate Fellowship.

** Research supported in part by an NSERC discovery grant.

7-Venn diagrams and he discovered 5 of them, but somehow overlooked a 6-th [3]. His search was in fact restricted to monotone Venn diagrams, which are those that can be drawn with convex curves [1].

A program was written to search for monotone simple symmetric 7-Venn diagrams and 23 of them were reported in the original version of the “Survey of Venn Diagrams” (Ruskey and Weston [9]) from 1997, but no description of the method was ever published and the isomorphism check was unjustified. Later Cao [2] checked those numbers, and provided a proof of the isomorphism check, but again no paper was ever published. In this paper, we justify the isomorphism check and yet again recompute the number of symmetric simple 7-Venn diagrams, using a modified version of the algorithm in [2].

1.2 Definitions

Let $\mathcal{C} = \{C_0, C_1, \dots, C_{n-1}\}$ be a collection of n finitely intersecting simple closed Jordan curves in the plane. The collection \mathcal{C} is said to be an n -Venn diagram if there are exactly 2^n nonempty and connected regions of the form $X_0 \cap X_1 \cap \dots \cap X_{n-1}$ determined by the n curves in \mathcal{C} , where X_i is either the unbounded open exterior or open bounded interior of the curve C_i . Each connected region corresponds to a subset $S \subseteq \{0, 1, \dots, n-1\}$. A region enclosed by exactly k curves is referred as a k -region or a k -set.

A *simple* Venn diagram is one in which exactly two curves cross each other at any point of intersection. In this paper we only consider simple diagrams. A Venn diagram is called *monotone* if every k -region ($0 < k < n$) is adjacent to both a $(k-1)$ -region and a $(k+1)$ -region. It is known that a Venn diagram is monotone if and only if it is isomorphic to some diagram in which all of the curves are convex [1].

A Venn diagram is *rotationally symmetric* (usually shortened to *symmetric*) if there is a fixed point p in the plane such that each curve C_i , for $0 \leq i < n$, is obtained from C_0 by a rotation of $2\pi i/n$ about p . There is also a second type of symmetry for diagrams drawn in the plane. Consider a rotationally symmetric Venn diagram as being projected stereographically onto a sphere with the south pole tangent to the plane at the point of symmetry p . The projection of the diagram back onto the parallel plane tangent to the opposite pole is called a *polar flip*. If the polar flip results in an isomorphic diagram then the diagram is *polar symmetric*. Figure 1 shows a 7-set polar symmetric Venn diagram (this diagram is known as “Victoria” [3]). In conceptualizing polar flips the reader may find it useful to think of the symmetric diagram as being projected on a cylinder, with the region that intersects all of the sets at the bottom of the cylinder and the empty region at the top of the cylinder. Then the polar flip is akin to turning the cylinder upside-down (see Figure 5).

Two Venn diagrams are generally said to be *isomorphic* if one of them can be changed into the other *or its mirror image* by a continuous transformation of the plane. However, when discussing rotationally symmetric diagrams we broaden this definition to allow for *polar flips* as well. Thus the underlying group of potential symmetries has order $4n$.

As was pointed out earlier, if an n -Venn diagram is symmetric then n is prime. Simple symmetric diagrams for $n = 2, 3, 5, 7$ have been found. The main purpose of this paper is to determine the total number of simple monotone symmetric 7-Venn diagrams. A nice poster of the set of resulting diagrams may be obtained at <http://www.cs.uvic.ca/~ruskey/Publications/Venn7/Venn7.html>.

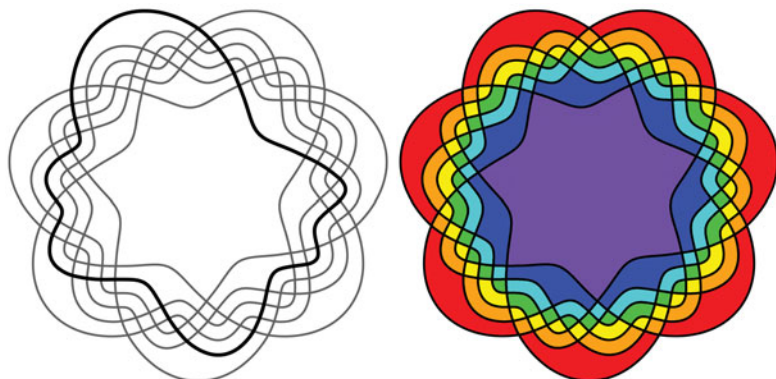


Fig. 1. “Victoria”: a simple monotone polar symmetric 7-Venn diagram

The paper is organized as follows. In Section 2 we outline the classical combinatorial embedding of planar graphs, which is our basic data structure for storing the dual graphs of Venn diagrams. In Section 3 we discuss the representation of Venn diagrams as strings of integers, focussing on those which were used by Grünbaum to manually check whether purported Venn diagrams were Venn diagrams or not, and, if so, whether they were isomorphic.

2 2-Cell Embedding

In this section we outline some of the theory that is necessary for the combinatorial embedding of Venn diagrams in the following sections.

Given a graph G and a surface S , a drawing of G on the surface without edge crossing is called an embedding of G in S . The embedding is 2-cell, if every region of G is homomorphic to an open disk. For a 2-cell embedding of a connected graph with n vertices, m edges and r regions in an orientable surface S_h with h handles we have Euler’s formula $n - m + r = 2 - 2h$.

Let $G = (V, E)$ be a finite connected (multi)graph with $V = \{v_1, v_2, \dots, v_n\}$. For each edge $e \in E$, we denote the oriented edge from v_i to v_j by $(v_i, v_j)_e$ and the opposite direction by $(v_j, v_i)_e$. For each vertex v_i , let E_i be the set of edges oriented from v_i ; i.e., $E_i = \{(v_i, v_j)_e : e \in E \text{ for some } v_j \in V\}$. Let Φ_i be the set of cyclic permutations of E_i . The following theorem proved in [12] shows that there is a one to one correspondence between the set of 2-cell embedding of G and the Cartesian product $\prod \Phi_i$.

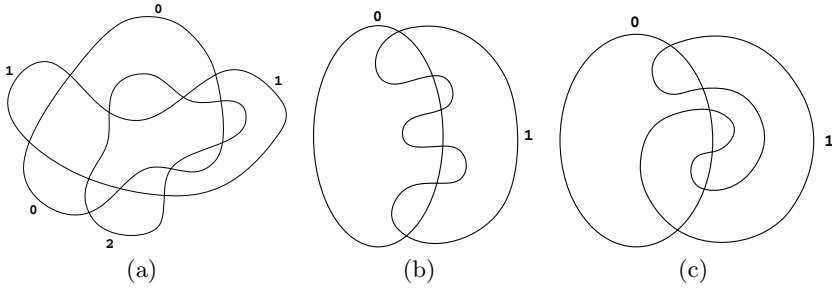


Fig. 2. Examples of normal and non-normal families of intersecting closed curves: (a) is a NFISC, (b) and (c) are not

Theorem 1. Let $G = (V, E)$ be a finite connected (multi)graph. Define E_i and Φ_i as above. Then each choice of permutations $(\phi_1, \phi_2, \dots, \phi_n)$ of $\Phi_1 \times \Phi_2 \times \dots \times \Phi_n$ determines a 2-cell embedding of G in some orientable surface S_h . Conversely, for any 2-cell embedding of G in S_h , there is a corresponding set of permutations that yields that embedding.

3 Representations of Symmetric Monotone Venn Diagrams

3.1 G-Encoding

A family of intersecting simple closed curves (or a FISC) is a collection of simple closed curves enclosing a common non-empty open region and such that every two curves intersect in finitely many points [1].

Definition 1. A normal FISC (or NFISC) is a FISC satisfying the following additional conditions:

- Every curve touches the infinite face,
- The collection is simple, i.e., exactly two curves meet at every point of intersection and they cross each other (each intersection is transverse).
- The collection is convex drawable; i.e., it can be transformed into a FISC with all curves convex by a homeomorphic transformation of the plane.

Let C be an NFISC consisting of n Jordan curves and call the diagram consisted of these n curves an n -diagram. Choosing an arbitrary curve as curve 0, we label all n curves by their clockwise appearance on the outmost region. Let M be the number of times the curves touch the infinity face, $M \geq n$. A G -encoding consists of $M + 1$ sequences and an $M \times n$ matrix F . The first sequence, call it $I = I_0, I_1, \dots, I_{M-1}$ has length M . Starting with curve 0, it specifies the curves encountered as we walk around the outer face of the n -diagram in clockwise direction. Thus, $I_i \in \{0, 1, \dots, n - 1\}$. Each element of I corresponds to a curve segment in the outer face of the diagram. For each curve c , the *first segment* is the one which corresponds to the first appearance of c in I . The other M sequences

(a)	i	I_i	w_i								
	0	0	1	2	2	2	1	2	1	1	
	1	1	2	0	2	0	2	0	2	2	0
	2	2	0	1	1	1	0	0	0	1	
	3	0	1	1	1	2	2	2	1	2	
	4	1	0	2	2	0	2	0	2	0	
			0	1	2	3	4	5	6	7	

(b)		0	1	2
	0	∞	4	5
	1	7	∞	2
	2	4	7	∞
	3	∞	6	7
	4	3	∞	6

Fig. 3. (a) G-encoding of Figure 2(a). (b) The corresponding F -matrix.

are denoted w_0, w_1, \dots, w_{M-1} . Sequence w_i records intersections along curve I_i as a sequence of integers, indicating the curves encountered at intersection points. As usual, the curves are traversed in a clock-wise order.

Among all intersections of the traversal starting at I_i with curve j , let $F[i, j]$ be the index of the first intersection with curve j after curve j touches the outer face for the first time. That is, if p_1, p_2, \dots, p_t are the indices of the intersections with curve j in sequence w_i , the first segment of curve j will eventually hit the outer face, say between intersections at positions p_{s-1} and p_s ; then $F[i, j] = p_s$.

Figure 3 shows the G-encoding of the 3-diagram of Figure 2(a). The first table shows the I and w_i sequences. The second table is the F matrix. Since the curves are not self intersecting, we define $F[i, j] = \infty$ if $j = I_i$. It is worth noting that in general the w_i sequences may have different lengths.

By constructing a circular list of oriented edges for each vertex (point of intersection), it can be shown that there is a correspondence between a 2-cell embedding of an NFISC n -diagram and its G-encoding.

Theorem 2. *Each G-encoding of an NFISC of n Jordan curves uniquely determines a 2-cell embedding of the n -diagram in some sphere S_0 .*

Note that for a non-NFISC, the G-encoding does not necessarily determine a unique diagram. For example the two non-NFISC diagrams (b) and (c) in Figure 2 have the same G-encoding.

3.2 The Grünbaum Encoding

Grünbaum encodings were introduced by Grünbaum as a way of hand-checking whether two Venn diagrams are distinct. However, no proof of correctness of the method was ever published. The Grünbaum encoding of a simple symmetric monotone Venn diagram consists of four n -ary strings, call them w, x, y, z . String w is obtained by first labeling the curves from 0 to $n - 1$ according to their clockwise appearance on the outer face and then following curve 0 in a clockwise direction, starting at a point where it touches the outermost region and meets curve 1, recording its intersections with the other curves, until we reach again the starting point.

String x is obtained by first labeling the curves in the inner face starting at 0 in a clockwise direction and then by following curve 0 in a clockwise direction starting at the intersection with curve 1.

Strings y and z are obtained in a similar way but in a counter-clockwise direction. First, curves are re-labeled counter-clockwise as they appear on the outer face. Then strings y and z are obtained by following curve 0 in a counter-clockwise direction starting from the outermost and innermost regions respectively and recording its intersection with other curves. The Grünbaum encoding of the Venn diagram shown in Figure 1 is given below.

w : 1 4 2 5 3 6 1 6 3 5 3 6 2 5 1 6 1 5 3 6 2 5 1 4 2 6 1 6 2 5 1 4 2 4 1 6
 x : 1 6 3 5 3 6 2 5 1 6 1 5 3 6 2 5 1 4 2 6 1 6 2 5 1 4 2 4 1 6 1 4 2 5 3 6
 y : 1 6 3 5 3 6 2 5 1 6 1 5 3 6 2 5 1 4 2 6 1 6 2 5 1 4 2 4 1 6 1 4 2 5 3 6
 z : 1 4 2 5 3 6 1 6 3 5 3 6 2 5 1 6 1 5 3 6 2 5 1 4 2 6 1 6 2 5 1 4 2 4 1 6

Property 1. Each string of the Grünbaum encoding of a simple symmetric monotone n -Venn diagram has length $(2^{n+1} - 4)/n$.

Proof. Clearly each string will have the same length, call it L . An n -Venn diagram has 2^n regions, and in a simple diagram every face in the dual is a 4-gon. We can therefore use Euler’s relation to conclude that the number of intersections is $2^n - 2$. By rotational symmetry every intersection represented by a number in the encoding corresponds to $n - 1$ other intersections. However, every intersection is represented twice in this manner. Thus $nL = 2(2^n - 2)$, and hence $L = (2^{n+1} - 4)/n$. □

According to the definition of Grünbaum encoding, each string starts with 1 and ends with $n - 1$. Given string w of the Grünbaum encoding, we can compute the other three strings. Let $L = (2^{n+1} - 4)/n$ denote the length of the Grünbaum encoding, and let $w[i]$, $x[i]$, $y[i]$ and $z[i]$ be the i th element of w , x , y and z , respectively, where $0 \leq i \leq L - 1$. Then, clearly,

$$y[i] = n - w[L - i - 1] \quad \text{and} \quad z[i] = n - x[L - i - 1].$$

To obtain x , we first find out the unique location in w where all curves have been encountered an odd number of times (and thus we are now on the inner face), then shift w circularly at this location. The string z can be easily inferred from y in a similar manner.

Three isomorphic Venn diagrams may be obtained from any Venn diagram by “flipping” and/or “polar flipping” mappings. The strings x , y and z are the first strings of the Grünbaum encodings of these isomorphic diagrams. So we can easily verify isomorphisms of any Venn diagram using the Grünbaum encoding. Due to space limitations the proof of the following theorem is omitted.

Theorem 3. *Each Grünbaum encoding determines a unique simple symmetric monotone n -Venn diagram (up to isomorphism).*

Using Grünbaum encoding of a Venn diagram, we can also verify whether it is polar symmetric or not by the following theorem.

Theorem 4. *An n -Venn diagram is polar symmetric if and only if the two string pairs (w, z) and (x, y) of its Grünbaum encoding are identical.*

Proof. For a given Venn diagram D with Grünbaum encoding (w, x, y, z) there are three isomorphic Venn diagram obtained by horizontal, vertical and polar flips with Grünbaum encodings (y, z, w, x) , (x, w, z, y) and (z, y, x, w) respectively. Let D' denotes the Venn diagram obtained by polar flip mapping of D . If D is polar symmetric, then it remains invariant under polar flips So D and D' must have the same Grünbaum encoding, that is, $(w, x, y, z) = (z, y, x, w)$. Therefore, for a polar symmetric Venn diagram we have $w = z$ and $x = y$.

Conversely, suppose we have a Venn diagram D with Grünbaum encoding (w, x, y, z) such that $w = z$ and $x = y$. Then $(w, x, y, z) = (z, y, x, w)$. So the isomorphic Venn diagram D' obtained by polar flip mapping of D , has the same Grünbaum encoding as D . So by Theorem 3 D and D' are equivalent and the diagram is polar symmetric. \square

3.3 The Matrix Representations of Monotone Diagrams

Because of the property of symmetry, an n -Venn symmetric diagram may be partitioned into n identical sectors. Each sector is a pie-slice of the diagram between two rays from the point of symmetry offset by $2\pi/n$ radians from each other. So the representation of one sector is sufficient to generate the whole diagram.

Given a sector of a simple monotone n -Venn diagram, one can map it to a graph consisting of n intersecting polygonal curves (which we call *polylines*), as shown in Figure 4. Putting 0s between these n polylines and 1s at the intersections gives us a 0/1 matrix. We then can expand the matrix by appending identical matrix blocks to generate a matrix that represent the whole Venn diagram.

An n -Venn diagram has exactly 2^n regions. Among them one is most inside (inside all curves) and one is most outside (outside all curves). The rest of $2^n - 2$ regions are evenly distributed in each sector. Hence in each sector there are $(2^n - 2)/n$ regions. We use a 1 to indicate the starting point of a region and the ending point of the adjacent region. This implies that there are exactly $(2^n - 2)/n$ 1's in the matrix. So one can always use a $(n - 1) \times (2^n - 2)/n$ 0/1 matrix to represent one sector and use a $(n - 1) \times (2^n - 2)$ 0/1 matrix to represent the whole diagram.

If a matrix (a_{ij}) , $i = 0, 1, \dots, n - 2, j = 0, 1, \dots, (2^n - 2)/n - 1$, is a representation of a Venn diagram, then any matrix obtained by a shift of some number of columns is also a representation of the same diagram. Therefore we can always shift the representation matrix so that $a_{00} = 1$. The matrix with 1 at the first entry is called the *standard* representation matrix.

The matrix representation of a simple symmetric monotone Venn diagram of n curves has the following properties:

- (a) The total number of 1's in the matrix is $(2^n - 2)/n$, with one 1 in each column.
- (b) There are $\binom{n}{k}/n$ 1s in the k th row, for $k = 1, 2, \dots, n - 1$.
- (c) There are no two adjacent 1's in the matrix.

Note that different 0/1 matrices could represent isomorphic Venn diagrams. How do we know whether a given 0/1 matrix represents a “new” Venn diagram? The Grünbaum encoding provides a convenient way to solve this problem.

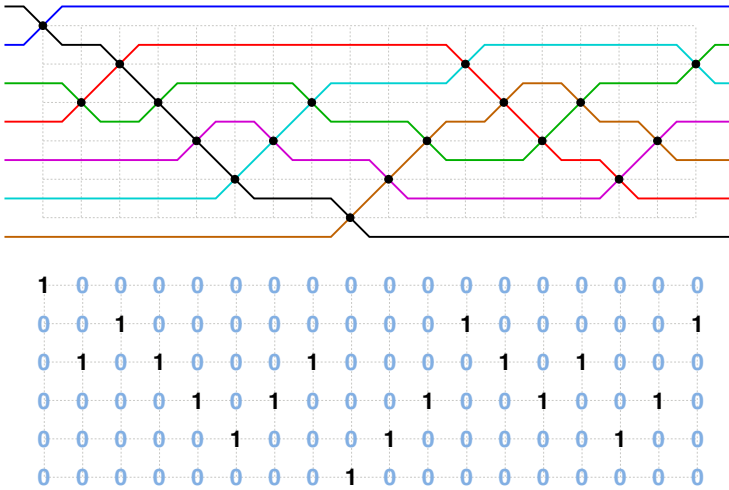


Fig. 4. Matrix representation of Victoria

4 The Algorithm

The algorithm to find all symmetric monotone Venn diagrams consists of the following four steps.

1. **Step one:** Generate all possible standard 0/1 matrices with $n - 1$ rows and $(2^n - 2)/n$ columns that satisfy (a), (b) and (c). To generate each row we are generating restricted combinations; e.g., all bitstrings of length 18 with k 1s, no two of which are adjacent.
2. **Step two:** Check validity. For each matrix V generated in step one, by appending it $n - 1$ times, we first extend the matrix to a matrix X that represents the whole potential Venn diagram. A valid matrix must represent exactly $2^n - 2$ distinct regions of the corresponding Venn diagram. The two other regions are the outermost and the innermost regions. Each region is specified by its rank defined as

$$\text{rank} = 2^0 x_0 + 2^1 x_1 + \dots + 2^{n-1} x_{n-1},$$

where $x_i = 1$ if the curve i is outside of the region and $x_i = 0$ otherwise. In order to check the regions and generate the Grünbaum encoding, an $n \times (2^n - 2)$ matrix C called the P-matrix is generated. The P-matrix gives us another representation of the curves of the Venn diagram (see Table [11](#)). The first column of C is set to $[0, 1, \dots, n - 1]^T$ and for each successive column j , $1 \leq j < 2^n - 2$, we use the same entries of column $j - 1$ and then swap $C_{ij}, C_{(i+1)j}$ if $X_{i(j-1)} = 1$. To check the validity of matrix X , we scan it column by column from left to right. Each 1 indicates the end of one region and start of another region. The entries in the same column of matrix C are used to compute

the rank of the regions. The generated matrix is a valid representation of a Venn diagram if $2^n - 2$ distinct regions are found by scanning the whole matrix, which will only occur if each of the rank calculations are different.

Table 1. The first 18 columns of the P-matrix of Victoria

0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	0	0	3	3	3	3	3	3	3	3	5	5	5	5	5	5	5
2	2	3	0	2	2	2	2	5	5	5	5	3	6	6	2	2	2
3	3	2	2	0	4	4	5	2	2	2	6	6	3	2	6	6	4
4	4	4	4	4	0	5	4	4	4	6	2	2	2	3	3	4	6
5	5	5	5	5	0	0	0	6	4	4	4	4	4	4	3	3	3
6	6	6	6	6	6	6	6	0	0	0	0	0	0	0	0	0	0

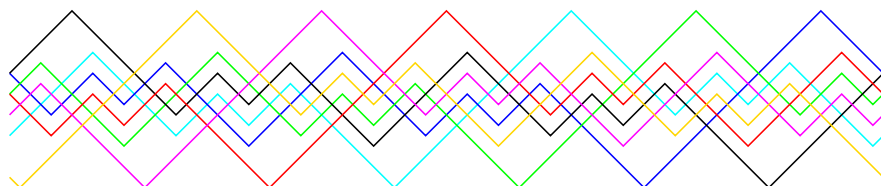


Fig. 5. Cylindrical representation of Victoria

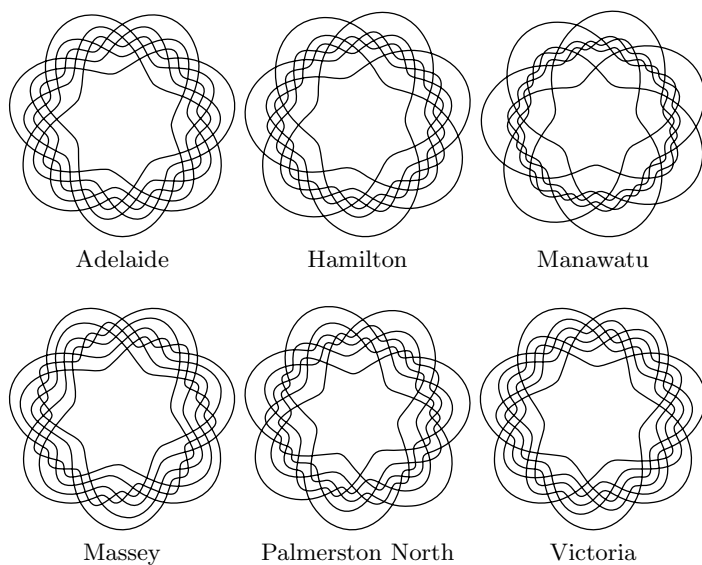


Fig. 6. All simple monotone polar symmetric 7-Venn diagrams

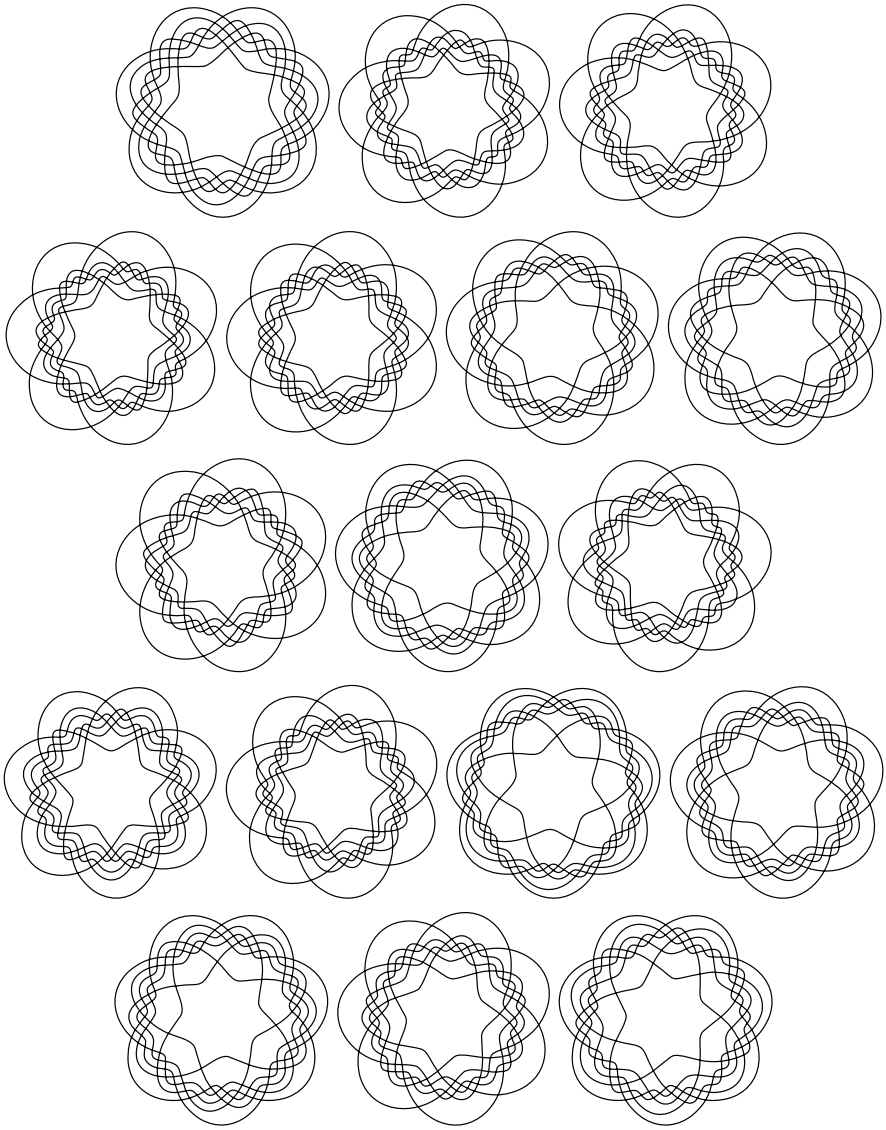


Fig. 7. All 17 simple monotone non-polar symmetric 7-Venn diagrams

3. **Step three:** Generate the Grünbaum encoding. To generate Grünbaum codes, we first relabel the polylines by the order of appearances in the first row so that they are labeled with $0, 1, \dots, n - 1$ (for Table 1 the relabeling permutation is 0124536). Following polyline 0 and recording its intersections with the other polylines, we have the first string w of the Grünbaum encoding. The other three strings, x, y and z , are computed from w .
4. **Step four:** Eliminate isomorphic solutions. By sorting the four strings of the Grünbaum encoding of each produced Venn diagram into lexicographic order and comparing them with the encodings of previously generated Venn diagrams, we eliminate all isomorphic solutions. If the current diagram is not isomorphic to any of previously discovered diagrams, then it will be added to the solution set.

Checking all possible 0/1 matrices for $n = 7$, we found exactly 23 non-isomorphic symmetric monotone Venn diagrams, of which 6 diagrams are polar symmetric. See Figures 6 and 7.

5 Drawing

The polyline diagram in figure 4 shows one sector of the cylindrical projection of Victoria. So given the matrix representation of a Venn diagram, one can easily get its cylindrical projection by computing the cylindrical coordinates of each intersection point. Because of the symmetry, it is sufficient to compute the coordinates only for the first curve. We also need extra points to specify peaks and valleys. To get a visually pleasing shape, we moved the points in such a way that at each point the line segments are perpendicular to each other. Figure 5 shows the resulting representation for Victoria.

The Cartesian coordinates of each point on the plane can be obtained from its cylindrical coordinates. Then we draw the first curve by applying spline interpolation to the computed coordinates. The other six curves are simply drawn by rotating the first curve about the point of symmetry. Figures 7 and 6 show drawings of all 23 diagrams, as constructed by this method.

6 Conclusions and Open Problems

A matrix representation of simple symmetric monotone Venn diagrams has been introduced. We proved that Grünbaum encoding can be used to check the isomorphism and polar symmetry of simple symmetric monotone Venn diagrams. Using an exhaustive search algorithm we verified that there are exactly 23 non-isomorphic simple symmetric monotone 7-Venn diagrams, which 6 of them are polar symmetric. Below is the list of some related open problems: (a) Find the total number of simple symmetric *non-monotone* 7-Venn diagrams. (b) Is there a simple symmetric Venn diagram for $n = 11$?

References

1. Bultena, B., Grünbaum, B., Ruskey, F.: Convex Drawings of Intersecting Families of Simple Closed Curves. In: 11th Canadian Conference on Computational Geometry, pp. 18–21 (1999)
2. Cao, T.: Computing all the Simple Symmetric Monotone Venn Diagrams on Seven Curves. Master's thesis, Dept. of Computer Science, University of Victoria (2001)
3. Edwards, A.W.F.: Seven-set Venn Diagrams with Rotational and Polar Symmetry. *Combinatorics, Probability, and Computing* 7, 149–152 (1998)
4. Griggs, J., Killian, C.E., Savage, C.D.: Venn Diagrams and Symmetric Chain Decompositions in the Boolean Lattice. *Electronic Journal of Combinatorics* 11(1), #R2 (2004)
5. Grünbaum, B.: Venn Diagrams and Independent Families of Sets. *Mathematics Magazine*, 13–23 (January-February 1975)
6. Grünbaum, B.: On Venn Diagrams and the Counting of Regions. *The College Mathematics Journal* 15, 433–435 (1984)
7. Henderson, D.W.: Venn diagrams for more than four classes. *American Mathematical Monthly* 70, 424–426 (1963)
8. Killian, C.E., Ruskey, F., Savage, C., Weston, M.: Half-Simple Symmetric Venn Diagrams. *Electronic Journal of Combinatorics* 11, #R86, 22 (2004)
9. Ruskey, F., Weston, M.: A survey of Venn diagrams. *The Electronic Journal of Combinatorics*, 1997. Dynamic survey, Article DS5 (online). Revised 2001 (2005)
10. Ruskey, F., Savage, C.D., Wagon, S.: The Search for Simple Symmetric Venn Diagrams. *Notices of the American Mathematical Society*, 1304–1311 (December 2006)
11. Wagon, S., Webb, P.: Venn Symmetry and Prime Numbers: A Seductive Proof Revisited. *American Mathematical Monthly* 115, 645–648 (2008)
12. White, A.T., Beineke, L.W.: Topological Graph Theory. In: Beineke, L.W., Wilson, R.J. (eds.) *Selected Topics in Graph Theory*. Academic Press, London (1978)

The Feline Josephus Problem

Frank Ruskey* and Aaron Williams

Dept. of Computer Science, University of Victoria, Canada

Abstract. In the classic Josephus problem, elements $1, 2, \dots, n$ are placed in order around a circle and a skip value k is chosen. The problem proceeds in n rounds, where each round consists of traveling around the circle from the current position, and selecting the k th remaining element to be eliminated from the circle. After n rounds, every element is eliminated. Special attention is given to the last surviving element, denote it by j . We generalize this popular problem by introducing a uniform number of lives ℓ , so that elements are not eliminated until they have been selected for the ℓ th time. We prove two main results: 1) When n and k are fixed, then j is constant for all values of ℓ larger than the n th Fibonacci number. In other words, the last surviving element stabilizes with respect to increasing the number of lives. 2) When n and j are fixed, then there exists a value of k that allows j to be the last survivor simultaneously for all values of ℓ . In other words, certain skip values ensure that a given position is the last survivor, regardless of the number of lives. For the first result we give an algorithm for determining j (and the entire sequence of selections) that uses $O(n^2)$ arithmetic operations.

Keywords: Josephus problem, Fibonacci number, Chinese remainder theorem, Bertrand's postulate, number theory, algorithm.

“un gatto ha sette vite”

1 Introduction

1.1 A Fanciful Scenario

In this subsection we describe some of the history that lead to the classic Josephus problem, and then invent a scenario that might have led to our version of the Josephus problem.

During the first Jewish-Roman war, the military leader Josephus (AD 37 - c. 100) and 40 of his countrymen hid from the Romans in the fallen city of Yodfat. With no hope for escape, the Jewish survivors agree to commit mass suicide rather than be captured and enslaved by the enemy. Josephus does not agree with the others, and instead convinces them to take part in a lethal game of chance:

* Research supported in part by an NSERC discovery grant.

“He whom the lot falls to first, let him be killed by him that hath the second lot, and thus fortune shall make its progress through us all.”
(Book 3, Chapter 8, Section 7 in *The Jewish War* [10].)

Whether by chance or “by the providence of God” Josephus survives this ordeal, and eventually becomes the Roman citizen and Jewish historian known as Titus Flavius Josephus.

By 1539, Girolamo Cardano described the situation as a mathematical puzzle [3]. Instead of drawing lots, the 41 men form a circle and then every 3rd person around the circle is successively selected for elimination. Using this interpretation, Josephus is recast as an erudite scholar who quickly determined that he should stand in the 31st position to avoid elimination.

In general, the classic *Josephus problem* has two parameters: n and k . A circle of n people is formed, and successively every k -th person is selected for elimination. As people are killed off, the circle shrinks, and the goal is to determine last surviving position j . In particular, Josephus solved the first instance of the problem by determining that $j = 31$ when $n = 41$ and $k = 3$. In all versions of the problem, including ours, Josephus knows where the circle begins and has the right to stand in whatever position he wishes.

Now imagine that Josephus’s countrymen agree with his orderly method of elimination, but suspect that Josephus has already determined where to stand. For this reason, they introduce a third parameter ℓ (for lives). Again the men form a circle, but this time they do not die until they have been selected (also called “hit”) for the ℓ -th time. Inspired by the Italian saying that “un gatto ha sette vite” (cats have seven lives)¹, we call this generalization the *Feline Josephus problem*. In the original problem $\ell = 1$.

To further complicate matters, Josephus’s countrymen hide the value of ℓ from their military leader. Undeterred, Josephus agrees to this change of plans, but only under one of the following two conditions: (a) he gets to specify a lower bound on ℓ , or (b) he gives up his right to specify where he stands, in exchange for choosing the value of k . Amazingly enough, Josephus continues to survive, and the main purpose of this paper is to tell the reader how he manages to do this.

In part, our motivation for studying (b) is the second bonus problem in *Concrete Mathematics* [6] which asks the following:

Suppose that Josephus finds himself in a given position j , but he has a chance to name the elimination parameter k such that every k th person is executed. Can he always save himself?

As we will show, the answer is yes, not only for $\ell = 1$, but for any $\ell \geq 1$.

1.2 Notation

The parameters to the feline Josephus problem are n (the number of people), k (the skip factor), and ℓ (the number of “lives”). It is important to note that we

¹ In some cultures the saying is “cats have nine lives.”

make no assumptions about the parameters, other than that they are positive integers. In particular, k or ℓ , or both, could be larger than n .

When the parameters are fixed, as they usually are, we introduce the following three notations. Let $\text{hit}(i)$ be the i th person that is selected (or “hit”). The sequence of all successive hits is called the *hit sequence*; it is

$$\text{hit}(1), \text{hit}(2), \dots, \text{hit}(\ell \cdot n).$$

Let $\text{kill}(i)$ be the i th person that is eliminated (or “killed”). Note that $j = \text{kill}(n) = \text{hit}(\ell \cdot n)$ denotes the last surviving element. The *kill sequence* is

$$\text{kill}(1), \text{kill}(2), \dots, \text{kill}(n).$$

Let $\text{round}(i)$ be the sequence of selections that take place after (but not including) the $(i - 1)$ st person is eliminated, up to (and including) the selection that eliminates the i th person. Thus the hit sequence may be written as

$$\text{round}(1), \text{round}(2), \dots, \text{round}(n).$$

Finally, let $\ell_r(i)$ represent the remaining lives for person i at the end of round r . In particular, $\ell_0(i) = \ell$ for all i , and the last surviving element is the unique value of i such that $\ell_{n-1}(i) > 0$.

Example 1: Let $n = 5, k = 2$, and $\ell = 1$. Then the hit sequence is

$$2, 4, 1, 5, 3,$$

which is the same as the kill sequence. If we now set $\ell = 3$, then we obtain the hit sequence

$$\underbrace{2, 4, 1, 3, 5, 2, 4, 1, 3, 5, 2, 4, 1, 5, 3}_{\text{round}(1)},$$

and the same kill sequence as before. The first round is indicated by the underbrace above; the other four rounds are singletons. This example is illustrated in Figure 1; it starts at the outer 1 and works inwards. In the figure white filled dots

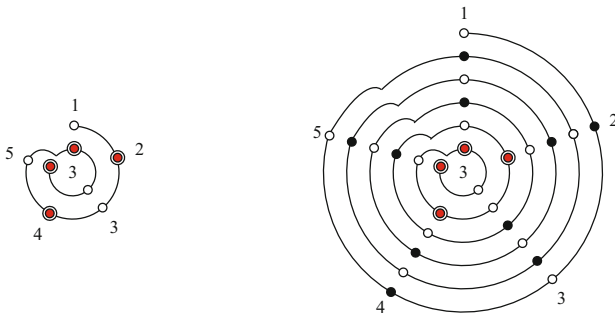


Fig. 1. The Josephus problem with $n = 5, k = 2, \ell = 1$ (left) and $\ell = 3$ (right)

are persons passed over and not hit, filled dots are persons hit, and a partially filled (red in color) dot is a person hit for the last time. The last survivor, person 3, is recorded in the center.

The reason that we did not continue with the circular arcs is that they would have spiralled around several times before hitting 3. In general it is desirable to avoid spiralling around the circle without hitting anybody. In round r this will happen if $k > n - r + 1$. Obviously the useless spirals can be eliminated by using $k \bmod n - r + 1$ in round r instead of k ; to save computation, this will be done later on in an algorithm.

One consequence of the observation in the above paragraph is that we may always reduce k to $k \bmod \text{lcm}\{1, 2, \dots, n\}$ without changing the hit sequence. In the OEIS $\text{lcm}\{1, 2, \dots, n\}$ is sequence A003418 [14]. The prime number theorem implies that $\text{lcm}\{1, 2, \dots, n\} \sim e^{n(1+o(1))}$, so it grows much slower than $n!$.

Example 1 is trivial in a sense, because n and k are relatively prime, (which we denote $n \perp k$, following [6]). For larger values of ℓ in the example above, the hit sequence will be $(2, 4, 1, 3, 5)^{\ell-1}, 2, 4, 1, 5, 3$. More generally, whenever $n \perp k$ the hit sequence will have the form $\pi^{\ell-1}\tau$ where π and τ are permutations of $\{1, 2, \dots, n\}$ and τ is the kill sequence for $\ell = 1$. Thus, the new problem is only interesting when n and k are not relatively prime. In particular, $41 \perp 3$ so the original instance of the Josephus problem would have been no more interesting with cats replacing humans. Next we will consider an example in which $n \not\perp k$.

Example 2: In this example $n = 6$ and $k = 4$. See Figure 2. If $\ell = 1$, then the hit sequence and the kill sequence are both 4, 2, 1, 3, 6, 5. If $\ell = 2$, then the hit sequence is 4, 2, 6, 4, 2, 1, 1, 3, 5, 6, 5, 3, and the kill sequence is 4, 2, 1, 6, 5, 3 which is different from the kill sequence when $\ell = 1$.

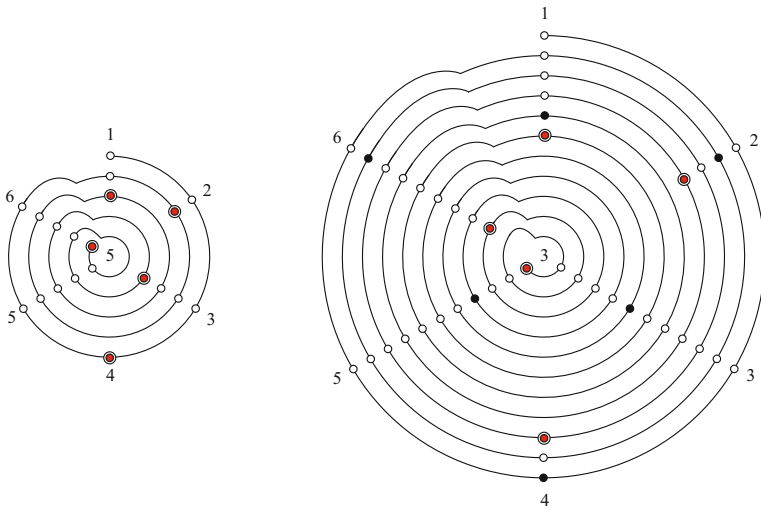


Fig. 2. The Josephus problem with $n = 6$, $k = 4$, $\ell = 1$ (left) and $\ell = 2$ (right)

1.3 History

There are several related research problems for the original Josephus problem: characterizing and computing the last survivor $\text{kill}(n)$ [11], [12], or the i th-last surviving element $\text{kill}(n-i+1)$ [9], [16], [7], [15], studying the combinatorial properties of $\text{kill}(1)\text{kill}(2)\cdots\text{kill}(n)$ as a permutation of $\{1, 2, \dots, n\}$ [13], [4], when considering $\text{kill}(n-r+1), \dots, \text{kill}(n)$ as an r -element subset of $\{1, 2, \dots, n\}$ [6] (especially when $r = n/2$ [5]). The problem plays an interesting role in the history of combinatorics [2], and is used for mathematical recreation [1] and education [8], [6]. Variations have also been examined [15], but the feline version of the problem appears to be new.

2 Josephus Meets Leonardo of Pisa

When n and k have a common factor, then the value of the last survivor is contingent upon the value of ℓ . As shown in Example 2, when $n = 6$ and $k = 4$, the last surviving element is $\text{kill}(n) = 5$ when $\ell = 1$, and changes to $\text{kill}(n) = 3$ when $\ell = 2$.

Example 3: For a more involved example, the values of $\text{kill}(n)$ appears below for $n = 12$, $k = 14642$, and $\ell = 1, 2, \dots, 10$ respectively:

$$\text{kill}(n) = 9, 1, 11, 11, 11, 11, 5, 5, 5, 1, 1.$$

Given this example, it is natural to fix n and k , and to consider the value of $\text{kill}(n)$ as $\ell \rightarrow \infty$. A priori, the ultimate behavior of $\text{kill}(n)$ is not clear: it could continue to fluctuate chaotically, or could settle into a repeating pattern. The main result of this section is that there is a single limiting value for $\text{kill}(n)$. Furthermore, this limiting value holds whenever ℓ exceeds the $(n+2)$ nd Fibonacci number.

To prove the result, we show that the value of $\ell_r(i)$ (the remaining lives for element i at the end of round r) is contained in either an “increasing” set or a “decreasing” set. By ensuring that these two sets do not overlap, it is possible to give an explicit formula for each $\text{round}(r)$ (the hits between the $(r-1)$ st and r th elimination). One consequence of this formula is the value of the last surviving element. Define the *increasing set* and *decreasing set* respectively as

$$\mathcal{I}(r) = \{0, 1, 2, \dots, F_r\} \text{ and } \mathcal{D}(r) = \{\ell, \ell - 1, \dots, \ell - F_{r+1} + 1\},$$

where F_i denotes the i th Fibonacci number. As usual the Fibonacci numbers are defined as, $F_0 = 0$, $F_1 = 1$, and $F_i = F_{i-2} + F_{i-1}$ for all $i \geq 2$. Note that

$$\mathcal{I}(0) \subseteq \mathcal{I}(1) \subseteq \dots \subseteq \mathcal{I}(n) \quad \text{and} \quad \mathcal{D}(0) \subseteq \mathcal{D}(1) \subseteq \dots \subseteq \mathcal{D}(n).$$

Lemma 1. *Let n and k be fixed positive integers. Then for any i satisfying $1 \leq i \leq n$ and any r satisfying $0 \leq r \leq n$,*

$$\ell_r(i) \in \mathcal{I}(r) \cup \mathcal{D}(r). \tag{1}$$

Proof. Our proof is by induction on r . The result is true when $r = 0$ since $\ell_0(i) = \ell$ and $\mathcal{D}(0) = \{\ell\}$. Now assume that (II) holds for all rounds previous to round r . We will use this assumption to prove that (II) holds for round r .

When $\mathcal{I}(r)$ and $\mathcal{D}(r)$ have a value in common then (II) certainly holds since the union of these two sets includes $\{0, 1, \dots, \ell\}$. Therefore, we may assume that the maximum value of $\mathcal{I}(r)$ is strictly less than the minimum value of $\mathcal{D}(r)$. It is also safe to ignore the case when i has been eliminated, since then $\ell_r(i) = 0$ and (II) clearly holds. Similarly, if i does not appear in $\text{round}(r)$ then (II) holds by induction.

In the remaining case, let $y = \text{kill}(r)$. Notice that $\ell_{r-1}(y) \leq \ell_{r-1}(i)$. Furthermore, i must appear in $\text{round}(r)$ either the same number of times as y , or one time fewer than y . That is,

$$\ell_r(i) \in \{\ell_{r-1}(i) - \ell_{r-1}(y), \ell_{r-1}(i) - \ell_{r-1}(y) + 1\}.$$

Inductively,

$$\{\ell_{r-1}(i), \ell_{r-1}(y)\} \subseteq \mathcal{I}(r - 1) \cup \mathcal{D}(r - 1)$$

and so there are four cases depending on which sets contain these two values. However, it is not possible for $\ell_{r-1}(i) \in \mathcal{I}(r - 1)$ and $\ell_{r-1}(y) \in \mathcal{D}(r - 1)$ since $\ell_r(y) \leq \ell_r(i)$ and our assumption that the maximum value of $\mathcal{I}(r)$ is strictly less than the minimum value of $\mathcal{D}(r)$. Furthermore, if $\ell_{r-1}(i) \in \mathcal{I}(r - 1)$ and $\ell_{r-1}(y) \in \mathcal{I}(r - 1)$ then obviously $\ell_r(i) \in \mathcal{I}(r - 1)$ and so (II) follows from $\mathcal{I}(r - 1) \subseteq \mathcal{I}(r)$. This leaves two cases to consider.

If $\ell_{r-1}(i) \in \mathcal{D}(r - 1)$ and $\ell_{r-1}(y) \in \mathcal{I}(r - 1)$ then

$$\begin{aligned} \ell_r(i) &\geq \min(\mathcal{D}(r - 1)) - \max(\mathcal{I}(r - 1)) \\ &= \ell - F_r + 1 - F_{r-1} \\ &= \ell - F_{r+1} + 1 = \min(\mathcal{D}(r)). \end{aligned}$$

On the other hand, if $\ell_{r-1}(i) \in \mathcal{D}(r - 1)$ and $\ell_{r-1}(y) \in \mathcal{D}(r - 1)$ then

$$\begin{aligned} \ell_r(i) &\leq \max(\mathcal{D}(r - 1)) - \min(\mathcal{D}(r - 1)) + 1 \\ &= \ell - (\ell - F_r + 1) + 1 \\ &= F_r = \max(\mathcal{I}(r)). \end{aligned}$$

Therefore, $\ell_r(i) \in \mathcal{I}(r) \cup \mathcal{D}(r)$ as claimed, and so (II) is true by induction. □

Lemma II proved that the remaining number of lives at the end of each round are always contained in the increasing or the decreasing set. The next lemma says that these two sets are disjoint, so long as ℓ is chosen to be large enough.

Lemma 2. *If $\ell \geq F_{n+2}$, then the maximum value in $\mathcal{I}(r)$ is less than the minimum value in $\mathcal{D}(r)$ for all $0 \leq r \leq n$.*

Proof. $\max(\mathcal{I}(r)) = F_r < F_{r+2} - F_{r+1} + 1 \leq \ell - F_{r+1} + 1 = \min(\mathcal{D}(r))$. □

2.1 Algorithmic Implications

Assuming the parameter ℓ is large enough, we will now show that it is possible to represent the hit sequence by a data structure that uses $O(n^2)$ space and that can be constructed in $O(n^2)$ time. For example, let us consider the scenario discussed at the beginning of this section. When $n = 12$, $k = 14642$, and $\ell \geq 9$, we will show below that the hit sequence for the feline Josephus problem is the following sequence of length $n\ell = 12 \cdot \ell$.

$$(2, 4, 6, 8, 10, 12)^{\ell-1}, 2, 3, 4, 6, 3, 1, 12, (3, 7, 9, 11)^{\ell-3}, 3, \\ 10, (1, 7, 9)^2, 1, 7, 9, 1, 8, (1, 11, 5)^2, 1, 11, 5^{\ell-2}, 1^{\ell-8}. \quad (2)$$

Notice that this sequence immediately implies that the last surviving element is 1 whenever $\ell \geq 9$. The remainder of this section describes how sequences of this form can be constructed algorithmically. The strings are guaranteed to represent the hit sequence for all $\ell \geq F_{n+2}$, although the above example illustrates that the string often represents the hit sequence for smaller values of ℓ as well.

Recall that the hit sequence is obtained by concatenating the sequences $\text{round}(r)$ for $r = 1, 2, \dots, n$. We will show that the sequence for $\text{round}(r)$ can be expressed as $\pi^h \tau$ where π is a sequence of unique elements from $\{1, 2, \dots, n\}$, as is τ . Either π or τ may be empty.

Let us first consider the problem of constructing a table of the values of $\ell_r(i)$ for $i = 1, 2, \dots, n$ and $r = 0, 1, \dots, n$. This process is best illustrated by an example, which is the same as the one that started this subsection. The following table shows the values of $n - r + 1 \bmod 14642$.

mod	12	11	10	9	8	7	6	5	4	3	2	1
14642	2	1	2	8	2	5	2	2	2	2	2	0

Table 1 shows $\ell_r(i)$ for our example. The rounds are shown below. We discuss how the table and rounds are obtained in the following paragraph.

$\text{round}(1) = (2, 4, 6, 8, 10, 12)^{\ell-1}, 2$	$\text{round}(2) = 3, 4$
$\text{round}(3) = 6$	$\text{round}(4) = 3, 1, 12$
$\text{round}(5) = (3, 7, 9, 11)^{\ell-3}, 3$	$\text{round}(6) = 10$
$\text{round}(7) = (1, 7, 9)^2, 1, 7$	$\text{round}(8) = 9$
$\text{round}(9) = 1, 8$	$\text{round}(10) = (1, 11, 5)^2, 1, 11$
$\text{round}(11) = 5^{\ell-2}$	$\text{round}(12) = 1^{\ell-8}$

The table is constructed column-by-column. Initially, we set $\ell_0(i) = \ell$. To determine column r from column $r - 1$, one needs to know the following information:

- **kill**($r - 1$) (the element eliminated at the end of the previous round)
- $\ell_{r-1}(i)$ (the remaining lives at the end of the previous round)
- $k \bmod n - r + 1$ (to determine the subset of elements that will be hit).

Table 1. A table of $\ell_r(i)$ when $n = 12$, $k = 14642$, and ℓ is sufficiently large. The dots (\cdot) indicate the value 0.

$\ell_r(i)$	rounds r												
	0	1	2	3	4	5	6	7	8	9	10	11	12
$i = 1$	ℓ	ℓ	ℓ	ℓ	$\ell-1$	$\ell-1$	$\ell-1$	$\ell-4$	$\ell-4$	$\ell-5$	$\ell-8$	$\ell-8$	0
$i = 2$	ℓ	0	\cdot	\cdot	\cdot	\cdot	\cdot	\cdot	\cdot	\cdot	\cdot	\cdot	\cdot
$i = 3$	ℓ	ℓ	$\ell-1$	$\ell-1$	$\ell-2$	0	\cdot	\cdot	\cdot	\cdot	\cdot	\cdot	\cdot
$i = 4$	ℓ	1	0	\cdot	\cdot	\cdot	\cdot	\cdot	\cdot	\cdot	\cdot	\cdot	\cdot
$i = 5$	ℓ	ℓ	ℓ	ℓ	ℓ	ℓ	ℓ	ℓ	ℓ	ℓ	$\ell-2$	0	\cdot
$i = 6$	ℓ	1	1	0	\cdot	\cdot	\cdot	\cdot	\cdot	\cdot	\cdot	\cdot	\cdot
$i = 7$	ℓ	ℓ	ℓ	ℓ	ℓ	3	3	0	\cdot	\cdot	\cdot	\cdot	\cdot
$i = 8$	ℓ	1	1	1	1	1	1	1	1	0	\cdot	\cdot	\cdot
$i = 9$	ℓ	ℓ	ℓ	ℓ	ℓ	3	3	1	0	\cdot	\cdot	\cdot	\cdot
$i = 10$	ℓ	1	1	1	1	1	0	\cdot	\cdot	\cdot	\cdot	\cdot	\cdot
$i = 11$	ℓ	ℓ	ℓ	ℓ	ℓ	3	3	3	3	3	0	\cdot	\cdot
$i = 12$	ℓ	1	1	1	0	\cdot	\cdot	\cdot	\cdot	\cdot	\cdot	\cdot	\cdot

Starting at $\text{kill}(r - 1)$, some subset of survivors will be repeatedly and cyclicly hit until some survivor’s life is reduced to one — the exact subset, call it S , depending on the value of $k \bmod n - r + 1$. That sequence of hits can be written as π^h where π is a permutation of S and $h = -1 + \min\{\ell_{r-1}(i) : i \in S\}$. Thereafter, the hitting continues in the order π until the first person with one life is hit; this person is $\text{kill}(r)$. This last sequence of hittings is some prefix of π , call it τ . The elements of τ form some subset of S , call it T . (When $h = 0$, it is natural to write $\text{round}(r) = \pi^h \tau = \tau$. Similarly, when $\pi = \tau$, it is natural to write $\text{round}(r) = \pi^h \tau = \pi^{h+1}$.) This technique for representing $\text{round}(r)$ as a sequence of the form $\pi^h \tau$ can always be followed, regardless of the value of ℓ .

It should also be clear that we can update the values of $\ell_r(i)$ from the information that we obtained in determining $\text{round}(r) = \pi^h \tau$. First, if $i \notin S$, then $\ell_r(i) = \ell_{r-1}(i)$. If $i \in S$ and $i \in \tau$, then $\ell_r(i) = \ell_r(i) - \ell_{r-1}(\text{kill}(r))$. If $i \in S$ and $i \notin \tau$, then $\ell_r(i) = \ell_r(i) - \ell_{r-1}(\text{kill}(r)) - 1$.

Lemmas [1](#) and [2](#) show that the values of $\ell_r(i)$ can be uniquely expressed as g or $\ell - g$ (where $g \leq F_{n+1}$), regardless of the specific value of $\ell \geq F_{n+2}$. That is why we left ℓ as a variable in Table [1](#). Furthermore, the values of the form g are always less than the values of the form $\ell - g$ whenever $\ell \geq F_{n+2}$. Therefore, τ is uniquely determined by π . The exponent h can also be uniquely expressed as g or $\ell - g$, where $g \leq F_{n+1}$. In particular, the last element of $\text{round}(n)$ is the final surviving element.

Theorem 1. *Given fixed values of n and k , the last surviving element $\text{kill}(n)$ is constant in the feline Josephus problem for all $\ell \geq F_{n+2}$.*

We now argue that the values $\ell_r(i)$ for $i = 1, 2, \dots, n$ can be determined in time $O(n)$. First we put the non-zero values of $\ell_{r-1}(i)$ into an array of size $n - r + 1$. This will allow us to advance by $k \bmod n - r + 1$ in constant time, which in turn will allow us to determine the set S and then the set T in $O(n)$ time. It should be clear that the remaining part of the update is $O(n)$. Thus the entire table

can be computed in $O(n^2)$ operations. We can compute π and τ for each round at the same time, also in $O(n^2)$ operations.

Of course other information can be efficiently recovered from the rounds if they are stored in an appropriate data structure, say an array that indexes into the information for each round. That information would include π , τ , and h . In particular, the kill sequence can be recovered in $O(n)$ time. To repeatedly determine a single value of $\text{hit}(t)$ we would want to keep a running total of the number of elements occurring in previous rounds. These totals would be stored as symbolic expressions of the form $a\ell + b$ for some integers a and b . We could then use binary search to determine the round r in which has the t th hit in time $O(\log n)$. Once we have r , determining $\text{hit}(t)$ takes constant time.

3 Saving Josephus

This section supposes that n is fixed, and that Josephus is assigned to a given position around the circle. In *Concrete Mathematics* [6] it is shown that Josephus can always save himself in this scenario, so long as he is allowed to choose the value of k . We prove that this result can be extended to the feline Josephus problem by constructing a suitable value of k . Furthermore, the constructed value of k does not depend on the specific value of ℓ .

As in [6], our solution relies on basic principles from number theory including the Chinese Remainder Theorem (CRT), and Bertrand’s Postulate. We use $a|b$ when a divides b , and $\text{lcm } \mathbf{S}$ for the least common multiple of a set of integers \mathbf{S} . After the proof, we point out why the solution in [6] does not generalize to the feline Josephus problem.

Theorem 2. *Suppose n and j are fixed and satisfy $1 \leq j \leq n$, and ℓ is an arbitrary positive integer. There exists a value of k such that $\text{kill}(n) = j$.*

Proof. If $j = n$ then $k = 1$ suffices. If $j = 1$ then $k = \text{lcm}\{1, 2, \dots, n\}$ suffices. If $n = 2$ then any even k suffices for $j = 1$, while any odd k suffices for $j = 2$. Therefore, assume $1 < j < n$ and $n > 2$. Bertrand’s postulate implies there is a prime p satisfying $n/2 < p < n$. Let \mathbf{P} represent the non-empty set $\{2, 3, \dots, n\} - \{p\}$. Notice that $p \perp i$ for all $i \in \mathbf{P}$. Therefore, $p \perp \text{lcm } \mathbf{P}$. We now split our construction into two cases depending on the value of j . **Case One:** $n/2 \leq j < n$. This case will be solved by constructing a value of k that results in the following hit sequence

$$(1, 2, \dots, n)^{\ell-1}, 1, 2, \dots, n-p, j+1, j+2, \dots, n, n-p+1, n-p+2, \dots, j. \tag{3}$$

This string is valid for all $\ell \geq 1$ and gives j as the last surviving element. To achieve this hit sequence, we choose k so that it satisfies the following congruences

$$k \equiv 1 \pmod{\text{lcm } \mathbf{P}} \tag{4}$$

$$k \equiv j + 1 - n \pmod{p}. \tag{5}$$

This choice of k is possible by the CRT. The congruence (4) implies the following congruences

$$k \equiv 1 \pmod{i} \text{ for all } i \in \mathbf{P}. \tag{6}$$

Now that the values of k modulo $1, 2, \dots, n$ have been determined, we can verify that (3) is indeed the hit sequence. This will be explained in three steps. First, the prefix $(1, 2, \dots, n)^{\ell-1}, 1, 2, \dots, n-p$ follows immediately from (6). Second, (5) implies that the next element selected and removed is $j+1$. Third, (6) implies that $j+2, j+3, \dots, n, n-p+1, n-p+2, \dots, j$ are the remaining elements selected and removed.

Case Two: $1 < j < n/2$. This case will be solved by constructing a value of k that results in the following hit sequence

$$n^\ell, (n-1)^\ell, \dots, (p+1)^\ell, \pi^{\ell-1}, j-1, j-2, \dots, 1, p, p-1, \dots, j \tag{7}$$

where π is a permutation of $\{1, 2, \dots, p\}$ beginning with $j-1$. This string is valid for all $\ell \geq 1$ and gives j as the last surviving element. To achieve this hit sequence, we choose k so that it satisfies the following congruences

$$k \equiv 0 \pmod{\text{lcm } \mathbf{P}} \tag{8}$$

$$k \equiv j-1 \pmod{p}. \tag{9}$$

This choice of k is possible by the CRT. The congruence (8) implies the following congruences

$$k \equiv 0 \pmod{i} \text{ for all } i \in \mathbf{P}. \tag{10}$$

Now that the values of k modulo $1, 2, \dots, n$ have been determined, we can verify that (7) is indeed the hit sequence. This will be explained in three steps. First, the prefix $n^\ell, (n-1)^\ell, \dots, (p+1)^\ell$ follows immediately from (10). Second, (9) and $j-1 \perp p$ imply that the next round is $\pi^{\ell-1}, j-1$ where π is some permutation of $\{1, 2, \dots, p\}$ that begins with $j-1$. Third, (10) implies that $j-2, j-3, \dots, 1, p, p-1, \dots, j$ are the remaining elements selected and removed. \square

This section concludes with a result that holds for the Josephus problem but not the feline Josephus problem. Consider the following hit sequences

$$3, 6, 4, 2, 5, 1 \qquad 4, 1, 3, 5, 2, 6.$$

The hit sequence on the left is for $n = 6$ and $k = 3$ and $\ell = 1$, while the hit sequence on the right is for $n = 6$ and $k = 58$ and $\ell = 1$. Notice that respective values in these hit sequences sum to $n + 1 = 7$. In other words, $k = 6$ and $k = 58$ provide identical hit sequences except they proceed in opposite clockwise/counter-clockwise directions around the circle. More generally, k and $\text{lcm}\{1, 2, \dots, n\} - k + 1$ provide *directionally-opposite* hit sequences when n is fixed and $\ell = 1$. On the other hand, consider the following hit sequences

$$3, 6, 3, 6, 4, 2, 1, 5, 4, 2, 5, 1 \qquad 4, 2, 6, 4, 1, 5, 2, 5, 6, 3, 3, 1.$$

The hit sequence on the left is for $n = 6$ and $k = 3$ and $\ell = 2$, while the hit sequence on the right is for $n = 6$ and $k = 58$ and $\ell = 2$. Notice that the hit sequences are no longer directionally-opposite, even when considering only the last surviving element. This observation led to the two-case proof of Theorem 2 that was not necessary in 6.

4 Open Problems

This paper has introduced a new variation of the Josephus problem known as the feline Josephus problem. Sections 2 and 3 have demonstrated that there are fun algorithmic and number theoretic problems that can be asked and solved within this generalized setting. This section suggests additional open problems arising from different perspectives, and then closes with an application of the algorithm from Section 2.1.

From an algorithmic perspective, it is natural to ask how efficiently the last surviving element j can be computed for arbitrary values of n , k , and ℓ . Similar questions are also natural for the kill sequence, hit sequence, and $\sigma(n)$ (discussed below).

From a mathematical perspective, it is natural to ask when the value of the last surviving element j can be characterized, either directly or in terms of the original Josephus problem. For example, we have already seen that the value of j does not depend on ℓ whenever n and k are relatively prime. For this reason, it may be useful to next consider situations where n is a prime power that is not relatively prime with k .

From the perspective of a person who wishes to avoid any pain as long as possible, what can be proven about the last person to be hit for the first time? In the original $\ell = 1$ Josephus problem, the last person to be hit is $\text{kill}(n)$. However, this is not necessarily true for $\ell > 1$. For example, the end of hit sequence for $n = 12$, $k = 14642$, and $\ell > 8$ is restated from (2) is $\dots 10, (1, 7, 9)^2, 1, 7, 9, 1, 8, (1, 11, 5)^2, 1, 11, 5^{\ell-2}, 1^{\ell-8}$. Notice that $\text{kill}(n) = 1$, whereas element 5 is the last element to be hit for the first time (in round 10). To provoke further questions, consider the following hit sequence for $n = 6$, $k = 10$, and $\ell \geq 2$:

$$(4\ 2\ 6)^{\ell-1}\ 4,\ 3^\ell,\ 6,\ 1\ 2,\ 1^{\ell-1},\ 5^\ell.$$

Notice that the last surviving element 5 is not hit until the final round. When do situations like this occur, and is there at least one such value of k for every value of n ?

Perhaps the most natural open problem is motivated by running the algorithm outlined in Section 2.1. For example, the table found at <http://webhome.cs.uvic.ca/~ruskey/Publications/Josephus/Table.pdf> contains output for $n = 3, 4, 5, 6$ and all k modulo $\text{lcm}\{1, 2, \dots, n\}$. From this table it is clear that the Fibonacci bound discussed in Section 2 is not tight. For example, in the $n = 6$ column of the table, $\ell - 2$ is the exponent with the lowest negative integer. Therefore, all of the hit sequences for $n = 6$ are valid for $\ell \geq 3$, as opposed to the bound $\ell \geq F_8 = 55$ from Theorem 1. More generally, let $\sigma(n)$ be the

lowest negative integer exponent in a hit sequence for n elements generated by our algorithm. The following values show $\sigma(n)$ in comparison with F_{n+2} for $n = 1, 2, \dots, 14$.

$n =$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\sigma(n)$	1	1	1	1	1	3	3	4	6	6	6	9	9	11	14	15
F_{n+2}	2	3	5	8	13	21	34	55	89	144	233	377	610	987	1597	2584

Notice that for small values of n , the correct bound for $\sigma(n)$ appears to be closer to n than F_{n+2} . Tightening the bound on $\sigma(n)$ promises further number theoretic and algorithmic fun.

References

1. Rouse Ball, W.W., Coxeter, H.S.M.: *Mathematical Recreations and Essays*. Dover Publications, Mineola (1987)
2. Biggs, N.L.: The roots of combinatorics. *Historia Math.* 6(2), 109–136 (1979)
3. Cardano, G.: *Practica Arithmetice et Mensurandi Singularis* (1539)
4. Dowdy, J., Mays, M.E.: Josephus permutations. *J. Combin. Math. Combin. Comput.* 6, 125–130 (1989)
5. Gardner, M.: *Mathematical Puzzles of Sam Loyd*, vol. 2. Dover Publications, Mineola (1960)
6. Graham, R.L., Knuth, D.E., Patashnik, O.: *Concrete Mathematics*. Addison-Wesley, Reading (1994)
7. Halbeisen, L., Hungerbühler, N.: The Josephus problem. *J. Théor. Nuombres Bordeaux.* 9, 303–318 (1997)
8. Herstein, I.N., Kaplansky, I.: *Matters Mathematical*. Harper & Row, New York (1974)
9. Jakobczyk, F.: On the generalized Josephus problem. *Glasgow Mathematical Journal* 14, 168–173 (1973)
10. Josephus, T.F.: *The Jewish War*, p. 75, ISBN 0-14-044420-3
11. Lloyd, E.L.: An $o(n \log m)$ algorithm for the Josephus problem. *J. Algorithms* 4(3), 262–270 (1983)
12. Odlyzko, A.M., Wilf, H.S.: Functional iteration and the Josephus problem. *Glasgow Mathematical Journal* 33(2), 235–240 (1991)
13. Robinson, W.J.: The Josephus problem. *Math. Gaz.* 4, 47–52 (1960)
14. Sloane, N.J.A.: The on-line encyclopedia of integer sequences, <http://www.research.att.com/~njas/sequences/A003418>
15. Thériault, N.: Generalizations of the Josephus problem. *Util. Math.* 58, 161–173 (2000)
16. Woodhouse, D.: The extended Josephus problem. *Rev. Mat. Hisp.-Amer.* 4(33), 207–218 (1973)

Scheduling with Bully Selfish Jobs

Tami Tamir

School of Computer science, The Interdisciplinary Center, Herzliya, Israel
tami@idc.ac.il

Abstract. In job scheduling with precedence constraints, $i \prec j$ means that job j cannot start being processed before job i is completed. In this paper we consider *selfish bully jobs* who do not let other jobs start their processing if they are around. Formally, we define the *selfish precedence-constraint* where $i \prec_s j$ means that j cannot start being processed if i has not started its processing yet. Interestingly, as was detected by a devoted kindergarten teacher whose story is told below, this type of precedence constraints is very different from the traditional one, in a sense that problems that are known to be solvable efficiently become NP-hard and vice-versa.

The work of our hero teacher, Ms. Schedule, was initiated due to an arrival of bully jobs to her kindergarten. Bully jobs bypass all other nice jobs, but respect each other. This natural environment corresponds to the case where the selfish precedence-constraints graph is a complete bipartite graph. Ms. Schedule analyzed the minimum makespan and the minimum total flow-time problems for this setting. She then extended her interest to other topologies of the precedence constraints graph and other special instances with uniform length jobs and/or release times.

1 Bully Jobs Arriving in Town

Graham school is a prestige elementary school for jobs. Jobs from all over the country are coming to spend their childhood in Graham school. Ms. Schedule is the kindergarten teacher and everybody in town admires her for her wonderful work with the little jobs. During recess, the jobs like to play outside in the playground. Ms. Schedule is known for her ability to assign the jobs to the different playground activities in a way that achieves many types of objectives (not all of them are clear to the jobs or to their parents, but this is not the issue of our story).

The jobs enjoy coming to school every morning. In addition to the national curriculum, they spend lot of time learning and practicing the rules Ms. Schedule is teaching them. For example, one of Ms. Schedules's favorite rules is called LPT [13]. They use it when playing on the slides in the playground. At first, each of the n jobs announces how long it takes him to climb up and slide down. Then, by applying the LPT rule they organize themselves quite fast (in time $O(n \log n)$) in a way that enables them to return to class without spending too much time outside. Ms. Schedule once told them that she will never be able to assign them to slides in a way that really minimizes the time they spend in the playground, but promised that this LPT rule provides a good approximation.

For years, everything went well at school. The jobs and their parents were very satisfied with the advanced educational program of Ms. Schedule, and the enrollment waiting list became longer and longer. Until the *bully jobs* came to town and joined the kindergarten.

Being very polite and well-mannered, the veteran jobs prepared a warm welcome party to the bully jobs. Ms. Schedule taught them the different kindergarten rules, and for the first few days no one noticed that the new jobs are different. It was only after a week that Ms. Schedule observed that the new bully jobs were not obeying the rules. Other jobs complained that sometimes, when they were waiting in lines, bully jobs passed them and climbed up the slide even if they were not first in line. One of the nice jobs burst into tears claiming that “I’m a very fast climber, according to SPT rule [21], I need to be first in line, but all these bully jobs are bypassing me”. Indeed, Ms. Schedule herself noticed that the bully jobs were bypassing others. She also noticed that as a result, the whole kindergarten timetable was harmed. The jobs had to spend much more time outside until they had all completed sliding.

Ms. Schedule decided to have a meeting with the bully jobs’ parents. In this meeting, it came clear to her that she will need to make a massive change in the kindergarten rules. The parents justified the inconsiderate behavior of their kids. “Our kids are *selfish*”, they said, “they will never obey your current rules. They will always bypass all the other kids. You should better not try to educate them, just accept them as they are”. Ms. Schedule was very upset to hear it, she was about to tell them that their kids must obey her rules, and otherwise will be suspended from school, but she was a bit afraid of their reaction¹, so she promised them to devise new rules for the kindergarten. The parents were satisfied and concluded: “Remember, bully jobs always bypass those that are in front of them in line. They also move from one line to another. But we, bullies, respect each other! bully jobs will not pass other bully jobs that were assigned before them in line”.

Ms. Schedule came back home tired and concerned, feeling she must design new rules for her kindergarten, taking into consideration what she have just learnt about the bully jobs.

2 Ms. Schedule Defining Her Goals

Ms. Schedule relaxed with a cup of good green tea. She decided that the first thing she needed is a formal definition of her new model. “In my setting”, she thought, “there is a set J of jobs, and a set M of m identical machines (slides). Each job is associated with a length (sliding time) p_j . Some of the jobs are *bully* and the other are *nice*. I will denote these sets B and N respectively, $B \cup N = J$. My rules assign jobs to slides, and determine the internal order of the jobs on each slide. The bully jobs, however, do not obey my assignment. Specifically, if a bully job can reduce its waiting time by passing other jobs in line or by moving to another line it will do so. On the other hand, bully jobs respect each other. If I assign them in some order

¹ Bully jobs tend to have bully parents.

to some line then their internal order will be kept. Moreover, if a bully moves to a different line, he will be last among the bullies who are already in line. Each of my assignment methods produces a schedule s of jobs on the slides, where $s(j) \in M$ denotes the slide job j is assigned to. The completion time, or flow-time of job j , denoted C_j , is the time when job j completes its processing. The load on a slide M_i in an assignment s is the sum of the sliding times of the jobs assigned to M_i , that is $\sum_{j|s(j)=M_i} p_j$. The *makespan* of a schedule, denoted C_{max} , is the load on the most loaded slide; clearly, this is also the maximal completion time of a job.”

2.1 Scheduling with Selfish Precedence-Constraints

Ms. Schedule thought that her problem, in some sense, is similar to the problem of scheduling with precedence constraints. In scheduling with precedence constraints, the constraints are given by a partial order precedence relation \prec such that $i \prec j$ implies that j cannot start being processed before i has been completed. Selfish-precedence is different. It is given by a partial order precedence relation \prec_s such that $i \prec_s j$ implies that j cannot start being processed before i is starting.

“I believe” she thought “that selfish precedence-constraints induces interesting problems that should be studied, especially in these days when it is very popular to deal with algorithmic game theory and selfish agents. A selfish job only cares about his delay and his completion time, it is OK with him that others are also doing well, but he is ready to hurt others in order to promote himself. This is exactly reflected by the fact that if $i \prec_s j$, then job i doesn’t mind if job j is processed in parallel with him, as long as it doesn’t start being processed before him”. Ms. Schedule decided to devote some of her valuable time to consider this new type of selfish precedence-constraints. “I’m not aware of any early work on this interesting setting”, she mentioned to herself.

“For a single machine, I don’t expect any interesting results.” Ms. Schedule figured out, “It is easy to see that with a single machine, a schedule is feasible under the constraints \prec if and only if it is feasible under the constraints \prec_s . Therefore all the results I see in my favorite web-site [1], carry over to selfish precedence constraints.”

“In fact, there is this issue of release times, which makes the precedence constraints different, already with a single machine” Ms. Schedule kept pondering “since bully jobs only care about *their* delay, they let other jobs be processed as long as they are not around (before their release time). It is only when they show up that they bypass others. Upon being released they push a following job away from the slide even if they already started climbing”. Ms. Schedule decided to elaborate on that issue of release times later (see Section 5), and to set as a primal goal the analysis of the basic problems of minimum makespan (Section 3) and minimum total flow-time (Section 4) for the precedence constraint setting she has in her kindergarten. In this paper we tell her story and reveal her results. Ms. Schedule kindly provided us with her complete work, however, due to space constraints, some of the proofs are missing and can be found in [22].

2.2 Complete-Bipartite Selfish Precedence-Constraints

“What I actually face”, Ms. Schedule kept thinking, “is the problem in which the selfish precedence-constraints graph is a *complete bipartite* $K_{b,n}$, where $b = |B|, n = |N|$, and $i \prec_s j$ for every $i \in B$ and $j \in N$.

As a first step, I would like to evaluate the potential loss from having bully jobs in my kindergarten. Similar to other equilibria notions, a schedule is a *Bully equilibrium* if no bully job can reduce its completion time by migrating to another machine or bypassing nice jobs. Indeed, since bully jobs bypass all nice jobs in their line and can also migrate from one machine to another, a schedule respects the $K_{b,n}$ constraints if and only if no nice job starts before a bully job. In fact, a schedule may respect the $K_{b,n}$ constraints, but not be a bully-equilibrium. This can happen if for example some machine is empty while another machine is loaded with more than a single bully job – in this case the precedence constraints hold but bullies will migrate to the empty machine. In general, if the gap between the completion times of the last bullies on different machines is larger than the length of the last bully on these machines, then this last bully will migrate. However, it is easy to see that for every reasonable objective function, in particular, all objective functions I’m about to consider, having a small gap between the completion times of the last bullies on different machines is a dominant property. Therefore, w.l.o.g., I would assume the following equivalence:”

Property 1. A schedule is a bully equilibrium if and only if it obeys the $K_{b,n}$ selfish precedence-constraints.

The Price of Bullying: Let S denote the set of all schedules, not necessarily obeying the selfish precedence-constraints. For a schedule $s \in S$, let $g(s)$ be some measure of s . For example, $g(s) = \max_j C_j(s)$ is the makespan measure, and $g(s) = \sum_j C_j$ is the total flow time measure.

Definition 1. Let $\Phi(I)$ be the set of Bully equilibria of an instance I . The price of bullying (PoB) for a measure $g(\cdot)$ is the ratio between the best bully equilibrium and an optimal solution. Formally, $PoB = \min_{s \in \Phi(I)} g(s) / \min_{s \in S} g(s)$.

Theorem 1. For the objective of minimizing the makespan, the price of bullying is $2 - \frac{1}{m}$.

Theorem 2. For the objective of minimizing the total flow-time, the price of bullying is $(n + b)/m$.

3 Makespan Minimization: $P|K_{b,n}, s\text{-}prec|C_{max}$

Ms. Schedule’s first goal was to minimize the recess length. She wanted all jobs to have a chance to slide once. She knew that the problem $P||C_{max}$ is strongly NP-hard, therefore, the best she could expect is a PTAS. With regular precedence constraints, an optimal solution for $P|K_{b,n}, prec|C_{max}$ consists of a concatenation of optimal solutions for each type of jobs, but with selfish precedence-constraints, this approach might not lead even to a good approximation. To

1/m	1/m	1/m	1/m	1
1				
1				
1				

$$C_{\max}(\text{double-OPT}) = 2$$

1/m	1
1/m	1
1/m	1
1/m	1

$$C_{\max}(\text{OPT}) = (m+1)/m$$

Fig. 1. The approximation ratio of double-OPT and double-LPT is at least $2 - \frac{2}{m+1}$

clarify this point better, Ms. Schedule drew Figure 1, and pointed out to herself that gluing independent optimal solutions to each type of jobs (denote this method double-OPT) can be far by a factor of at least $2 - \frac{2}{m+1}$ from an optimal solution. The main reason for this relative poor performance of double-OPT is that in an optimal schedule, the bully jobs might better be assigned in a non-balanced way. Ms. Schedule decided to consider and analyze known heuristics and also to develop a PTAS for a constant number of machines.

3.1 Adjusted List-Scheduling

Let the jobs stand in a single line, bully jobs first in arbitrary order followed by the nice job in arbitrary order. Then assign them to slides greedily according to this order. Each job goes to the first available slide. The starting times of the jobs form a non-decreasing sequence; in particular, every bully job is starting not later than every nice job. Therefore, the resulting schedule is feasible. Moreover, it is a possible execution of the known List Scheduling algorithm [2], therefore it produces a $(2 - \frac{1}{m})$ -approximation to the makespan, even compared to the makespan with no selfish precedence-constraints.

3.2 Double-LPT

Let the jobs stand in a single line, bully jobs first in non-increasing sliding-time order, followed by the nice job in non-increasing sliding-time order. Then assign them to slides greedily according to this order. Each job goes to the first available slide. As this is a possible execution of the adjusted list-scheduling algorithm, the resulting assignment is feasible. However, as was detected by Ms. Schedule when considering double-OPT assignments, the actual approximation ratio of this heuristic is not much better than the $(2 - \frac{1}{m})$ -guaranteed by list-scheduling, and does not resemble the known bounds of LPT (of $(\frac{4}{3} - \frac{1}{3m})$ [13] or $(1 + \frac{1}{k})$ where k is the number of jobs on the most loaded machine [5]). Note that for the instance considered in Figure 1, the double-OPT schedule is achieved also by double-LPT. As Ms. Schedule was able to show, this ratio of $2 - \frac{2}{m+1}$ is the worst possible for double-LPT. Formally,

Theorem 3. *The approximation ratio of double-LPT for $P|K_{b,n}, s\text{-prec}|C_{\max}$ is $2 - \frac{2}{m+1}$.*

3.3 A PTAS for $P_m|K_{b,n}, s\text{-prec}|C_{max}$

Ms. Schedule was familiar with several PTASs for the minimum makespan problem [15,10]. She was even working on implementing one with the little jobs in their Drama class, hoping to have a nice show for the end-of-year party. However, knowing that double-OPT may be far from being optimal, she understood that a similar double-PTAS approach will not lead to approximation ratio better than $2 - \frac{2}{m+1}$, independent of ε . “I must develop a new PTAS, in which the assignment of bully and nice jobs is coordinated”. She thought.

Ms. Schedule was able to solve the problem for a constant number of machines. She used the idea of grouping small jobs, as introduced in the PTAS of Sahni for $P_m||C_{max}$ [20]. Given an instance $I = B \cup N$, let P denote the total processing time of all jobs, and let p_{max} denote the longest processing time of a job (bully or nice). Let $C = \max(P/m, p_{max})$.

“The first step of my scheme is to modify the instance I into a simplified instance I' . Given I, ε , partition the jobs into small jobs – of length at most εC , and big jobs – of length larger than εC . Let P_S^B, P_S^N denote the total length of small bully and small nice jobs, respectively. The modified instance I' consists of all big jobs in I together with $\lfloor P_S^B / (\varepsilon C) \rfloor$ bully jobs and $\lfloor P_S^N / (\varepsilon C) \rfloor$ nice jobs of length εC . The second step is to solve optimally the problem for I' , while the third step is to transform it back into a schedule of I .” thought Ms. Schedule. “This final stage is the one in which my PTAS and its analysis are different from Sahni’s.”

Theorem 4. *There exists a PTAS for $P_m|K_{b,n}, s\text{-prec}|C_{max}$.*

4 Minimizing Total Flow-Time: $P|K_{b,n}, s\text{-prec}|\sum C_j$

Before the bully jobs arrived, one of Ms. Schedule favorite rules was SPT [21]. She used it when she wanted to minimize the total flow-time of the jobs. Ms. Schedule kept in her cupboard a collection of dolls that she called *dummy jobs* and used them from time to time in her calculations. Whenever Ms. Schedule wanted the jobs to use SPT rule, she first added to the gang some of her dummy jobs, so that the total number of (real and dummy) jobs divided m . The dummy jobs did not require any time in the slides (i.e., their sliding time was 0) so it was never clear to the little jobs why the dummies were needed, but Ms. Schedule explained them that it helps her in her calculations. When applying SPT rule, the jobs sorted themselves by their sliding time, and were assigned to *heats*. The m fastest jobs formed the 1st heat, the m next jobs formed the 2nd heat and so on. The internal assignment of jobs from the same heat to slides didn’t matter to Ms. Schedule.

After the bully jobs arrived it was clear to everyone that these jobs must be assigned to early heats, even if they were slow. For a single slide, it was not difficult to find a schedule achieving minimum total flow-time.

Theorem 5. *The problem $1|K_{b,n}, s\text{-prec}|\sum C_j$ is polynomially solvable.*

On the other hand, for more than one slide. Ms. Schedule couldn't come up with an efficient assignment rule. She told the principal that this was one of the problems she will never be able to solve. The principal couldn't accept it. "I know you are having a difficult quarter with these bullies, but you should try harder. I suggest to simply extend the assignment rule for a single slide", he told Ms. Schedule, "Just let the bullies arrange themselves by SPT rule, and then let the nice jobs join them greedily - also according to SPT order. Let me show you the following for instances with a *single* nice job. As you will see, these selfish precedence-constraints are totally different from the regular ones".

Theorem 6. *$P2|K_{b,1}, prec|\sum C_j$ is NP-hard, but $P|K_{b,1}, s\text{-prec}|\sum C_j$ is solvable in polynomial time.*

Proof. "For the hardness of $P2|K_{b,1}, prec|\sum C_j$ " said the principal, "I will use a reduction from the bi-criteria problem $P2||F_h(C_{max}/\sum C_j)$. In this problem, it is desired to minimize the total flow-time as the primary objective and minimize the makespan as the secondary objective. This problem is known to be NP-hard [2]. Since the single nice job can only start its execution after all preceding jobs complete their execution, it is easy to see that this bi-criteria problem can be reduced to our problem."

"And now, let me show you my optimal algorithm for $P|K_{b,1}, s\text{-prec}|\sum C_j$ ", the principal continued. "First, using your dummy jobs, we can assume that the number of bully jobs is zm for some integer z . Next, sort the bully jobs from shortest to longest and assign them to the machines greedily - using SPT rule. The jobs $(k-1)m+1, \dots, km$ form the k -th heat. Each machine is getting in turn one job from each heat. In particular, and this is the important part, the shortest job from each heat goes to M_1 . Finally, assign the nice job to M_1 . To complete the proof I will show you the following claim", the principal concluded.

Claim. The resulting assignment is feasible and achieves minimum total flow-time.

"Now that we have a proof for a single nice job" said the principal, "we only need to extend it by induction for any number of nice jobs". Ms. Schedule was not impressed. She drew Figure 2 on the whiteboard in the principal's office and said: "For more than a single nice job, your algorithm is not optimal". The principal looked at her doubtfully, but she continued, "as you can see, the total flow-time of the bully jobs is not necessarily minimal in an optimal schedule. Interestingly, while for a single nice job there is a distinction between regular and selfish precedence-constraints, for many nice jobs, the problem is NP-hard in both settings."

Theorem 7. *The problem $P2|K_{b,n}, s\text{-prec}|\sum C_j$ is NP-hard.*

Proof. Ms. Schedule used a reduction from the problem $P2||F_h(C_{max}/\sum C_j)$. "As you claimed five minutes ago", she told the principal, "this bi-criteria problem is known to be NP-hard [2]. It remains NP-hard if the number of jobs and their total length are even integers. Consider an instance \mathcal{I} of $2k$ jobs having

1	1	4		4			
2	2	2	1	1	1	1	

(a)

1	2	2	4		1
1	2	4		1	1

(b)

Fig. 2. (a) An optimal schedule. The bully jobs (in grey) are not scheduled according to SPT, $\sum C_j = 65$. (b) The best schedule under the constraint that bully jobs obey SPT, $\sum C_j = 66$.

lengthes $a_1 \leq a_2 \leq \dots \leq a_{2k}$, such that $\sum_i a_i = 2B$ ". "For this instance", the principal broke in, "the minimum total flow time is

$$T = k(a_1 + a_2) + (k - 1)(a_3 + a_4) + \dots + (a_{2k-1} + a_{2k}),$$

and it is obtained by SPT". "You are right", Ms. Schedule nodded. "However, there are many ways to achieve this value. The hardness of $P2||F_h(C_{max}/\sum C_j)$ tells us that it is NP-hard to decide if among these optimal schedules there is a one with makespan B ."

"Given \mathcal{I} , I will build the following input for $P2|K_{b,n}, s\text{-prec}|\sum C_j$: There are $2k + 1$ bully jobs whose lengthes are a_1, \dots, a_{2k} and B . In addition, there are $n = B$ nice jobs of length 1". "It is easy to solve the problem for the bully jobs only", said the principal, "since you have an odd number of jobs, add one of your dummy jobs of length 0, and apply SPT rule. I can tell you that the resulting total flow-time will be

$$T' = (k + 1)a_1 + k(a_2 + a_3) + (k - 1)(a_4 + a_5) + \dots + (a_{2k} + B)."$$

"Once again, you are right", said Ms. Schedule, "but the main issue here is that the optimal solution for the whole instance is not necessarily achieved by minimizing the total flow-time of the bully jobs. The more important question is whether there is an assignment of the bullies with a particular gap in the loads between the machines. The following claim completes my hardness proof.

Claim. The instance \mathcal{I} has a schedule with total flow-time T and makespan B if and only if the solution to $P2|K_{b,n}, s\text{-prec}|\sum C_j$ has value $T + \frac{3}{2}B^2 + \frac{5}{2}B$.

5 Selfish Precedence-Constraints with Release Times

One significant difference between regular and selfish precedence-constraints is the influence of *release times*. If a job i is not around yet, other jobs can start their processing, even if i precedes them. However, if i is released while a job j such that $i \prec_s j$ is processed, then i pushes j a way and starts being processed right away (assuming that no job who precedes i was also released). Job j will have to restart its processing on some other time (independent of the partial processing it already experienced). This affect of release times is relevant for any precedence-constraints topology, not only for complete bipartite graphs.

Example: Let $J = \{J_1, J_2, J_3\}, p_1 = p_2 = 2, p_3 = 1, r_1 = r_2 = 0, r_3 = 3$, and $J_3 \prec_s J_1, J_3 \prec_s J_2$. Then it is possible to process J_1 in time $[0, 2]$. Indeed

$J_3 \prec_s J_1$, but J_3 is not around yet along the whole processing. J_2 may start its processing at time 2, but will be pushed away by J_3 upon its release at time 3. J_3 will be processed in time $[3, 4]$, and J_2 will be processed in time $[4, 6]$. Two processing units are required for J_2 even-though it was allocated one already.

Ms. Schedule noticed that when recess begins, the nice jobs were always out in the playground on time, while the bully jobs tended to arrive late to the playground². She therefore decided to consider the case in which for every nice job $j \in N, r_j = 0$, while bully jobs have arbitrary release times. She denoted this type of instance by $r_j(B)$. Recall that upon an arrival of a bully job, he must start sliding right away (unless there are other bullies sliding, because, as we already know, bully jobs respect each other). Ms. Schedule decided to consider the minimum makespan problem for this setting.

5.1 Hardness Proof for a Very Simple Instance

It is known that $1|prec, r_j|C_{max}$ is solvable in polynomial time for any precedence-constraints graph [16]. This is not the case with selfish precedence-constraints: The problem is NP-hard already for $K_{b,n}$. In fact, already the special case of $K_{1,n}$ which is an out-tree of depth 1, and when all nice jobs are available at time $t = 0$, can be reduced from the subset-sum problem.

Theorem 8. *The problem $1|K_{1,n}, s-prec, r_j(B)|C_{max}$ is NP-hard.*

5.2 A PTAS for $1|K_{b,n}, s-prec, r_j(B)|C_{max}$

Ms. Schedule decided to develop a PTAS for a single slide for the problem she is facing. Her first observation was that any feasible schedule of this type alternates between sliding time of bullies and nice jobs (see Figure 3). Formally, the schedule consists of alternating B -intervals and N -intervals. A B -interval begins whenever a bully job arrives and no other bully is sliding, and continues as long as some bully job is around. The N -intervals are simply the complement of the B -intervals. During N -intervals, nice jobs may slide. In particular, during the last N -interval (after all bullies are done) nice jobs who are still in line can slide. The finish time of this interval, N_k , for some $k \leq b$, determines the makespan of the whole schedule. Given the release times and the sliding times of the bullies, the partition of time into B - and N - intervals can be done in a straightforward way – by assigning the bully jobs greedily one after the other whenever they are available. “Given the partition into B - and N -intervals”, thought MS. Schedule, “my goal is to utilize the N -intervals in the best possible way. In fact, I need to *pack* the nice jobs into the N -intervals, leaving as few idle time of the slide as possible. The slide might be idle towards the end of an N -interval, when no nice job can complete sliding before a bully shows up. Given $\varepsilon > 0$, my PTAS consists of the following steps:

² Because they were busy pushing and calling names everybody on their way.

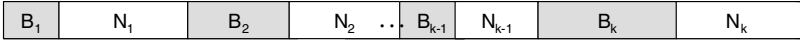


Fig. 3. The structure of any feasible schedule

1. Assign the bully jobs greedily. This determines the B - and N -intervals. Let k be the number of B -intervals.
2. Build the following instance for the multiple-knapsack (MK) problem:
 - $k - 1$ knapsacks of sizes $|N_1|, |N_2|, \dots, |N_{k-1}|$.
 - n items, where item j has size and profit p_j (sliding time of nice job j).
3. Run a PTAS for the resulting MK problem [3], with ε as a parameter.
4. Assign the nice jobs to the first $k - 1$ N -intervals as induced by the PTAS. That is, jobs that are packed into a knapsack of size $|N_i|$ will be scheduled during N_i . Assign the remaining nice jobs, which were not packed by the PTAS, to N_k with no intended idle.

Let C_{ALG} denote the makespan of the schedule produced by the PTAS. Let C^* denote the optimal minimum makespan.

Claim. Let $\varepsilon > 0$ be the PTAS parameter, then $C_{ALG} \leq (1 + \varepsilon)C^*$.

6 Selfish Precedence-Constraints of Unit-Length Jobs

Summer arrived. The jobs prepared a wonderful end-of-year show. The parents watched proudly how their jobs were simulating complex heuristics. No eye remained dry when the performance concluded with a breathtaking execution of a PTAS for the minimum makespan problem. At the end of the show they all stood and saluted the jobs and Ms. Schedule for their efforts.

Ms. Schedule decided to devote the summer vacation to extending her research on selfish precedence-constraints. During the school year, she only had time to consider the complete bipartite-graph case, and she was looking forward for the summer, when she will be able to consider more topologies of the precedence graph.

That evening, she wrote in her notebook: The good thing about bully jobs is that they do not avoid others be processed simultaneously with them. Among all, it means that the scheduler is more flexible. If for example we have two jobs and two machines, they can be processed simultaneously even if one of them is bully. With regular precedence-constraints, many problems are known to be NP-hard even if jobs have unit-length or if the precedence-constraints have limited topologies. For example, $P|p_i = 1, prec|C_{max}$ is NP-hard [23], as well as $P|p_i = 1, prec|\sum C_j$ [18]. These hardness results are not valid for selfish precedence-constraints. Formally,

Theorem 9. *The problems $P|p_i = 1, s\text{-}prec|C_{max}$ and $P|p_i = 1, s\text{-}prec|\sum C_j$ are solvable in polynomial time.*

Proof. Let n, m be the number of jobs and machines, respectively. Consider any *topological sort* of the selfish precedence-constraints graph. Since the graph is induced by a partial order relation, such a sort always exist. An optimal schedule simply assign the first m jobs in the first heat, that is, schedule them in time $[0, 1]$. The next heat will consist of the next m jobs in the topological sort, and so on. The makespan of this schedule is $\lceil n/m \rceil$ which is clearly optimal. This schedule is also optimal with respect to total flow-time, as it is a possible output of algorithm SPT on the same input without the selfish precedence-constraints. The schedule is feasible: if $i \prec_s j$ then i appears before j in the topological sort. Therefore, i is not assigned to a later heat than j . They may be assigned to the same heat, which is acceptable by job i .

7 Summary and Discussion

A new school year was about to begin. The jobs had wonderful time in the summer and were very excited to return to 1st-grade at Graham school. Ms. Schedule summarized her results for the 1st-grade teacher, Ms. Worst-case, who was full of concerns towards getting the bully jobs to her room.

“I focused on selfish precedence-constraints given by a complete bipartite graph”, she started “essentially, this models the bully-equilibrium problem we have at school. I first analyzed the price of bullying for the two objectives I found most important: minimum makespan and total-flow time. Next, I analyzed the well-known heuristics List-Scheduling and LPT, and I developed a PTAS for the minimum makespan problem. I then considered the problem of minimizing the total flow-time. I have a hardness proof for instances with many nice jobs and an optimal algorithm for instances with a single nice job. I suggest that you consult with the principal regarding this problem. He is not as dumb as he seems.

If the bully jobs keep being late also in 1st grade, you can use my PTAS for minimizing the makespan when bullies have release times. Finally, while for regular precedence-constraints, many problems are NP-hard already with unit-length jobs and very restricted topologies of the precedence graph, I showed that with selfish precedence-constraints, and any precedence graph, minimizing both the makespan and the total flow-time can be solved in linear-time.

I trust you to consider the following problems during the next school year:”

(i) The only objectives I considered are minimum makespan and total flow-time. It would be very interesting to consider instances in which jobs have due dates, and the corresponding objectives of minimizing total or maximal tardiness and lateness. For regular precedence-constraints, these problems are known to be NP-hard already for unit-length jobs and restricted precedence topologies [18,19].

(ii) As I’ve just told you, the hardness of minimizing the total flow depends on the number of nice job. Can you find the precise value of n for which the problem becomes NP-hard? This value might be a constant or a function of m, b , or the sliding times. Also, as the problem is closely related to the bi-criteria problem $P2||F_h(C_{max}/\sum C_j)$, it is desirable to check if heuristics suggested for it (e.g., in [6,9]) are suitable also for our setting.

(iii) It would be nice to extend my PTAS for $1|K_{b,n}, s\text{-}prec, r_j(B)|C_{max}$ for parallel slides. Note that for this setting, a late-arriving bully pushes a way only a single nice job (of his choice, or not, depending on your authorization).

(iv) As is the case with other scheduling problems, it would be nice to extend the results to uniformly related or unrelated machines, and to consider additional precedence-constraints graphs, such as chains and in/out-trees.

(v) Another natural generalization for the total flow-time objective, is when jobs have weights. For regular precedence-constraints, the problem $1|prec|\sum w_j C_j$ is known to be NP-hard, and several approximation algorithms are known [174].

References

1. Brucker, P., Knust, S.: Complexity results for scheduling problems, <http://www.mathematik.uni-osnabrueck.de/research/OR/class/>
2. Bruno, J.L., Coffman, E.G., Sethi, R.: Algorithms for minimizing mean flow-time. In: IFIPS Congress, vol. 74, pp. 504–510 (1974)
3. Chekuri, C., Khanna, S.: A PTAS for the multiple knapsack problem. In: Proc. of the 11th ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 213–222 (2000)
4. Chekuri, C., Motwani, R.: Precedence constrained scheduling to minimize sum of weighted completion times on a single machine. *Discrete Applied Mathematics* 98(1-2), 29–38 (1999)
5. Coffman, E.G., Sethi, R.: A generalized bound on LPT sequencing. In: Joint Int. Conf. on Measurements and Modeling of Computer Systems, SIGMETRICS (1976)
6. Coffman, E.G., Sethi, R.: Algorithms minimizing mean flow-time: schedule length properties. *Acta Informatica* 6, 1–14 (1976)
7. Dror, M., Kubiak, W., Dell’Olmo, P.: Strong-weak chain constrained scheduling. *Ricerca Operativa* 27, 35–49 (1998)
8. Du, J., Leung, J.Y.-T., Young, G.H.: Scheduling chain-structured tasks to minimize makespan and mean flow-time. *Inform. and Comput.* 92(2), 219–236 (1991)
9. Eck, B.T., Pinedo, M.: On the minimization of the makespan subject to flowtime optimality. *Operations Research* 41, 797–800 (1993)
10. Epstein, L., Sgall, J.: Approximation schemes for scheduling on uniformly related and identical parallel machines. *Algorithmica* 39(1), 43–57 (2004)
11. Garey, M.R., Johnson, D.S.: *Computers and Intractability: A guide to the theory of NP-completeness*. W. H. Freeman and Company, San Francisco (1979)
12. Graham, R.L.: Bounds for Certain Multiprocessing Anomalies. *Bell Systems Technical Journal* 45, 1563–1581 (1966)
13. Graham, R.L.: Bounds on Multiprocessing Timing Anomalies. *SIAM J. Appl. Math.* 17, 263–269 (1969)
14. Hu, T.C.: Parallel sequencing and assembly line problems. *Oper. Res.* 9, 841–848 (1961)
15. Hochbaum, D.S., Shmoys, D.B.: Using dual approximation algorithms for scheduling problems: Practical and theoretical results. *J. of the ACM* 34(1), 144–162 (1987)
16. Lawler, E.L.: Optimal sequencing of a single machine subject to precedence constraints. *Management Sci.* 19, 544–546 (1973)
17. Lawler, E.L.: Sequencing jobs to minimize total weighted completion time subject to precedence constraints. *Ann. Discrete Math.* 2, 75–90 (1978)

18. Lenstra, J.K., Rinnooy Kan, A.H.G.: Complexity of scheduling under precedence constraints. *Oper. Res.* 26(1), 22–35 (1978)
19. Leung, J.Y.-T., Young, G.H.: Minimizing total tardiness on a single machine with precedence constraints. *ORSA J. Comput.* 2(4), 346–352 (1990)
20. Sahni, S.: Algorithms for scheduling independent tasks. *J. of the ACM* 23, 555–565 (1976)
21. Smith, W.E.: Various optimizers for single-stage production. *Naval Research Logistics Quarterly* 3, 59–66 (1956)
22. Tamir, T.: Scheduling with Bully Selfish Jobs, <http://www.faculty.idc.ac.il/tami/Papers/FUNfull.pdf>
23. Ullman, J.D.: NP-complete scheduling problems. *J. Comput. System Sci.* 10, 384–393 (1975)

O(1)-Time Unsorting by Prefix-Reversals in a Boustrophedon Linked List

Aaron Williams

Dept. of Computer Science, University of Victoria, Canada

Abstract. Conventional wisdom suggests that $O(k)$ -time is required to reverse a substring of length k . To reduce this time complexity, a simple and unorthodox data structure is introduced. A *boustrophedon linked list* is a doubly-linked list, except that each node does not differentiate between its backward and forward pointers. This lack of information allows substrings of any length to be reversed in $O(1)$ -time. This advantage is used to efficiently unsort permutations using prefix-reversals. More specifically, this paper presents two algorithms that visit each successive permutations of $\langle n \rangle = \{1, 2, \dots, n\}$ in worst-case $O(1)$ -time (i.e. loopless). The first visits the permutations using a prefix-reversal Gray code due to Zaks [22], while the second visits the permutations in co-lexicographic order. As an added challenge, both algorithms are non-probing since they rearrange the data structure without querying its values. To accomplish this feat, the algorithms are based on two integer sequences: A055881 in the OEIS [17] and an unnamed sequence.

Keywords: unsorting, permutations, lexicographic order, prefix-reversal, Gray code, loopless algorithm, boustrophedon linked list, BLL, integer sequences.

1 Introduction

Suppose $\mathbf{p} = p_1 \dots p_n$ is a string containing n symbols. The following operations replace its substring $p_i p_{i+1} \dots p_{j-1} p_j$ where $1 \leq i < j \leq n$. A *transposition* of the i th and j th symbols replaces the substring by $p_j p_{i+1} \dots p_{j-1} p_i$. A *shift* of the i th symbol into the j th position replaces the substring by $p_{i+1} \dots p_{j-1} p_j p_i$. A *reversal* between the i th and j th symbols replaces the substring by $p_j p_{j-1} \dots p_{i+1} p_i$.

Reversals are the most powerful of these operations. This is because any transposition or shift can be accomplished by at most two reversals. Similarly, reversals are also the most expensive of these operations. More precisely, $O(k)$ -time is required to reverse a substring of length k in an array or linked list, whereas $O(1)$ -time is sufficient for array transpositions and linked list shifts.

Reversals are a bottleneck when *unsorting permutations* in *lexicographic order*. Unsorting begins with the string $1\ 2\ \dots\ n$ and concludes when this string has been rearranged in all $n! - 1$ ways. (This is dual to sorting, which starts with an arbitrary string over $\langle n \rangle$ and rearranges it into $1\ 2\ \dots\ n$.) In lexicographic order, worst-case $O(n)$ -time is required to create successive permutations of $\langle n \rangle$

when the string is stored in an array or linked list. To illustrate why this is true, consider the permutations of $\langle 8 \rangle$ given below in lexicographic order

1 2 3 4 5 6 7 8, ..., 3 8 7 6 5 4 2 1, 4 1 2 3 5 6 7 8, ..., 8 7 6 5 4 3 2 1.

Let $\mathbf{p} = 3\ 8\ 7\ 6\ 5\ 4\ 2\ 1$ and $\mathbf{q} = 4\ 1\ 2\ 3\ 5\ 6\ 7\ 8$. Notice that no positions match in \mathbf{p} and \mathbf{q} ($p_i \neq q_i$ for all $1 \leq i \leq n$). Therefore, if the string is stored in an array then every entry needs to be changed when rearranging \mathbf{p} into \mathbf{q} . Similarly, no substrings of length two match in \mathbf{p} and \mathbf{q} (i.e., $\{p_1p_2, p_2p_3, \dots, p_{n-1}p_n\} \cap \{q_1q_2, q_2q_3, \dots, q_{n-1}q_n\} = \emptyset$). Therefore, if the string is stored in a linked list then every forward pointer needs to be changed when rearranging \mathbf{p} into \mathbf{q} .

The object of this paper is to reduce the worst-case from $O(n)$ -time to $O(1)$ -time. In other words, the object is to create a *loopless* algorithm for unsorting permutations in lexicographic order. This goal is achieved by using two integer sequences from Section 2, and a data structure from Section 4. The same goal is achieved for a second order of permutations discussed in Section 3.

This section concludes with additional context on (un)sorting, permutations, algorithms, and reversals. Reversing a prefix of a string is known as a *prefix-reversal*. *Pancake sorting* refers to sorting algorithms that use prefix-reversals, with focus on upper-bounds on the number of reversals used in sorting an arbitrary permutation [7,2]. The minimum number of reversals needed to sort a particular permutation is known as *sorting by reversals*. This problem arises in biology when determining hereditary distance [1,6]. In *burnt pancake sorting* [4] and *signed sorting by reversals* [9], each pancake or element has two distinct sides; the boustrophedon linked list is an analogous data structure (except individual nodes are unaware of their orientation). Algorithms for unsorting permutations were surveyed as early as the 1960s [14]. The term *loopless* was coined by Ehrlich [5], and loopless algorithms for unsorting permutations (using non-lexicographic orders) exist using transpositions in arrays [18,8], and shifts in linked lists [13,20]. However, lexicographic order has distinct advantages including linear-time ranking [15]. The subject of *combinatorial generation* is devoted to efficiently unsorting various combinatorial objects. Section 7.2.1 of Knuth's *The Art of Computer Programming* [10,11,12] is an exceptional reference.

2 Integer Sequences

This section defines two integer sequences, explains their connection to staircase strings, and discusses their efficient generation. The first sequence is denoted \mathcal{R} and is known as OEIS A055881 [17], while the second sequence is denoted \mathcal{T} and is otherwise unnamed. The first $4! = 24$ terms of each sequence appear below

$\mathcal{R} = 1, 2, 1, 2, 1, 3, 1, 2, 1, 2, 1, 3, 1, 2, 1, 2, 1, 3, 1, 2, 1, 2, 1, 4, \dots$ (OEIS A055881)

$\mathcal{T} = 1, 1, 1, 2, 1, 1, 1, 1, 1, 2, 1, 2, 1, 1, 1, 2, 1, 3, 1, 1, 1, 2, 1, 1, \dots$ (unnamed).

The i th values in each sequence are respectively denoted r_i and t_i . The first $n! - 1$ values in each sequence are respectively denoted by \mathcal{R}_n and \mathcal{T}_n . That is,

$\mathcal{R}_n = r_1, r_2, \dots, r_{n!-1}$ and $\mathcal{T}_n = t_1, t_2, \dots, t_{n!-1}$. The sequences can be defined recursively as follows: Let $\mathcal{R}_2 = r_1 = 1$ and $\mathcal{T}_2 = t_1 = 1$, and then for $n > 2$,

$$\mathcal{R}_n = \overbrace{\mathcal{R}_{n-1}, n, \mathcal{R}_{n-1}, n, \dots, \mathcal{R}_{n-1}, n, \mathcal{R}_{n-1}, n, \mathcal{R}_{n-1}}^{n \text{ total copies of } \mathcal{R}_{n-1}} \tag{1}$$

$$\mathcal{T}_n = \mathcal{T}_{n-1}, 1, \mathcal{T}_{n-1}, 2, \dots, \mathcal{T}_{n-1}, n-2, \mathcal{T}_{n-1}, n-1, \mathcal{T}_{n-1}. \tag{2}$$

To explain the origins of these sequences, let us consider the staircase strings of length $n - 1$. A *staircase string* of length $n - 1$ is a string $\mathbf{a} = a_1 a_2 \dots a_{n-1}$ with the property that $0 \leq a_i \leq i$ for each $1 \leq i \leq n - 1$. In other words, the i th symbol can take on the $i + 1$ values in $\{0, 1, \dots, i\}$, and so staircase strings are simply the *mixed-radix strings* with bases $2, 3, \dots, n$. Notice that there are $n!$ staircase strings of length $n - 1$.

Sequences \mathcal{R} and \mathcal{T} can also be defined by their connection to staircase strings in co-lex order. (Strings are ordered from right-to-left in *co-lexicographic* (*co-lex*) order, and this order is often more convenient than lexicographic order.) When the i th staircase string in co-lex order is followed by the $(i + 1)$ st staircase string in co-lex order, then r_i is the position of the rightmost symbol that changes value, and t_i is the new value. For example, $1\ 2\ 3\ 4\ 1\ 0\ 0\ \dots$ is the 240th staircase string in co-lex order and $0\ 0\ 0\ 0\ 2\ 0\ 0\ \dots$ is the 241st staircase string in co-lex order. Therefore, $r_{240} = 5$ and $t_{240} = 2$. For a more complete example, the staircase strings of length 3 appear in column (a) of Table 1 in co-lex order, while columns (b) and (c) contain \mathcal{R}_4 and \mathcal{T}_4 . (Table 1 appears on page 371 and columns (b) and (c) have been offset by half a row to emphasize the transition between successive staircase strings.) To generate \mathcal{R} and \mathcal{T} , one can simply augment Algorithm M (Mixed-radix generation) in 7.2.1.1 of [10] while using $m_i = i$ for the bases, for all $1 \leq i \leq n - 1$.

To generate \mathcal{R} and \mathcal{T} by a loopless algorithm, we can again follow [10]. Algorithm H in [10] generates multi-radix strings in *reflected Gray code* order. In this order, successive strings differ in ± 1 at a single position, and all “roll-overs” are suppressed. Furthermore, the position whose value changes is simply the rightmost position whose value changes in co-lex order. Within Algorithm H, each position is assigned a *direction* from $\{+1, -1\}$, and *focus pointers* determine the positions whose value will change. TR(n) gives pseudocode for Algorithm H to the right of Table 1 in a **Fun** function. Lines 10-15 output successive values of \mathcal{R} and \mathcal{T} into arrays r and t , which are indexed by $i = 1, 2, \dots, n! - 1$. The staircase strings are stored in array a , the directions are stored in array d , and the focus pointers are stored in array f . Columns (d),(e), and (f) in Table 1 respectively provide the values stored in these three arrays when TR(n) is run with $n = 4$. (Within our programs all array indexing is 1-based, and if a is an array then $a[i]$ denotes its i th entry.) Using TR(n), successive entries of \mathcal{R} and \mathcal{T} can be computed in worst-case $O(1)$ -time. In Section 5, we will modify TR(n) so that its i th iteration provides the r_i th and t_i th pointers into a (boustrophedon) linked list. The remaining columns in Table 1 are explained in Section 3.

Table 1. Relationships between staircase strings, permutation orders, and sequences. Columns (b), (c), (h), (j), (k), (m) respectively contain $\mathcal{R}_4, \mathcal{T}_4, \mathcal{R}_4 + 1, \mathcal{T}_4, \mathcal{R}_4 + 1, \mathcal{R}_4$. Sequences \mathcal{R}_n and \mathcal{T}_n are generated by loopless algorithm TR(n) in arrays r and t .

staircase in co-lex			staircase in Gray code			perms in Zaks'		perms in co-lex				% Loopless \mathcal{R}_n and \mathcal{T}_n Fun TR(n)
(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)	(j)	(k)	(l)	
000	1	1	000	+++	1234	4321	2	4321	1	2	1	1: $r := [-1, \dots, -1]$ (size $n!-1$)
100	2	1	100	-++	2234	3421	3	3421	1	3	2	2: $t := [-1, \dots, -1]$ (size $n!-1$)
010	1	1	110	-++	1234	2431	2	4231	1	2	1	3: $f := [1, 2, \dots, n]$ (size n)
110	2	2	010	+++	2234	4231	3	2431	2	3	2	4: $d := [1, 1, \dots, 1]$ (size $n-1$)
020	1	1	020	+++	1334	3241	2	3241	1	2	1	5: $a := [0, 0, \dots, 0]$ (size $n-1$)
120	3	1	120	---+	3234	2341	4	2341	1	4	3	6: $i := 1$
001	1	1	121	---+	1234	1432	2	4312	1	2	1	7: while $f[1] < n$
101	2	1	021	+-+	2234	4132	3	3412	1	3	2	8: $j := f[1]$
011	1	1	011	+-+	1234	3142	2	4132	1	2	1	9: $f[1] := 1$
111	2	2	111	---+	2234	1342	3	1432	1	2	1	10: $r[i] := j$
021	1	1	101	-++	1334	4312	2	3142	1	2	1	11: if $d[j] = 1$
121	3	2	001	+++	3234	3412	4	1342	2	4	3	12: $t[i] := a[j] + 1$
002	1	1	002	+++	1234	2143	2	4213	1	2	1	13: else
102	2	1	102	-++	2234	1243	3	2413	1	3	2	14: $t[i] := j - a[j] + 1$
012	1	1	112	-++	1234	4213	2	4123	1	2	1	15: end if
112	2	2	012	+++	2234	2413	3	1423	2	3	2	16: $a[j] := a[j] + d[j]$
022	1	1	022	+++	1334	1423	2	2143	1	2	1	17: if $a[j] = 0$ OR $a[j] = j$
122	3	3	122	---+	3234	4123	4	1243	3	4	3	18: $d[j] := -d[j]$
003	1	1	123	---	1244	3214	2	3214	1	2	1	19: $f[j] := f[j + 1]$
103	2	1	023	+-+	2244	2314	3	2314	1	3	2	20: $f[j + 1] := j + 1$
013	1	1	013	+-+	1244	1324	2	3124	1	2	1	21: end if
113	2	2	113	---	2244	3124	3	1324	2	3	2	22: $i := i + 1$
023	1	1	103	-++	1434	2134	2	2134	1	2	1	23: end while
123			003	+-+	4234	1234		1234				

3 Permutations of Permutations

This section discusses two orders (or permutations) of the permutations of $\langle n \rangle$. The first is Zaks' prefix-reversal Gray code [22], and the second is co-lex order. These two orders can be created from the sequences found in Section 2. In particular, \mathcal{R} and \mathcal{T} have been named after the Reversal and Transposition operations that produce the two orders.

Zaks' original paper describes a poor waiter who must flip n pancakes of different sizes into all $n!$ possible stacks. In order to do this, Zaks proposes an order where "in $1/2$ of these steps he will reverse the top 2 pancakes, in $1/3$ of them the top 3, and, in general, in $(k-1)/k!$ of them he will reverse the top k pancakes". As Zaks points out, the waiter can achieve such an ordering directly from sequence \mathcal{R} . In particular, his $(i+1)$ st stack is obtained from his i th stack by flipping the top $r_i + 1$ pancakes. In the parlance of permutations, Zaks proves that a prefix-reversal Gray code for the permutations of $\langle n \rangle$ is obtained by successively reversing the prefix of length $r_i + 1$ in the i th permutation. This is illustrated for

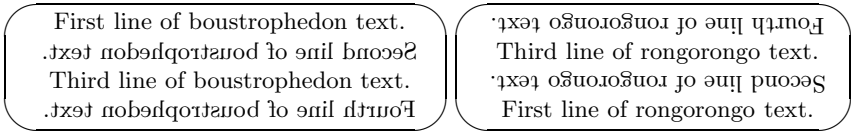
$n = 4$ in Table [11](#), with columns (g) and (h) providing the permutations of $\langle 4 \rangle$ in Zaks' order and $\mathcal{R}_4 + 1$. (For convenience, $\mathcal{R}_n + 1$ denotes the sequence obtained by adding 1 to each entry in sequence \mathcal{R}_n .) Zaks' paper shows how to generate $\mathcal{R}_n + 1$ using a loopless algorithm, and also proves that the average value of $\mathcal{R}_n + 1$ is less than $e \cong 2.8$. For this reason, it is possible to generate Zaks' prefix-reversal Gray code with a *constant amortized-time algorithm (CAT)*, meaning that each successive permutation is created in amortized $O(1)$ -time. Such an algorithm can be constructed from $\text{TR}(n)$ by initializing an array $p = [1, 2, \dots, n]$, and then by replacing lines [10–15](#) with a loop that reverses its first $j + 1$ entries.

As Knuth points out at the beginning of Section 7.2.1.2 in *The Art of Computer Programming* [\[10\]](#), an iterative algorithm for creating the permutations of $\langle n \rangle$ in co-lex order dates back to the 14th-century. This historical algorithm is based on scanning the current permutation. Less obviously, co-lex order can be created using \mathcal{R} and \mathcal{T} . In particular, the $(i + 1)$ st permutation can be obtained from the i th permutation in two steps. In the first step, the symbols in positions t_i and $r_i + 1$ are transposed. In the second step, the prefix of length r_i is reversed. This is illustrated for $n = 4$ in Table [11](#). Column (i), (j), (k), and (l) provide the permutations of $\langle 4 \rangle$ in co-lex order, \mathcal{T}_4 , $\mathcal{R}_4 + 1$, and \mathcal{R}_4 . As in the case of Zaks' algorithm, $\text{TR}(n)$ can be modified to provide an array-based CAT algorithm for creating permutations in co-lex order.

By using sequences \mathcal{R} and \mathcal{T} , the difference between successive permutations in Zaks' order and co-lex order can be described in worst-case $O(1)$ -time. Section [4](#) will allow these differences to be performed in worst-case $O(1)$ -time.

4 Boustrophedon Linked Lists

In Greek, *βουστροφηδόν* means “as the ox turns while plowing”. In English, *boustrophedon* is commonly used to describe ancient texts that are reflected horizontally on every second line [\[19\]](#). A variation from Rapa Nui is *rongorongongo* that rotates every second line 180° .



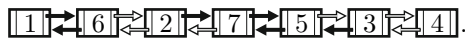
The term boustrophedon also describes back-and-forth motion without reflections or rotations. For example, dot-matrix printers are more efficient when printing *boustrophedonically*. Similarly, Major League Eating world-recorder holder Jason “Crazy Legs” Conti often eats sweet corn in a *boustrophedonic* pattern [\[21\]](#). Scientifically, the *boustrophedon transform* and *boustrophedon cellular decomposition* appear in mathematics [\[16\]](#) and computer science [\[3\]](#), respectively.

As in a standard doubly-linked list (DLL), the first node of a *boustrophedon linked list* (BLL) is the *head* and the last node is the *tail*. Similarly, each node in a BLL has a *value* and is connected to its two *adjacent* nodes by a *forward* pointer (.fwd) and a *backward* pointer (.bwd). However, in a BLL the forward pointers do

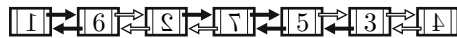
not necessarily point towards the tail. In other words, the backward and forward pointers of any node in a BLL can be independently interchanged. This includes the head (resp. tail), where one pointer must be NULL and the other is directed to the second (resp. second-last) node. Nodes are *normal* or *reversed* if their forward pointer is directed towards the tail or head, respectively.

Despite the uncertainty at each node, a BLL containing n nodes can be traversed in $O(n)$ -time. Furthermore, this $O(n)$ -time traversal could transform the BLL into a DLL by “flipping” each reversed node. As in a DLL, nodes in a BLL can be inserted, removed, transposed, and shifted in worst-case $O(1)$ -time. However, a BLL has a distinct advantage over a DLL: any sublist can be reversed in worst-case $O(1)$ -time. This section describes these results, and builds a small “boustrophedon toolkit” that includes the concepts of twin-pointers and balancing. Again it is stressed that no additional directional information is stored within each node of the BLL nor within an auxiliary data structure. (One can consider BLL alternatives that add an extra bit of information to each node of a DLL. This bit could not denote whether a node is normal or reversed without precluding $O(1)$ -time reversals. However, this worst-case time could be obtained by having the first i bits determine whether the i th node is normal or reversed.)

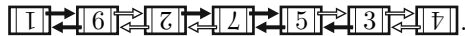
Before discussing these results, let us have fun considering pictographic representations for a BLL. In each proposal the nodes appear left-to-right from head-to-tail, with backward pointers in black (\rightarrow), forward pointers in white (\Leftarrow), and NULL pointers omitted at the head and tail. Thus, normal and reversed internal nodes appear respectively as $\leftarrow \boxed{} \rightarrow$ and $\leftarrow \boxed{} \Leftarrow$. A BLL storing the permutation $\mathbf{p} = 1\ 6\ 2\ 7\ 5\ 3\ 4$ is shown below in its *standard representation*



From the head $\boxed{1}$ to the tail $\boxed{4}$, successive nodes are reached by traveling backwards, forwards, backwards, backwards, forwards, and forwards (as seen by the pointers on the top row). To draw attention to the reversed nodes, we can also represent this BLL using its *boustrophedonic representation* (where the values of reversed nodes are reflected horizontally)



or its *rongorongo representation* (where reversed nodes values are rotated 180°)



Rongorongo representation is nice since reversals are visualized as 180° rotations.

In a BLL it is often helpful to use two pointers when one would suffice in a DLL. This *twin-pointer* approach is demonstrated for traversing the nodes in a BLL from head to tail. Say that pointers x and y are *adjacent* if they point to two nodes that are adjacent in the BLL. When x and y are adjacent pointers, then x 's *other* node is its adjacent node that is not y . (If x is the head or tail of the BLL, then either x 's other node or y will be NULL.) The simple logic needed to determine $\text{other}(x, y)$ appears below, (All variables in this section are pointers to nodes in a BLL.) Using this routine we can easily traverse a BLL in $\text{traverse}(\text{head})$,

where pointer x traverses the BLL while pointer y follows one node behind, and t is a temporary variable.

```

% Traverse BLL          % Other neighbor of x      % Flip reversed ↔ normal
Fun traverse(head)    Fun other(x, y)          Fun flip(x)
  y := NULL            if x.fwd = y              t := x.fwd
  x := head            return x.bwd              x.fwd := x.bwd
  while x ≠ NULL      else if x.bwd = y          x.bwd := t
    t := x              return x.fwd
    x := other(x, y)    end if
    y := t
  end while
    
```

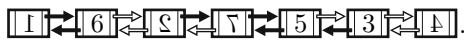
Clearly, `traverse(head)` takes $O(n)$ -time when the BLL contains n nodes. The above code can be expanded to convert the BLL into a DLL in $O(n)$ -time by remembering the orientation of successive nodes starting from head and “flipping” those nodes that are reversed. Pseudocode for `flip(x)` appears above and also changes normal nodes into reversed nodes.

Another way to overcome the lack of information in a BLL is to make a portion of the BLL behave like a standard DLL. In a DLL, $x.fwd = y$ implies $y.bwd = x$, so long as $y \neq \text{NULL}$. In other words, proceeding forwards and then backwards will return you to the initial node, so long as the intermediate node exists. On the other hand, the same statement is not true in a BLL unless x and y both point to normal nodes, or reversed nodes. We say that an adjacent pair of nodes $\{x, y\}$ is *balanced* or *imbalanced* depending on `isBalanced(x, y)`.

```

% Is adjacent pair {x, y} balanced?          % Balance pair {x, y}
Fun isBalanced(x, y)                        Fun balance(x, y)
  return (x = NULL) OR (y = NULL) OR        if NOT isBalanced(x, y)
  (x.fwd = y AND y.bwd = x) OR              flip(x)
  (x.bwd = y AND y.fwd = x)                 end if
    
```

Pictorially, adjacent nodes are balanced if and only if there is one white and one black pointer between them. For example, the previously-drawn BLL for $p = 1\ 6\ 2\ 7\ 5\ 3\ 4$ appears below (in its boustrophedonic representation)



Notice that the only balanced pairs are the nodes containing $\{2, 7\}$ and $\{5, 3\}$. Any pair of adjacent nodes can be assured to be balanced by calling `balance(x, y)`. For example, flipping the node containing 2 causes the nodes containing $\{6, 2\}$ to become balanced at the expense of imbalancing the nodes containing $\{2, 7\}$



When x is not in an imbalanced pair, then x is a *balanced node*. Otherwise, if x is in an imbalanced pair, then x is an *imbalanced node*. To change a node from imbalanced to balanced, either the node itself or one of its neighbors must be flipped. Any node can be assured to be balanced by calling `balance(x)`.

```

% Balance  $x$ 
Fun balance( $x$ )
  if NOT isBalanced( $x, x.fwd$ ) AND NOT isBalanced( $x, x.bwd$ )
    flip( $x$ )
  else if NOT isBalanced( $x, x.fwd$ )
    flip( $x.fwd$ )
  else if NOT isBalanced( $x, x.bwd$ )
    flip( $x.bwd$ )
  end if

```

By balancing nodes and pairs of nodes, BLL deletions and insertions proceed as in a DLL. In turn, these routines allow simple worst-case O(1)-time transpositions and shifts. Balancing also allows simple worst-case O(1)-time reversals.

<pre> % Remove node x Fun delete(x) balance(x) $x.bwd.fwd := x.fwd$ $x.fwd.bwd := x.bwd$ </pre>	<pre> % Transpose x and y Fun transpose(x, y) if $x = y$ return end if if $x = \text{NULL}$ return else if $y = \text{NULL}$ return end if balance(x) $f_x := x.fwd$ $b_x := x.bwd$ delete(x) if $f_x = y$ insert($x, y, y.fwd$) else if $b_x = y$ insert($x, y, y.bwd$) else $f_y := y.fwd$ $b_y := y.bwd$ delete(y) insert(x, f_y, b_y) insert(y, f_x, b_x) end if </pre>	<pre> % Reverse between u x % through between y v Fun reverse(u, x, y, v) if $x = y$ return end if balance(u, x) balance(y, v) $f_x := x.fwd$ $f_y := y.fwd$ if $f_x = u$ $x.fwd = v$ $u.bwd = y$ else $x.bwd = v$ $u.fwd = y$ end if if $f_y = v$ $y.fwd = u$ $v.bwd = x$ else $y.bwd = u$ $v.fwd = x$ end if </pre>
<pre> % Insert x between u v Fun insert(x, u, v) $x.bwd := v$ $x.fwd := u$ balance(u, v) if $u.fwd = v$ $u.fwd := x$ $v.bwd := x$ else $u.bwd := x$ $v.fwd := x$ end if </pre>	<pre> % Shift x between u v Fun shift(x, u, v) if $x = u$ OR $x = v$ return end if delete(x) insert(x, u, v) </pre>	

In $\text{shift}(x, u, v)$, u and v are adjacent. In $\text{reverse}(u, x, y, v)$, u and x are adjacent, as are y and v ; nodes between x and y are reversed while u and v are stationary.

5 Algorithms

At this point we have seen how Zaks' order and lexicographic order can be created using transpositions and reversals (Section 3), we have shown that the indices

r_i and t_i that describe these transpositions and reversals can be computed in worst-case $O(1)$ -time (Section 2), and we have presented a data structure that allows for worst-case $O(1)$ -time transpositions and reversals (Section 4). This section ties these results together by providing worst-case $O(1)$ -time algorithms for creating permutations in Zaks' order and in lexicographic order.

The first hurdle is that $\text{TR}(n)$ creates integers r_i and t_i , while pointers to the r_i th and t_i th nodes are required for BLL transpositions and reversals. As an intermediate step, algorithms $\text{R}(n)$ and $\text{T}(n)$ create the sequences \mathcal{R}_n and \mathcal{T}_n using a static linked list. In particular, head is initialized to $\boxed{1} \rightarrow \boxed{2} \rightarrow \dots \leftarrow \boxed{n}$ which can be viewed as a standard DLL or a BLL with no reversed nodes. At each iteration, a pointer points to its r_i th node (on line 17 in $\text{R}(n)$) and the t_i th node (on line 17 in $\text{T}(n)$) while the linked list is never changed.

<pre> % Loopless generation of \mathcal{R}_n in a BLL Fun $\text{R}(n)$ 1: head := $\boxed{1} \rightarrow \boxed{2} \rightarrow \dots \leftarrow \boxed{n}$ 2: $R := [\text{NULL}, \text{NULL}, \dots, \text{NULL}]$ (size n) 3: $r := [-1, -1, \dots, -1]$ (size $n!-1$) 4: $f := [1, 2, \dots, n]$ (size n) 5: $d := [1, 1, \dots, 1]$ (size $n-1$) 6: $a := [0, 0, \dots, 0]$ (size $n-1$) 7: $i := 1$ 8: while $f[1] < n$ 9: $j := f[1]$ 10: $f[1] := 1$ 11: $a[j] := a[j] + d[j]$ 12: if $R[1] = \text{NULL}$ 13: $R[1] := \text{head}$ 14: end if 15: $\text{Rnode} := R[1]$ 16: $R[1] := \text{NULL}$ 17: $r[i] := \text{Rnode.value}$ 18: if $a[j] = 0$ OR $a[j] = j$ 19: $d[j] := -d[j]$ 20: $f[j] := f[j + 1]$ 21: $f[j + 1] := j + 1$ 22: if $R[j + 1] = \text{NULL}$ 23: $R[j] := \text{Rnode.fwd}$ 24: else 25: $R[j] := R[j + 1]$ 26: end if 27: $R[j + 1] := \text{NULL}$ 28: end if 29: $i := i + 1$ 30: end while </pre>	<pre> % Loopless generation of \mathcal{T}_n in a BLL Fun $\text{T}(n)$ 1: head := $\boxed{1} \rightarrow \boxed{2} \rightarrow \dots \leftarrow \boxed{n}$ 2: $T := [\text{NULL}, \text{NULL}, \dots, \text{NULL}]$ (size n) 3: $t := [-1, -1, \dots, -1]$ (size $n!-1$) 4: $f := [1, 2, \dots, n]$ (size n) 5: $d := [1, 1, \dots, 1]$ (size $n-1$) 6: $a := [0, 0, \dots, 0]$ (size $n-1$) 7: $i := 1$ 8: while $f[1] < n$ 9: $j := f[1]$ 10: $f[1] := 1$ 11: $a[j] := a[j] + d[j]$ 12: if $T[j] = \text{NULL}$ 13: $T[j] := \text{head}$ 14: end if 15: $\text{Tnode} := T[j]$ 16: $T[j] := \text{Tnode.fwd}$ 17: $t[i] := \text{Tnode.value}$ 18: if $a[j] = 0$ OR $a[j] = j$ 19: $d[j] := -d[j]$ 20: $f[j] := f[j + 1]$ 21: $f[j + 1] := j + 1$ 22: $T[j] := \text{NULL}$ 23: end if 24: $i := i + 1$ 25: end while </pre>
---	---

% Loopless permutations in Zaks' order

Fun Zaks(n)

```

1: head := [n] → ... ← [2] ↔ [1]
2: R := [NULL, NULL, ..., NULL] (size n)
3: S := [NULL, NULL, ..., NULL] (size n)
4: f := [1, 2, ..., n] (size n)
5: d := [1, 1, ..., 1] (size n - 1)
6: a := [0, 0, ..., 0] (size n - 1)
7: while f[1] < n
8:   j := f[1]
9:   f[1] := 1
10:  a[j] := a[j] + d[j]
11:  if R[1] = NULL
12:    R[1] := head
13:    S[1] := other(head, NULL)
14:  end if
15:  Rnode := R[1]
16:  Snode := S[1]
17:  R[1] := NULL
18:  S[1] := NULL
19:  if a[j] = 0 OR a[j] = j
20:    d[j] := -d[j]
21:    f[j] := f[j + 1]
22:    f[j + 1] := j + 1
23:    if R[j + 1] = NULL
24:      temp := R[j]
25:      R[j] := S[j]
26:      S[j] := other(S[j], temp)
27:    else
28:      R[j] := R[j + 1]
29:      S[j] := S[j + 1]
30:    end if
31:    R[j + 1] := NULL
32:    S[j + 1] := NULL
33:  end if
34:  temp := other(Snode, Rnode)
35:  reverse(temp, Snode, head, NULL)
36:  if f[j] = j + 1
37:    R[j] := head
38:  end if
39:  head := Snode
40:  visit(head)
41: end while

```

% Loopless permutations in co-lex order

Fun Lex(n)

```

1: head := [n] → ... ← [2] ↔ [1]
2: R := [NULL, NULL, ..., NULL] (size n)
3: S := [NULL, NULL, ..., NULL] (size n)
4: T := [NULL, NULL, ..., NULL] (size n)
5: U := [NULL, NULL, ..., NULL] (size n)
6: f := [1, 2, ..., n] (size n)
7: d := [1, 1, ..., 1] (size n - 1)
8: a := [0, 0, ..., 0] (size n - 1)
9: while f[1] < n
10:  j := f[1]
11:  f[1] := 1
12:  a[j] := a[j] + d[j]
13:  if R[1] = NULL
14:    R[1] := head
15:    S[1] := other(head, NULL)
16:  end if
17:  Rnode := R[1]
18:  Snode := S[1]
19:  R[1] := NULL
20:  S[1] := NULL
21:  if T[j] = NULL
22:    T[j] := head
23:    U[j] := other(head, NULL)
24:  end if
25:  T[j] := Unode
26:  U[j] := other(Unode, Tnode)
27:  if a[j] = 0 OR a[j] = j
28:    d[j] := -d[j]
29:    f[j] := f[j + 1]
30:    f[j + 1] := j + 1
31:    if R[j + 1] = NULL
32:      temp := R[j]
33:      R[j] := S[j]
34:      S[j] := other(S[j], temp)
35:    else
36:      R[j] := R[j + 1]
37:      S[j] := S[j + 1]
38:    end if
39:    R[j + 1] := NULL
40:    S[j + 1] := NULL
41:    T[j] := NULL
42:    U[j] := NULL
43:  end if
44:  transpose(Snode, Tnode)
45:  if Tnode = head
46:    head := Snode
47:  end if
48:  if f[j] = j + 1
49:    R[j] := Tnode
50:  end if
51:  if Rnode ≠ Tnode
52:    reverse(Tnode, Rnode, head, NULL)
53:    head := Rnode
54:  else if Snode ≠ head
55:    reverse(Tnode, Snode, head, NULL)
56:    head := Snode
57:  end if
58:  visit(head)
59: end while

```

To understand $R(n)$, observe that r_i equals the first focus pointer in $TR(n)$ (see lines [84-10](#)). Therefore, we mimic the focus pointers in f using real pointers

in a new array R . More precisely, if $f[i] = j$ then $R[i]$ will point to the j th node in the linked list. One caveat is that `NULL` is used when a focus pointer points to itself. That is, $R[i] = \text{NULL}$ when $f[i] = i$. This convention limits the changes made to R when reversing prefixes in the final algorithms. Given this description, we can equate instructions involving f with instructions involving R in $R(n)$. In particular, line 9 matches with lines 12-14, line 10 matches with line 16, line 20 matches with lines 22-26, and line 21 matches with line 27. To track the $(r_i + 1)$ st node instead of the r_i th node, line 13 can simply be changed to $R[1] := \text{other}(\text{head}, \text{NULL})$. In $\text{Lex}(n)$ and $\text{Zaks}(n)$, the r_i th node is tracked in array R , the $(r_i + 1)$ st node is tracked in array S , and the resulting twin-pointers give a BLL-friendly alternative to line 23.

To understand $T(n)$ consider its recursive formula originally found in (2)

$$T_n = T_{n-1}, 1, T_{n-1}, 2, \dots, T_{n-1}, n-2, T_{n-1}, n-1, T_{n-1}.$$

The values in $1, 2, \dots, n-1, 1$ are obtained by “marching” the pointers forward in line 16, before “retreating” in line 22. To track the $(t_i + 1)$ st node instead of the t_i th node, line 13 can simply be changed to $T[j] := \text{other}(\text{head}, \text{NULL})$. In $\text{Lex}(n)$, the t_i th node is tracked in array T , the $(t_i + 1)$ st node is tracked in array U , and the resulting twin-pointers give a BLL-friendly alternative to line 16.

The second hurdle is to update these pointers during each transposition and reversal. The updates are simplified by the fact that $f[i] = i$ for $1 \leq i < r[i]$ during the i th iteration of $\text{TR}(n)$. Therefore, the corresponding `NULL` values in R , S , T , and U do not require updating. The additional code in the final algorithms account for the remaining updates to R , S , T , U , and `head`. (A tail pointer can also be easily updated.)

When investigating the final algorithms on page 376, notice the absence of “.value”. We call the algorithms *non-probing* since they never query values stored in any node. One application is that the algorithms will function regardless of the initial values pointed to by `head`. In particular, *co-lex* and *reverse co-lex* are created by initializing `head := [n] → ... ← [2] ← [1]` and `head := [1] ← [2] → ... ← [n]` respectively. Furthermore, *lexicographic* and *reverse lexicographic* are created by maintaining `tail` and replacing `visit(head)` with `visit(tail)`.

Both algorithms are implemented in C, and are available by request.

6 Open Problems

This paper uses a BLL to unsort permutations in worst-case $O(1)$ -time in lexicographic order. Can this result be extended to multiset permutations? Where else does a BLL provide advantages over a DLL? Do non-probing algorithms have applications to privacy or security? What other combinatorial objects can be generated without probing? $\text{TR}(n)$ gives a common framework for understanding lexicographic order and Zaks’ order. Which other orders of permutations can be explained by the specialization of Algorithm H 10 to staircase strings or *downward staircase strings* (using bases $n, n - 1, \dots, 2$)?

References

1. Bafna, V., Pevzner, P.A.: Genome rearrangements and sorting by reversals. *SIAM J. Comput.* 25(2), 272–289 (1996)
2. Chitturi, B., Fahle, W., Meng, Z., Morales, L., Shields, C.O., Sudborough, I.H., Voit, W.: An $(18/11)n$ upper bound for sorting by prefix reversals. *Theoretical Computer Science* 410, 3372–3390 (2009)
3. Choset, H.: Coverage of known spaces: The boustrophedon cellular decomposition. *Journal Autonomous Robots* 9(3), 247–253 (2000)
4. Cohen, D.S., Blum, M.: On the problem of sorting burnt pancakes. *Discrete Applied Mathematics* 61, 105–120 (1995)
5. Ehrlich, G.: Loopless algorithms for generating permutations, combinations and other combinatorial configurations. *Journal of the ACM* 20(3), 500–513 (1973)
6. Fertin, G., Labarre, A., Rusu, I., Tannier, E., Vialette, S.: *Combinatorics of Genome Rearrangements*. MIT Press, Cambridge (2009)
7. Gates, W.H., Papadimitriou, C.H.: Bounds for sorting by prefix reversal. *Discrete Mathematics* 27, 47–57 (1979)
8. Goldstein, A.J., Graham, R.L.: Sequential generation by transposition of all the arrangements of n symbols. Bell Telephone Laboratories (Internal Memorandum), Murray Hill, NJ (1964)
9. Kaplan, H., Shamir, R., Tarjan, R.E.: Faster and simpler algorithm for sorting signed permutations by reversals. In: *SODA 1997: Proceedings of the eighth annual ACM-SIAM symposium on discrete algorithms*, New Orleans, Louisiana, United States, pp. 178–187 (1997)
10. Knuth, D.E.: *The Art of Computer Programming*, volume 4 fascicle 2: Generating All Tuples and Permutations. Addison-Wesley, Reading (2005)
11. Knuth, D.E.: *The Art of Computer Programming*, volume 4 fascicle 3: Generating All Combinations and Partitions. Addison-Wesley, Reading (2005)
12. Knuth, D.E.: *The Art of Computer Programming*, volume 4 fascicle 4: Generating All Trees, History of Combinatorial Generation. Addison-Wesley, Reading (2006)
13. Korsh, J.F., Lipschutz, S.: Generating multiset permutations in constant time. *Journal of Algorithms* 25, 321–335 (1997)
14. Lehmer, D.H.: Teaching combinatorial tricks to a computer. In: *Combinatorial Analysis. Proc. of Symposium Appl. Math.*, vol. 10, pp. 179–193. American Mathematical Society, Providence (1960)
15. Mares, M., Straka, M.: Linear-time ranking of permutations. In: Arge, L., Hoffmann, M., Welzl, E. (eds.) *ESA 2007. LNCS*, vol. 4698, pp. 187–193. Springer, Heidelberg (2007)
16. Millar, J., Sloane, N.J.A., Young, N.E.: A new operation on sequences: the boustrophedon transform. *Journal of Comb. Theory, Series A* 76(1), 44–54 (1996)
17. Sloane, N.: (2010), <http://www.research.att.com/~njas/sequences/A055881>
18. Steinhaus, H.: *One Hundred Problems in Elementary Mathematics*. Pergamon Press, Oxford (1963); Reprinted by Dover Publications in 1979 (1979)
19. Wikipedia (2009), <http://en.wikipedia.org/wiki/Boustrophedon>
20. Williams, A.: Loopless generation of multiset permutations using a constant number of variables by prefix shifts. In: *SODA 2009: The Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, New York, USA (2009)
21. YouTube. 2009 National Sweet Corn Eating Championship (2009), <http://www.youtube.com/watch?v=EUy56pBA-HY>
22. Zaks, S.: A new algorithm for generation of permutations. *BIT Numerical Mathematics* 24(2), 196–204 (1984)

Author Index

- Arthur, David 307
Asano, Tetsuo 16, 28
- Ben-Zwi, Oren 37
Bernasconi, Anna 41
Boothe, Peter 53
Bressan, Marco 68
Brisaboa, Nieves R. 77
Burcsi, Péter 89
- Cantone, Domenico 166
Cao, Tao 331
Chatzigiannakis, Ioannis 4
Chien, Yu-Feng 102
Cicalese, Ferdinando 89, 113
Ciriani, Valentina 41
Clifford, Raphaël 307
Cohen, Nathann 121
Coudert, David 121
- Demaine, Erik D. 28, 133
Demaine, Martin L. 28, 133
Diochnos, Dimitrios I. 145
- Elmasry, Amr 156
- Faro, Simone 166
Fici, Gabriele 89
Fleischer, Rudolf 178
Flocchini, Paola 190
Focardi, Riccardo 202
Forišek, Michal 214
- Gagie, Travis 113
Ghosh, Arpita 228
Giaquinta, Emanuele 166
Grossi, Roberto 1
- Halldórsson, Magnús M. 237
Healy, Patrick 249
Holzer, Markus 260, 319
Hon, Wing-Kai 102
- Jalsenius, Markus 307
Jensen, Claus 156
- Katajainen, Jyrki 156
Kellett, Matthew 190
Kostitsyna, Irina 272
Kranakis, Evangelos 284
Krizanc, Danny 284
- Lipták, Zsuzsanna 89
Luaces, Miguel R. 77
Luccio, Fabrizio 41
Luccio, Flaminia L. 202
- Macula, Anthony J. 113
Mahdian, Mohammad 228, 296
Mamakani, Khalegh 331
Mason, Peter C. 190
Mazauric, Dorian 121
McKenzie, Pierre 260
Milanič, Martin 113
Montanaro, Ashley 307
Mylonas, Georgios 4
- Navarro, Gonzalo 77
Nepomuceno, Napoleão 121
Nisse, Nicolas 121
- Orlandi, Alessio 1
Ottaviano, Giuseppe 1
- Pagli, Linda 41
Panagopoulou, Panagiota N. 4
Peserico, Enoch 68
Polishchuk, Valentin 272
- Raghavan, Prabhakar 3
Ruepp, Oliver 319
Ruskey, Frank 331, 343
- Sach, Benjamin 307
Santoro, Nicola 190

Seco, Diego 77

Shachnai, Hadas 237

Spirakis, Paul G. 4

Tamir, Tami 355

Triesch, Eberhard 113

Uehara, Ryuhei 28, 133

Uno, Takeaki 133

Uno, Yushi 133

Williams, Aaron 343, 368

Woeginger, Gerhard J. 178

Wolfvitz, Guy 37