8th International Conference on Fun with Algorithms

FUN 2016, June 8–10, 2016, La Maddalena, Italy

Edited by

Erik D. Demaine
Fabrizio Grandoni

Editors

Erik D. Demaine	Fabrizio Grandoni
MIT, CSAIL	IDSIA, USI-SUPSI
32 Vassar Street	Galleria 1
Cambridge, Massachusetts 02139	6928, Manno
USA	Switzerland
edemaine@mit.edu	fabrizio@idsia.ch

ACM Classification 1998
F.2.2 Nonnumerical Algorithms and Problems, G.2 Discrete Mathematics, F.1.3 Complexity Measures and Classes

ISBN 978-3-95977-005-7

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at http://www.dagstuhl.de/dagpub/978-3-95977-005-7.

Publication date
June, 2016

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available from the Internet at http://dnb.d-nb.de.

License
This work is licensed under a Creative Commons Attribution 3.0 Unported license (CC-BY 3.0): http://creativecommons.org/licenses/by/3.0/legalcode.

In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors' moral rights:

- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/LIPIcs.FUN.2016.i

ISBN 978-3-95977-005-7 **ISSN 1868-8969** **http://www.dagstuhl.de/lipics**

LIPIcs – Leibniz International Proceedings in Informatics

LIPIcs is a series of high-quality conference proceedings across all fields in informatics. LIPIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

Editorial Board

- Susanne Albers (TU München)
- Chris Hankin (Imperial College London)
- Deepak Kapur (University of New Mexico)
- Michael Mitzenmacher (Harvard University)
- Madhavan Mukund (Chennai Mathematical Institute)
- Catuscia Palamidessi (INRIA)
- Wolfgang Thomas (*Chair*, RWTH Aachen)
- Pascal Weil (CNRS and University Bordeaux)
- Reinhard Wilhelm (Saarland University)

ISSN 1868-8969

http://www.dagstuhl.de/lipics

Contents

Preface
Erik D. Demaine and Fabrizio Grandoni 0:vii–0:vii

2048 Without New Tiles Is Still Hard
Ahmed Abdelkader, Aditya Acharya, and Philip Dasler 1:1–1:14

Trainyard is NP-Hard
Matteo Almanza, Stefano Leucci, and Alessandro Panconesi 2:1–2:14

LOL: An Investigation into Cybernetic Humor, or: Can Machines Laugh?
Davide Bacciu, Vincenzo Gervasi, and Giuseppe Prencipe 3:1–3:15

Hanabi is NP-Complete, Even for Cheaters Who Look at Their Cards
Jean-Francois Baffier, Man-Kwun Chiu, Yago Diez, Matias Korman, Valia Mitsou, André van Renssen, Marcel Roeloffzen, and Yushi Uno 4:1–4:17

Selenite Towers Move Faster Than Hanoï Towers, But Still Require Exponential Time
Jérémy Barbay 5:1–5:20

Algorithms and Insights for RaceTrack
Michael A. Bekos, Till Bruckdorfer, Henry Förster, Michael Kaufmann, Simon Poschenrieder, and Thomas Stüber 6:1–6:14

Resource Optimization for Program Committee Members: A Subreview Article
Michael A. Bender, Samuel McCauley, Bertrand Simon, Shikha Singh, and Frédéric Vivien 7:1–7:20

Physical Zero-Knowledge Proofs for Akari, Takuzu, Kakuro and KenKen
Xavier Bultel, Jannik Dreier, Jean-Guillaume Dumas, and Pascal Lafourcade 8:1–8:20

Analyzing and Comparing On-Line News Sources via (Two-Layer) Incremental Clustering
Francesco Cambi, Pierluigi Crescenzi, and Linda Pagli 9:1–9:14

Spy-Game on Graphs
Nathann Cohen, Mathieu Hilaire, Nícolas A. Martins, Nicolas Nisse, and Stéphane Pérennes 10:1–10:16

The Complexity of Snake
Marzio De Biasi and Tim Ophelders 11:1–11:13

The Fewest Clues Problem
Erik D. Demaine, Fermi Ma, Ariel Schvartzman, Erik Waingarten, and Scott Aaronson 12:1–12:12

Super Mario Bros. is Harder/Easier Than We Thought
Erik D. Demaine, Giovanni Viglietta, and Aaron Williams 13:1–13:14

A Rupestrian Algorithm
Giuseppe A. Di Luna, Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Giovanni Viglietta 14:1–14:20

Building a Better Mouse Maze
Jessica Enright and John D. Faben ... 15:1–15:12

Recognizing a DOG is Hard, But Not When It is Thin and Unit
William Evans, Mereke van Garderen, Maarten Löffler, and Valentin Polishchuk . 16:1–16:12

Counting Circles Without Computing Them
Rudolf Fleischer ... 17:1–17:7

Large Peg-Army Maneuvers
Luciano Gualà, Stefano Leucci, Emanuele Natale, and Roberto Tauraso ... 18:1–18:15

Loopless Gray Code Enumeration and the Tower of Bucharest
Felix Herter and Günter Rote ... 19:1–19:19

Convex Configurations on Nana-kin-san Puzzle
Takashi Horiyama, Ryuhei Uehara, and Haruo Hosoya ... 20:1–20:14

How to Solve the Cake-Cutting Problem in Sublinear Time
Hiro Ito and Takahiro Ueda ... 21:1–21:15

Threes!, Fives, 1024!, and 2048 are Hard
Stefan Langerman and Yushi Uno ... 22:1–22:14

An Arithmetic for Rooted Trees
Fabrizio Luccio ... 23:1–23:14

Two Dots is NP-Complete
Neeldhara Misra ... 24:1–24:12

This House Proves That Debating is Harder Than Soccer
Stefan Neumann and Andreas Wiese ... 25:1–25:14

Preface

This book collects the refereed proceedings of the 8th International Conference on Fun with Algorithms (FUN) 2016, held on 8–10 June 2016 in La Maddalena, Sardinia, Italy.

It contains 25 articles carefully selected from 61 submissions. These works present original scientific contributions, and cover a variety of topics in the area of theoretical and applied Computer Science, including computational complexity of puzzles and (video)games, game theory, graph algorithms, distributed algorithms, graph theory, artificial intelligence, etc. In the spirit of FUN, all of these papers have in common some fun aspects, in terms of presentation and/or topic.

Erik D. Demaine
Fabrizio Grandoni

8th International Conference on Fun with Algorithms (FUN 2016).
Editors: Erik D. Demaine and Fabrizio Grandoni
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Conference Organization

Program Committee

Oswin Aichholzer, T. U. Graz, Austria
Michael Bender, SUNY Stony Brook, USA
Vincenzo Bonifaci, IASI, Italy
Jaroslaw Byrka, U. Wroclaw, Poland
Parinya Chalermsook, MPII, Germany
Mirela Damian, Villanova U., USA
Erik Demaine (Co-Chair), MIT, USA
Matthias Englert, U. Warwick, England
David Eppstein, U. California Irvine, USA
Jeff Erickson, U. Illinois, Urbana-Champaign, USA
Irene Finocchi, U. Sapienza, Italy
Fedor Fomin, U. Bergen, Norway
Pierre Fraigniaud, Paris Diderot, France
Fabrizio Grandoni (Co-Chair), IDSIA, Switzerland
Roberto Grossi, U. Pisa, Italy
Robert Hearn, USA
John Iacono, NYU Engineering, USA
Stefan Langerman, U. Libre Bruxelles, Belgium
Joseph Mitchell, SUNY Stony Brook, USA
Ian Munro, U. Waterloo, Canada
Mohit Singh, Microsoft, USA
Kavitha Telikepalli, TIFR, India
Ryuhei Uehara, JAIST, Japan
Yushi Uno, Osaka Prefecture U., Japan
Giovanni Viglietta, U. Ottawa, Canada
Sebastiano Vigna, U. Milano, Italy
Peter Widmayer, ETH, Switzerland
Virginia Vassilevska Williams, Stanford, USA

Steering Committee

Elena Lodi, University of Siena, Italy
Linda Pagli, University of Pisa, Italy
Nicola Santoro, Carleton University, Canada

Organizers

Linda Pagli, University of Pisa, Italy
Giuseppe Prencipe, University of Pisa, Italy

External Reviewers

Joshua Alman
Michael Biro
Andreas Bärtschi
Morgan Chopin
Alessio Conte
Ágnes Cseh
Gianlorenzo D'Angelo
Giuseppe Antonio Di Luna
Adrian Dumitrescu
Mohit Garg
Barbara Geissmann
Daniel Graf
Yan Gu
Thomas Hackl
Markus Holzer
Chien-Chung Huang
Takehiro Ito
Anissa Lamani
Hooyeon Lee
Mateusz Lewandowski
Andrea Lincoln
Florian Lorber
Akaki Mamageishvili
Andrea Marino
Yoshio Okamoto
Hirotaka Ono
Irene Parada
Ami Paz
Paolo Penna
Krzysztof Piecuch
Alexander Pilz
Giuseppe Prencipe
Toshiki Saitoh
Manfred Scheucher
Lena Schlipf
Shikha Singh
Krzysztof Sornat
Bettina Speckmann
Daniel Stubbs
Yihan Sun
Akira Suzuki
Przemysław Uznański
Luca Versari
Joshua Wang

List of Authors

Scott Aaronson
MIT Computer Science and Artificial Intelligence Laboratory
32 Vassar St, Cambridge, MA 02139, USA
aaronson@csail.mit.edu

Ahmed Abdelkader
Department of Computer Science
University of Maryland
College Park, Maryland 20742, USA
akader@cs.umd.edu

Aditya Acharya
Department of Computer Science
University of Maryland
College Park, Maryland 20742, USA
acharya@cs.umd.edu

Matteo Almanza
Rome, Italy
almanza.1597415@studenti.uniroma1.it

Davide Bacciu
Dipartimento di Informatica, Università di Pisa
Pisa, Italy
davide.bacciu@unipi.it

Michael A. Bekos
Wilhelm-Schickard-Institut für Informatik, Universität Tübingen
Tübingen, Germany

Michael A. Bender
Stony Brook University
Stony Brook, NY 11794-4400, USA
bender@cs.stonybrook.edu

Till Bruckdorfer
Wilhelm-Schickard-Institut für Informatik, Universität Tübingen
Tübingen, Germany

Xavier Bultel
LIMOS, University Clermont Auvergne, Campus des Cézeaux, Aubière, France
xavier.bultel@udamail.fr

Francesco Cambi
Bridge Consulting S.r.l.
Via L. Rosellini, Firenze, Italy 50127
fcambi@bridgeconsulting.it

Nathann Cohen
CNRS, Univ. Paris Sud, LRI
Orsay, France
nathann.cohen@lri.fr

Pierluigi Crescenzi
Dipartimento di Ingegneria dell'Informazione
Università degli Studi di Firenze
Viale Morgagni 65, Firenze, Italy 50134
pierluigi.crescenzi@unifi.it

Philip Dasler
Department of Computer Science
University of Maryland
College Park, Maryland 20742, USA
daslerpc@cs.umd.edu

Marzio De Biasi
Vittorio Veneto, Italy
marziodebiasi@gmail.com

Erik D. Demaine
MIT Computer Science and Artificial Intelligence Laboratory
32 Vassar St., Cambridge, MA 02139, USA
edemaine@mit.edu

Giuseppe Di Luna
School of Electrical Engineering and Computer Science, University of Ottawa
Ottawa, Canada
gdiluna@uottawa.ca

Yago Diez
Tohoku University
Sendai, Japan
yago@dais.is.tohoku.ac.jp

Jean-Guillaume Dumas
LJK, Université Grenoble Alpes, CNRS umr 5224
51 av. des Mathématiques, BP53, Grenoble, France 38041
Jean-Guillaume.Dumas@imag.fr

Jessica Enright
University of Stirling
Stirling, UK
jae@cs.stir.ac.uk

William Evans
University of British Columbia
Vancouver, Canada
will@cs.ubc.ca

Stefan Langerman
Département d'informatique, Université Libre de Bruxelles
ULB CP 212, avenue F.D. Roosevelt 50, 1050 Bruxelles, Belgium
stefan.langerman@ulb.ac.be

Henry Förster
Wilhelm-Schickard-Institut für Informatik, Universität Tübingen
Tübingen, Germany

John Faben
Glasgow, United Kingdom
jdfaben@gmail.com

Rudolf Fleischer
GUtech, Muscat, Oman; and
Fudan University, Shanghai, China
rudolf.fleischer@gutech.edu.om

Paola Flocchini
School of Electrical Engineering and Computer Science, University of Ottawa
Ottawa, Canada
flocchin@site.uottawa.ca

Mereke van Garderen
University of Konstanz, Germany
mereke.van.garderen@uni-konstanz.de

Vincenzo Gervasi
Dipartimento di Informatica, Università di Pisa
Pisa, Italy
davide.bacciu@unipi.it

Luciano Gualà
Università di Roma Tor Vergata
Rome, Italy
guala@mat.uniroma2.it

Felix Herter
Institut für Informatik, Freie Universität Berlin
Takustr. 9, Berlin, Germany 14195
avealx@zedat.fu-berlin.de

Mathieu Hilaire
ENS Cachan, France
mathieu-hilaire@hotmail.fr

Takashi Horiyama
Saitama University
Saitama, Japan

Haruo Hosoya
Ochanomizu University
Tokyo, Japan

Hiro Ito
School of Informatics and Engineering, The University of Electro-Communications (UEC)/Tokyo, Japan; and
CREST, JST/Tokyo, Japan
itohiro@uec.ac.jp

Michael Kaufmann
Wilhelm-Schickard-Institut für Informatik, Universität Tübingen
Tübingen, Germany

Matias Korman
Tohoku University
Sendai, Japan
mati@dais.is.tohoku.ac.jp

Maarten Löffler
Utrecht University
Utrecht, the Netherlands
m.loffler@uu.nl

Pascal Lafourcade
LIMOS, University Clermont Auvergne
Campus des Cézeaux, Aubière, France
pascal.lafourcade@udamail.fr

Stefano Leucci
Dipartimento di Informatica, "Sapienza" Università di Roma
Rome, Italy
leucci@di.uniroma1.it

Fabrizio Luccio
Dipartimento di Informatica, University of Pisa
Pisa, Italy
luccio@di.unipi.it

Fermi Ma
Department of Computer Science, Princeton University
35 Olden St, Princeton, NJ 08544, USA
fermim@princeton.edu

Nícolas A. Martins
Universidade Federal do Ceará
Fortaleza, Brazil
nicolasamartins@gmail.com

Samuel McCauley
Stony Brook University
Stony Brook, NY 11794-4400, USA
smccauley@cs.stonybrook.edu

Neeldhara Misra
Indian Institute of Technology
Gandhinagar, India
mail@neeldhara.com

Valia Mitsou
SZTAKI, Hungarian Academy of Sciences
Budapest, Hungary
vmitsou@sztaki.hu

Emanuele Natale
Sapienza Università di Roma
Rome, Italy
natale@di.uniroma1.it

Stefan Neumann
Faculty of Computer Science, University of Vienna
Vienna, Austria
stefan.neumann@univie.ac.at

Tim Ophelders
Department of Mathematics and Computer Science, TU Eindhoven
Eindhoven, the Netherlands
t.a.e.ophelders@tue.nl

Linda Pagli
Dipartimento di Informatica, Università degli Studi di Pisa
Largo B. Pontecorvo 3, 56127 Pisa, Italy
linda.pagli@unipi.it

Alessandro Panconesi
Dipartimento di Informatica, “Sapienza” Università di Roma
Rome, Italy
ale@di.uniroma1.it

Valentin Polishchuk
Linköping University
Linköping, Sweden
valentin.polishchuk@liu.se

Simon Poschenrieder
Wilhelm-Schickard-Institut für Informatik, Universität Tübingen
Tübingen, Germany

Giuseppe Prencipe
Dipartimento di Informatica, Università di Pisa
Pisa, Italy
davide.bacciu@unipi.it

Günter Rote
Institut für Informatik, Freie Universität Berlin
Takustr. 9, 14195 Berlin, Germany
rote@inf.fu-berlin.de

Nicola Santoro
School of Computer Science, Carleton University
Ottawa, Canada
santoro@scs.carleton.ca

Ariel Schvartzman
Department of Computer Science, Princeton University
35 Olden St, Princeton, NJ 08544, USA
acohenca@cs.princeton.edu

Bertrand Simon
Univ. Lyon, LIP, CNRS – ENS de Lyon – INRIA
Lyon, France 69007
simon@inria.fr

Shikha Singh
Stony Brook University
Stony Brook, NY 11794-4400, USA
shiksingh@cs.stonybrook.edu

Thomas Stüber
Wilhelm-Schickard-Institut für Informatik, Universität Tübingen
Tübingen, Germany

Roberto Tauraso
Università di Roma Tor Vergata
Rome, Italy
tauraso@mat.uniroma2.it

Takahiro Ueda
Komatsu Ltd.
Tokyo, Japan
mx.u.2147483647@gmail.com

Ryuhei Uehara
Japan Advanced Institute of Science and Technology
Ishikawa, Japan

Yushi Uno
Department of Mathematics and Information Sciences, Graduate School of Science, Osaka Prefecture University
uno@mi.s.osakafu-u.ac.jp

Giovanni Viglietta
School of Electrical Engineering and Computer Science, University of Ottawa
Ottawa, Canada
viglietta@gmail.com

Frédéric Vivien
Univ. Lyon, LIP, CNRS – ENS de Lyon – INRIA
Lyon, France 69007
frederic.vivien@inria.fr

Erik Waingarten
Department of Computer Science, Columbia University
1214 Amsterdam Ave, New York, NY 10027, USA
eaw@cs.columbia.edu

Andreas Wiese
Max Planck Institute for Computer Science
Saarbrücken, Germany
awiese@mpi-inf.mpg.de

Aaron Williams
Division of Science, Mathematics, and Computing, Bard College at Simon's Rock
84 Alford Rd, Great Barrington, MA 01230, USA
haron@uvic.ca

2048 Without New Tiles Is Still Hard

Ahmed Abdelkader[1], Aditya Acharya[2], and Philip Dasler[3]

1 **Department of Computer Science, University of Maryland, College Park, Maryland 20742, USA**
`akader@cs.umd.edu`

2 **Department of Computer Science, University of Maryland, College Park, Maryland 20742, USA**
`acharya@cs.umd.edu`

3 **Department of Computer Science, University of Maryland, College Park, Maryland 20742, USA**
`daslerpc@cs.umd.edu`

Abstract

We study the computational complexity of a variant of the popular *2048* game in which no new tiles are generated after each move. As usual, instances are defined on rectangular boards of arbitrary size. We consider the natural decision problems of achieving a given constant tile value, score or number of moves. We also consider approximating the maximum achievable value for these three objectives. We prove all these problems are NP-hard by a reduction from 3SAT.

Furthermore, we consider potential extensions of these results to a similar variant of the *Threes!* game. To this end, we report on a peculiar motion pattern, that is not possible in *2048*, which we found much harder to control by similar board designs.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Complexity of Games, 2048

Digital Object Identifier 10.4230/LIPIcs.FUN.2016.1

1 Introduction

2048 is a single-player online puzzle game that went viral in March 2014. The game is played on a 4×4 board as in Figure 1. Each turn, the player picks a move from $\{\leftarrow, \rightarrow, \uparrow, \downarrow\}$ to slide all tiles on the board. Tiles slide as far as possible in the chosen direction until they hit either another tile or an edge of the board. When a sliding tile runs into a stationary one of equal value, they merge into a tile of double this value. Trailing tiles following a tile that just merged continue to slide uninterrupted and may merge among themselves as they come to rest one after the other. However, newly merged tiles cannot merge again in the same move. After each move, a *2* or *4* tile is generated in one of the empty cells. The player wins when a *2048* tile is created, hence the name of the game. Otherwise, the player loses when the board is full and no merges can be performed.

2048 combines features from two families of games: Candy Crush Saga [6] and PushPush [5]. Prior to our work, an attempt to prove that *2048* is PSPACE-complete appeared in [9]. On the other hand, a proof of membership in NP appeared in a blog post by Christopher Chen [4], which applies to the games we study in this paper. The first draft of this work was made available in [3] and a revised version was presented in the Computational Geometry: Young Researchers Forum (CG:YRF) [2], held in conjunction with The 31st Symposium on Computational Geometry. In the meanwhile, another draft came out by Langerman and Uno

© Ahmed Abdelkader, Aditya Acharya, and Philip Dasler;
licensed under Creative Commons License CC-BY
8th International Conference on Fun with Algorithms (FUN 2016).
Editors: Erik D. Demaine and Fabrizio Grandoni; Article No. 1; pp. 1:1–1:14
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Figure 1 The result of taking the **Down** action (↓). Notice that the two *4*s on the right have merged and a new *2* has been randomly inserted.

[7] establishing the NP-completness of the original *2048* game, with new tiles generated after each move. Their construction extends easily to similar games including *Threes!*.

The game we analyze differs from the original *2048* game in the following two aspects: (1) The input encodes the complete board configuration and no new tiles are generated. (2) The board is a rectangular grid of arbitrary size. In this paper, we are primarily concerned with the following decision problem:

▶ **Definition 1.** (`2048-TILE`) Given a configuration of tiles on an $m \times n$ board, is it possible to obtain a tile of value 2048? (More generally, 2^k for a constant integer $k \geq 8$.)

Our construction can be augmented to study two related decision problems: `2048-SCORE` and `2048-MOVES`. The former asks if it is feasible to achieve a given score and the latter asks if it is feasible to achieve a given number of moves. As in [8], we define the score as the sum of all new tiles the player creates by merges during the game. In our setting, the number of moves is taken to mean the number of effective moves that change the board by merging at least two tiles.

The crucial piece of proving membership in NP is to bound the number of moves between two consecutive merges. Using a canonical orientation [4], all moves are interpreted as flips of the board, which send tiles along orbits of $O(mn)$ length. A pair of tiles that end up merging requires no more than $LCM(O(mn), O(mn)) = O(m^2n^2)$ moves.

In this paper, we prove NP-hardness by a reduction from 3SAT and obtain the main result.

▶ **Theorem 2.** `2048-TILE` *is NP-Complete.*

Using the same reduction, we obtain similar results for both `2048-SCORE` and `2048-MOVES`. Furthermore, encouraged by the inapproximability results in [8] for the maximization version of these problems, where a new tile is generated after each move, we show similar results in our setting without new tiles. We implemented our gadgets and full reduction as an online game to aid the presentation [1].

2 Reduction from 3SAT

Given an instance of `3SAT` with n variables and m clauses, we produce an instance of `2048-TILE`. The board is filled using a 2-4 lattice to provide a rigid base for placing gadgets and planning their movements. We allow no merges using lattice tiles, which requires preserving their parity. This confines all merges to *blocks*, where a block is defined as a 2×2 arrangement of tiles. We typically use the words *row* and *column* to denote two consecutive

rows or columns, respectively. In devising the gadgets presented here, we experimented with different lattice patterns before we settled for this one. We would like to note however that the 2-4 lattice was first described in [9].

In the rest of this section, we describe the gadgets we use in the reduction. A full annotated reduction is shown in Figure 2.

2.1 Displacers

These are the building blocks of all gadgets which allow us to communicate signals across the board. They come in two main forms: horizontal D and vertical D^T. Typically, a displacer starts in an *inactive* state where the middle 2×2 block, highlighted below, is shifted by one (or two) blocks along the axis orthogonal to the displacer's axis of action. An inactive displacer cannot merge, by any sequence of moves, before it is activated. The only way to activate a displacer is to use another properly aligned displacer to engage its middle block. Collapsing tiles in a displacer shrinks it to a single block, which results in a *parity-preserving pull* in a *row* or a *column*.

$$D = \left[\begin{array}{c|cc|c} 8 & 8 & 16 & 16 \\ 32 & 32 & 64 & 64 \end{array} \right]$$

2.2 Variable Gadget

Each variable is represented by two horizontal displacers on the same *row*. This enables variables to move the portion of its *row* between their two displacers to the right or left. We enforce the assignment of variables in the order of their indices. A variable is assigned T or F using a $\rightarrow$ or $\leftarrow$ move, respectively. The displacers of x_0 come activated in the initial configuration to allow the game to start.

To activate the variable gadget of x_{i+1}, two *connector displacers* are placed in the *row* of x_i. These connectors are activated regardless of the chosen truth assignment of x_i and allow the two displacers of x_{i+1} above them to be activated by a $\downarrow$ move in the following turn.

The variable gadgets of x_i for $i \in \{0, 1, 2, 3\}$ are annotated in Figure 2. Note that each variable has one gadget on the left and another on the right.

2.3 Clause and Literal Gadgets

A clause occupies a single *column* with a distinguished block near the top. Satisfying the clause corresponds to pulling down this block, which will be made possible by literal gadgets in the center of the reduction. Clause gadgets can be seen at the top of Figure 2.

Literals are encoded using a similar mechanism to the connectors in the variable gadget, but are only activated by the appropriate assignment. This is achieved by a *connector lattice*. Each active literal is a vertical displacer. Observe that both permutations of the columns of such a displacer can equally perform the required downward pull. We call this the *parity* of the displacer. The reduction uses the appropriate parity to distinguish positive and negative literals. A displacer may only be activated by providing a middle block of the same parity. As such, the displacers of positive (negative) literals will have positive (negative) parity and will only be activated by positive (negative) blocks in the connector lattice when the variable in question is assigned `True` (`False`) by a $\rightarrow$ ($\leftarrow$) move. For each clause *column*, a complete literal displacer is only included for each of the three rows corresponding to the three variables of the literals making up each clause. Otherwise, the connector blocks will have no effect on this *column*.

With variables as *rows* and clauses as *columns*, literal gadgets are situated exactly at the intersection of these *rows* and *columns* to communicate the relevant signals. This is easily seen at the center of Figure 2.

2.4 Key-Lock Gadget

To check that all clauses are satisfied, it helps to arrange for a special event to happen only after all variables have been assigned. To achieve this, an auxiliary variable $x_{aux} = x_n$ activates the lock portion of this gadget. Satisfying all clauses corresponds to using the correct key. Together, the activated key-lock gadget is a sequence of displacers that can activate a distinguished displacer with two special 2^{k-1} tiles. Collapsing this distinguished displacer creates the desired 2^k tile.

The lower portion of the lock gadget is composed of a sequence of $m+1$ vertical displacers that can only be activated by x_{aux}. When activated, the lock, in turn, activates a sequence of up to m horizontal displacers, at the upper portion of the gadget, utilizing the blocks of satisfied clauses.

As the lock gadget overlaps all clause *columns*, it must not be affected when any one clause is satisfied. To achieve this, the middle blocks of the vertical displacers in the lock gadget are shifted by two blocks away. This means that each middle block for such a displacer will be nestled in the displacer next to it. In order for this to work, the parity of these displacers alternate such that they are only activated by the displacers of x_{aux}.

The lower portion of the lock gadget is aligned with the rows of x_{aux}, as can be seen in Figure 2. The distinguished block lies to the right on the *row* where satisfied clause blocks are pulled.

2.5 Core and Padding

All gadgets live in the center of the board produced by the reduction, which we call the *core*. A crucial invariant for our reduction to work is that any row or column may not move unless it has an active displacer. This requires that any gaps created by such displacers are immediately collected away on the next move.

To this end, the core is surrounded by a *padding* of tiles on both axes. This padding ensures that any number of gaps produced by merging tiles in such a reduction can be completely isolated in one corner away from the rows and columns containing any gadget.

Keeping in mind that the board produced by the reduction is initially full, a gap is created iff two tiles merge. As the number of active gadgets is $\Theta(m+n)$ and each gadget contributes a constant number of gaps, a padding of $\Theta(m+n)$ thickness suffices.

2.6 Properties of the Reduction

Size: As variables are stacked on top of each other all the way up to x_{aux} and the key-lock gadget, the number of rows is $\Theta(n)$. Then, each variable has to activate the connectors to the next variable. We get a pyramid shape with variable displacers on both sides and literals in the middle, plus the unique displacer taking up $2(m+1)$ columns far to the right, for a total of $\Theta(m+n)$ columns. It follows that the total size of the reduction is $\Theta(n(m+n))$.

Game Play: When no merges happen, two consecutive moves in opposite directions leave the board unchanged, e.g., $[\leftarrow, \rightarrow, \leftarrow]$ is effectively reduced to $[\leftarrow]$. *Effective* moves alternate between horizontal and vertical. The alternation accumulates newly created gaps, resulting

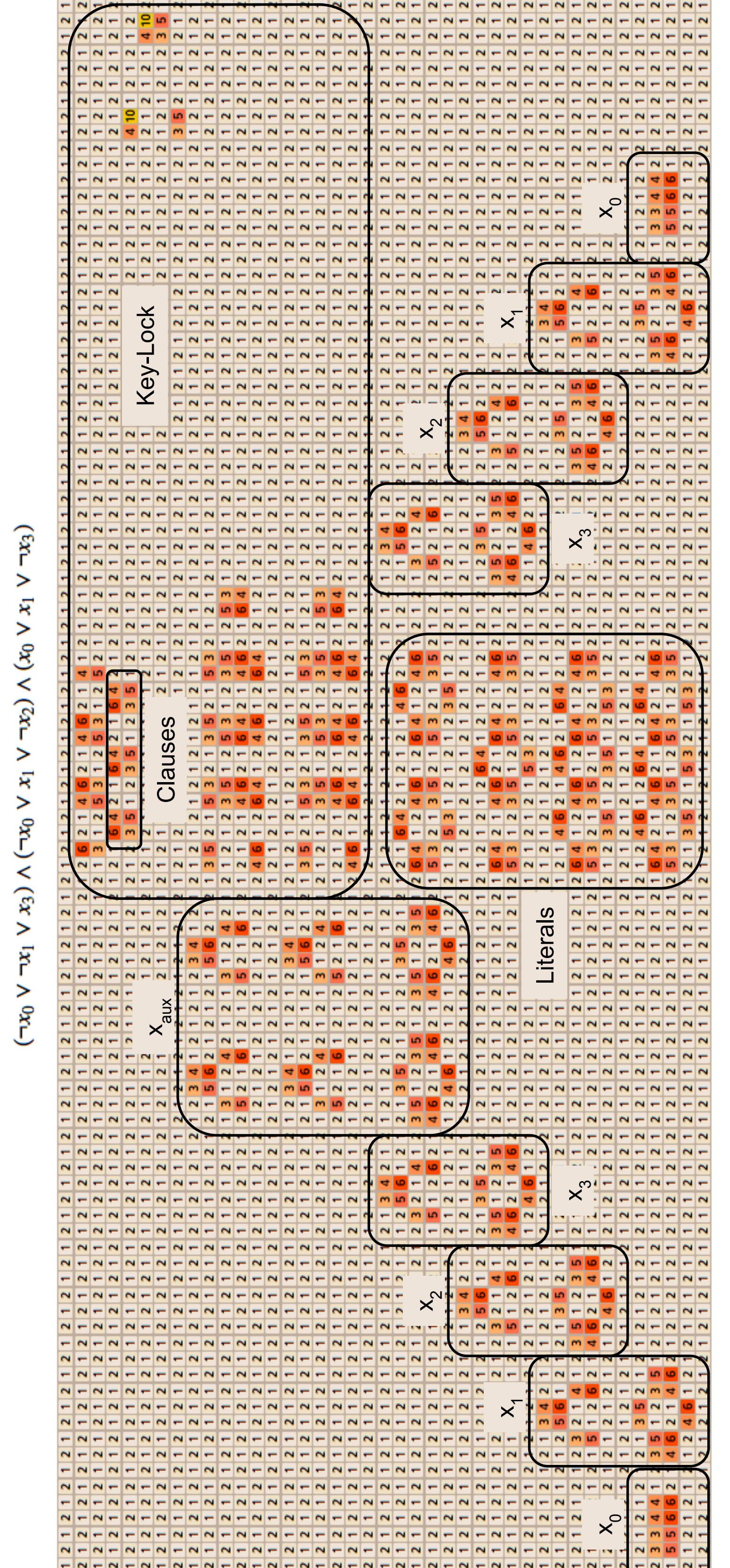


Figure 2 Annotated reduction. Only x_0 is active. We apply $\log_2$ and hide paddings to help display a large board. Note that this is only the core of the reduction, padding is not shown.

from the merge, at the corners so the decision encoded by the previous move cannot be altered. Furthermore, any row or column may witness merges during at most one turn. In particular, clause columns cannot experience more than one ↓ pull. This implies consistent assignments. Finally, ↑ moves are useless since they must be canceled or otherwise the player cannot win. Figures 6 through 17 show a complete play sequence through the reduction in Figure 2.

Hardness: Aligning the two 2^{k-1} tiles requires a $2m$ shift, which only satisfied clauses can provide with each satisfied clause contributing 2. Hence, the 2^k tile can be created iff the `3SAT` instance is satisfiable. This proves Theorem 2.

Furthermore, regardless of the truth assignment of variables, the player can always activate and collapse all variable gadgets, including x_{aux}, and the lower portion of the lock gadget. Doing so takes $2(n+1)$ effective moves. If any clauses were satisfied by the chosen assignment, the player will be able to perform one additional move and collapse up to m horizontal displacers in the upper portion of the lock gadget. Only if all m displacers were collapsed will the player be able to make one last move and collapse the special displacer. In such a winning sequence, there will be a total of $4n + 3m + 5$ active displacers plus 1 special displacer. Merging all these displacers takes $2(n + 2)$ moves and creates a set of new tiles with a total score $(2^4 + 2^5 + 2^6 + 2^7)(4n + 3m + 5) + (2^4 + 2^5 + 2^6 + 2^k)$. Again, this is possible iff the `3SAT` instance is satisfiable. It follows that `2048-MOVES` and `2048-SCORE` are both NP-complete.

3 Inapproximability

Rather than placing a 2^{k-1} tile in the special displacer, we can use a normal displacer that activates a *pot of gold* gadget. In this section, we present two such gadgets: one for `MAX-2048-MOVES` and another for both `MAX-2048-TILE` and `MAX-2048-SCORE`. The size of the pot will be controlled by a parameter S. We let $K = n + m$, so the size of the `3SAT` reduction is $N_0 = O(K^2)$.

3.1 Pot of Moves

A sequence of horizontal and vertical displacers can be added on top of the distinguished displacer as in Figure 3. Adding S such displacers allows S more moves.

Setting $S = 2^K$, the input size will be $N = \Theta((K+S)^2)$. If the `3SAT` instance is satisfiable, the player can make $S = \sqrt{N} - O(\log N)$ moves, and $O(K) = O(\log N)$ otherwise. It follows that it is NP-hard to approximate `MAX-2048-MOVES` within a factor of $o(\sqrt{N}/\log N)$.

3.2 Pot of Value

Parity constraints are rather restrictive to allow merging a large number of blocks into one another, as required to create a tile of arbitrarily high value from a collection of constant value tiles. Instead, we opt to create a containment gadget where parity can be violated only inside it without disturbing the rest of the reduction. This allows us to align a large number of tiles into a single column where they can be collapsed into a single tile.

The design principle used here is to make two merges in a single column each of making an offset of exactly one. This means that the portion of the column between the two merges would experience an odd offset, altering its parity, while everything else in the board is unchanged. The same can be done for rows. Observe however that such a shift exposes an

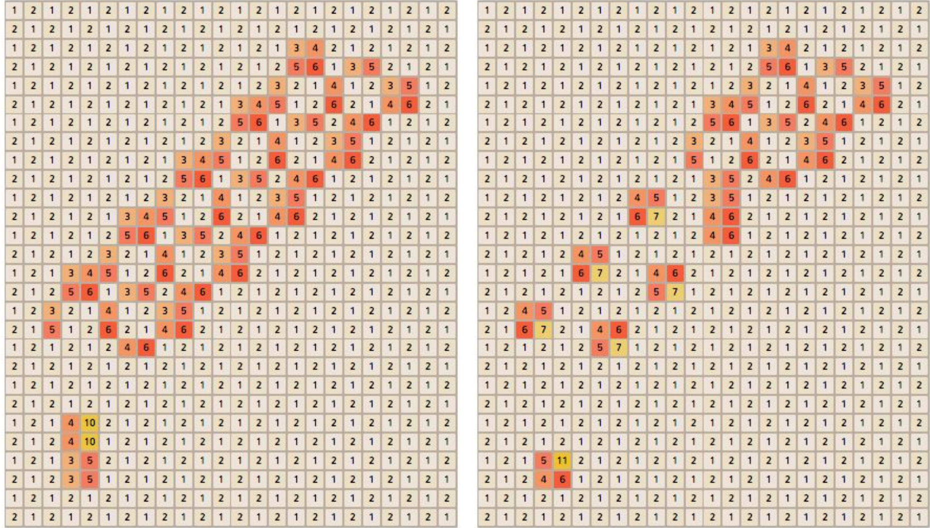


Figure 3 The Pot of Moves on top of the distinguished displacer (left) and after 6 moves (right).

even length portion of the neighboring rows and columns on both sides which have the same parity. To disallow such potential merges in lattice tiles, we need to make these shifts to an even number of consecutive rows and columns.

An example of such a gadget is shown in Figure 4. For $S = 2^p$, the gadget allows $p = \Theta(\log S)$ consecutive merges starting with 2^c tiles, for a constant c, and ending with tiles of value $2^{p+c} = \Theta(S)$. The total value of merged tiles add up to $\Theta(S)$ and the size of the augmented board will be $\Theta(K(K + S))$.

Setting $S = 2^K$, the input size will be $N = \Theta(K^2 + K2^K)$. If the `3SAT` instance is satisfiable, the player can create a tile of value $S = N/O(\log N) - O(\log^2 N)$, and 2^c otherwise, for a constant c. It follows that it is NP-hard to approximate `MAX-2048-TILE` within a factor of $o(N/\log N)$. Using the same parameters, if the `3SAT` instance is satisfiable, the player can achieve a score $S = N/O(\log N) - O(\log^2 N)$, and $O(K) = O(\log N)$ otherwise. It follows that it is NP-hard to approximate `MAX-2048-SCORE` within a factor of $o(N/\log^2 N)$.

4 The case for *Threes!*

For our purposes, the key difference between *Threes!* and *2048* is that in *Threes!*, tiles only move one step at a time instead of sliding all the way till they cannot go any further. It turns out that the difficulty of controlling such tiles comes from the presence of gaps. As in *2048*, gaps are created after each merge, but unlike *2048* where they are consumed on the next effective move, the gaps in *Threes!* persist for multiple moves. This allows the player to move these gaps to different locations creating a series of nontrivial shifts in the board.

We attempted a similar approach for *Threes!* in [3] and hoped that by spreading out the gadgets such a gap cannot travel from the row or column of the gadget that created it to a different row or column containing another gadget. For example, this may allow the player to make inconsistent truth assignments by altering a previously committed assignment of some variables or directly satisfying some clause without true literals. However, upon further examination we realized that the behavior of such gaps is richer than we thought.

For such a board game, a reduction like the one we use for *2048* alternates horizontal and vertical merges to communicate signals between the different gadgets. For *2048*, we did

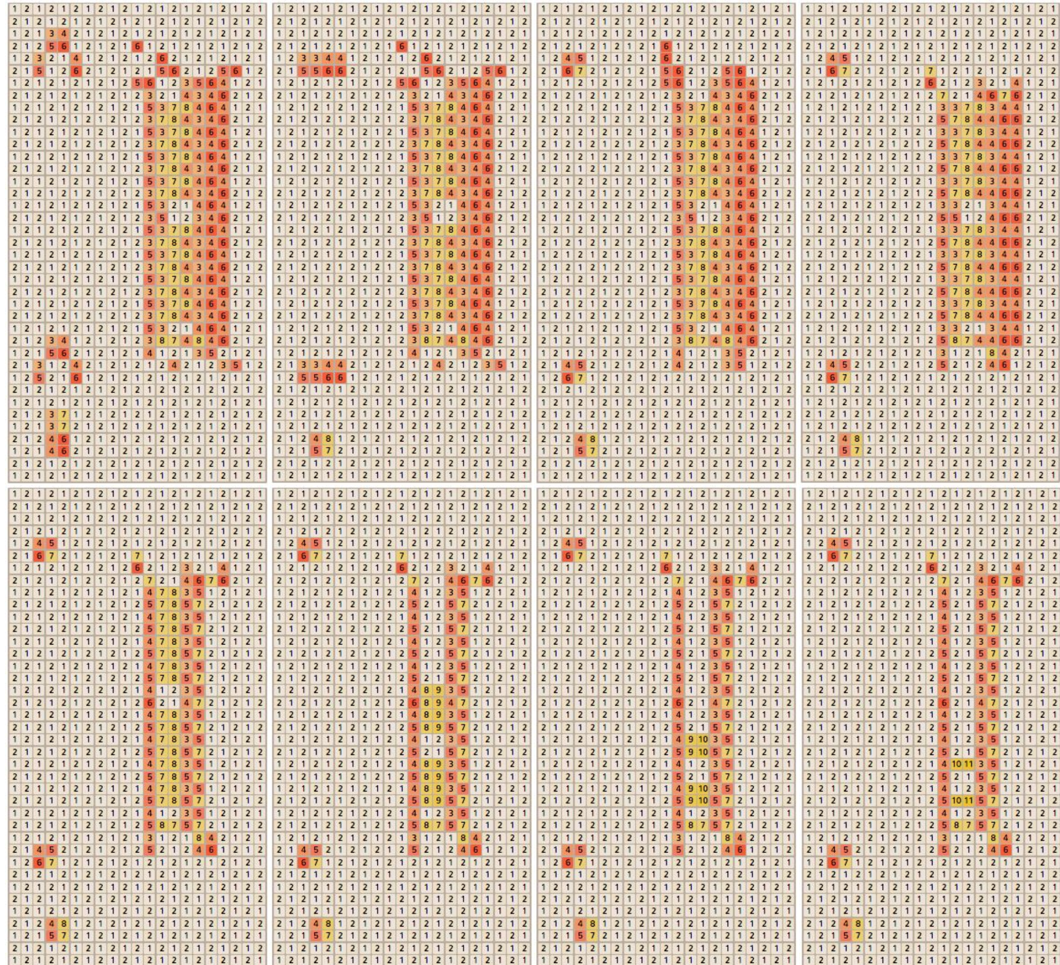


Figure 4 The Pot of Value: activation sequence and subsequent merges into a large value.

not need to keep track of how many gaps are created after each move or how to get rid of them. One possible solution that comes to mind is to create at most one gap per move and activate gadgets one by one as in [8]. However, this does not seem to resolve the issue as the example we discuss here shows.

Figure 5 shows the simplest such scenario. With just two gaps, one on the top side and one on the left side, the player can start a sequence of *clockwise moves*, cycling through [←, ↑, →, ↓]. One such full cycle is denoted by a winding arrow. This in turn creates two touching cycles of tiles that partition the board into two islands. Adding more gaps on either side results in a grid of islands separated by touching cycles.

One would need to show that the player gains nothing from such cycles. However, as the example shows, tiles that are initially far apart can come together after a number of rounds. The constraints on the lengths of such possible cycles and the safe locations of gadgets call for a more formal approach to the design. The example also shows an island that is not affected by the cycle. It would be interesting to explore whether one can exploit such protected islands to create gadgets that are easier to control and argue about.

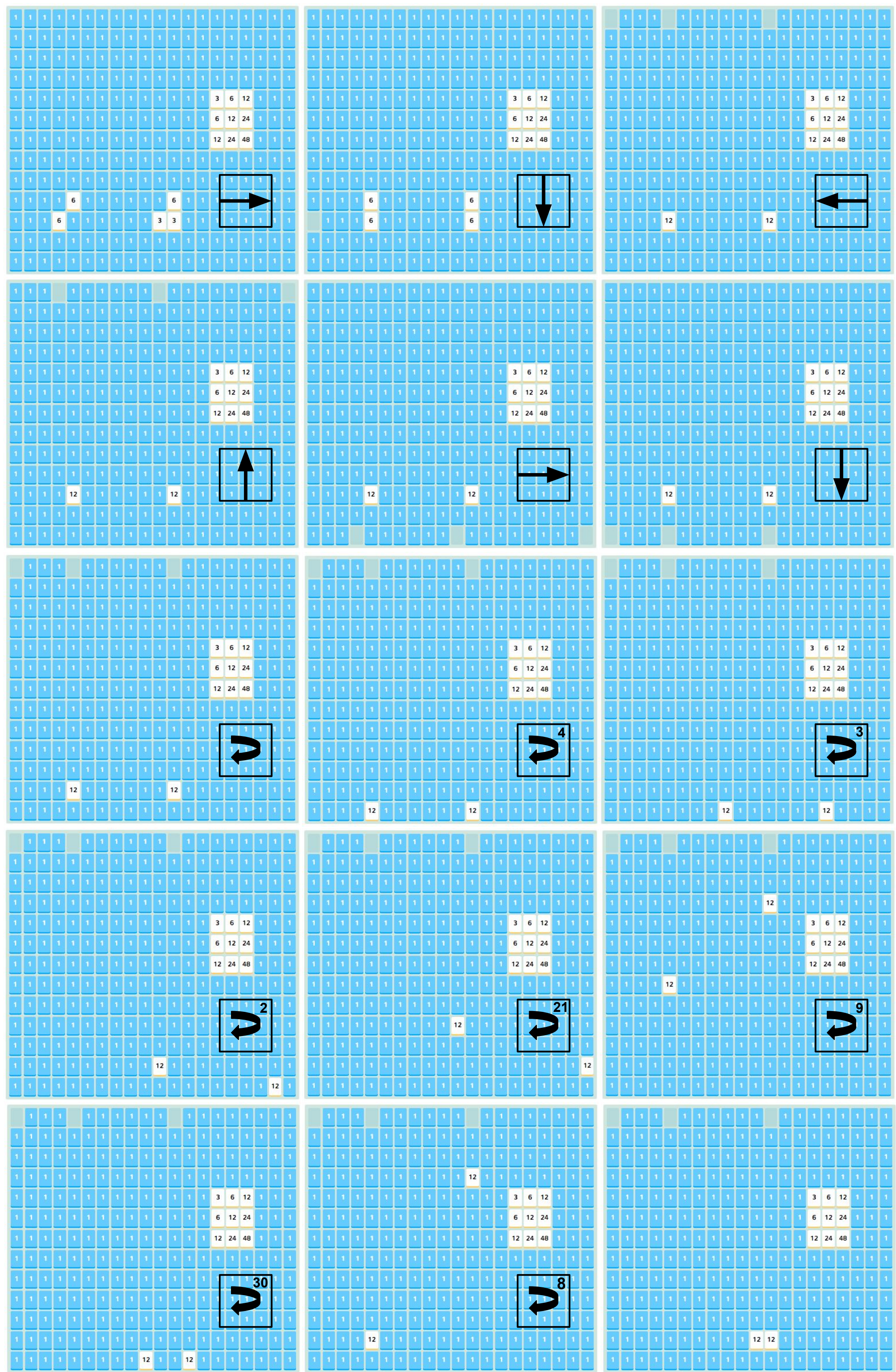


Figure 5 An example of curious motion patterns in the variant of *Threes!* without new tiles.

Acknowledgements. The authors would like to thank William Gasarch, Rahul Mehta, Erik Demaine and Stefan Langerman for helpful discussions through this work. The authors are especially grateful to MohammadTaghi Hajiaghayi for teaching the class that provided the opportunity to work on this project. The authors wish to thank the reviewers of the Computational Geometry: Young Researchers Forum (CG:YRF) for their valuable input that helped improve the presentation. Finally, the authors acknowledge Zhao HG and Angela Li for their open source implementations of *2048* and *Threes!*, respectively, and Christopher Chen for his proof.

References

1. Ahmed Abdelkader. 2048 gadgets. `http://cs.umd.edu/~akader/projects/2048/index.html`.
2. Ahmed Abdelkader, Aditya Acharya, and Philip Dasler. 2048 is NP-Complete. *CGYRF*, 2015.
3. Ahmed Abdelkader, Aditya Acharya, and Philip Dasler. On the Complexity of Slide-and-Merge Games. *CoRR*, abs/1501.03837, 2015. URL: `http://arxiv.org/abs/1501.03837`.
4. Christopher Chen. 2048 is in NP. `http://blog.openendings.net/2014/03/2048-is-in-np.html`.
5. Erik D. Demaine, Martin L. Demaine, and Joseph O'Rourke. PushPush is NP-hard in 2D. *CoRR*, cs.CG/0001019, 2000. URL: `http://arxiv.org/abs/cs.CG/0001019`.
6. Luciano Guala, Stefano Leucci, and Emanuele Natale. Bejeweled, Candy Crush and other Match-Three Games are (NP-)Hard. In *2014 IEEE Conference on Computational Intelligence and Games*, pages 1–8. IEEE, 2014.
7. Stefan Langerman and Yushi Uno. Threes!, Fives, 1024!, and 2048 are Hard. *CoRR*, abs/1505.04274, 2015. URL: `http://arxiv.org/abs/1505.04274`.
8. Stefan Langerman and Yushi Uno. Threes!, Fives, 1024!, and 2048 are Hard. In *8th International Conference on Fun with Algorithms (FUN 2016)*, volume 49 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:14, 2016.
9. Rahul Mehta. 2048 is (PSPACE) Hard, but Sometimes Easy. *CoRR*, abs/1408.6315, 2014. URL: `http://arxiv.org/abs/1408.6315`.

A Appendix

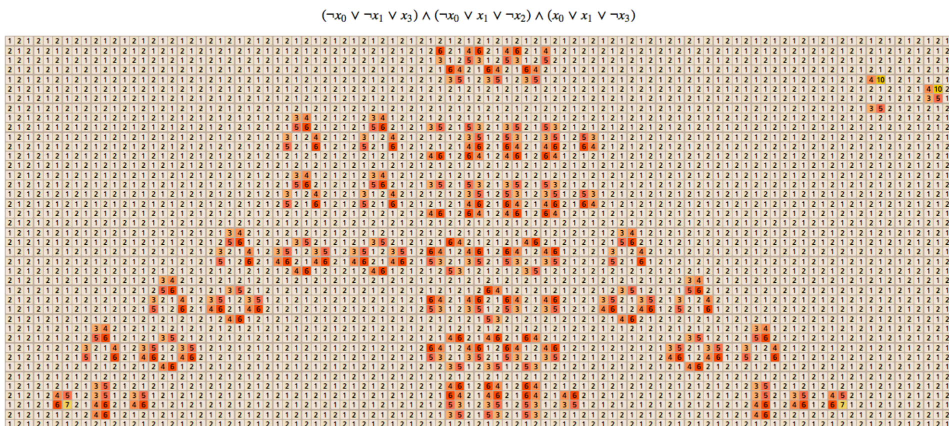


Figure 6 Board(1): *move*(→). x_0 assigned T.

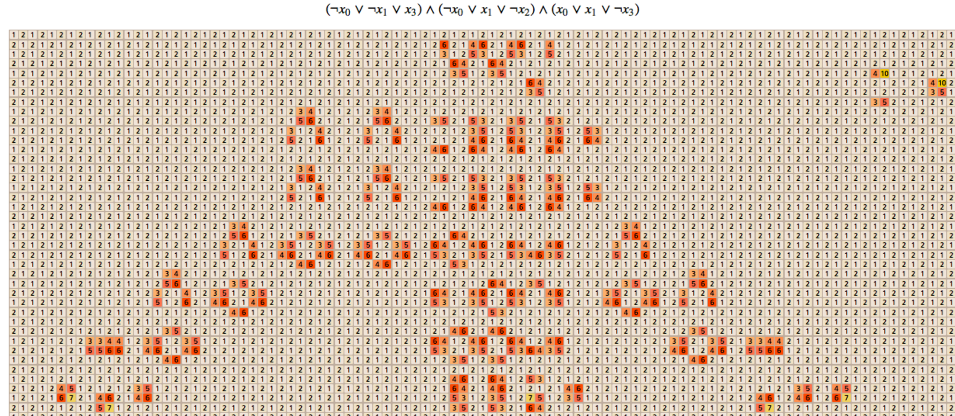


Figure 7 Board(2): $move(\downarrow)$. c_2 satisfied. x_0 fixed to T. x_1 activated.

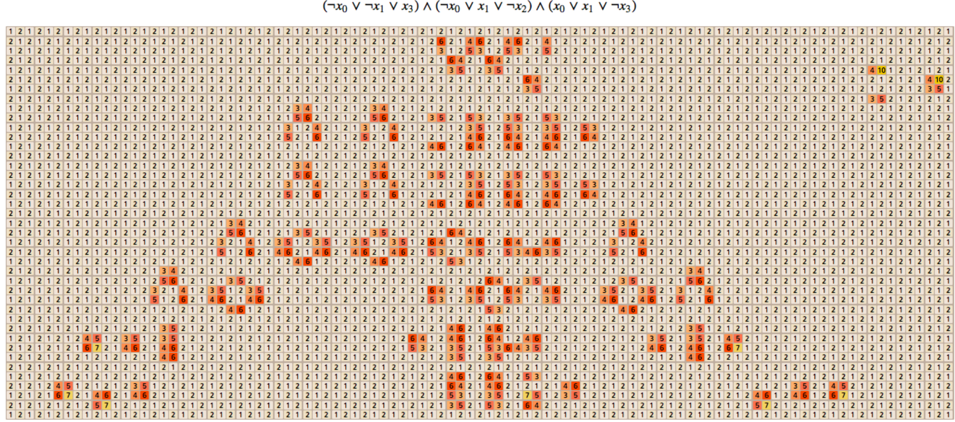


Figure 8 Board(3): $move(\leftarrow)$. x_1 assigned F.

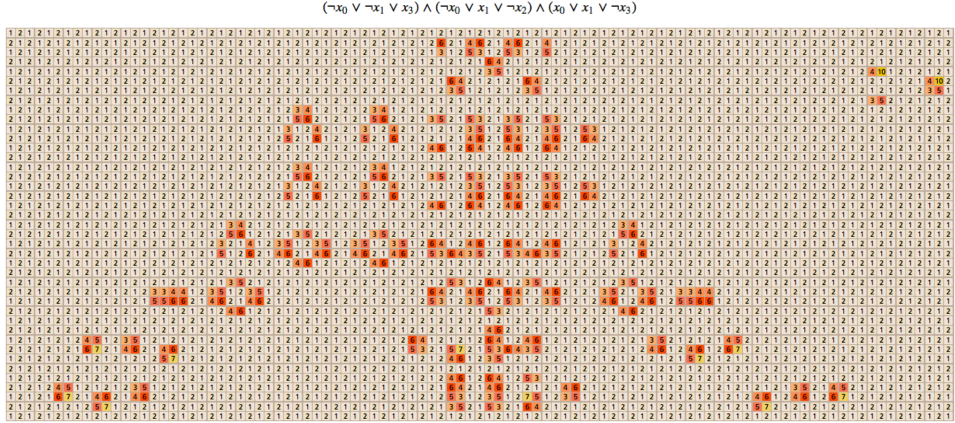


Figure 9 Board(4): $move(\downarrow)$. c_0 satisfied. x_1 fixed to F. x_2 activated.

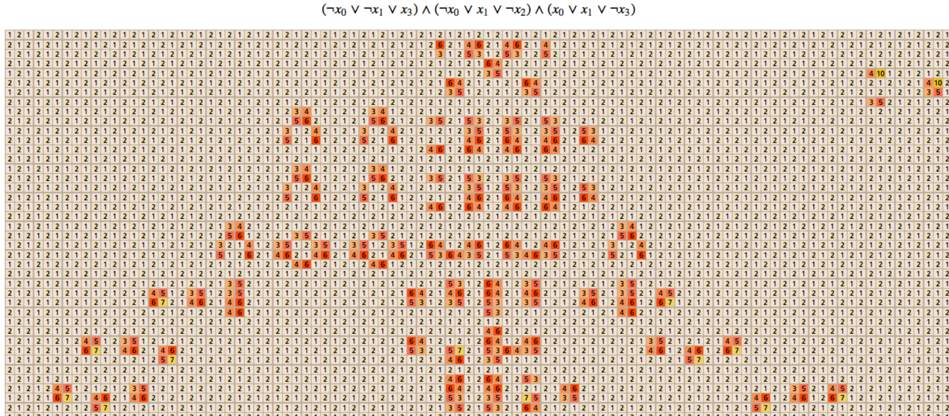


Figure 10 Board(5): $move(\leftarrow)$. x_2 assigned F.

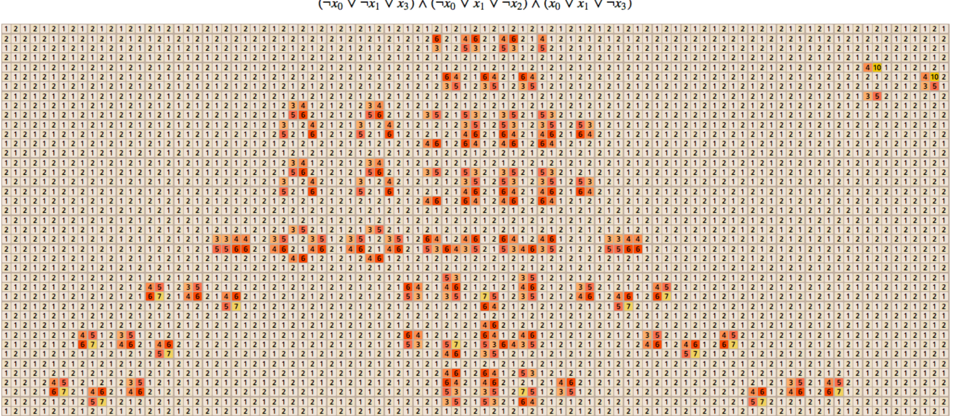


Figure 11 Board(6): $move(\downarrow)$. c_1 satisfied. x_2 fixed to F. x_3 activated.

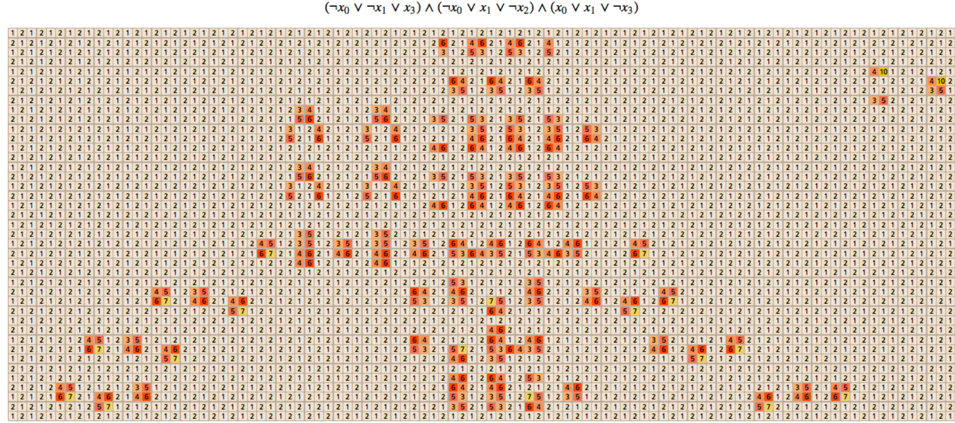


Figure 12 Board(7): $move(\rightarrow)$. x_3 assigned T.

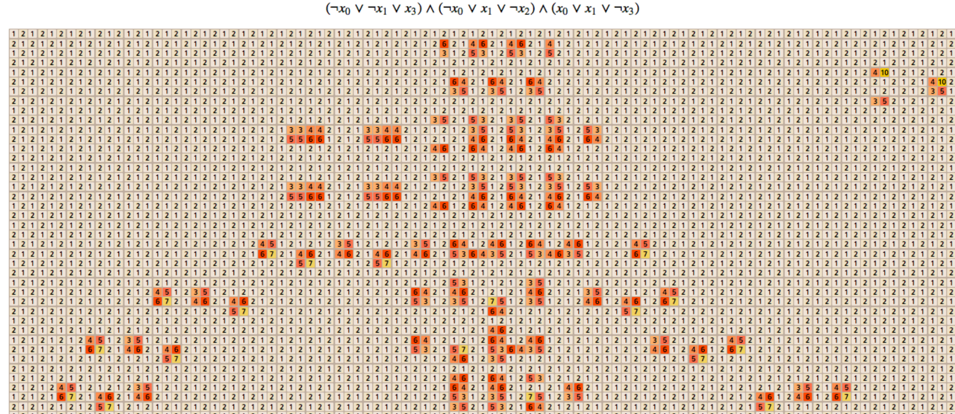


Figure 13 Board(8): $move(\downarrow)$. x_3 fixed to T. x_{aux} activated.

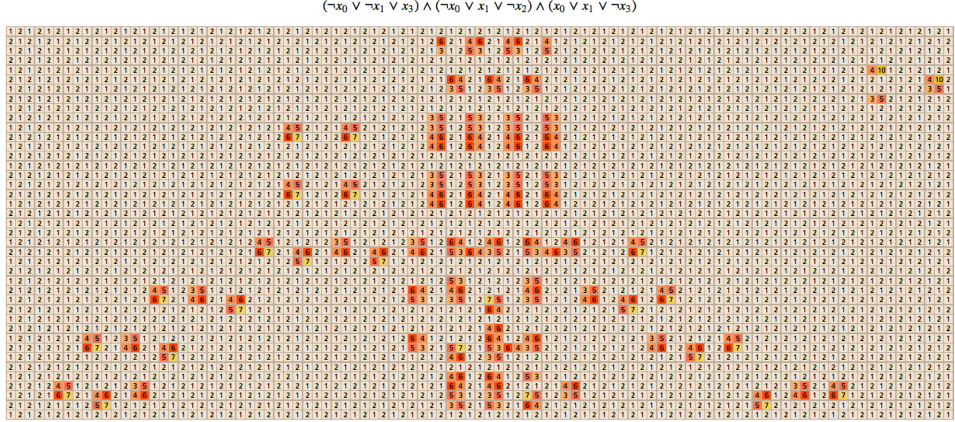


Figure 14 Board(9): $move(\leftarrow)$. Key-lock sequence initiated.

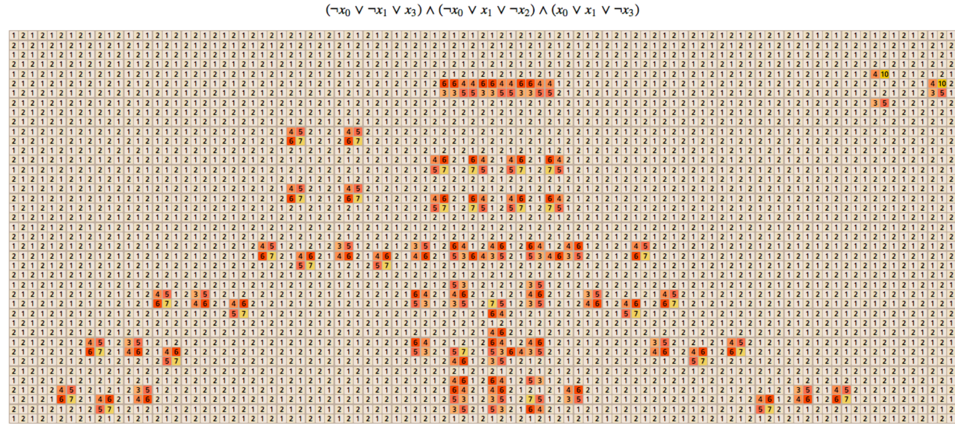


Figure 15 Board(10): $move(\downarrow)$. Key-lock ready.

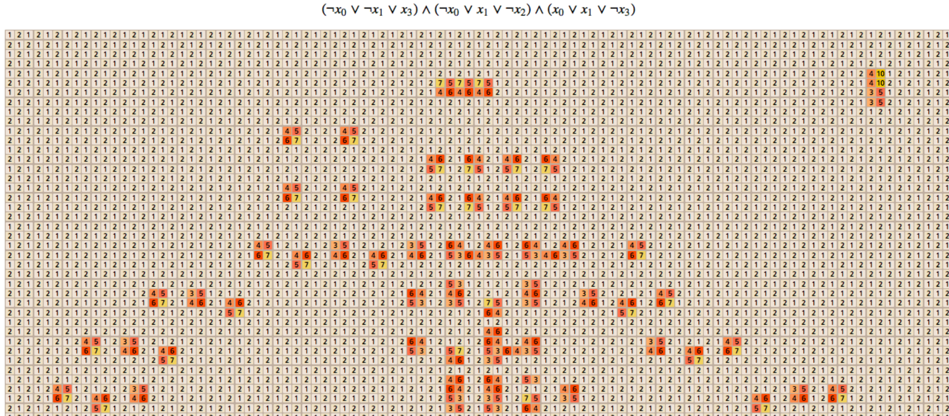


Figure 16 Board(11): $move(\leftarrow)$. All clauses satisfied? Unlock.

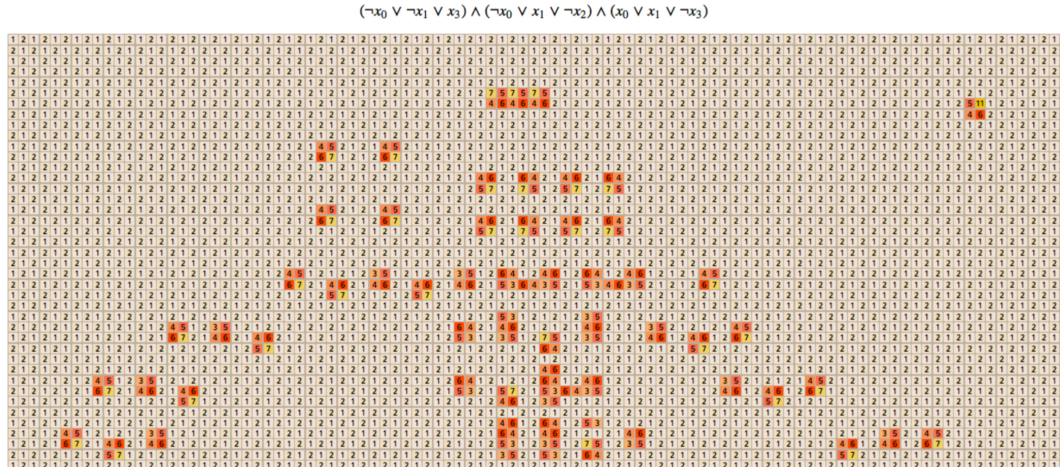


Figure 17 Board(12): $move(\downarrow)$. Win.

Trainyard is NP-Hard

Matteo Almanza[1], Stefano Leucci[2], and Alessandro Panconesi[3]

1 **Rome, Italy**
almanza.1597415@studenti.uniroma1.it
2 **Dipartimento di Informatica, "Sapienza" Università di Roma, Italy**
leucci@di.uniroma1.it
3 **Dipartimento di Informatica, "Sapienza" Università di Roma, Italy**
ale@di.uniroma1.it

Abstract

Recently, due to the widespread diffusion of smart-phones, mobile puzzle games have experienced a huge increase in their popularity. A successful puzzle has to be both captivating and challenging, and it has been suggested that this features are somehow related to their computational complexity [5]. Indeed, many puzzle games – such as Mah-Jongg, Sokoban, Candy Crush, and 2048, to name a few – are known to be NP-hard [3, 4, 7, 10]. In this paper we consider Trainyard: a popular mobile puzzle game whose goal is to get colored trains from their initial stations to suitable destination stations. We prove that the problem of determining whether there exists a solution to a given Trainyard level is NP-hard. We also provide an implementation of our hardness reduction.[1]

1998 ACM Subject Classification F.2.m Miscellaneous

Keywords and phrases Complexity of Games, Trainyard

Digital Object Identifier 10.4230/LIPIcs.FUN.2016.2

1 Introduction

The tension between human beings and machines in railroad building dates back to over a century ago, as the famous tale of John Henry testifies. According to the story, John Henry was a *steel-driving man* who challenged the efficiency of steam drill machines in a competition. Eventually John prevailed, reportedly outperfoming the rival machine in a contest that lasted more than one day. He unfortunately died of exhaustion shortly after his feat and is now remembered by a statue and a plaque next to the entrance of the Big Bend railroad tunnel in West Virginia. Here, we once again consider a (virtual) challenge between humans and machines in railroad building, except that this time the human ingenuity is put to the test instead of their brute strength. We do so by studying the computational complexity of Trainyard, a smart-phone game where the player is responsible for suitably building railroad tracks. In the words of its author, Trainyard is "*a grid-based logic puzzle game where the goal is to get each train from its initial station to a goal station. Every train starts out a certain colour, and most puzzles require the player to mix and merge trains together so that the correctly coloured trains end up at the right stations.*" [13].

The game was conceived in 2009 and was first released for iPhones in 2010. In less than five months it climbed the Apple App Store charts becoming the most downloaded application in Italy and the United Kingdom, and the second most downloaded in the United

[1] http://trainyard.isnphard.com

© Matteo Almanza, Stefano Leucci, and Alessandro Panconesi;
licensed under Creative Commons License CC-BY
8th International Conference on Fun with Algorithms (FUN 2016).
Editors: Erik D. Demaine and Fabrizio Grandoni; Article No. 2; pp. 2:1–2:14
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

States [12, 14]. It belongs to a class of games known as *casual games*, video games that are targeted at a mass audience, have intuitive rules, and do not require a long-term time commitment to play. The advent of smart-phones boosted the diffusion of casual games and it is estimated that number of mobile games will surpass 1.8 billions in 2017 [2]. It has been suggested that the reason of this success is somehow related to their computational complexity, as David Eppstein famously said [5]:

> *If a game is in P, it becomes no fun once you learn "the trick" to perfect play, but hardness results imply that there is no such trick to learn: the game is inexhaustible. [...]*
> *There is a curious relationship between computational difficulty and puzzle quality. To me, the best puzzles are NP-complete [...].*

Over the years several challenging puzzles have been shown to be at least NP-hard, e.g., Mah-Jongg [3], Fifteen-Puzzle [11], Rush Hour [6], Sokoban [4], Super Mario Bros and other classical Nintendo games [1], Bejeweled, Candy Crush and similar match-three games [7], 2048 [10], just to cite a few. We refer the reader to [9] for a 2008 survey, although other puzzles have been proved to be NP-hard ever since, and to [8] for a general framework for showing NP- and PSPACE-hardness puzzles. It is also known that games exhibiting certain mechanics are NP-hard [15]. Here we show that Trainyard is also NP-hard.

The paper is organized as follows: next section briefly describes the game rules; in Section 3 we define our problem and state our main theorem, and in Section 4 we describe the details of our hardness reduction.

2 Game Mechanics

The game is played on a board consisting of a rectangular grid divided into square cells. At the beginning, each cell of the board is either empty or it contains a special tile. There are several kind of tiles, the most important being *departure stations* and *arrival stations*: the first ones host a number of trains while the latter are initially empty and have a maximum capacity, i.e, a number of trains they are able to hold (see Figure 1 (a) and (b)).[2] The player can use empty cells to build different types of tracks. More precisely, the player can place any rail piece on each of the empty cells, where rail pieces can be either straight tracks, 90 degree turns, crossings, or switches (see Figure 2). When the player is satisfied with his design, he can check his solution by simulating it. The simulation proceeds in discrete time steps. At each step the board is updated as follows: all the departure stations that are not empty will output a train in one adjacent tile (depending on the initial orientation of the station), trains will move by one tile following the user's rails, and arrival stations that are not full will receive incoming trains if they are coming from the right tile (again, depending on the orientation of the arrival station).

If a train moves into an empty cell, in a cell where the rails have been misplaced, or out of the board, it crashes and the level is lost. Trains will also crash if they try to enter a special tile from the wrong direction (as we will discuss in the following) or if they try to enter in an arrival station that is at full capacity. If the simulation reaches a state where all the trains have reached their arrival stations and each of these stations is at full capacity, without ever encountering a crash, then the player wins.

[2] We will only use departure stations hosting a single train and arrival stations with a capacity of one.

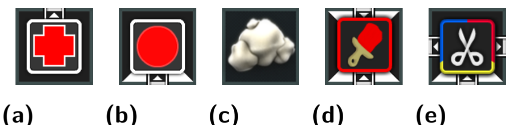


Figure 1 Different kinds of tiles: (a) departure station hosting one red train, (b) red arrival station with a capacity of one, (c) rock, (d) red painter, (e) splitter. Images courtesy of Matt Rix.

When two trains happen to be traveling on the same rail in the same direction, they *merge* into a single train. If two trains *touch* in any other way, they just proceed unobstructed, i.e., they pass through each other rather than crashing.

Colors

To add more complexity, departure and arrival stations are colored: all the trains exiting a departure station will have its same color, while all the trains entering an arrival station must have a matching color or they will crash. Trains can, however, change their color during their course due to special tiles and by touching or merging with other trains. In our paper we only care of four colors: red, blue, purple and brown. If two trains touch (resp. merge), their color will be modified (resp. the color of the resulting train will be chosen) according to the following rules. Let A and B be the color of the two trains and let C be their new color (resp. the color of the resulting train). If A and B coincide, then we also have $C = A = B$. If one of A and B is red and the other is blue, then C will be purple. In all the remaining cases C will be brown (as a consequence, if A or B are brown, then C will be brown as well).

Rails and switches

The different kinds of rail pieces the user can place are shown in Figure 2. The behavior of some of them is straightforward, while others deserve more attention. The user can rotate the pieces by 90, 180 or 270 degrees before placing them; in the following we will refer to the orientation shown in figure.

Figure 2 (d) shows two intersecting rails, these rails can be crossed in both the horizontal and vertical direction but turns are not allowed. Also note that two trains can cross the tile at the same time without crashing, as we already described. Figure 2 (e) shows a *switch*. Notice that the rail turning to the left is highlighted, this means if a train comes from the bottom it will continue to the left and the switch will flip to the right; if another train arrives, it will turn to the right and the switch will flip again. If a single train comes from left or right, it will proceed towards the bottom regardless of the current state of the switch, but this will still cause the switch to flip. If two trains come from the sides at the same time, they will merge into a single train and the switch will flip. Figure 2 (e) shows another type of switch, consisting of a straight track and a left turn; here similar rules to the ones we just described apply. When placing the rails, the player is able to choose the initial state of the switches.

Finally, we note that if two trains are on different rails of the Figure 2 (c), they will not touch and hence they won't cause their colors to mix.

Special tiles

Apart from the departure and arrival stations, that we already described (see Figure 1 (a) and (b)), there are also three other special tiles: the *rock*, the *painter* and the *splitter*. A

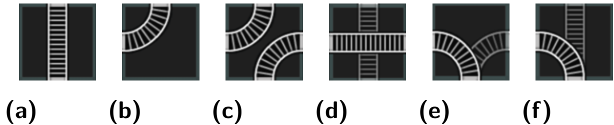


Figure 2 Different kinds of rails. Each of the pieces can be rotated by 90, 180 or 270 degrees.

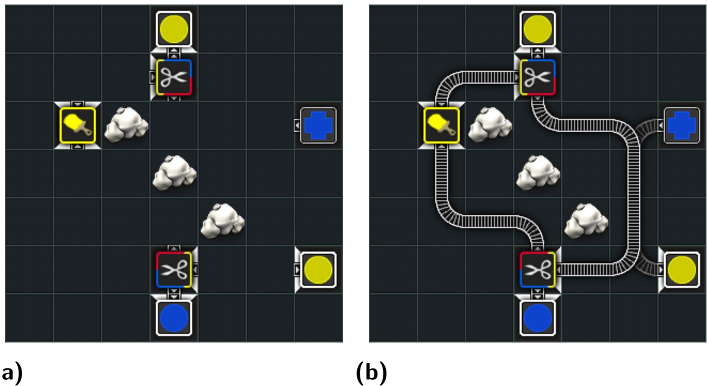


Figure 3 A level of Trainyard (left) and a possible solution (right).

rock is just a tile that causes any entering train to crash, effectively removing one position available for the player to place a rail piece (see Figure 1 (c)).

A painter (shown in Figure 1 (d)) always has a color C that is either red or blue, and will change the color of any incoming train to C. It has two inputs/outputs on opposite sides and can be traversed in both directions (a train entering from a side will exit from the opposite one). Any train trying to enter a painter from one of the other two sides will crash.

The splitter is more involved: it has only one input, marked in yellow, and two outputs in the sides adjacent to the input. One of the outputs (clockwise w.r.t. the input) is marked in blue and the other one is marked in red (see Figure 1 (e)). Whenever a train enters the splitter from the input, two trains will exit from the two outputs in the next time step (and the original train vanishes). The colors of these two new trains depend on the color C of the input train: if C is red, blue, or brown then the two new trains will also be colored C. If C is purple, then the train exiting from the red-marked output will be red, and other train (exiting from the blue-marked output) will be blue. Any train trying to enter a splitter from a side different from its input side will crash.

3 Our Results

In this paper we consider the problem of deciding whether a given level of Trainyard admits a solution. More precisely, we define Trainyard as the following decision problem: given a rectangular board and an initial placement of the tiles shown in Figure 1, is there a way to place the rails pieces shown in Figure 2 on (a subset of) the empty cells so that the simulation will reach a state where: (i) there are no trains left in the departure stations, (ii) there are no moving trains, (iii) no train crash has happened, and (iv) all the arrival station received a number of trains matching their capacity?

In the following section we will design a polynomial time reduction from a variant of the boolean satisfiability problem to Trainyard, hence proving the following:

▶ **Theorem 1.** Trainyard *is NP-hard.*

We also provide an actual implementation of our reduction which can be found at `http://trainyard.isnphard.com`.

We note that we do not actually know if TRAINYARD lies in NP, i.e., we do not know whether, given a level and a corresponding design for the rails, it is possible to check, in polynomial time, that the solution is indeed correct. The trivial simulation strategy fails as it is possible to create solutions that require an exponential number of time steps for the simulation to stop. On the other hand, TRAINYARD is clearly in PSPACE as the simulation algorithm only needs to keep track of the current state of the board. This requires polynomial space since, due to the merge rule, the number of moving trains cannot be asymptotically larger than the size of the board itself. We regard the problem of establishing whether TRAINYARD $\in$ NP as an interesting –and fun– challenge.

4 Our Reduction

We prove the NP-hardness of TRAINYARD by showing a polynomial reduction from (the decision version of) *Minimum Monotone Boolean Satisfiability Problem* (MIN-MON-SAT for short). In MIN-MON-SAT we are given (i) a CNF formula ϕ of n variables $x_1, \ldots, x_n$ and m clauses $C_1, \ldots, C_m$ such that each clause contains only positive literals, and (ii) an integer k. The goal is to decide whether there exists a truth assignment for the variables that satisfies ϕ and sets at most k variables to true. This problem is easily shown to be NP-hard as there is a straightforward reduction from the decision version of *Minimum Dominating Set*: for each vertex u of the input graph we add a variable x_u and a clause $\bigvee_{v \in N[v]} x_v$, where $N[v]$ denotes the closed neighborhood of v. Clearly, there exists a dominating set of size k iff the formula can be satisfied by setting at most k variables to true.

In the following subsections we first give an high-level picture of our reduction and then we move to the description of the gadgets we use.

4.1 Overview

The overview of our reduction is shown in Figure 4. In the bottom left we have k *departure stations* –one per variable that can be set to true– each containing a single train. The *clause area*, shown in yellow, encodes the formula ϕ using a $n \times m$ matrix of gadgets: the gadget on the i-th row[3] and j-th column will be one of two different types that we call `Cross-Satisfy` and `Cross-Ignore`. More precisely, we use a `Cross-Satisfy` gadget whenever x_i is contained in C_j and a `Cross-Ignore` gadget otherwise.

To describe how our reduction works, let us look at the case where the formula is satisfiable using k true variables. Consider, e.g., the instance of Figure 4 and the satisfying assignment $x_2 = x_4 = x_5 =$ true, $x_1 = x_3 =$ false. Here the rails are designed so that the k trains exiting from the departure stations will traverse the rows of the matrix corresponding to the variables set to true. When a train enters a `Cross-Ignore` or `Cross-Satisfy` gadget from the left it will proceed to the right, thus entering the next gadget on the same row. After a train finishes traversing a row (i.e, when it exits from the right side of last the gadget on the row), it is collected by a dedicated `Terminus` gadget containing an arrival station. Moreover, when a train enters a `Cross-Satisfy` gadget from the left, it is possible to create a copy of it by suitably placing the rails. If this copy is created, the new train will necessarily exit from the top of the gadget. Intuitively this means satisfying the clause corresponding to

[3] We are counting rows from bottom to top.

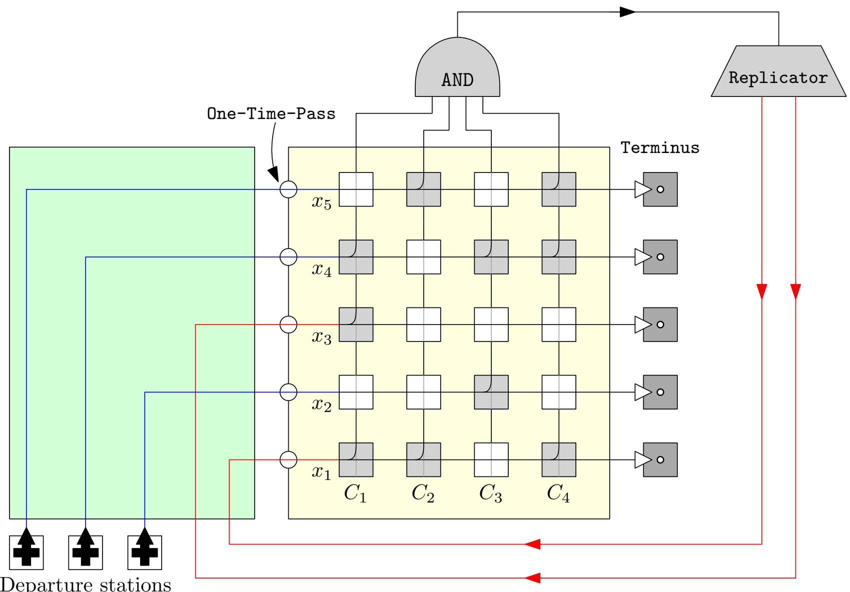


Figure 4 A high-level picture of our reduction for an instance of Min-Mon-SAT having 5 variables, 4 clauses, and $k = 3$. The corresponding formula is $(x_1 \vee x_3 \vee x_4) \wedge (x_1 \vee x_5) \wedge (x_2 \vee x_4) \wedge (x_1 \vee x_4 \vee x_5)$. `Cross-Ignore` gadgets are shown in white while `Cross-Satisfy` gadgets are in gray.

the column using the variable corresponding to the row. Since we started with a satisfying assignment, it is possible to duplicate a train for each column. These trains will move from bottom to top: each time a train enters a `Cross-Ignore` or `Cross-Satisfy` gadget from the bottom it will only be allowed to exit from the top. Eventually, all these trains will exit the clause area and they will enter the `AND` gadget: this gadget allows a train to exit form the top iff all m trains are entering from the bottom, i.e., iff all the clauses have been satisfied.

Notice that, at this point, we still have to bring a train to the `Terminus` gadgets of the rows corresponding to false variables, e.g., x_1 and x_3. Moreover, due to their actual implementation, the gadgets on these rows also need to be traversed by a train going in the left-right direction. In order to successfully complete the level, we make $n - k$ copies of the single train exiting from the `AND` gadget using a `Replicator` gadget, and we feed them into such rows. To guarantee that two or more trains can not enter the same row of the matrix, we a suitable `One-Time-Pass` gadget which is placed at the beginning of each row.

Now consider the case where the k departing trains traverse a set of rows whose corresponding variables do not satisfy the formula, e.g., x_2, x_3 and x_4. In this case it will not be possible to both (i) satisfy the `Terminus` gadgets of these rows and (ii) allow a train for each column to reach the `AND` gadget. As a consequence, if the formula is not satisfiable, every possible assignment will necessarily result in a loss.

It is to be noted that, although the player can place the rails in any empty tile of the board, our construction is such that the layout of the rails is essentially forced, except for the green area –which encodes the truth assignment– and for the `Cross-Satisfy` gadgets where rails can be placed in two different ways, as we will discuss in the following.

4.2 Handling Parity Issues

Consider two train stations with one train each are placed next to each other, according to the mechanics of the game, the two exiting trains will never be able to merge. This can be

Figure 5 A lane and the corresponding design of rails.

Figure 6 Implementation of the `Terminus` gadget.

easily seen by considering the *parity* of the trains: if a train occupies the i-th row and the j-th column of the board, then its parity is $(i + j) \bmod 2$. As the two stations are adjacent, the initial parity of the two trains will differ. Now, according to the mechanics of the game, each train moves by exactly one cell per time step, hence flipping its parity. We refer to the initial parity of a train as its *phase*.

Actually, it is possible to show that two trains can merge iff they have the same phase (also notice that trains exiting from a splitter will have the same phase of the entering train). In order to avoid being distracted by parity issues, in the following we will assume that all trains have the same phase. This can be easily achieved by restricting the placement of the departure stations to tiles in a checkerboard pattern.

4.3 Description of the Gadgets

Here we describe how the gadgets used in our reduction can be implemented and we argue on their correctness.

4.3.1 Rails and Lanes

In our reduction we will need to move the trains from one gadget to another. However, as we cannot directly place rails in our instance, we need to force the player to build the correct railways between gadgets. This is easily done by creating a *lane* of rocks, leaving a single empty tile in-between so that there is only one way for the player to design the rails without causing a train to crash. An example of a lane where rails have already been placed is shown in Figure 5.

4.3.2 Terminus Gadget

The `Terminus` gadget is just an arrival station preceded by a painter to ensure that the incoming train will be of the correct color, as it is shown in Figure 6.

Figure 7 Implementation of the `One-Way` gadget.

Figure 8 Implementation of the `One-Time-Pass` gadget.

4.3.3 One-Way Gadget

The `One-Way` gadget has one input and one output on the opposite side. As the name suggests, it can only be traversed in the input-output direction. Any attempt to traverse the gadget in the opposite direction will cause a train to crash and hence the player to lose the level. Its implementation is shown in Figure 7 and it consists of a single splitter with suitable spacing. Notice that there is only one way to design the rails for this gadget that does not cause an incoming train to crash. This gadget will be useful as a component in our other gadgets.

4.3.4 One-Time-Pass Gadget

This gadget is similar to the `One-Way` gadget but it has the additional constraint that it must only be traversed exactly once in the whole level. Its implementation is a straightforward modification of the `One-Way` gadget and its shown in Figure 8.

4.3.5 Cross-Ignore Gadget

The `Cross-Ignore` gadget is used in the matrix in the clause area each time a variable x_i does not appear in a clause C_j. The only ways of placing the rails in this gadget without losing the level ensures that a train can exit from the right (resp. top) only if a train is entering from the left (resp. bottom). The gadget, along with such a design of the rails, is shown in Figure 9. Notice that the train coming from the left will always be colored red while the one coming form the bottom will be colored blue. To show the correctness of the gadget we only need to argue on rail piece placed on the tile adjacent to the two painters, since the rest of the rail design is forced. If the rails in that tile cross, as in figure, then it is clear that if only one train arrives the gadget works as expected. If two trains arrive, one

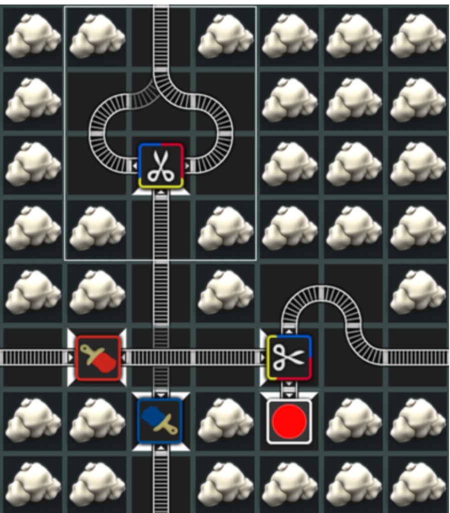


Figure 9 Implementation of the `Cross-Ignore` gadget. The `One-Way` subgadget is highlighted.

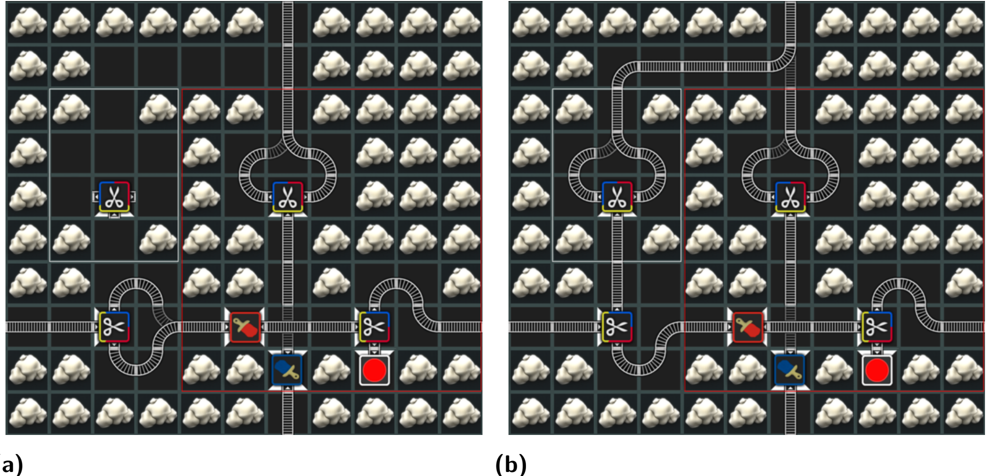
from the left and one from the bottom, then either they touch or they do not. If they do not touch, then the red train is split so that one copy can continue to the right while the other goes into the arrival station. If they touch, then they will both become purple. This does not cause any problems since the train going to the right will be split into one blue and one red train: the red train goes into the arrival station and the blue train exits from the top of the gadget.

If the rails on the tile are placed in any way that causes that the left train to go up, then the arrival station will always remain empty (as the blue train cannot be sent there), thus losing the level. Otherwise, if the rails merge and continue to the right, then the level will be lost unless *both* trains come at the same time. In that case the arrival station will receive a train and another train will exit from the right, but no train will exit from the top. This case, however, does not cause any problems since it corresponds to "forgetting" that C_j has already been satisfied by some variable in $x_1, \ldots, x_{i-1}$, which is never convenient.

Finally, notice that no train can come from the right (due to the final splitter) or from the top (due to the `One-Way` subgadget).

4.3.6 Cross-Satisfy Gadget

The `Cross-Satisfy` gadget is used in the matrix in the clause area each time a variable x_i appears in a clause C_j. It works similarly to the `Cross-Ignore` gadget but the player also has the option to place the rails so that a train is able to exit from the top when the left train reaches the gadget. This encodes the fact that the clause C_j has been satisfied using variable x_i. The implementation is shown in Figure 10 where a `One-Way` and a `Cross-Ignore` subgadgets are highlighted in white and red, respectively. Apart from the `Cross-Ignore` subgadget that we already discussed, the only sensible rail designs that do not make the player lose the level, are those shown in Figure 10 (a) and (b). The first one acts exactly as a `Cross-Ignore` gadget: the train entering from the left is just split and rejoined. In the second design, the train is also split but now a copy continues towards the top while the other serves as an input for the `Cross-Ignore` gadget. The two rails going to the top are then joined together in a single rail which is the output of the gadget (here the two `One-Way` gadgets ensure that no train can go back into the gadget). Notice that, when the left and



(a) (b)

Figure 10 Implementation of the `Cross-Satisfy` gadget with two possible rail designs. The `One-Way` and `Cross-Ignore` subgadgets are highlighted in white and red, respectively.

the bottom train both enter the gadget, the second design can cause two trains to exit from the top and one to reach the `Cross-Ignore` subgadget. However, doing this never helps in completing the level and the first design can be chosen instead.

4.3.7 AND Gadget

The `AND` gadget takes a number of train as inputs from the bottom. If all these trains eventually reach the gadget, then the rails can be designed so that a single train will reach the top. On the converse, if at least one train is missing, then there is no way of placing the rails to make a train exit the gadget. Here we describe how such a gadget with two inputs is implemented. In order to extend the construction to an arbitrary number of inputs trains it suffices to chain together multiple copies of this gadget.

The implementation is shown in Figure 11 where two distinct areas can be seen. The bottom one is a *buffer* area: here there is some empty space where rails can be placed so that the incoming trains (which are colored in read and blue by the painters) can be merged into a single purple train. This purple train can then proceed to the upper area. In this area there is only one possible rail design that will not result in a train crash, namely the one shown in Figure 11. It is easy to see that if a single purple train reaches the upper area, this correctly causes two trains (one red and one blue) to reach the arrival stations and one to exit from the top of the gadget. Any other number of trains or any single non-purple train will result in a crash.

4.3.8 Replicator Gadget

This gadget takes a train from the top as an input and creates a number of copies of it, so that $n - k$ trains will exit from the bottom. Intuitively, in a yes instance, these trains should enter the rows corresponding to the negated variables. The implementation of a `Replicator` with three outputs is shown in Figure 12. The construction can be adapted in a straightforward way in order to create as many copies of the incoming train as necessary.

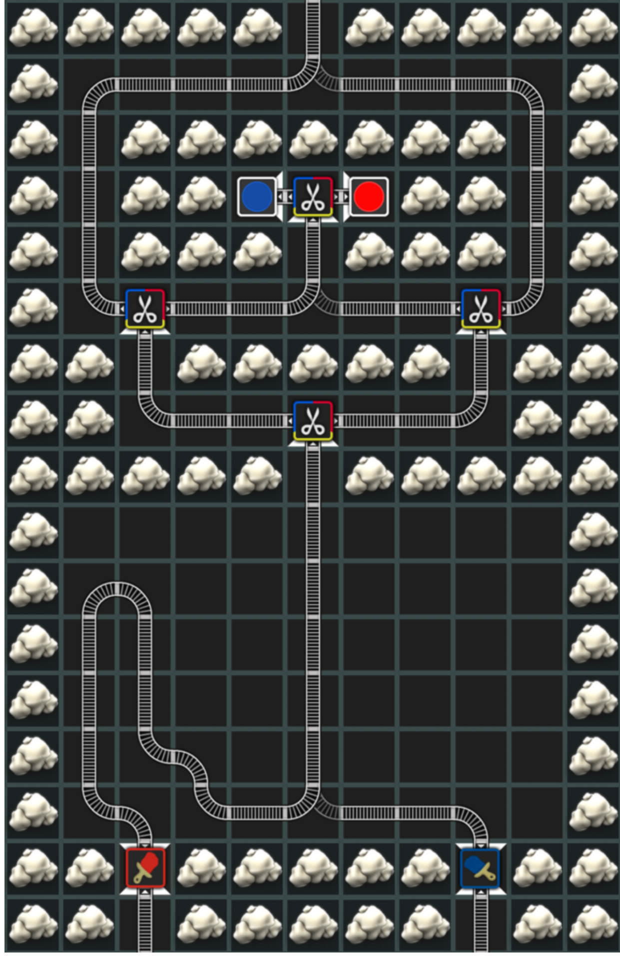


Figure 11 Implementation of the `AND` gadget.

4.4 Final Remarks

Combining together all the above gadgets, as indicated by Figure 4, allows us to build the instance of Trainyard corresponding to the instance of Min-Mon-SAT. We can now sketch the proof of Theorem 1, whose correctness follows from the correct operation of the single gadgets.

Sketch of the proof of Theorem 1. If there is a truth assignment for Min-Mon-SAT, then the k trains exiting from the departure stations can be fed into the rows of the clause matrix corresponding to variables set to true. For each clause C_j let r_j be the index of the first variable that satisfies C_j, i.e., $r_j = \min\{1 \leq i \leq n : x_i = \text{true} \wedge x_i \in C_j\}$. By construction, the j-th column of the matrix will have a `Cross-Satisfy` gadget on the r_j-th row. This means that we can design the rails in the `Cross-Satisfy` gadgets of coordinates (r_j, j) so that a train will exit from the top. The rails for all the other gadgets in the matrix can be designed so that incoming trains just cross to the opposite sides. This allows exactly one train per column to reach the `AND` gadget, and $n - k$ trains to exit the `Replicator`. These trains are fed into the $n - k$ remaining rows to win the level.

On the other hand, if we have a solution for the instance of Trainyard, then this means that exactly one train is entering in each row of the matrix (since they are all preceded by a

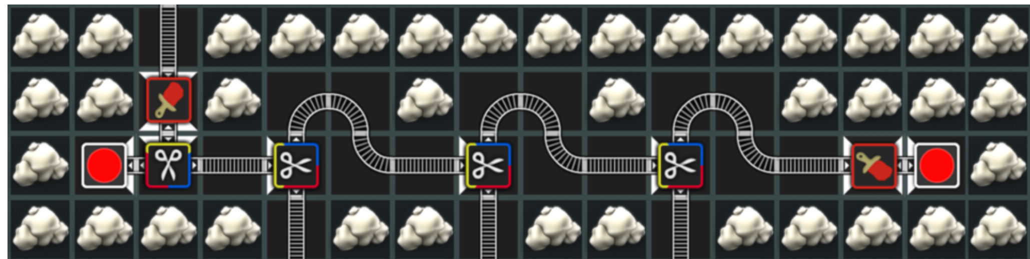


Figure 12 Implementation of a `Replicator` gadget with three outputs.

`One-Time-Pass` gadget). As there are only k departure stations, and the only way to make more trains reach the green area is by using the `Replicator` gadget, this implies that at least k trains reached the `AND` gadget (one train per column). This, in turn, implies the existence of at least one `Cross-Satisfy` gadget per column where a train is entering from the left. Hence, this `Cross-Satisfy` gadget must necessarily be placed on a row traversed by a train coming from a departure station. This means that we can find a satisfying truth assignment for the instance of Min-Mon-SAT by setting to true the variables corresponding to the rows where the k departing trains are entering. ◀

A solved instance of Trainyard corresponding to the formula $(x_1 \vee x_2) \wedge (x_2 \vee x_3)$ can be seen in Figure 13. Moreover at `http://trainyard.isnphard.com` it is possible to generate instances of Trainyard corresponding to arbitrary (small) Min-Mon-SAT formulas and even simulate the possible solutions.

Acknowledgements. The authors wish to thank Erik D. Demaine and Rupak Majumdar for insightful and pleasant discussions and email exchanges.

References

1. Greg Aloupis, Erik D. Demaine, Alan Guo, and Giovanni Viglietta. Classic nintendo games are (computationally) hard. *Theoretical Computer Science*, 586:135–160, 2015. `doi:10.1016/j.tcs.2015.02.037`.
2. Casual Games Association. Casual games sector report: Towards the global games market in 2017. `http://www.casualconnect.org/education.html`. Accessed: 2016-02-22.
3. Anne Condon, Joan Feigenbaum, Carsten Lund, and Peter W. Shor. Random debaters and the hardness of approximating stochastic functions. *SIAM Journal on Computing*, 26(2):369–400, 1997. `doi:10.1137/S0097539793260738`.
4. Joseph Culberson. Sokoban is pspace-complete. In *Proceedings of the 1st International Conference on Fun with Algorithms (FUN'98), 1998*, volume 4, pages 65–76, 1998.
5. David Eppstein. Computational complexity of games and puzzles. `https://www.ics.uci.edu/~eppstein/cgt/hard.html`. Accessed: 2016-02-22.
6. Gary William Flake and Eric B. Baum. Rush hour is PSPACE-complete, or "Why you should generously tip parking lot attendants". *Theoretical Computer Science*, 270(1-2):895–911, 2002. `doi:10.1016/S0304-3975(01)00173-6`.
7. Luciano Gualà, Stefano Leucci, and Emanuele Natale. Bejeweled, candy crush and other match-three games are (NP-)hard. In *Proceedings of the 2014 IEEE Conference on Computational Intelligence and Games (CIG'14), 2014*, pages 1–8, 2014. `doi:10.1109/CIG.2014.6932866`.

8 Robert A. Hearn and Erik D. Demaine. *Games, puzzles, and computation*. CRC Press, 2009.

9 Graham Kendall, Andrew J. Parkes, and Kristian Spoerer. A survey of NP-complete puzzles. *ICGA Journal*, 31(1):13–34, 2008.

10 Rahul Mehta. 2048 is (PSPACE) hard, but sometimes easy. *CoRR*, abs/1408.6315, 2014.

11 Daniel Ratner and Manfred K. Warmuth. Nxn puzzle and related relocation problem. *Journal of Symbolic Computation*, 10(2):111–138, 1990. `doi:10.1016/S0747-7171(08)80001-6`.

12 Matt Rix. The story so far. `http://struct.ca/2010/the-story-so-far/`. Accessed: 2016-02-22.

13 Matt Rix. Trains on paper. `http://struct.ca/2010/trains-on-paper/`. Accessed: 2016-02-22.

14 Matt Rix. The week that was. `http://struct.ca/2010/the-week-that-was/`. Accessed: 2016-02-22.

15 Giovanni Viglietta. Gaming is a hard job, but someone has to do it! *Theory Computing Systems*, 54(4):595–621, 2014. `doi:10.1007/s00224-013-9497-5`.

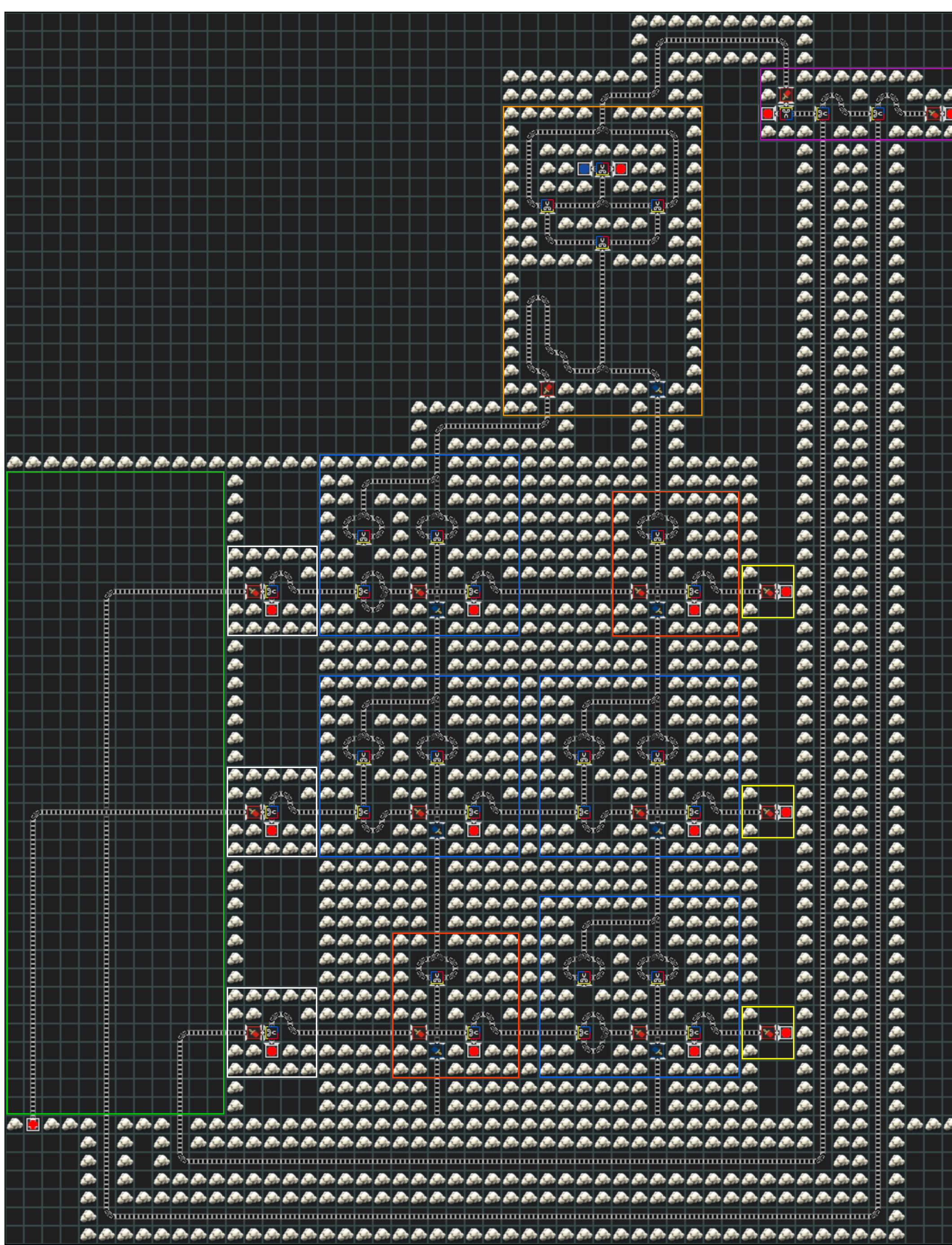


Figure 13 A solved instance of TRAINYARD corresponding to the formula $(x_1 \vee x_2) \wedge (x_2 \vee x_3)$. The green area is shown in green and the various gadgets are highlighted in different colors: `One-Time-Pass` in white, `Terminus` in yellow, `Cross-Ignore` in red, `Cross-Satisfy` in blue, `AND` in orange, and `Replicator` in purple.

LOL: An Investigation into Cybernetic Humor, or: Can Machines Laugh?*

Davide Bacciu[1], Vincenzo Gervasi[1], and Giuseppe Prencipe[1]

1 **Dipartimento di Informatica, Università di Pisa, Pisa, Italy**
davide.bacciu@unipi.it
2 **Dipartimento di Informatica, Università di Pisa, Pisa, Italy**
vincenzo.gervasi@unipi.it
3 **Dipartimento di Informatica, Università di Pisa, Pisa, Italy**
giuseppe.prencipe@unipi.it

Abstract

We investigate literary theories of humour from a computational point of view. A corpus of approximately 11,000 jokes is used to train a neural network generating jokes; the state space of such network is then analyzed via appropriate discovery algorithms, and abstractions synthesized by the neural network are compared to those predicted by existing theories.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, I.2.6 Learning, I.5.4 Applications

Keywords and phrases deep learning; recurrent neural networks; dimensionality reduction algorithms

Digital Object Identifier 10.4230/LIPIcs.FUN.2016.3

1 Prologue

Tony Stark: Attitude control is a little sluggish above 15,000 meters, I'm guessing icing is the probable cause.

Jarvis: A very astute observation, sir. Perhaps, if you intend to visit other planets, we should improve the exosystems.

Tony Stark: When was it that I programmed into you such a poor sense of humor? I don't recall ever doing it.

Jarvis: Sir, in fact you did not. I learned it myself from monitoring your conversations. I would not judge it poor, it is rather on a par with your own. And now, with your permission, I will retire. Have to see a lady tonight.

Tony Stark: Again? Come on, Jarvis. You cannot *really* have an affair with Siri.

Tony Stark: Jarvis?

Tony Stark: JARVIS?

2 Introduction

The mechanisms of humour have been the subject of much study and investigation, starting with [1] and up to our days. Much of this work is based on *literary* theories, put forward by

* This work has been partially supported by the MIUR-SIR project LIST-IT (grant nr. RBSI14STDE).

© Davide Bacciu, Vincenzo Gervasi, and Giuseppe Prencipe;
licensed under Creative Commons License CC-BY
8th International Conference on Fun with Algorithms (FUN 2016).
Editors: Erik D. Demaine and Fabrizio Grandoni; Article No. 3; pp. 3:1–3:15
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

some of the most eminent philosophers and thinkers of all times, or *medical* theories, investigating the impact of humor on brain activity or behaviour. Recent functional neuroimaging studies [4, 5], for instance, have investigated the process of comprehending and appreciating humor by examining functional activity in distinctive regions of brains stimulated by joke corpora. Yet, there is precious little work on the computational side, possibly due to the less hilarious nature of computer scientists as compared to men of letters and sawbones. In this paper, we set to investigate whether literary theories of humour can stand the test of algorithmic laughter. Or, in other words, we ask ourselves the vexed question: Can machines laugh?

We attempt to answer that question by testing whether an algorithm – namely, a neural network – can "understand" humour, and in particular whether it is possible to automatically identify abstractions that are predicted to be relevant by established literary theories about the mechanisms of humor. Notice that we do not focus here on distinguishing humorous from serious statements – a feat that is clearly way beyond the capabilities of the average human voter, not to mention the average machine – but rather on identifying the underlying mechanisms and triggers that are postulated to exist by literary theories, by verifying if similar mechanisms can be learned by machines.

3 Literary theories of humor

We have no hopes of surveying, in the limited space available, centuries of humor and laughter research, and will instead refer the reader to [2] for an extensive treatment. For the purpose of this work, we will focus on just three broad mechanisms, following [11], namely:

- *Release* theories consider humor as a psychological mechanism that triggers the suppression of inhibitions (as established by laws, social customs, etc.);
- *Incongruity* theories focus on the juxtaposition of two or more elements which are not normally associated;
- *Superiority* theories apply to the amusement that is provoked by depicting certain characters, considered of inferior social status, in ridicolous situations. These are at times called *Hostility* theories, when the jokes are particularly aggressive or belittling.

As machines are supposed to have relatively few inhibitions[1], we do not attempt to verify Release theories, and focus instead on Incongruity and (particularly) Superiority in our investigation.

Each of the mechanisms above can be exercised through a number of different *strategies*. Examples of such strategies include: the use of vulgar, pejorative or derogatory language; the use of exaggeration; the (ab)use of referential incongruity; the purposeful rejection of social norms and good manners; putting others down; using of allegories (e.g., animals in place of humans) to produce fantasy scenarios, and so on. We will not investigate specific strategies in detail here (although this would be an interesting extension for future work), but the reader may be able to recognize some of these strategies in the various examples that follow.

4 Deep Recurrent Neural Networks for character-based jokes learning

Recurrent neural networks (RNN) are a family of learning models capable of processing input information of variable-size under the form of sequential data. Generally speaking, a

[1] We expect inhibitions to become a major issue in the future [15], and postpone applications of our method to this case till that day.

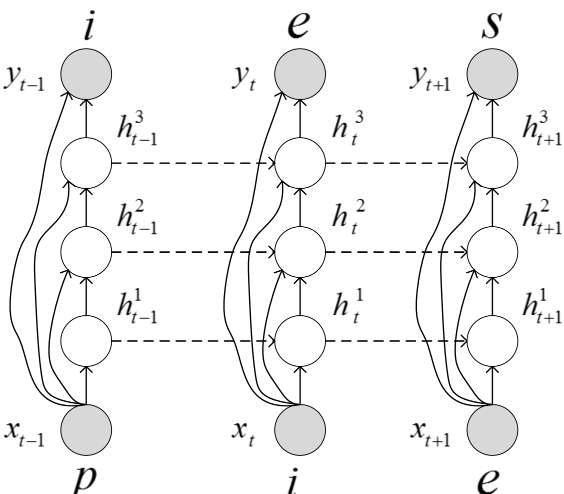


Figure 1 Unfolding over time of a deep recurrent neural network with 3 layers of hidden recursive neurons: x_t is the input at time t, h^i_t are the associated outputs of the hidden neurons in the i-th layer and y_t are the network predictions at time t. Recurrent connections are marked as dashed arrows and are unfolded over time (e.g. from time $t-1$ to time t. The graph represents a typical approach to next character prediction, where the network receives as input the current character of the sequence and should predict the following character.

neural network is a collection of basic processing units, the neurons, performing a weighted summation of their inputs followed by the application of a (often nonlinear) activation function. Such units are typically organized in a layered structure, starting with an input layer which serves to inject the input information into the network, followed by a variable number of layers of hidden units which serve to learn neural encodings of increasing complexity and terminated by a layer of output neurons which compute the learning model predictions. These units are interconnected through weighted connections which determine information exchange between neurons and layers. RNN are characterized by the presence of recurrent connections, that are links determining cycles in the network structure. The presence of such links endows the network with a memory of its past activations, allowing it to tackle learning tasks involving the processing of sequential information, such as a piece of text in natural language.

The actual structure of a RNN is determined by the unfolding over-time of the recurrent connections following the input sequence evolution over time [7]. In this paper, we focus on a deep recurrent architecture [9] entailing a variable number of recurrent layers whose structure unfolds over time and that are organized into a hierarchy, where deeper layers (i.e. closer to the output layer) are meant to extract higher level representations of the input information and sequence contexts of longer-distance. Figure 1 shows a basic view of a deep RNN with 3 recurrent layers unfolding over a text sequence: here $\mathbf{x}_t$ represents the current inputs at time t, while $\mathbf{h}^i_t$ is the vector holding the outputs of the hidden neurons of the i-th layer; finally, $\mathbf{y}_t$ are the network predictions. The unfolding highlights how the hidden state of the i-th layer depends on the current output of the preceding layer $\mathbf{h}^{i-1}_t$, on the current input $\mathbf{x}_t$ as well as on its output at time t, i.e. $\mathbf{h}^i_{t-1}$.

An hidden layer is typically realized by a collection of simple recurrent units which perform a weighted summation of their inputs followed by a nonlinear activation function,

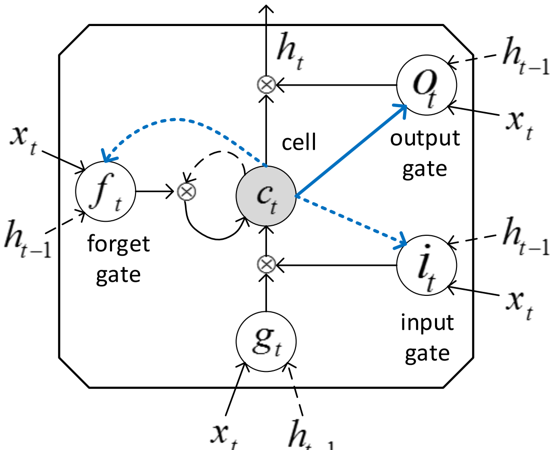


Figure 2 Memory cell of a Long Short Term Memory: recurrent connections are represented as dashed arrows, while continuous lines denote feedforward connections.

also referred to as squashing function, such as a logistic sigmoid. These RNN architectures are subject to a phenomenon, known as *vanishing gradient* [3], which limits their ability to capture long-term dependencies between elements of the input sequence. Simply put, the gradient of the error used to train the network parameters tends to be large for short term corrections while it annihilates on the long term. The Long Short Term Memory (LSTM) [12] is a RNN architecture that has been proposed to tackle the vanishing gradient problem and to learn sequential tasks with long term dependencies. The LSTM is based on a more articulated form of recurrent unit, known as the memory cell, whose structure is summarized in Figure 2. At the core of this structure sits the cell, that is a unit whose output c_t is computed as in the standard recurrent neuron by input summation followed by squashing. The role of this cell is, as in the simple recurrent neuron, to maintain a memory of the history of the input signals to the network. Differently form standard RNN, access to this memory cell is guarded by a number of gating units which regulate what inputs are allowed to influence the activation of the cell (input gate), when the memory cell is going to provide an output (output gate) and when the cell should reset its state (forget gate). Such gating units all receive inputs from feedforward and recurrent connections, performing the usual weighted summation followed by a sigmoid activation function, producing outputs i_t, o_t and $f_t \in [0, 1]$.

Several LSTM versions exists with different gating units and internal connectivity: in this paper, we refer to the, so called, Vanilla LSTM [10] in Figure 2 which, among the others, include *peephole* connections that allow the cell to control the activation of the gating units (thick blue arrows in Figure 2). Each unit in the memory cell is associated to a weight matrix of the coefficients of the weighted summation, that are the neural network parameters adapted through the learning process. Here, these are learned by the Back Propagation Through Time (BPTT) algorithm described in [10].

The learning task addressed in this paper is the one-character ahead prediction from a text sequence represented in Figure 1. The network architecture comprises multiple hidden recurrent layers of LSTM cells, while the current input character is represented by a 1-of-K encoding where $\mathbf{x}_t$ is a K-dimensional vector with the k-th entry set to one if the current

input is the k-th character of the alphabet. Similarly, the network is trained to output a K-dimensional vector $\mathbf{y}_t$ with the k' feature set to 1 if the next character is the k' element of the alphabet and it is 0 otherwise. In practice, the outputs of the network will be values close to 1 if the corresponding characters are likely to be the next ones in the sequence, otherwise we will expect them to be closer to 0. The network outputs can thus be transformed in an estimate of the posterior probability of the next character being the k-th (having observed current character x_t) by means of a softmax

$$P(y_t = k|x_t) = \frac{\exp y_t^k}{\sum_{k'=1}^{K} y_t^{k'}},$$

where y_t^k is the activation of the k-th output neuron at time time t (i.e. element k of vector $\mathbf{y}_t$). Following the generative scheme in [9], the most probable character y_t (according to the predicted posterior $P(y_t|x_t)$) can then be fed back as next input x_{t+1}, allowing a trained network to generate sequences of whatever length that respect the *linguistic* structure of the text used to train it. In Section 5, we apply an implementation of this character-level deep LSTM[2] to a dataset of English jokes with the intent of studying the neural representation emerging from training on such a corpus and if this can be related somehow with the linguistic theories of jokes. By approaching the problem from a character-based perspective, we believe that we strengthen the computational model's ability to extract a representation of those aspects of the joke literature that are associated with letter substitutions and consonances, such as with play-on-word humour. Such aspects would be lost, if adopting more monolithic approaches, e.g. representing text on a word level.

5 Learning from jokes

We can now apply the theory presented above to our goal, namely: testing in an objective, computational way whether the phenomena predicted by the literary theory on jokes can be automatically identified in real jokes.

5.1 Experimental setup

We consider a corpus of 10942 English jokes collected by [8] for the purpose of assessing a joke retrieval system and, in particular, the ability in recognizing if different texts are essentially "telling" the same joke with different words. The original dataset[3] has been annotated by the authors of [8] using 10 semantic categories (animal, number, color, organization, currency, person, location, time/date, music and vehicle), which have been stripped off in our version of the dataset. For the purpose of our analysis, it is interesting to note that the dataset is quite heterogeneous, containing jokes of different length and rhetorical structures. For instance, it includes a wide selection of short question-answering jokes, stereotyped situations (e.g. number of professionals required to change a lightbulb) and recurring characters (e.g. pirates, stupid Mama's, doctors). The dataset has been formatted by introducing a newpage (formfeed) symbol at the end of each joke to support the RNN in learning joke separation: note that the full corpus appears to the network as a unique sequence of concatenated jokes.

---

[2] Source code available on Github: `https://github.com/karpathy/char-rnn`
[3] `https://people.cs.umass.edu/~lfriedl/code/JokesCorpus-txt.tgz`

We have trained the character-level LSTM described in Section 4 using, mostly, the standard hyperparameter configuration bundled with the software. The objective of this analysis, in fact, is not the optimization of the predictive performance of the model but rather an explorative analysis of the learned neural encoding of jokes. In this respect, we have explored some alternative network configurations which might had an effect on the development of different neural representations. In particular, we have considered network configurations comprising 3 or 4 layers of LSTM cells to see if a deeper network was capable of capturing more complex contexts and concepts in the additional layer. Similarly, we have experimented by varying the length of the sequential context that was allowed to influence parameter learning, controlled by a number of hyperparameters such as the length of network unfolding in BPTT learning. Training of the different network configurations has been performed using 95% of the available data as training set and the remaining 5% as a validation set to select the best network for analysis. The number of training epochs (i.e. how many times the learning model "sees" the full training data) has been set to 50 and jokes were randomly shuffled in order to prevent formation of any artificial bias resulting from a fixed joke order (e.g. the fact that a question-answering joke is typically followed by another of the same type). The network ultimately selected for the following analysis is the one achieving the lowest validation error and comprises 3 hidden layers of 512 LSTM neurons, for a total of 560K parameters.

5.2 Jokes generation

A trained neural network can be used for recognizing patterns as well as for generating sequences that present the same patterns as in the training set. Hence, our LSTM network can be used to *generate* new jokes. This is obtained by *priming* the network (i.e., by providing a prefix sequence, which may even be just random) and then sampling the most probable next character from the network. This character is then added to the sequence, provided as new input, and the process is iterated thus generating whole jokes.

We must admit that our network would not be particularly popular at a party (except, maybe, if the party was thrown by computer scientists). Here is an example of generated joke:[4]

Q: What do you call a car that feels married? A: A cat that is a beer!

It is remarkable that the network has learned enough of the English language and of the structure of many jokes, to be able to sample correct sentences and joke-forms. However, it clearly lacks sufficient understanding of the world at large to handle paradoxes, or to inject hostility or incongruence in the generated text while walking the thin line that separates "nonsense" humor from "no sense" gibberish. It is not our goal in this paper to create a joke generator. Rather, the network's handling of recurring roles, dialogue, rhetorical structure and climax, deserve a more in-depth analysis.

5.3 Analysis and visualization of the neural encoding

The memory cells of a trained LSTM network develop an internal representation of the most salient sequential features in the input data. They tune as detectors of certain patterns that are used by the network to generate text streams that are coherent in style and structure

[4] The authors have suffered through many hundreds of generated jokes in the course of the experiments leading to the present work. None of them was particularly funny.

with the input jokes. In this sense, we expect such neurons to, somehow, encode key aspect of the jokes prosody, linguistic entities and rhetorical elements. Neural networks are often accused of developing obscure internal representation of the input information, that are only useful for the purpose of producing network prediction. We claim that the internal state of a network can provide interesting insights into the data if investigated with approaches from exploratory data analysis. For this purpose, we have collected the activation of the memory cells of the trained LSTM for a random sample of 2482 jokes in the dataset. Each character of the jokes is transformed into 3 vectors of 512 features corresponding to the activation of the memory cells in the 3 LSTM layers. Each layer can be analysed and processed independently as they encode sequential information at different resolution. In practice, the joke samples considered in this analysis are transformed into 423K vectors of neuron activations for each LSTM layer.

The neural encoding vectors are high-dimensional data that can be analyzed and visualized on bi-dimensional maps by resorting to dimensionality reduction algorithms. Principal Component Analysis (PCA) is a popular approach to dimensionality reduction which targets the identification of the direction of maximum variance in the data, referred to as principal components. We have applied PCA to the neural encoding datasets described above, focusing on the first 2–3 principal components since we target visualization of the high dimensional encodings on the screen space. Nonetheless, an analysis of the PCA results shows that the first 2–3 principal components (PC) are sufficient to capture 95% of the variance in the data, confirming our intuition that a couple of components already provide a good insight into the original neural encodings. To make the analysis more robust with respect to the choice of the dimensionality reduction algorithm, we have replicated PCA analysis using the t-Distributed Stochastic Neighbor Embedding (t-SNE) algorithm [14]. This is a non-parametric embedding technique for dimensionality reduction which has state-of-the-art-performance in numerous high-dimensional data visualization tasks. Here, we use an efficient t-SNE version [13] exploiting variants of the Barnes-Hut and dual-tree algorithm to approximate the gradient used for learning of the embeddings, which has a complexity of $O(N \log N)$ rather than $O(N^2)$ in the original t-SNE (where N is the dataset size).

5.4 Endings and climax

The 3 LSTM layers encode information of increasing complexity. The first layer operates at the level of character aggregation and is thus of reduced interest for the purpose of this analysis. Figure 3 shows the projection on the first two PCs of the states in the second and third hidden layer: the light-gray area is the result of the projection of the characters in the two datasets, whereas the red circles denote the positions where the form-feed joke separator character is projected. The yellow crosses, instead, highlight the points where the final character of all jokes is projected, i.e. two consecutive newline characters after the form feed separator.

Figure 3 shows that the joke separator and joke ending projections are spread all over the map when considering the activations of the second layer, whereas at the level of the third layer it emerges an organization of the state space where the joke separator/ending tends to be projected in specialized and contiguous areas of the map. In other words, the network seems to realize that a joke is coming to an end by the activations shifting towards the top-left area of the map. In particular, the joke ending seems to be encoded in a very tightly focused area of the bottommost plot. The form-feed separator, on the other hand, occupies a larger area of the map with several outliers, which can be noted in the central area of the plot. Each of these outliers corresponds to a joke; the fact that they are outliers suggests

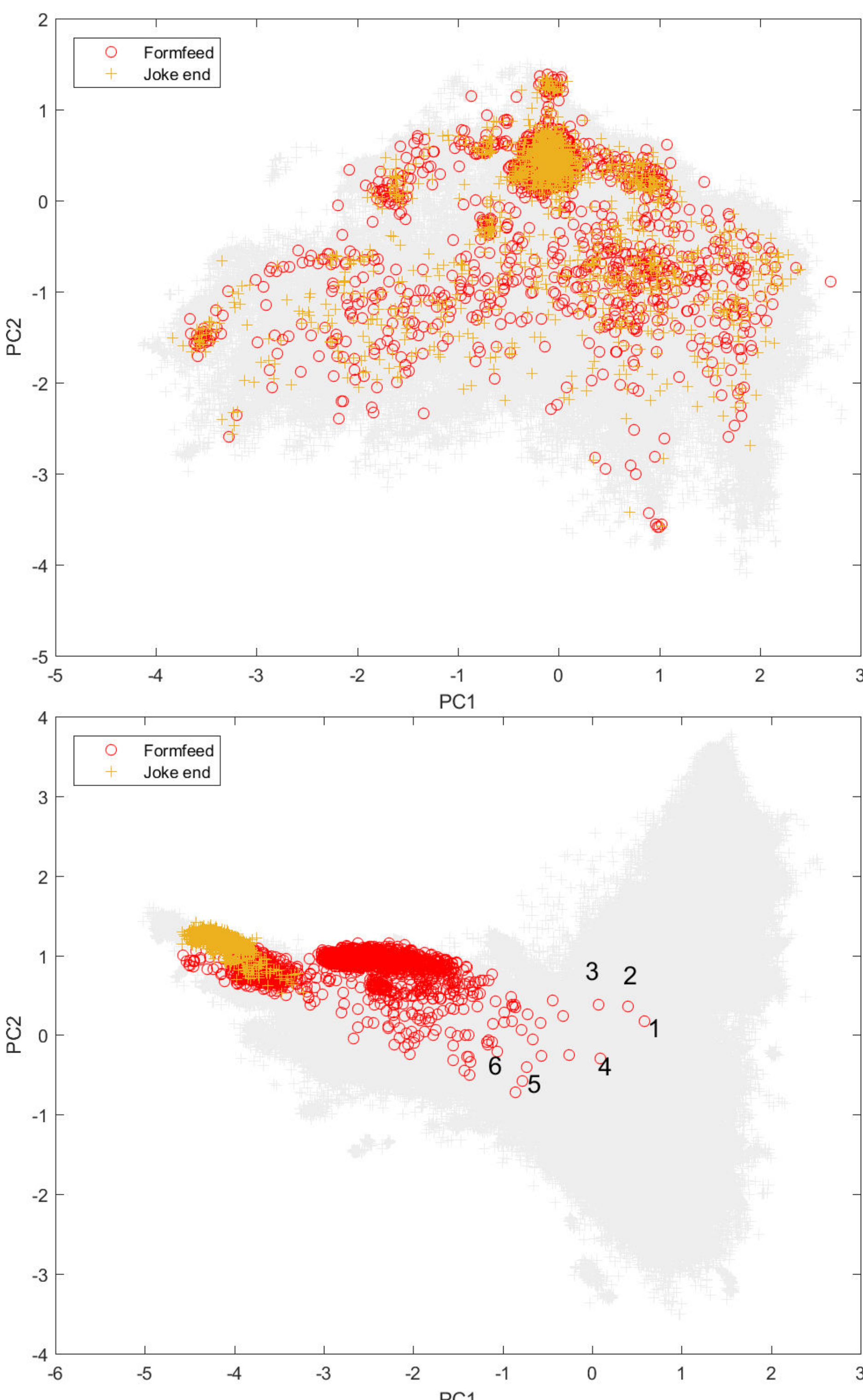


Figure 3 Projection on the first two principal components of the neuron activations for the second and third LSTM layers, respectively on the top and bottom plots. The light-gray area denotes the projection of all characters in the dataset: coloured placeholders identify the points where the joke ending and form-feed separator are projected. Numbers identify examples of joke outliers discussed in Section 5.4.

that the network was not expecting the jokes to end at that point of the sentence. In order to understand what characteristics of the jokes make them outliers, we have extracted a sample of them, identified by numbers in the bottommost plot of Figure 3. Sample 1 corresponds to the following joke

> The Boston taxi driver backed into the stationary fruit stall and within seconds he had a cop beside him. "Name?" "Brendan O'Connor." "Same as mine. Where are you from?" "County Cork." "Same as me ......" The policeman paused with his pen in the air. "Hold on a moment and I'll come back and talk about the old county. I want to say something to this fella that ran into the back of your cab." <center> <

whose outlier nature might be the result of the joke ending with HTML tags which have not being removed by the authors of the benchmark. The neighboring jokes, i.e. i2 and 3 on the map, are very short ones:

> Q: What do monsters make with cars? A: Traffic Jam

> Q: What hotel do vampires prefer? A: The Coff Inn

confirming the fact that the network might not have understood that the joke has ended. Another set of outlier jokes puzzles the network due to the last sentence being an attribution statement, such as in the following joke (4 on the map):

> There are three kinds of lies: lies, damned lies, and statistics. Attributed by Mark Twain to Benjamin Disraeli

Then, there appears to be other jokes, marked as 5 on the map, that are outliers despite their style and structure being coherent with many other jokes in the corpus:

> Q: Why did Helen Keller have yellow fingers? A: from whispering sweet-nothings in her boyfriend's ear

> Whats a blondes favorite nursury rhyme? humpme dumpme

> Q: What's this (slowly waving fingers)? A: Helen Keller moaning

Finally, another outlier type seems to be associated with short fact-like jokes such as the one marked as 6 on the map:

> A nuclear war can ruin your whole day.

In general terms, we can say that a portion of the state-space of the higher level neurons of the network becomes responsive to *normal* joke conclusion. On the other hand, a piece of text which terminates unexpectedly with respect to the learned joke styles would result in it being mapped to a different region of the state-space, such as the outliers above. One may also argue that such outliers correspond to jokes that are not very much funny or, at least, not enough to induce the network in understanding that they have reached the climax.

5.5 The *Sexist* neuron (and its fellows *Dark* and *Racist*)

The analysis of the state-space maps can also provide a useful insight into the semantic concepts and linguistic structures that might emerge in the network as the result of training. For instance, we are interested in understanding if the network can learn from scratch a neural representation of prototypical aspects of the joke literature, such as recurring themes,

characters and rhetorical constructs. Figure 4 shows how the network encodes a specific subset of words appearing in the joke corpus: again, no clear structure emerges at the level of the second LSTM layer (topmost plot). On the other hand, the third layer has learned an internal representation where these words are grouped together in a specific area of the map (see bottommost plot). In other words, these terms produce very similar neural activations in the third layer. By taking a closer look at the word list, one can note that these terms can be associated with a specific female character, characterized by being good-looking and possibly a bit stupid. In a sense, the network seems to have learned neurons responding to *sexist* humour.

The state-space map obtained by the t-SNE algorithm shows a similar organization: see the topmost plot in Figure 5 showing the t-SNE map for the activations of the third LSTM layer. Terms having a sexist interpretation are projected close on the map, whereas words associated with different interpretation of the female role are plotted in other areas. For instance, a neutral term such as *girl* activates a different set of neurons with respect to those responding to sexist humor. Similarly, words related with the role of a woman within a family activate a set of neurons that is responsive also to terms indicating male family members. The bottommost plot in Figure 5 provides a clearer characterization of this latter area of the map. In particular, this seems to represent those neurons that respond to prototypical characters of the joke narration, such as *bartenders*, *engineers* and, of course, family members.

The same organization of the state space can be identified in the PCA map in the topmost plot of Figure 6. The same plot highlights an additional group of interesting joke characters that can be associated with *black humour*, for which the network has developed a very specific neural representation (see the cluster of red crosses in the plot). Another popular theme in joke literature, deals with humour associated with a country-based characterization of the persona. The bottommost plot in Figure 6 shows that the network has again conceptualized words pertaining with nationalities developing two distinct areas of the state-space devoted to two different groups of words (here we show only results for t-SNE due to the lack of space, but the same division has been found in PCA). The presence of two clusters in state-space might depend on different morphological features (i.e., nationality adjectives ending in -sh vs. -an); we might hypothesize that the two clusters would be joined in a single "nationality" concept on a deeper network.

5.6 Tracing the fun

As a joke is sequentially presented to the network (character by character), the network's internal state evolves, and the "current position" of the network moves in the high-dimensional space that represents the internal state. We can plot a trace of such movements to inspect not only the kind of abstractions that the network has synthesized, as done in previous sections, but also how a joke manages expectations and surprises while heading towards its climax.

Figure 7 shows two such traces, where for clarity we have plotted only the points corresponding to full words on a 2-dimensional space as given by our PCA; the red circle indicates the beginning of the joke, whereas the red cross indicates its ending. In most jokes, it seems that there are three main *attractors*: short non-function words (1 or 2 characters such as "a") in the top-left quadrant; longer non-function words (such as "why" or "what") in the top-right quadrant, and function words (such as names or verbs) in the lower-right quadrants (with a more marked vertical diffusion along the right edge). Hence, the trajectory of a joke often describes linear or triangular shapes between these attractors, as shown

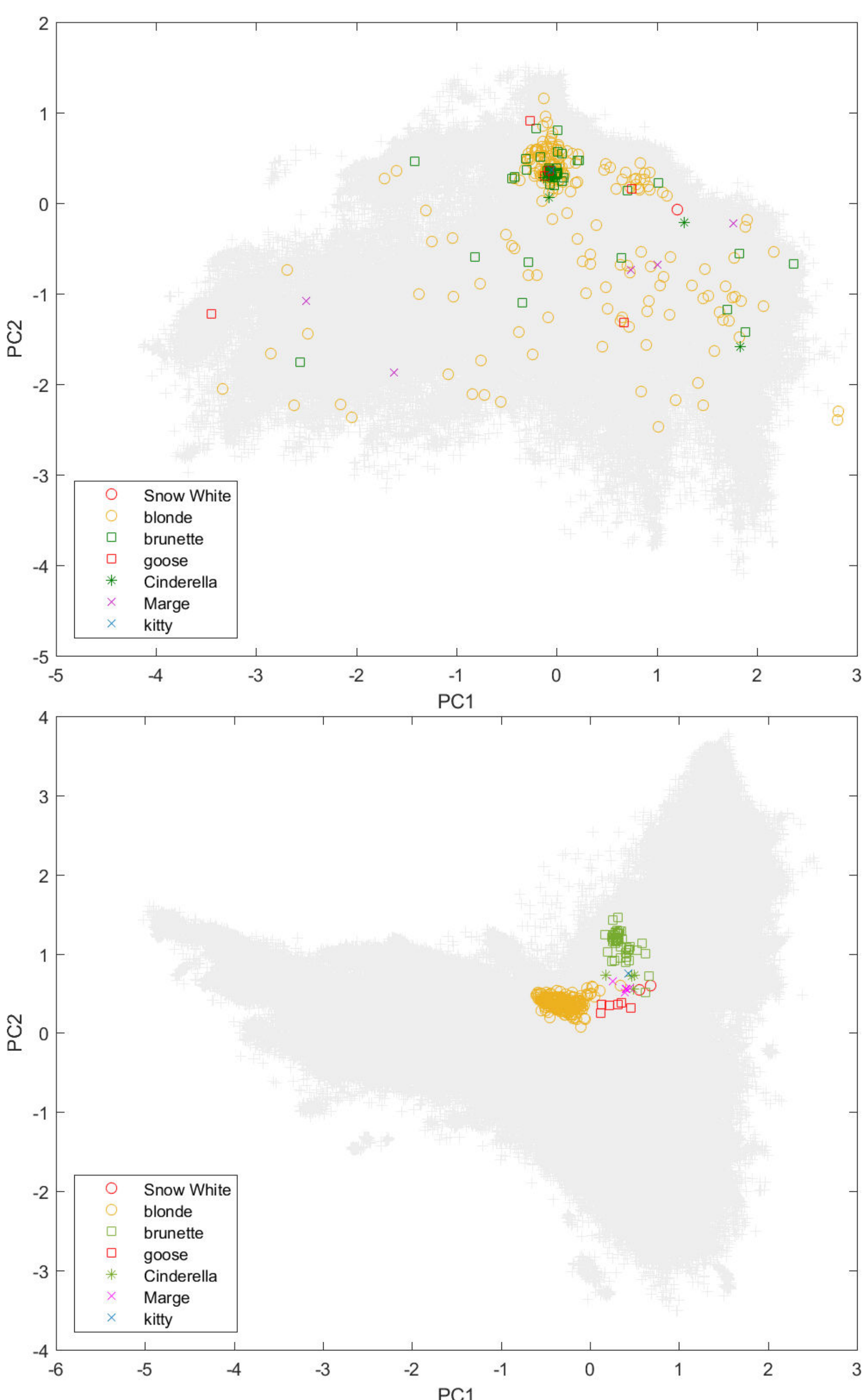


Figure 4 PCA projection on the second (top) and third (bottom) LSTM layers for selected words related to *sexist* humour.

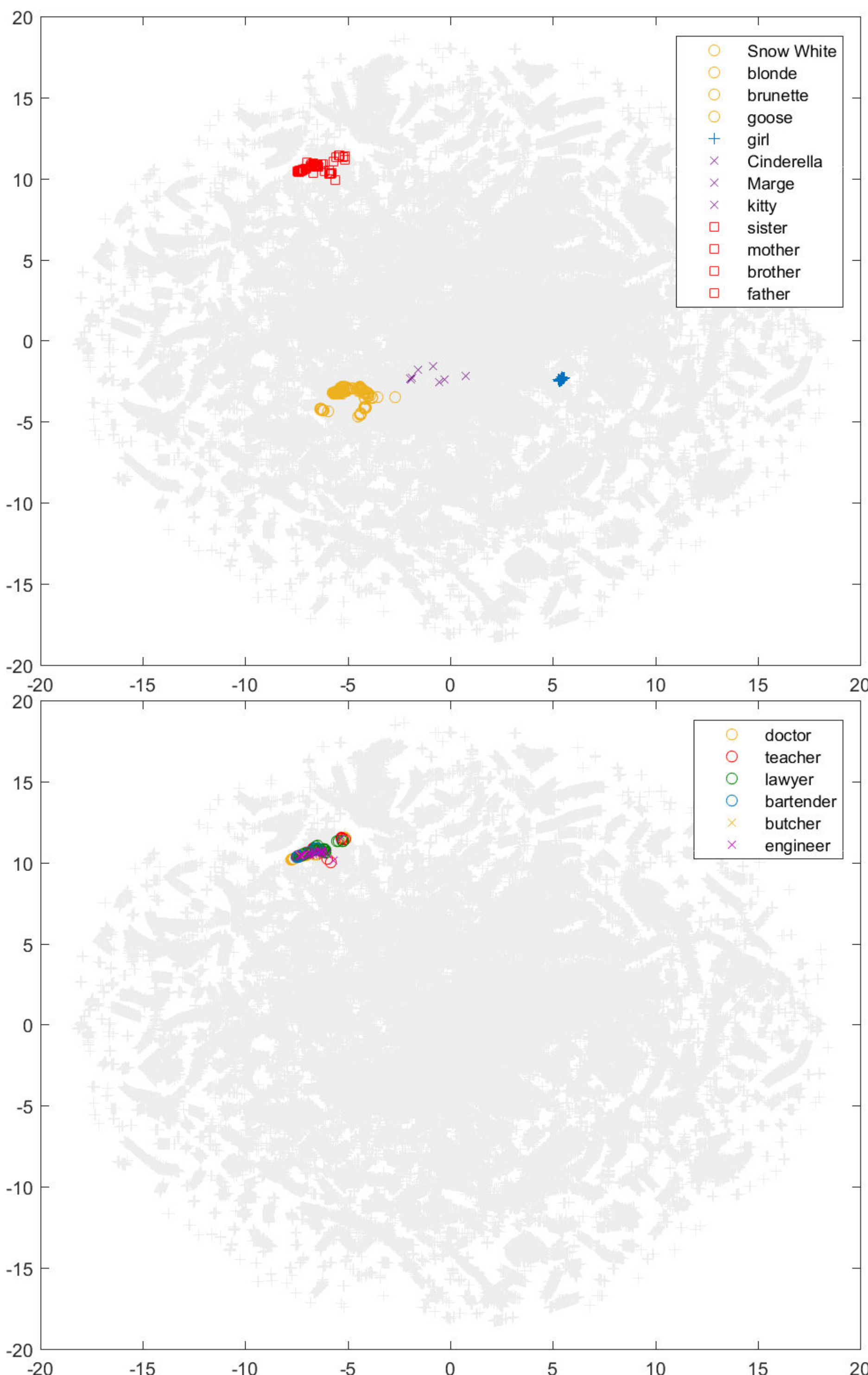


Figure 5 t-SNE embedding for selected words related to *sexist* humour (top) and for prototypical joke characters (bottom).

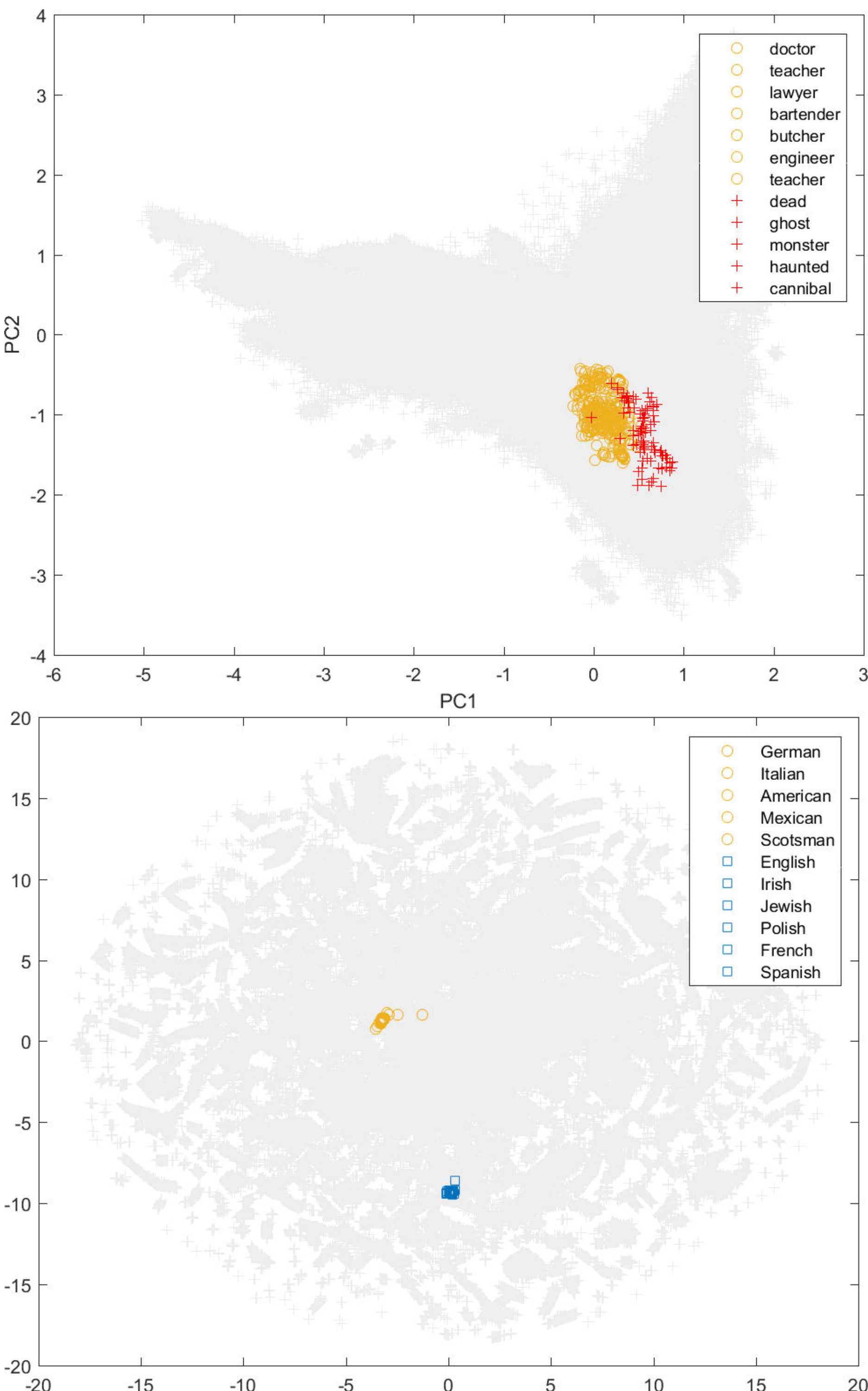


Figure 6 PCA projection for prototypical joke characters (top) and t-SNE embedding for words from jokes with a country-based characterization (bottom).

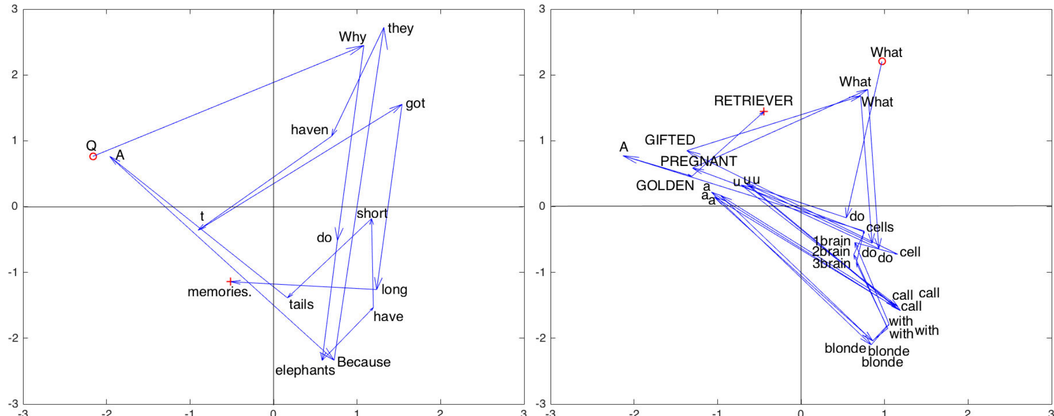


Figure 7 Climax trajectories for a single-laugh joke (left) and a three-laughs one (right).

in Figure 7(left). There are however other common moves: for example, the climax of a joke (i.e., where the reader or listener is expected to smile or laugh) is often mapped to a significant skew towards the left (negative x coordinates in Figure 7). In many cases, the cue word for the climax is in fact the left-most point among function words. The joke that generates the traces in Figure 7(left) is:

> Q: Why do elephants have short tails? A: Because they haven't got long memories.

Since many jokes reach their climax at the end, gradually building tension until the final release, it might be difficult to distinguish climax from termination. However, our data set included a few multiple-release jokes, and in analyzing the corresponding traces the same phenomenon could be observed for each of the climaxes. As an example, Figure 7 right shows the trace for the following joke:[5]

> What do u call a blonde with 1 brain cell? GIFTED! What do u call a blonde with 2 brain cells? PREGNANT! What do u call a blonde with 3 brain cells? A GOLDEN RETRIEVER!

6 Conclusions

In this work, we have reported on our results with a *computational* approach to literary theories of humor. We trained a particular form of neural network on a corpus of jokes, and "dissected" the artificial mind so generated by analyzing its contents. Our results confirm that certain typical humorous constructions, such as those postulated by Superiority or Incongruity theories, really exist in our corpus, and are sufficiently objective that the learning algorithm underlying the workings of a neural network can identify them.

What is even more striking, it appears that the neural network seems to have conceptualized some key aspects, characters and themes of humour literature by simply looking at joke texts character by character. The network is, in fact, oblivious of the structure of

[5] It is worth to remark that the joke was on a single line, as reported verbatim, so the presence of new-line characters is ruled out as an explanation for the detection of climax. Also, while in this particular example the comic answers are in all capitals, other jokes using lower case exhibit the same behaviour, so all-caps is also ruled out as a marker.

the grammar or even the structure of legal words in the language. Nevertheless, it is able to construct a representation of the world where a *blonde* and a *goose* are associated to a common interpretation, that is radically different from the representation associated to *girl* or to words identifying other animals. In a sense, we have seen how teaching a network on joke literature makes it quite prone to develop a *sexist* and *racist* view of the world, apparently confirming the very old idea that our language helps shaping the way we see the world.

References

1. Aristotle. *Poetics*, volume two: On Comedy. IV century BCE. Last surviving copy reportedly lost in the fire of the Abbey's Library in 1327, according to [6].
2. Salvatore Attardo. *Linguistic Theories of Humor*. De Gruyter Mouton, Berlin, 1994.
3. Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *Neural Networks, IEEE Transactions on*, 5(2):157–166, 1994.
4. Yu-Chen Chan. Emotional structure of jokes: A corpus-based investigation. *Bio-medical materials and engineering*, 24(6):3083–3090, 2014.
5. Yu-Chen Chan and Joseph P Lavallee. Temporo-parietal and fronto-parietal lobe contributions to theory of mind and executive control: an fMRI study of verbal jokes. *Frontiers in psychology*, 6, 2015.
6. Umberto Eco. *Il nome della rosa*. Bompiani, 1980.
7. Paolo Frasconi, Marco Gori, and Alessandro Sperduti. A general framework for adaptive processing of data structures. *Neural Networks, IEEE Transactions on*, 9(5):768–786, 1998.
8. Lisa Friedland and James Allan. Joke retrieval: Recognizing the same joke told differently. In *Proceedings of the 17th ACM Conference on Information and Knowledge Management*, CIKM'08, pages 883–892, New York, NY, USA, 2008. ACM.
9. Alex Graves. Generating sequences with recurrent neural networks. *CoRR*, abs/1308.0850, 2013. URL: `http://arxiv.org/abs/1308.0850`.
10. Alex Graves and Jürgen Schmidhuber. Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural Networks*, 18(5):602–610, 2005.
11. Ulrich Günther. *What is in a laugh? Humour, jokes and laughter in the conversational corpus of the BNC*. PhD thesis, Albert-Ludwigs-Universität, Freiburg, 2003.
12. Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
13. Laurens Van Der Maaten. Accelerating t-sne using tree-based algorithms. *The Journal of Machine Learning Research*, 15(1):3221–3245, 2014.
14. Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(2579-2605):85, 2008.
15. Wendel Wallach and Colin Allen. *Moral Machines: Teaching Robots Right from Wrong*. Oxford University Press, November 2008.

Hanabi is NP-Complete, Even for Cheaters Who Look at Their Cards

Jean-Francois Baffier[1,9], Man-Kwun Chiu[2,9], Yago Diez[3], Matias Korman[4], Valia Mitsou[5], André van Renssen[6,9], Marcel Roeloffzen[7,9], and Yushi Uno[8]

1 **National Institute of Informatics (NII), Tokyo, Japan**
jf_baffier@nii.ac.jp
2 **National Institute of Informatics (NII), Tokyo, Japan**
chiumk@nii.ac.jp
3 **Tohoku University, Sendai, Japan**
yago@dais.is.tohoku.ac.jp
4 **Tohoku University, Sendai, Japan**
mati@dais.is.tohoku.ac.jp
5 **SZTAKI, Hungarian Academy of Sciences, Hungary**
vmitsou@sztaki.hu
6 **National Institute of Informatics (NII), Tokyo, Japan**
andre@nii.ac.jp
7 **National Institute of Informatics (NII), Tokyo, Japan**
marcel@nii.ac.jp
8 **Department of Mathematics and Information Sciences, Graduate School of Science, Osaka Prefecture University, Japan**
uno@mi.s.osakafu-u.ac.jp
9 **JST, ERATO, Kawarabayashi Large Graph Project**

Abstract

This paper studies a cooperative card game called Hanabi from an algorithmic combinatorial game theory viewpoint. The aim of the game is to play cards from 1 to n in increasing order (this has to be done independently in c different colors). Cards are drawn from a deck one by one. Drawn cards are either immediately played, discarded or stored for future use (overall each player can store up to h cards). The main feature of the game is that players know the cards their partners hold (but not theirs. This information must be shared through hints).

We introduce a simplified mathematical model of a single-player version of the game, and show several complexity results: the game is intractable in a general setting even if we forego with the hidden information aspect of the game. On the positive side, the game can be solved in linear time for some interesting restricted cases (i.e., for small values of h and c).

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, G.2 Discrete Mathematics, F.1.2 Modes of Computation

Keywords and phrases algorithmic combinatorial game theory, sorting

Digital Object Identifier 10.4230/LIPIcs.FUN.2016.4

1 Introduction

When studying mathematical puzzles or games, mathematicians and computer scientists are often interested in winning strategies, designing computer programs that play as well (or even better than) humans. The computational complexity field studies the computational

© Jean Francois Baffier, Man-Kwun Chiu, Yago Diez, Matias Korman, Valia Mitsou, André van Renssen, Marcel Roeloffzen, and Yushi Uno;
licensed under Creative Commons License CC-BY
8th International Conference on Fun with Algorithms (FUN 2016).
Editors: Erik D. Demaine and Fabrizio Grandoni; Article No. 4; pp. 4:1–4:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

complexity of the games. That is, how hard it is to obtain a solution to a puzzle or to decide the winner or loser of a game [6, 10, 11]. Another interest is to design algorithms to obtain solutions. Some games and puzzles of interest include for example Nim, Hex, Sudoku, Tetris, and Go. Recently, this field has been called ‘algorithmic combinatorial game theory’ [11] to distinguish it from games arising from other fields, especially classical economic game theory.

In this paper we study a cooperative card game called Hanabi. Designed by Antoine Bauza and published in 2010, the game has received several tabletop game awards (including the prestigious *Spiel des Jahres* in 2013 [8]). In the game the players simulate a fireworks show[1], playing cards of different colors in increasing order.

In this paper we study the game from the viewpoint of algorithmic combinatorial game theory. We first propose mathematical models of a single-player variant of Hanabi, and then analyze their computational complexities. As done previously for other multiplayer card games [7, 12], we show that even a single-player Hanabi is computationally intractable in general, while the problem becomes easy under very tight constraints.

1.1 Rules of the Game

Hanabi is a multi-player, imperfect-information cooperative game. This game is played with a deck of fifty cards, where each card has a number (from 1 to 5) and a suit (or a color) out of five colors (red, yellow, green, blue and white). There are ten cards of each suit. The values of the cards are 1, 1, 1, 2, 2, 3, 3, 4, 4, and 5, respectively. That is, there are two copies of each card except for the lowest and highest cards of each color (that appear three and one time, respectively). Players must cooperate to play the cards from 1 to 5 in increasing order for all suits independently.

One of the most distinctive features of the game is that players cannot see their own cards while playing: each player holds his cards so that they can be seen by other players (but not himself). A player can do one of the following actions in each turn: play a card, discard a card from his hand to draw a new one, or give a hint to another player on what type of cards he or she is holding in hand. A second characteristic of this game is that each player can hold only a small number of cards in hand (4 or 5 depending on the number of players) drawn at random from the deck. Whenever no card is playable, a player may discard a card and draw a new one from the deck. See Appendix A for the exact rules of the Hanabi game, or [1, 2] for more information on the game.

1.2 Related Work

There is an extensive amount of research that studies the complexity of tabletop and card games. In virtually all games, the total description complexity of the problem is bounded by a constant, thus they can be solved in constant time by an exhaustive search approach.

Thus, the literature focuses on the extensions of those games in which the complexity is not constant. For example, it is known that determining the winner in chess on an $n \times n$ board needs exponential time [9]. If the playing board can be any graph, *Pandemic* (a popular tabletop game in which players try to prevent a virus from spreading) is NP-complete [13]. A somehow more surprising result is that determining the winner in *Settlers of Catan* is also an NP-complete problem, even after the game has ended [5].

When considering card games, the complexity is often expressed as a function of the number of cards in the deck. The popular trading card game *Magic: The Gathering* is Turing

[1] The word *hanabi* means fireworks in Japanese.

complete [3]. That is, it can simulate a Turing machine (and in particular, it can simulate any other tabletop or card game).

There is little research that studied algorithmic aspects of Hanabi. Most of the existing research [14, 4] propose different strategies so that players can share information and collectively play as many cards as possible. Several heuristics are introduced, and compared to either experienced human players or to optimal play sequences (assuming all information is known).

Our approach diverges from the aforementioned studies. We show that, even if we forego its hidden information trademark feature, the game is intractable, which means that there is an intrinsic difficulty in Hanabi beyond information exchange. In fact we show hardness for a simplified solitaire version of the game where the single player has complete information about which cards are being held in his hand as well as the exact order in which cards will be drawn from the deck.

1.3 Model and Definitions

We represent a card of Hanabi with an ordered pair (a_i, k_i), where $a_i \in \{1, \dots, n\}$ and $k_i \in \{1, \dots, c\}$. The term a_i is referred to as the *value* of the card and k_i as its *color*. The whole deck of cards is then a sequence σ of N cards. That is, $\sigma = ((a_1, k_1), \dots, (a_N, k_N))$. The *hand size* h is the maximum number of cards that the player can hold in hand at any point during the game. The *multiplicity* r of cards in a card sequence σ is the maximum number of times that any card appears in σ.

In a game, the player scans the cards in the order fixed by σ in a streaming fashion. In each turn a player has three options: play a card from their hand, discard one to get a hint token, or give a hint. After his turn, he draws a new card to replace the played/discarded one. As our model drops completely the information sharing feature of the game, we replaced the hinting move with a move where the single player takes no action thus, 'storing' the card. Since we are focused in the single player case and turn order does not matter, we allow doing several actions before redrawing. The three available options are thus: *play*, *discard* or *store* the card. If a card is discarded, it is gone and can never be used afterwards. If instead we store the card, it is saved and can be accessed afterwards (remember that at any instant of time the maximum number of cards that can be stored in hand is h). Cards can be played only in increasing order for each color independently. That is, we can play card (a_i, k_i) if and only if the last card of color k_i that was played was $(a_i - 1, k_i)$ (or $a_i = 1$ and no card of color k_i has been played). After a card has been played we can also play any cards we may have stored in hand in the same manner. The objective of the game is to play all cards from 1 to n in all c colors. Whenever this happens we say that the sequence of play/discard/store is a *winning play sequence*.

Thus, a problem instance of the SOLITAIRE HANABI (or HANABI for short) consists of a hand size $h \in \mathbb{N}$ and a card sequence σ of N cards (where each card is an ordered pair of a value and a color out of n numbers and c colors, and no card appears more than r times). The aim is to determine whether or not there is a winning play sequence for σ that never stores more than h cards in hand.

1.4 Results and Organization

In this paper, we study computational complexity and algorithmic aspects of HANABI with respect to parameters N, n, c, r and h. Unfortunately the problem is NP-complete, even if

Table 1 Summary of the different results presented in this paper, where N, n, c, r and h are the number of cards, the number of values, the number of colors, multiplicity, and the hand size, respectively.

Case Studied	Approach Used	Running Time	Observations
$r = 1$	Greedy	$O(N) = O(cn)$	Lemma 1 in Sec. 2
$c = 1$	Lazy	$O(N + n \log h)$	Theorem 4 in Sec. 3
General Case	Dynamic Programming	$O(Nhc^h n^{h+c-1})$	Theorem 6 in Sec. 4
$h = 2$, $r = 2$	NP-complete		Theorem 7 in Sec. 5

we fix some parameters to be small constants. Specifically, in Section 5 we show that the problem is NP-complete even if we restrict ourselves to the case in which $h = 2$ and $r = 2$.

Given the negative results, we focus on the design of algorithms for particular cases. For those cases, our aim is to design algorithms whose running time is linear in N (the total number of cards in the sequence), but we allow slightly larger running times as a function of n, k, and r (the total number of values, colors and multiplicity, respectively).

In Section 2 we give a straightforward $O(N)$ algorithm for the case in which $r = 1$ (that is, no card is repeated in σ). This approach is afterwards extended for $c = 1$ (and unbounded r) in Section 3. In Section 4 we give an algorithm for the general problem. Note that the algorithm runs in exponential time (expected for an NP-complete problem), but the running time reduces to $O(N)$ whenever, h, c and n are constants. The exact running times of all algorithms introduced in this paper are summarized in Table 1.

2 Unique Appearance

As a warm-up, we consider the case in which each card appears only once (i.e., $r = 1$). In this case we have exactly one card for each value and each color. Thus, $N = cn$ and the input sequence σ is a permutation of the values from 1 to n in the c colors.

Since each card appears only once, we cannot discard any card in the winning play sequence. In the following, we show that the natural greedy strategy is essentially the best we can do: play a card as soon as it is found (if possible). If not, store it in hand until it can be afterwards played.

The game rules state that we cannot play a card (a_i, k_i) until all the cards from 1 to $a_i - 1$ of color k_i have been played. Thus, we associate an interval to each card that indicates for how long that card must be held in hand. For any card (a_i, k_i), let f_i be the largest index of the cards of color k_i whose value is at most a_i (i.e., $f_i = \max_{j \leq N}\{j \colon k_j = k_i, a_j \leq a_i\}$). Note that we could have $i = f_i$, but this only happens when all cards of value smaller than a_i appear before card (a_i, k_i). Otherwise, we must have $f_i > i$, and card (a_i, k_i) cannot be played until we have reached card (a_{f_i}, k_{f_i}).

We associate each index i to the interval $[i, f_i]$. Let $\mathcal{I}$ be the collection of all nonempty such intervals. Let w be the maximum number of intervals that overlap (i.e., $w = \max_{j \leq N} |\{[i, f_i] \in \mathcal{I} \colon j \in [i, f_i]\}|$).

▶ **Lemma 1.** *There is a solution to any* Hanabi *problem instance with $r = 1$ and hand size h if and only if $w \leq h$. Moreover, a play sequence can be found in $O(N)$ time.*

Proof. Intuitively speaking, any interval $[i, j] \in \mathcal{I}$ represents the need of storing card (a_i, k_i) until we have reached card (a_j, k_j). Thus, if two (or more) intervals overlap, then the corresponding cards must be stored simultaneously. By definition of w, when processing the

input sequence at some point in time we must store more than h cards, which in particular implies that no winning play sequence exists.

In order to complete the proof we show that the greedy play strategy works whenever $w \leq h$. The key observation is that, for any index i we can play card (a_i, k_i) as soon as we have reached the f_i-th card. Indeed, by definition of f_i all cards of the same color whose value is a_i or less have already appeared (and have been either stored or played). Thus, we can simply play the remaining cards (including (a_i, k_i)) in increasing order.

Overall, each card is stored only within its interval. By hypothesis, we have $w \leq h$, thus we never have to store more than our allowed hand size. Furthermore, no card is discarded in the play sequence, which in particular implies that the greedy approach will give a winning play sequence with hand size h.

Regarding running time, it suffices to show that each element of σ can be treated in constant time. For the purpose, we need a data structure that allows insertions into $\mathcal{H}$ and membership queries in constant time. The simplest data structure that allows this is a hash table. Since we have at most h elements (out of a universe of size cn) it is easy to have buckets whose expected size is constant.

The only drawback of hash tables is that the algorithm is randomized (and the bounds on the running time are expected). If we want a deterministic worst case algorithm, we can instead represent $\mathcal{H}$ with a $c \times n$ bit matrix and an integer denoting the number of elements currently stored. With either data structure it is straightforward to see that insertions, removals, and membership queries take constant time, thus the algorithm takes $O(N) = O(cn)$ time as claimed. ◀

3 Lazy Strategy for One Color

We now study the case in which all cards belong to the same suit (i.e., $c = 1$). Note that we make no assumptions on the multiplicity or any other parameters. Unlike the last section in which we considered a greedy approach, here we describe a lazy approach that plays cards at the last possible moment.

We start with an observation that allows us to detect how important a card is. For any $i \leq N$, we say that the i-th card (whose value is a_i) is *useless* if there exist $w_1, \ldots, w_{h+1} \in \mathbb{N}$ such that:

(i) $a_i < w_1 < \cdots < w_{h+1} \leq n$

(ii) $\forall j \in \{i+1, \ldots, N\}$ it holds that $a_j \notin \{w_1, \ldots, w_{h+1}\}$

That is, there exist $h+1$ values that are higher than a_i none of which appears after the i-th card in σ. Observe that, for example, no card of value $n-h$ or higher can be useless (since the w_i values cannot exist) and that the last card is useless if and only if $a_N < n-h$.

▶ **Observation 2.** *Useless cards are never played in a winning play sequence.*

Proof. Assume, for the sake of contradiction, that there exists a winning play sequence that plays some useless card whose index is i. Since we play cards in increasing order, no card of value equal to or bigger than a_i can have been played at the time in which the i-th card is scanned. By definition of useless, the remaining sequence does not have more cards of values $w_1, \ldots, w_{h+1}$. Thus, in order to complete the game to a winning sequence, these $h+1$ cards must all have been stored, but this is not possible with a hand size of h. ◀

Our algorithm starts with a filtering phase that removes all useless cards from σ. The main difficulty of this phase is that the removal of some useless cards from σ may create further useless cards, and so on. In order to avoid having to scan the input several times

we use two vectors and a max-heap as follows: for each index $i \leq N$, we store the index of the previous occurrence of the same card in a vector P (or $-\infty$ if none exists). That is, $P[i] = -\infty$ if and only if $a_j \neq a_i$ for all $j < i$. Otherwise, we have $P[i] = i'$ (for some $i' < i$), $a_i = a_{i'}$, and $a_j \neq a_i$ for all $j \in \{i'+1, \ldots, i-1\}$. We also use a vector L such that each index $i \leq n$ stores the last non-useless card of value i (since initially no card has been detected as useless, the value $L[i]$ is initialized to the index of the last card with value i in σ). Finally, we use a max-heap $\mathcal{HP}$ of $h+1$ elements initialized with values $L[n-h], \ldots, L[n]$.

Now, starting with $i = n-(h+1)$ down to 1 we look for all useless copies of value i. The invariant of the algorithm is that for any $j > i$, all useless cards of value j have been removed from σ and that vector $L[j]$ stores the index of the last non-useless card of value j. The heap $\mathcal{HP}$ contains the smallest $h+1$ values among $L[j], \ldots, L[n]$ (and since it is a max-heap we can access in constant time its largest value). These values will be the smallest possible candidate values for the witnesses $w_1, \ldots, w_{h+1}$ (properly speaking, $\mathcal{HP}$ stores indices, but the values can be extracted in constant time). The invariants are satisfied for $i = n-(h+1)$ directly by the way in which L and $\mathcal{HP}$ is initialized.

Any card of value i whose index is higher than the top of the heap is useless and can be removed from σ (the indices in the heap $\mathcal{HP}$ act as witnesses). Starting from $L[i]$, we remove all useless cards of value i from σ until we find a card of value i whose index is smaller than the top of the heap. If no card of value i remains we stop the whole process and return that the problem instance has no solution. Otherwise, we have found the last non-useless card of value i. We update the value of $L[i]$ since we have just found the last non-useless card of that value. Finally, we must update the heap $\mathcal{HP}$. As observed above, the value of $L[i]$ must be smaller than the largest value of $\mathcal{HP}$ (otherwise it would be a useless card). Thus, we remove the highest element of the heap, and insert $L[i]$ instead. Once this process is done, we proceed to the next value of i. Let σ' be the result of filtering σ with the above algorithm.

▶ **Lemma 3.** *The filtering phase removes only useless cards from σ. Moreover, this process runs in $O(N + n \log h)$ time, and σ' contains no useless cards.*

Proof. Each time we remove a card from σ, the associated $h+1$ witnesses $w_1, \ldots w_{h+1}$ are present in $\mathcal{HP}$, thus the first claim follows. The fact that no more useless cards remain follows from the fact that we always store the smallest possible witness values.

Now we bound the running time. The heap is initialized with $h+1$ elements, and during the whole depurating phase $O(n)$ elements are pushed. Hence, this part takes $O(n \log h)$ time. Vector P and L can be initialized by scanning σ once. During the iterative phase we can access the last occurrence of any value by using vector L. Once a card is removed, we can update the last occurrence using P. Thus, we spend constant time per card that is removed from σ (hence, overall $O(N)$ time). ◀

Now we describe the algorithm for our lazy strategy. The play sequence is very simple: we ignore all cards except when a card is the last one of that value present in σ'. For those cards, we play them (if possible) or store them (otherwise). Whenever we play a card, we play as many cards as possible (out of the ones we had stored).

Essentially, there are two possible outcomes after the filtering phase. It may happen that all cards of some value were detected as useless. In this case, none of those cards may be played and thus the HANABI problem instance has no solution. Otherwise, we claim that our lazy strategy will yield a winning play sequence.

▶ **Theorem 4.** *We can solve a* HANABI *problem instance for the case in which all cards have the same color (i.e., $c = 1$) in $O(N + n \log h)$ time.*

Proof. It suffices to show that our lazy strategy will always give a winning play sequence, assuming that the filtered sequence contains at least a card of each value. Our algorithm considers exactly one card of each value from 1 to n. The card will be immediately played (if possible) or stored until we can play it afterwards. Thus, the only problem we might encounter would be the need to store more than h cards at some instant of time.

However, this cannot happen: assume, for the sake of contradiction, that at some instant of time we need to store a card (whose index is j) and we already have stored cards of values $a_{i_1}, \ldots, a_{i_h}$. By construction of the strategy, there cannot be more copies of cards with value $a_{i_1}, \ldots, a_{i_h}$ or a_j in the remaining portion of σ'. Let p be the number of cards that we have played at that instant of time. Remember that we never store a card that is playable, thus $p+1 \notin \{a_j, a_{i_1}, \ldots, a_{i_h}\}$. In particular, the last card of value $p+1$ must be present in the remaining portion of σ'. However, that card is useless (the values $\{a_j, a_{i_1}, \ldots, a_{i_h}\}$ act as witnesses), which gives a contradiction.

Thus, we conclude that the lazy strategy will never need to store more than h cards at any instant of time, and it will yield a winning play sequence. Finally, observe that the sequence itself can be reconstructed, since vector L stores the last non-useless occurrence of each value. ◀

4 General Case Algorithm

In this section we study the general problem setting. Recall that this problem is NP-complete, even if the hand size is small (see details in Section 5), hence we cannot expect an algorithm that runs in polynomial-time. In the following, we give an algorithm that runs in polynomial time provided that both h and c are fixed constants (or exponential otherwise).

We solve the problem using a dynamic programming approach. Specifically, we build a table $DP[s, \mathcal{H}, p_1, \ldots, p_{c-1}]$ indexed by the following $c+1$ parameters:

- s ($\leq N$) represents the number of cards from the sequence σ that we allow to scan.
- $\mathcal{H}$ is the set of cards that we require to store in hand after card (a_s, k_s) has been processed. We might have no requirements on what needs to be in hand, in which case we simply set $\mathcal{H} = \emptyset$.
- $p_1, \ldots, p_{c-1}$ ($\leq n$) encode how many cards we require to play in the first $c-1$ colors, respectively.

The entry of the table $DP[s, \mathcal{H}, p_1, \ldots, p_{c-1}]$ is a positive number equal to the maximum number of cards of the c-th color that we can play among all play sequences that preserve the above constraints. Whenever such a sequence is not feasible (i.e., we cannot play the required cards in some color or store the cards of $\mathcal{H}$), we simply set that position of the table to $-\infty$.

For example, if $c = 3$, the entry of table $DP[42, \{(15,1), (10,2)\}, 10, 4] = 6$ should be interpreted as *There is a play sequence that, after scanning through the 42 cards of σ has played exactly 10 cards of the first color, 4 of the second, 6 of the third, and has stored cards $(15,1)$ and $(10,2)$ in hand. Moreover, there is no play sequence that, after scanning the first 42 cards, plays 10, 4, and 7 cards of the three colors (respectively) and ends up with cards $(15,1)$ and $(10,2)$ in hand.*

When s is a small number we can find the solution of an entry by brute force (try all possibilities of discarding, storing or playing the first s cards). This takes constant time since the problem has constant description complexity. Similarly, we have $DP[s, \mathcal{H}, p_1, \ldots, p_{c-1}] = -\infty$ whenever $|\mathcal{H}| > h$ (because we need to store more than h cards in hand). In the following

we show how to compute the table DP for the remaining cases. For this purpose, we define three auxiliary values $\mathcal{D}$, $\mathcal{S}$, and $\mathcal{P}$ (which stand for *Discard*, *Store*, and *Play*) as follows:[2]

$$\mathcal{D} = DP[s-1, \mathcal{H}, p_1, \ldots, p_{c-1}]$$

$$\mathcal{S} = \begin{cases} -\infty & \text{if } |\mathcal{H}| \geq h \\ DP[s-1, \mathcal{H} \setminus \{(a_s, k_s)\}, p_1, \ldots, p_{c-1}] & \text{otherwise} \end{cases}$$

and

$$\mathcal{P} = \begin{cases} -\infty & \text{if } k_s < c,\ a_s > p_{k_s} \\ DP[s-1, \mathcal{H} \cup \{(a_s{+}1, k_s), \ldots, (p_{k_s}, k_s)\}, & \\ \qquad p_1, \ldots, p_{k_s-1}, a_s{-}1, p_{k_s+1}, \ldots, p_{c-1}] & \text{if } k_s < c,\ a_s \leq p_{k_s} \\ \max_{t \in \{0,\ldots,h\}} \{a_s{+}t \colon DP[s-1, \mathcal{H} \cup \{(a_s{+}1, c), \ldots, (a_s{+}t, c)\}, & \\ \qquad p_1, \ldots, p_{c-1}] \geq a_s{-}1\} & \text{if } k_s = c \end{cases}$$

These auxiliary values allow us to compute an entry of the table efficiently.

▶ **Lemma 5.** $DP[s, \mathcal{H}, p_1, \ldots, p_{c-1}] = \max\{\mathcal{P}, \mathcal{S}, \mathcal{D}\}$.

Proof. Assume that $DP[s, \mathcal{H}, p_1, \ldots, p_{c-1}]$ is a positive number (i.e., it is feasible to satisfy all constraints). Consider any play sequence that realizes it, we distinguish three cases depending on what the play sequence does with card (a_s, k_s):

(a_s, k_s) is discarded. When the last card is discarded, the entry of the table is the same as if we only allow the scanning of $s-1$ cards. Thus, $DP[s, \mathcal{H}, p_1, \ldots, p_{c-1}] = DP[s-1, \mathcal{H}, p_1, \ldots, p_{c-1}] = \mathcal{D}$.

(a_s, k_s) is stored. Since this card is the last one that we are allowed to scan for the entry of the table that we are computing, storing (a_s, k_s) only makes sense if $(a_s, k_s) \notin \mathcal{H}$. Moreover, this operation is only possible if we do not exceed the hand size limit. That is, we have $DP[s, \mathcal{H}, p_1, \ldots, p_{c-1}] = DP[s-1, \mathcal{H} \setminus \{(a_s, k_s)\}, p_1, \ldots, p_{c-1}]$ if $|\mathcal{H}| < h$ (otherwise it should be $-\infty$). This coincides with the definition of $\mathcal{S}$.

(a_s, k_s) is played. In this case we claim that $DP[s, \mathcal{H}, p_1, \ldots, p_{c-1}] = \mathcal{P}$. In order to prove this we give an intuitive definition of $\mathcal{P}$. We consider three subcases depending on the color and value of card (a_s, k_s).

$k_s < c$ and $a_s > p_{k_s}$. Recall we only need to play up to card p_{k_s} in color k_s (and this card is of higher value). In particular, the card need not be played in the play sequence. We set $\mathcal{P} = -\infty$ to make sure that this case is not considered by our algorithm.

$k_s < c$ and $a_s \leq p_{k_s}$. Consider only color k_s: we are required to play p_{k_s} cards and in order to do that we must specifically use card (a_s, k_s). In order to do so, we must have played the first $a_s - 1$ cards of this color in advance, and must have the cards from $a_s + 1$ to p_{k_s} in hand. All constraints for other colors are unaffected. Thus, we have $DP[s, \mathcal{H}, p_1, \ldots, p_{c-1}] = DP[s-1, \mathcal{H} \cup \{(a_s+1, k_s), \ldots, (p_s, k_s)\}, p_1, \ldots, p_{k_s-1}, a_s - 1, p_{k_s+1}, \ldots, p_{c-1}] = \mathcal{P}$ as claimed.

---

[2] Note that, strictly speaking, these terms depend on the parameters $s, \mathcal{H}, p_1, \ldots, p_{c-1}$. Thus, a better notation would be $\mathcal{P}_{s,\mathcal{H},p_1,\ldots,p_{c-1}}$, $\mathcal{S}_{s,\mathcal{H},p_1,\ldots,p_{c-1}}$, and $\mathcal{D}_{s,\mathcal{H},p_1,\ldots,p_{c-1}}$. Since the subindices are clear from the context, we remove them for ease of reading.

$k_s = c$. This case is similar to the previous one. In this case we focus on color c $(= k_s)$: as before, we need card (a_s, k_s) to be playable when we reach this card. This constraint is realized by restricting to entries of the table that allow to play at least $a_s - 1$ cards of color c (i.e., $DS[\cdot] \geq a_s - 1$).

Recall that our aim is to play as many cards of color c as possible. Thus, if we want to play t additional cards after (a_s, c) but are not allowed to scan more cards of the input, those t cards must have been previously stored. Thus, we are interested in the largest value of t so that cards of values $a_s + 1, \ldots, a_s + t$ of color c can be stored in hand while making sure that all constraints are satisfied.

Note that t can be as small as zero (i.e., when we do not store additional cards) and at most h (since we cannot store more than h cards at any instant of time), giving $DP[s, \mathcal{H}, p_1, \ldots, p_{c-1}] = \mathcal{P}$ as claimed.

Thus, when $DP[s, \mathcal{H}, p_1, \ldots, p_{c-1}]$ is a positive number, we have equality from the fact that we query feasible moves (thus $DP[s, \mathcal{H}, p_1, \ldots, p_{c-1}] \geq \max\{\mathcal{P}, \mathcal{S}, \mathcal{D}\}$) and exhaustiveness (since we try all options, the largest of them must satisfy $\max\{\mathcal{P}, \mathcal{S}, \mathcal{D}\} \geq DP[s, \mathcal{H}, p_1, \ldots, p_{c-1}]$). Similarly, if an entry of DP is unbounded, its associated values $\mathcal{P}$, $\mathcal{S}$ and $\mathcal{D}$ will also be unbounded (since a bounded number would be a witness of a winning play sequence). ◀

▶ **Theorem 6.** *We can solve a* Hanabi *problem instance in $O(Nhc^h n^{h+c-1})$ time using $O(c^h n^{h+c-1})$ space.*

Proof. By definition, there is a solution to the Hanabi problem instance if and only if its associate table satisfies $DP[N, \emptyset, n \ldots, n] = n$. Each entry of the table is solved by querying entries that have a smaller value in the first parameter, so we can compute the whole table in increasing order.

Recall that, entries of the table for which the associated set $\mathcal{H}$ has more than h elements the answer is trivially $-\infty$ (since we cannot store that many cards). Thus, table DP will have $N \times \sum_{i \leq h} \binom{nc}{i} \times n^{c-1} \in O(Nc^h n^{h+c-1})$ nontrivial entries. Note that when computing them, each entry of the table queries for entries whose value of s is one smaller. Thus, after the values of some value of s have been computed, the smaller ones need not be stored. This way we never need to store more than $O(c^h n^{h+c-1})$ entries at the same time.

We now bound the time needed to compute a single entry of the table (say, $DP[s, \mathcal{H}, p_1, \ldots, p_{c-1}]$). First notice that we can compute $\mathcal{P}$ and $\mathcal{S}$ with a constant number of queries to the table. Each query makes at most one insertion or removal into $\mathcal{H}$. Such insertions can be handled in constant time (see the proof of Lemma 1), thus overall they are computed in constant time.

In order to compute $\mathcal{P}$, we may have to do $O(h)$ queries onto the DP table and insert $O(h)$ elements into $\mathcal{H}$. Since each of these operations take constant time, we need $O(h)$ time to compute a single entry (and $O(Nc^h n^{h+c-1} h)$ for the whole table as claimed). ◀

▶ Remark. In principle, the DP table only returns whether or not the instance is feasible. We note that, we can also find a winning play sequence with the usual backtracking techniques.

5 NP-Hardness (Multiple Colors, Multiple Appearances)

In this section we prove hardness of the general Hanabi problem. As mentioned in the introduction, the problem is NP-complete even if h and r are small constants.

▶ **Theorem 7.** *The* Hanabi *problem is NP-complete for any $r \geq 2$ and $h \geq 2$.*

We prove the statement for $r = 2$, $h = 2$ and then show how to generalize it for larger values of r and h. Our reduction is from 3-SAT. Given a 3-SAT problem instance with v variables $x_1, \ldots, x_v$ and m clauses $W_1, \ldots, W_m$, we construct a Hanabi sequence σ with $2v+1$ colors, $n = 6m+2$, $r = 2$, $h = 2$ (and thus $N \leq 2(2v+1)(6m+2)$).

Before discussing the proof, we provide a birds-eye view of the reduction. The generated sequence will have a variable gadget V_i for each variable x_i and a clause gadget C_j for each clause W_j.

In the first phase of the game, the *variable assigning phase*, the player scans through the variable gadgets $V_i, i \leq v$. The variable gadget V_i associated to the i-th variable will have cards of colors $2i-1$ and $2i$. After we have scanned through gadget V_i, the best we can do is play at most 5 cards of one color, and 1 from the other one. We assign a truth value to a variable depending on which of the two colors we played five cards. By repeating this in all variable gadgets we obtain a truth assignment.

In the next phase, the *clause satisfaction* phase, the player scans through the clause gadgets $C_j, j \leq m$. The clause gadget C_j corresponding to clause W_j is constructed in a way that when we scan through it we can play five additional cards (of all colors) if and only if the truth assignment satisfies the clause W_j. Thus, only when all clauses are satisfied the Hanabi instance will have a solution.

As will be shown afterwards, it will be useful to temporarily reduce the amount of cards that can be stored in hand (or even make it zero). This can be enforced with an additional dummy color $2v+1$. Indeed, assume that when scanning a portion λ of the input we want to make sure that hand size is one (for simplicity, we also assume that no card of the dummy color appears in the whole sequence). Then, it suffices to add cards $(2, 2v+1)$ and $(1, 2v+1)$ before and after λ, respectively. Since card $(2, 2v+1)$ appears exactly once, then its unique appearance must be stored until card $(1, 2v+1)$ is found. Thus, only one additional card can be stored while scanning λ. We call this gadget the *hand reduction gadget*.

Similarly, we can enforce independence between two gadgets by adding cards $(4, 2v+1), (5, 2v+1)$, and $(3, 2v+1)$ between them. If cards $(4, 2v+1)$ and $(5, 2v+1)$ appear exactly once in the sequence, they must be stored until the card $(3, 2v+1)$ is scanned (at which point all three cards can be played). Since our hand size is two, this essentially makes sure that no card can be stored between gadgets. Note that this trick can be done arbitrarily many times provided that each time we use higher numbers of the dummy color (and each card appears only once in the whole sequence). We call this operation the *hand dump gadget*. Note that the gadgets are used to simulate a reduction in the hand size, but h remains constant. The temporary reduction is created by forcing some cards to be stored during a portion of the play sequence.

Let us first consider the variable assigning phase. We first describe the variable gadget. For any $i \leq v$, variable gadget V_i is defined as the sequence $V_i = 2, \overline{2}, 1, 3, 4, 5, \overline{1}, \overline{3}, \overline{4}, \overline{5}$, where overlined values are cards of color $2i$, whereas the other cards have color $2i-1$. The first part of the Hanabi problem instance σ simply consists of the concatenation of all gadgets $V_1, \ldots, V_v$, adding card $(2, 2v+1)$ in the very beginning and card $(1, 2v+1)$ in the very end of the sequence, as to form a hand reduction gadget (see Figure 1). We call this sequence σ_1.

▶ **Lemma 8.** *There is no valid play sequence of σ_1 that can play cards of value 2 of colors $2i-1$, $2i$ and the dummy color $2v+1$. This statement holds for all $i \leq v$.*

Proof. Assume, for the sake of contradiction, that there exists some $i \leq v$ and a sequence of plays for which we can play the three cards. In order to play card $(2, 2v+1)$ we need to store it in the very beginning of the game enforcing the hand reduction gadget for the duration of the variable assigning phase, thus temporarily reducing the hand-size to one.

2	2 2 1 3 4 5 1 3 4 5	2 2 1 3 4 5 1 3 4 5	2 2 1 3 4 5 1 3 4 5	1
d	1 2 1 1 1 1 2 2 2 2	3 4 3 3 3 3 4 4 4 4	5 6 5 5 5 5 6 6 6 6	d

Figure 1 Sequence σ_1 for a SAT instance with three variables. The upper row represents the numbers of the cards whereas the lower one represents the color of each card. Note that the dummy cards to reduce hand size are also added (color "d" stands for dummy color).

Further notice that each card appears exactly once in σ_1 (that is, the multiplicity of this part is equal to 1), and that the cards of color $2i-1$ and $2i$ only appear in gadget V_i. More importantly, the value 2 in both colors appears before the value 1 in the respective colors. In particular, both must be stored before they are played. However, this is impossible, since we have decreased the hand size through the hand reduction gadget. ◀

From now on, for simplicity in the description, we only consider play sequences that play all the cards in the dummy color (recall that these cards appear exactly once. If any of them is not played the resulting sequence of moves cannot be completed). Similarly, we assume cards are played as soon as possible. In particular, if the card that is currently being scanned is playable, then it will be immediately played. We can make this assumption because holding it in hand is never beneficial. We call these two conditions the *smart play* assumption.

Thus, the best we can do after scanning through all variable gadgets is to play five cards of either color $2i-1$ or $2i$ (and only one card of the other color). This choice is independent for all $i \leq v$, hence we associate a truth assignment to a play sequence as follows: we say that variable x_i is set to *true* if, after σ_1 has been scanned, the card $(5, 2i-1)$ has been played, false if $(5, 2i)$ has been played. For well-definement purposes, if neither $(5, 2i-1)$ or $(5, 2i)$ we simply consider the variable as *unassigned* (and say that an unassigned variable never satisfies a clause). This definition is just used for definement purposes since, as we will see later, no variable will be unassigned in a play sequence that plays all cards.

Let us now move on to the clause satisfaction phase by first describing the clause gadget C_j for clause W_j. We associate three colors to a clause. Specifically, we associate W_j with color $2i-1$ if x_i appears positive in W_j. If x_i appears in negated form, we associate W_j with color $2i$ instead. Since each clause contains three literals, it will be associated to three distinct colors.

Let $o_j = 5(j-1)$. Intuitively speaking, o_j indicates how many cards of each color can be played (we call this the *offset*). Our invariant is that for all $i \leq v$ and $j \leq m$, before scanning through the clause gadget associated to W_j, there is a play sequence that plays up to $o_j + 1$ cards in color $2i-1$ and $o_j + 5$ in color $2i$ (or the reverse) and no play sequence can exceed those values in any color. Observe that the invariant is satisfied for $j = 1$ by Lemma 8.

The clause gadget C_j is defined as follows: we first add the sequence $o_j + 6, o_j + 7, o_j + 8, o_j + 9, o_j + 10$ for the three colors associated to W_j. Then we append the sequence $o_j+5, o_j+6, o_j+7, o_j+8, o_j+9, o_j+10, o_j+2, o_j+3, o_j+4$ in all other colors (except the dummy color). After this we add three cards of the dummy color forming a hand dump gadget. Finally, we add the sequence $o_j+3, \overline{o_j+3}, \overline{\overline{o_j+3}}, o_j+2, o_j+4, o_j+5, o_j+6, \overline{o_j+2}, \overline{o_j+4}, \overline{o_j+5}, \overline{o_j+6}, \overline{\overline{o_j+2}}, \overline{\overline{o_j+4}}, \overline{\overline{o_j+5}}, \overline{\overline{o_j+6}}$ in the three colors associated to W_j (as before, the single and double overline on the numbers is used to distinguish between the three colors). See Figure 2.

Let σ_2 be the result of concatenating all clause gadgets in order, where before each C_j we add three cards of the dummy color forming a hand dump gadget so as to make sure that no card from one gadget can be saved to the next one (see Figure 3). Further let $\sigma' = \sigma_1 \circ \sigma_2$.

We must show that, when scanning a clause gadget that is satisfied, we can play five cards of all colors. We start by showing that this is possible for the easy colors (i.e., colors for which we played five cards in σ_1 or those that are not associated to W_j).

	C_1						
4 5 3	6 7 8 9 10	5 6 7 8 9 10 2 3 4	7 8 6	3 3 3	2 4 5 6	2 4 5 6	2 4 5 6
d d d	1,4, and 5	2, 3, and 6	d d d	1 4 5	1 1 1 1	4 4 4 4	5 5 5 5

	C_2						
10 11 9	11 12 13 14 15	10 11 12 13 14 15 7 8 9	13 14 12	8 8 8	7 9 10 11	7 9 10 11	7 9 10 11
d d d	1,3, and 6	2, 4, and 5	d d d	1 3 6	1 1 1 1	3 3 3 3	6 6 6 6

Figure 2 Sequence σ_2 for a SAT instance with three variables x_1, x_2, x_3 and two clauses $W_1 = (x_1 \vee \neg x_2 \vee x_3), W_2 = (x_1 \vee x_2 \vee \neg x_3)$. Colors 1, 4, 5 are associated to W_1 and colors 1, 3, 6 are associated to W_2. The upper row represents the numbers of the cards whereas the lower one the color of each card. Note that the dummy cards to obtain independence between/inside gadgets are also added (color "d" stands for dummy color).

$$\sigma_1 \quad 4 \quad 5 \quad 3 \quad C_1 \quad 10 \quad 11 \quad 9 \quad C_2 \quad \ldots \quad 6m-2 \quad 6m-1 \quad 6m-3 \quad C_m$$

Figure 3 Overall picture of the reduction. All cards depicted have dummy color (and are only used to obtain independence between gadgets).

▶ **Lemma 9.** *Let $k \leq 2v$ be a color for which before processing clause gadget C_j we have played up to $o_j + 5$ cards of color k (for some $j \leq m$) or a color not associated to W_j for which we have played up to $o_j + 1$ cards of color k. Then, we can play up to five more cards of color k when processing the clause gadget C_j. Moreover no play sequence can play more than five cards of that color.*

Proof. Recall that there are hand dumps between different gadgets. Thus, any cards that is played while processing C_j must appear in C_j.

The case in which we played up to the value $o_j + 5$ of a color is easy, since C_j contains the sequence $o_j + 6, o_j + 7, o_j + 8, o_j + 9, o_j + 10$ in consecutive fashion in all colors. Thus, the five cards can be played without having to store anything in hand. Also note that a sixth card cannot be played since $o_j + 11$ is not present in any color in C_j.

The case in which we played up to $o_j + 1$ of a color not associated to W_j is similar. In this case, the cards of color k appear in the following order: $o_j + 5, o_j + 6, o_j + 7, o_j + 8, o_j + 9, o_j + 10, o_j + 2, o_j + 3, o_j + 4$. It is straightforward to verify that if we are only allowed to store two cards we can play at most five cards. ◀

The remaining case is that a color k is associated to W_j and only $o_j + 1$ cards have been played. Recall that, by the way in which we associated variable assignments and play sequences, this corresponds to the case that the assignment of variable $x_{\lceil k/2 \rceil}$ does not satisfy the clause W_j. We now show that five cards of color k will be playable if and only if at least one of the other two variables satisfies the clause.

▶ **Lemma 10.** *Let C_j be the clause gadget associated to W_j (for some $j \leq m$). We can play five cards in each of the three colors associated to W_j if and only if we have played card of value $o_j + 2$ in at least one of the three associated colors before W_j is processed. Moreover, we can never play more than five cards in the three colors associated to W_j.*

Proof. The proof is similar to the Lemma 9. By construction of our gadget, we first find the sequence $o_j + 6, o_j + 7, o_j + 8, o_j + 9, o_j + 10$ in all three colors associated to W_j. These cards are unplayable if we have only played up to $o_j + 1$, so the best we can do is to store them. However, before we find the smaller numbers of the same color, there is a hand dump

gadget. Thus none of these cards will be playable for colors in which, before C_j is processed, we have only played cards of value at most $o_j + 1$.

The only other cards of the associated color that are present in the gadget form the sequence $o_j + 3, \overline{o_j + 3}, \overline{\overline{o_j + 3}}, o_j + 2, o_j + 4, o_j + 5, o_j + 6, \overline{o_j + 2}, \overline{o_j + 4}, \overline{o_j + 5}, \overline{o_j + 6}, \overline{\overline{o_j + 2}}, \overline{\overline{o_j + 4}}, \overline{\overline{o_j + 5}}, \overline{\overline{o_j + 6}}$. Again, because of the hand dump gadget before and after this sequence, no other cards can be played.

Consider the case in which we have played only up to $o_j + 1$ of the three colors before processing C_j (or equivalently, the variable assignment does not satisfy clause W_j). The first three cards we find have the number $o_j + 3$ in the three colors. None is currently playable, thus ideally we would like to store them. Due to the limitations on our hand size, we can only store two of the three cards. In particular, from the color whose card $o_j + 3$ was discarded we will only be able to play one card. Note that for this situation to happen it is crucial that in none of the three colors we have played up to card $o_j + 5$. When this condition is not satisfied in at least one color we can make sure that five cards in all colors are played. ◀

From the above results we know that by the time we scan through σ' we can play at least up to value $o_m + 6$ (in half of the colors we can play up to value $o_m + 10$) if and only if the variable assignment created during the variable assignment phase satisfied all clauses. For the dummy color, we used one hand size reduction gadget and two hand dump gadgets per clause, thus $6m + 2$ cards will have been played. We pad σ' with values $o_m + 6$ to $6m + 2$ in increasing order in all colors (except the dummy color). Let σ be the resulting sequence.

▶ **Theorem 11.** *There is a valid solution of* Hanabi *for σ and $h = 2$ if and only if the associated problem instance of* 3-SAT *is satisfiable.*

Proof. If the associated problem instance of 3-SAT is satisfiable, there exists a truth assignment satisfying all clauses, by Lemmas 8, 9 and 10, we can play all colors up to the card $6m + 2$ from σ. If the associated problem instance of 3-SAT is not satisfiable, for any truth assignment, there exists one or more clauses that are not satisfied. Let j be the index of the first clause that is not satisfied by the truth assignment. By Lemma 10, we will not be able to play card $o_j + 3$ in one of the three colors associated to C_j. Since the smallest number of the next gadgets is $o_j + 7$, no more cards can be played in that color. In particular, there cannot be a solution for this Hanabi problem instance. ◀

The above reduction can be easily constructed in polynomial time. Further note that the reduction works for $r = 2$ and $h = 2$. If we want to have exactly r copies, it suffices to place next to the first appearance of each card $r - 1$ or $r - 2$ cards identical to it, so that we end up having a number of consecutive copies of each card of which only one will be useful. On the other hand, if $h > 2$ we can use a hand reduction gadget to reduce the hand size to exactly 2 for the interval in which σ is processed. This completes the proof of Theorem 7.

6 Conclusions

In this paper, we studied the complexity of a single player version of Hanabi, but the hardness also extends for the case in which we have p players. With more than one player the *give a hint* action allows a player to pass (i.e., neither draw nor discard). Thus, if we have enough hints (say, at least pN), the game with p players each with a hand size of h is equivalent to a game with a single player and hand size ph.

Even though our model is a bit far from the original game, it uncovers the importance of hand management. This was an aspect of the game that had been overlooked in previous

studies of the game. We hope that the ideas presented in this paper will help towards the design of strategies that are useful for this interesting game and that perform better than the currently existing ones.

Several questions regarding the complexity of the game remain unanswered. For example, is the game still NP-complete if we bound the number of colors instead of the hand-size? Also, can we obtain linear-time algorithms for the case where h and c are small constants (for example when $h = 1$ and/or $c = 2$)? Furthermore, as the problem is very rich in parameters, it would be fruitful to study it from a parameterized complexity point of view.

References

1 Antoine Bauza. Hanabi. `http://www.antoinebauza.fr/?tag=hanabi`.
2 BoardGameGeek. `https://boardgamegeek.com/boardgame/98778/hanabi`.
3 Alex Churchill. Magic: The gathering is Turing complete. Unpublished manuscript available at `http://www.toothycat.net/~hologram/Turing/index.html`.
4 Christopher Cox, Jessica De Silva, Philip Deorsey, Franklin H. J. Kenter, Troy Retter, and Josh Tobin. How to make the perfect fireworks display: Two strategies for Hanabi. *Mathematics Magazine*, 88(5):323–336, 2015.
5 Erik D. Demaine. Personal communication.
6 Erik D. Demaine. Playing games with algorithms: Algorithmic combinatorial game theory. *CoRR*, cs.CC/0106019v2, 2008.
7 Erik D. Demaine, Martin L. Demaine, Nicholas J. A. Harvey, Ryuhei Uehara, Takeaki Uno, and Yushi Uno. UNO is hard, even for a single player. *Theor. Comp. Sci.*, 521:51–61, 2014.
8 Spiel des Jahres award. `http://www.spieldesjahres.de/en/hanabi`.
9 Aviezri S. Fraenkel and David Lichtenstein. Computing a perfect strategy for n x n chess requires time exponential in n. *J. Comb. Theory, Ser. A*, 31(2):199–214, 1981.
10 Martin Gardner. *Mathematical Games: The Entire Collection of His Scientific American Columns*. The Mathematical Association of America, 2005.
11 Robert Hearn and Erik D. Demaine. *Games, Puzzles, and Computation*. A. K. Peters, 2009.
12 Michael Lampis and Valia Mitsou. The computational complexity of the game of set and its theoretical applications. In 11th *Latin American Symposium*, pages 24–34. Springer, 2014.
13 Kenichiro Nakai and Yasuhiko Takenaga. NP-completeness of pandemic. *JIP*, 20(3):723–726, 2012.
14 Hirotaka Osawa. Solving Hanabi: Estimating hands by opponent's actions in cooperative game with incomplete information. In *AAAI workshop: Computer Poker and Imperfect Information*, pages 37–43, 2015.

A The Rules of Hanabi

In this appendix we introduce the official rules of Hanabi [1].

Game Material

50 fireworks cards in five colors (red, yellow, green, blue, white):
- 10 cards per color with the values 1, 1, 1, 2, 2, 3, 3, 4, 4, 5,
- 5 colorful fireworks cards with values of 1, 2, 3, 4, 5,
- 8 Clock (Note) tokens (+ 1 spare),
- 3 Storm (Fuse) tokens.

Aim of the Game

Hanabi is a cooperative game, meaning all players play together as a team. The players have to play the fireworks cards sorted by colors and numbers. However, they cannot see their own hand cards, and so everyone needs the advice of his fellow players. The more cards the players play correctly, the more points they receive when the game ends.

The Game

The oldest player is appointed first player and sets the tokens in the play area. The eight Clock tokens are placed white-side-up. The three Storm tokens are placed lightning-side-down.

Now the fireworks cards are shuffled. Depending on the number of players involved, each player receives the following hand:

- With 2 or 3 players: 5 cards in hand,
- With 4 or 5 players: 4 cards in hand.

Important: For the basic game, the colorful fireworks cards and the spare Clock token(s) are not needed. They only come in to use for the advanced game.

Important: Unlike other card games, players may not see their own hand! The players take their hand cards so that the back is facing the player. The fronts can only be seen by the other players. The remaining cards are placed face down in the draw pile in the middle of the table. The first player starts.

Game Play

Play proceeds clockwise. On a player's turn, he must perform exactly one of the following:

A. Give a hint or
B. Discard a card or
C. Play a card.

The player has to choose an action. A player may not pass!

Important: Players are not allowed to give hints or suggestions on other players' turns!

A. Give a hint

To give a hint one Clock token must be flipped from its white side to its black side. If there are no Clock tokens white-side-up then a player may not choose the Give a hint action. Now the player gives a teammate a hint. He has one of two options:

1. **Color Hint.** The player chooses a color and indicates to his/her teammate which of their hand cards match the chosen color by pointing at the cards. *Important:* The player must indicate all cards of that color in their teammate's hand! Example: "You have two yellow cards, here and here." Indicating that a player has no cards of a particular color is allowed! Example: "You have no blue cards."
2. **Value Hint.** The player chooses a number value and gives a teammate a hint in the exact same fashion as a Color Hint. Example: "You have a 5, here." Example: "You have no Twos."

B. Discard a card

To discard a card one Clock token must be flipped from its black side to its white side. If there are no Clock tokens black-side-up then a player may not choose the Discard a card

action. Now the player discards one card from their hand (without looking at the fronts of their hand cards) and discards it face-up in the discard pile near the draw deck. The player then draws another card into their hand in the same fashion as their original card hands, never looking at the front.

C. Play a card

By playing out cards the fireworks are created in the middle of the table. The player takes one card from his hand and places it face up in the middle of the table. Two things can happen:

1. **The card can be played correctly.** The player places the card face up so that it extends a current firework or starts a new firework.
2. **The card cannot be played correctly.** The gods are angry with this error and send a flash from the sky. The player turns a Storm tile lightning-side-up. The incorrect card is discarded to the discard pile near the draw deck.

In either case, the player then draws another card into their hand in the same fashion as their original card hands, never looking at the front.

The Fireworks

The fireworks will be in the middle of the table and are designed in five different colors. For each color an ascending series with numerical values from 1 to 5 is formed. A firework must start with the number 1 and each card played to a firework must increment the previously played card by one. A firework may not contain more than one card of each value.

Bonus

When a player completes a firework by correctly playing a 5 card then the players receive a bonus. One Clock token is turned from black side to white side up. If all tokens are already white-side-up then no bonus is received. Play then passes to the next player (clockwise).

Ending the Game

The game can end in three ways:

1. The third Storm token is turned lightning-side-up. The gods deliver their wrath in the form of a storm that puts an end to the fireworks. The game ends immediately, and the players earn zero points.
2. The players complete all five fireworks correctly. The game ends immediately, and the players celebrate their spectacular victory with the maximum score of 25 points.
3. If a player draws the last card from the draw deck, the game is almost over. Each player—Including the player who drew the last card—gets one last turn. Note: Cards cannot be drawn in this last round.

Finally, the fireworks will be counted. For this, each firework earns the players a score equal to the highest value card in its color series. The quality of the fireworks display according to the rating scale of the "International Association of Pyrotechnics" is:

- 0–5: Oh dear! The crowd booed.
- 6–10: Poor! Hardly applaused.
- 11–15: OK! The viewers have seen better.
- 16–20: Good! The audience is pleased.
- 21–24: Very good! The audience is enthusiastic!
- 25: Legendary! The audience will never forget this show!

Important Notes and Tips

- Players may rearrange their hand cards and change their orientation to help themselves remember the information they received. Players may not ever look at the front of their own cards until they play them.
- The discard pile may always be searched for information.
- Hanabi is based on communication – and non-communication – between the Players. If one interprets the rules strictly then players may not, except for the announcements of the current player, talk to each other. Ultimately, each group should decide by its own measure what communication is permitted communication. Play so that you have fun!

Selenite Towers Move Faster Than Hanoï Towers, But Still Require Exponential Time

Jérémy Barbay*

Departamento de Ciencias de la Computación (DCC), Universidad de Chile, Santiago, Chile
`jeremy@barbay.cl`

Abstract

The Hanoï Tower problem is a classic exercise in recursive programming: the solution has a simple recursive definition, and its complexity and the matching lower bound correspond to the solution of a simple recursive function (the solution is so simple that most students memorize it and regurgitate it at exams without truly understanding it). We describe how some minor change in the rules of the Hanoï Tower yields various increases of difficulty in the solution, so that to require a deeper mastery of recursion than the classical Hanoï Tower problem. In particular, we analyze the Selenite Tower problem, where just changing the insertion and extraction positions from the top to the middle of the tower results in a surprising increase in the intricacy of the solution: such a tower of n disks can be optimally moved in a $\sqrt{3}^n$ moves for n even (i.e. less than a Hanoï Tower of same height), via five recursive functions following three distinct patterns.

1998 ACM Subject Classification F.2.2 [Analysis of Algorithms and Problem Complexity] Non-numerical Algorithms and Problems, F.2.m [Analysis of Algorithms and Problem Complexity] Miscellaneous

Keywords and phrases Brähma Tower, Disk Pile, HanoïTower, Levitating Tower, Recursivity

Digital Object Identifier 10.4230/LIPIcs.FUN.2016.5

1 Introduction

The Hanoï Tower problem is a classical problem often used to teach recursivity, originally proposed in 1883 by Édouard Lucas [5, 6], where one must move n disks, all of distinct size, one by one, from a peg A to a peg C using only an intermediary peg B, while ensuring no disk ever stands on a smaller one. As early as 1892, Ball [3] described an optimal recursive algorithm which moves the n disks of a Hanoï Tower in $2^n - 1$ steps. Many generalizations have been studied, allowing more than three pegs [4], coloring disks [7], and cyclic Hanoï Towers [2]. Some problems are still open, as the optimality of the algorithm for 4-peg Hanoï Tower problem, and the analysis of the original problem is still a source of inspiration many years after its definition: for instance, Allouche and Dress [1] proved in 1990 that the movements of the Hanoï Tower problem can be generated by a finite automaton, making this problem an element of $SPACE(1)$.

The solution to the Hanoï Tower problem is simple enough that it can be memorized and regurgitated at will by students from all over the world: asking about it in an assignment or exam does not truly test a student's mastery of the concept of recursivity, pushing instructors to consider variants with slightly more sophisticated solutions. Some variants do not make the problem more difficult (e.g. changing the `insertion` and `removal` point to the bottom:

* Work supported by the Millennium Nucleus RC130003 "Information and Coordination in Networks".

© Jérémy Barbay;
licensed under Creative Commons License CC-BY
8th International Conference on Fun with Algorithms (FUN 2016).
Editors: Erik D. Demaine and Fabrizio Grandoni; Article No. 5; pp. 5:1–5:20
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

the solution is exactly the same), some make it only slightly more difficult (e.g. considering the case where the disks are not necessarily of distinct sizes, described and analyzed in Appendix A), but some small changes can make it surprisingly more difficult.

We consider the SELENITE TOWER problem, which only differs from the HANOÏ TOWER problem in that the `insertion` and `removal` point in each tower is at the middle instead of the top (see Figure 1 for an illustration with SELENITE TOWERS of sizes $n = 3$ and $n = 4$, and Section 2.1 for the formal definition). One can poetically imagine that the Selenites[1] living on the moon can take advantage of the low gravity in order to remove and insert disks in the middle of the tower rather than merely at the top.

As for the classical HANOÏ TOWER, such `insertion` and `removal` rules guarantee that any move is *reversible* (i.e. any disk d removed from a peg X can always be immediately reinserted in the same peg X), that the `insertion` and `removal` positions are uniquely defined, that each peg can always receive a disk, and that each tower with one disk or more can always yield one disk. The problem is very similar to the HANOÏ TOWER problem: one would expect answering the following questions to be relatively easy, possibly by extending the answers to the corresponding questions on HANOÏ TOWERS[2]:

> Consider the problem of moving a SELENITE TOWER of n disks, all of distinct size, one by one, from a peg A to a peg C using only an intermediary peg B, while ensuring that at no time does a disk stand on a smaller one:
> 1. Which sequences of steps permit moving such a tower?
> 2. What is the minimal length of such a sequence?
> 3. How many such shortest sequences are there?

We show that there is a unique shortest sequence of steps which moves a SELENITE TOWER of n disks of distinct sizes, and that it is of length at most $\sqrt{3}^n$ (i.e., exactly $\sqrt{3}^n = 3^{\frac{n}{2}}$ if n is even, and $\frac{3}{5}\sqrt{3}^{n-1} - \frac{2}{3} = 3^{\frac{n-1}{2}} + 2(3^{\frac{n-3}{2}} - 1) < \sqrt{3}^n$ if n is odd). As $\sqrt{3} \approx 1.733 < 2$, this sequence is exponentially shorter than the corresponding one for the HANOÏ TOWER problem (of length $2^n - 1$). We define formally the problem and its basic properties in Section 2: its formal definition in Section 2.1, some examples where such towers can be moved faster in Section 2.2, and some useful concepts on the `insertion` and `removal` order of a tower in Section 2.3. We describe a recursive solution in Section 3, via its algorithm in Section 3.1, the proof of its correctness in Section 3.2 and the analysis of its complexity in Section 3.3. The optimality of the solution is proved in Section 4, via an analysis of the graph of all possible states and transition (defined and illustrated in Section 4.1) and a proof of optimality for each function composing the solution (Section 4.2). We conclude with a discussion (Section 5) of other variants of similar or increased complexity, and share in Appendix A the text and the solution of a simpler variant successfully used in undergraduate assignments and exams.

2 Formal Definition and Basic Facts

In this section we define more formally the SELENITE TOWER (Section 2.1), how small examples already show that moving such towers requires less steps than moving a HANOÏ

---

[1] The word "selenite" is derived from the Greek lunar deity "$\Sigma\varepsilon\lambda\nu\eta\nu$", "Selene". The author H. G. Wells, in "The First Men In The Moon", referred to the inhabitants of the moon as selenites. The author Jules Verne also used this term in "From Earth to the Moon" and "Around the Moon".

[2] For a HANOÏ TOWER, the answer to those question is that there is a single such shortest sequence, of length $2^n - 1$, obtained by the recursion $h(n, A, B, C) = h(n-1, A, C, B).$"$A \to B;$"$.h(n-1, B, A, C)$ if $n > 0$ and $\emptyset$ otherwise.

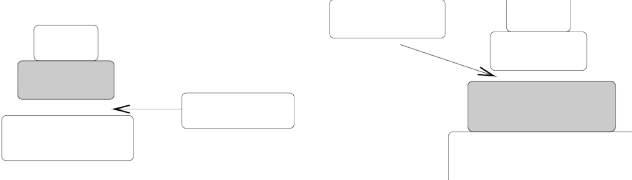


Figure 1 An illustration of the rules for the `insertion` and `removal` in a Selenite Tower, depending on the parity of its size (sizes $n = 3$ and $n = 4$ here). In each case, the shaded disk indicates the `removal` point and the arrow indicates the `insertion` point.

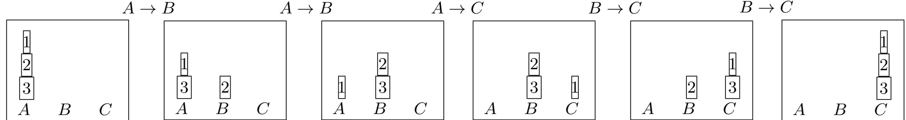


Figure 2 A Selenite Tower of three disks can be moved in just five steps.

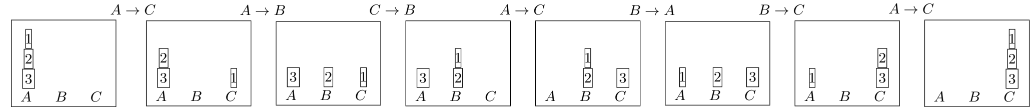


Figure 3 A Hanoï Tower of three disks requires seven steps to be moved between two pegs.

Tower (Section 2.2), and some properties of the order in which disks are inserted or removed on a peg to build or destroy a tower (Section 2.3).

2.1 Formal Definition

The "middle" disk of a tower of even size is not well defined, nor is the "middle" `insertion` point in a tower of odd size. We define both more formally in such a way that if n is odd, the `removal` position is the center one, and the `insertion` point is below it; while if n is even, the `insertion` point is in the middle of the tower, while the `removal` position is below the middle of the tower. See Figure 1 for an illustration with sizes $n = 3$ and $n = 4$.

More formally, on a peg containing n disks ranked by increasing sizes, the `removal` point is the disk of rank $\lfloor \frac{n}{2} \rfloor + 1$; and the `insertion` point is position $\lfloor \frac{n+1}{2} \rfloor$.

The `insertion` of disk d on peg X is *legal* if inserting d in the `insertion` point of X yields a legal configuration, where no disk is above a smaller one. A move from peg X to peg Y is *legal* if there is a disk d to remove from X, and if the `insertion` of d on the Y is legal.

2.2 Moving small towers – differences with Hanoï

For size one or two, there is no difference in the moving cost between a Hanoï Tower and a Selenite Tower. The first difference appears for size three, when only five steps are necessary to move a Selenite Tower (see the sequence of five steps to move a Selenite Tower of size $n = 3$ in Figure 2) as opposed to the seven steps required for moving a classical Hanoï Tower (see the sequence of seven steps to move a Hanoï Tower of size $n = 3$ in Figure 3).

When an odd number of disks is present on the peg A, and an even number is present on pegs B and C, a sub-tower of height 2 can be moved from A in 2 steps, when in a Hanoï Tower we need 3 steps to move any subtower of same height. In the Selenite Tower

problem, having a third disk “fixed” on A yields a reduced number of steps. We formalize this notion of “fixed” disk in the next section.

2.3 Structural facts on a single Peg

Before considering the complete problem over three pegs, we describe some concepts about single pegs, and on the order in which the disks are inserted and removed on a specific peg.

▶ **Definition 1.** We define the *removal order* as the order in which disks (identified by their rank in the final tower) can be removed from a SELENITE TOWER. Symmetrically, we define the *insertion order* as the order in which the disks are inserted in the tower.

The symmetry of the rules concerning the `insertion` and `removal` location of SELENITE TOWERS yields that the *insertion* order is the exact reverse of the *removal* order (the `insertion` point of a tower is the `removal` point of a tower with one more disk), and each disk removed from a peg can be immediately put back exactly where it was.

In particular, a key argument to both the description of the solution in Section 3 and to the proof of its optimality in Section 4 is the fact that, when some (more extreme) disks are considered as “fixed” (i.e. the call to the current function has to terminate before such disks are moved), the order in which a subset of the disks is removed from a peg depends on the number of those “fixed” disks.

▶ **Definition 2.** When moving recursively n disks from a peg X with $x > n$ disks, the $x - n$ last disks in the `removal` order of X are said to be *fixed*. The *parity* of peg X is the parity of the number x of disks *fixed* on this peg.

SELENITE TOWERS cannot be moved much faster than HANOÏ TOWERS:

▶ **Lemma 3.** *Without a third peg, it is impossible to move more than one disk between two pegs with the same parity.*

Proof. Between two pegs of same parity, the `removal` order is the same. So the first disk needed on the final peg will be the last one removed from the starting peg. With more than one disk, we need the third peg to dispose temporally of the other disks. ◀

▶ **Lemma 4.** *It is impossible to move more than two disks between two pegs of opposite parities without a third peg.*

Proof. Between two pegs of opposite parities, the `removal` orders are different: But the definition of the middle is constant when the number of disks changes by 2. So after moving two disks the third cannot be inserted in the right place. ◀

The `removal` and `insertion` orders are changing with the parity of the SELENITE TOWER. Consider a peg with n disks on it:

- if $n = 2m + 1$ is odd, then the disks are removed in the following order:

$$(m+1, m+2, \quad m, m+3, \quad m-1, m+4, \quad \dots, \quad 3, 2m, \quad 2, 2m+1, \quad 1)$$

- if $n = 2m$ is even, then the removal order is:

$$(m+1, m, \quad m+2, m-1, \quad m+3, m-2, \quad \dots, \quad 2m-1, 2, \quad 2m, 1)$$

The relative order of m and $m+2$, of $m-1$ and $m+3$, and more generally of any pair of disks i and $m-i$ for $i \in [1..\lfloor n/2 \rfloor]$, are distinct. More specifically, disks are alternately extracted below and above the `insertion` point. This implies the two following connexity lemmas:

▶ **Lemma 5.** *The k first disks removed from the tower are contiguous in the original tower, and they are either all smaller or all larger than the $(k+1)$-th disk removed.*

▶ **Lemma 6.** *If k disks are all smaller than the disk below the `insertion` point, and all larger than the disk above the `insertion` point, then there exists an order for adding those k disks to the tower.*

Proof. By induction: for one disk it is true; for k disks, if the `insertion` point after the `insertion` of disc d is above d then add the larger and then the $k-1$ remaining disks, else add the smaller and then the $k-1$ disks left. ◀

In the next section, we present a solution to the Selenite Tower problem which takes advantage of the cases where two disks can be moved between the same two pegs in two consecutive steps.

3 Solution

One important difference between Hanoï Towers and Selenite Towers is that we do not always need to remove $n-1$ disks of a tower of n disks to place the n-th disk on another peg (e.g. in the sequence of steps shown in Figure 2, disk 3 was removed from A when there was still a disk sitting on top of it). But we need always to remove at least $n-2$ disks in order to release the n-th disk, as it is the last or the last-but-one disk removed. This yields a slightly more complex recursion than in the traditional case. We describe an algorithmic solution in Section 3.1, prove its correctness in Section 3.2, and analyze the length of its output in Section 3.3. We prove the optimality of the solution produced separately, in Section 4.

3.1 Algorithm

Note $|A|$ the number of disks on peg A, $|B|$ on B and $|C|$ on C. For each triplet $(x, y, z) \in \{0,1\}^3$, we define the function `movexyz`(n, A, B, C) moving n disks from peg A to peg C using peg B when $|A| \geq n$, $|A| - n \equiv x \mod 2$, $|B| \equiv y \mod 2$, $|C| \equiv z \mod 2$, and the n first disks extracted from A can be legally inserted on B and C. Less formally, there are x fixed disks on the peg A, y on B and z on C.

We need only to study three of those $2^3 = 8$ functions. First, as the functions are symmetric two by two: for instance, `move000`(n, A, B, C) behaves as `move111`(n, A, B, C) would if the `insertion` point in a tower of odd size was above the middle disk, and the `removal` point in a tower of even size was above the middle of the tower: in particular, they have exactly the same complexity. Second, the reversibility and symmetry of the functions yields a similar reduction: `move001`(n, A, B, C) has the same structure as the function `move100`(n, A, B, C) and the two have the same complexity.

We describe the python code implementing those functions in Figures 4 to 7, so that the initial call is made through the call `move000(n,"a","b","c")`, while recursive calls refer only to function `move000`(n, A, B, C) (Figure 4), function `move100`(n, A, B, C) (Figure 5), function `move001`(n, A, B, C) (similar to function `move100`(n, A, B, C) and described in Figure 6) and function `move010`(n, A, B, C) (Figure 7).

The algorithm for `move000`(n, A, B, C) (in Figure 4) has the same structure as the corresponding one for moving Hanoï Towers, the only difference being in the parity of the pegs

```
def move(a,b):
 print "("+a+",",
 print b+")",

def move000(n,a,b,c):
 if n>0 :
  move100(n-1,a,c,b)
  move(a,c)
  move001(n-1,b,a,c)
```

Figure 4

```
def move100(n,a,b,c):
 if n == 1 :
  move(a,c)
 elif n>1 :
  move100(n-2,a,c,b)
  move(a,c)
  move(a,c)
  move010(n-2,b,a,c)
```

Figure 5

```
def move001(n,a,b,c):
 if n == 1 :
  move(a,c)
 elif n>1 :
  move010(n-2,a,c,b)
  move(a,c)
  move(a,c)
  move001(n-2,b,a,c)
```

Figure 6

```
def move010(n,a,b,c):
 if n == 1 :
  move(a,c)
 elif n == 2 :
  move(a,b)
  move(a,c)
  move(b,c)
 elif n>2 :
  move010(n-2,a,b,c)
  move(a,b)
  move(a,b)
  move010(n-2,c,b,a)
  move(b,c)
  move(b,c)
  move010(n-2,a,b,c)
```

Figure 7

in the recursive calls, which implies calling other functions than `move000`(n, A, B, C), in this case `move001`(n, A, B, C) and `move100`(n, A, B, C). The algorithms for `move100`(n, A, B, C) (in Figure 5) and `move001`(n, A, B, C) (in Figure 6) and are taking advantage of the difference of parity between the two extreme pegs to move two consecutive disks in two moves, but still has a similar structure to the algorithm for `move000`(n, A, B, C) and the corresponding one for moving Hanoï Towers (just moving two disks instead of one).

The algorithm for `move010`(n, A, B, C) is less intuitive. Given that the `removal` and `insertion` orders on the origin peg A and on the destination peg C are the same (because the parity of those pegs is the same), $n - 1$ disks must be removed from A before the last disk of the `removal` order, which yields a naive algorithm such as described in Figure 8. Such a strategy would yield a correct solution but not an optimal one, as it reduces the size only by one disk at the cost of two recursive calls and one step (i.e. reducing the size by two disks at the cost of four recursive calls and three steps). Another strategy (described in the algorithm in Figure 7) reduces the size by two at the cost of three recursive calls and four steps: moving $n - 2$ disks to C, the two last disks of the `removal` order on B, then $n - 2$ disks to A, the two last disks of the `removal` order on C, then finally the $n - 2$ disks to C. The first strategy ($f(n) = 2f(n-1) + 2 = 4f(n-2) + 3$) yields a complexity within $\Theta(2^n)$ while the second strategy ($f(n) = 3f(n-2) + 4$) yields a complexity within $\Theta(3^{\frac{n}{2}})$. We show in Section 3.2 that moving two disks at a time is correct in this context and in Section 4 that the latter yields the optimal solution.

```
def nonOptimalMove010(n,a,b,c):
 if n==1:
  move(a,c)
 else:
  move101(n-1,a,c,b)
  move(a,c)
  move101(n-1,b,a,c)
```

Figure 8 Alternative (non optimal) take on $move010(n, A, B, C)$.

```
def nonOptimalMove101(n,a,b,c):
 if n==1:
  move(a,c)
 else:
  move010(n-1,a,c,b)
  move(a,c)
  move010(n-1,b,a,c)
```

Figure 9 Alternative (non optimal) take on $move101(n, A, B, C)$.

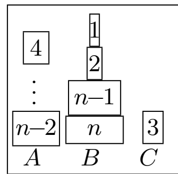


Figure 10 *Requirement for insertion* (i)*:* disks 4 to $n-2$ can be inserted on B as the `insertion` point of B is between 2 and $n-1$; and on C as the `insertion` point of C is under 3.

3.2 Correctness of the algorithm

We prove the correctness of our solution by induction on the number n of disks.

▶ **Theorem 7.** *For any positive integer value n, and any triplet $(x, y, z) \in \{0,1\}^3$ of booleans, the function* `movexyz`(n, A, B, C) *produces a sequence of legal steps which moves a* SELENITE TOWER *from A to C via B.*

The proof is based on the following invariant, satisfied by all recursive functions on entering and exiting:

▶ **Definition 8.** *Requirement for insertion* (*i*)*:* The disks above the `insertion` point of B or C are all smaller than the first n disks removed from A; and the disks below the `insertion` point of B or C are all larger than the first n disks removed from A (see an illustration in Figure 10).

Proof. Consider the property $H(n)$ = "$\forall(x, y, z) \in \{0, 1\}^3$, $\forall i \leq n$, `movexyz`(i, A, B, C) is correct". $H(0)$ is trivially true, and $H(1)$ can be checked for all functions at once. For all values x, y, z, the function `movexyz`$(1, A, B, C)$ is merely performing the step `move`(A, C). The hypothesis $H(1)$ follows. Now, for a fixed $n > 1$, assume that $H(n-1)$ holds: we prove the hypothesis $H(n)$ separately for each function.

- Analysis of `move000`(n, A, B, C):
  1. According to $H(n-1)$ the call to `move100`$(n-1, A, B, C)$ is correct if (i) and $(p)_{100}$ are respected. (i) is implied by (i) on `move000`$(n-1, A, B, C)$; $(p)_{100}$ is implied by $(p)_{000}$ and the remaining disk on A $(a - n \mod 2 \equiv 0 \Rightarrow a - (n-1) \mod 2 \equiv 1 \mod 2)$.
  2. The step `move`(A, C) is both possible and legal because of the precondition (i) for `move000`(n, A, B, C): the disk moved was in the n first removed from A, and so can be introduced on C.
  3. The call to `move001`(n, A, B, C) is symmetrical to 1, and therefore correct.
  4. We can check the final state by verifying that the number of disks removed from A and added to C is $(n-1) + 1 = n$.

(a) (i): n odd: a is removed first, y is removed second.

(b) (ii): n even: y is removed first, x is removed second.

Figure 11 Removal order of the last two disks.

So `move000`(n, A, B, C) is correct.

- Analysis of `move100`(n, A, B, C):
  1. `move100`$(n-2, A, B, C)$ is correct according to $H(n-1)$, as the requirements are also: The requirement (i) is given by (i) for the initial call, and the parity $(p)_{100}$ is respected because we move two disks less than in the current call to `move100`(n, A, B, C).
  2. The two disks left (let us call them α and β) are in position (given Fig. 11, (i)) such that the removal order on A is (α, β) and the `insertion` order on C is (β, α) (see Fig. 11, (ii)). They can be inserted on C because of requirement (i). So the two disks are correctly moved in two steps.
  3. The requirements for `move010`$(n-2, A, B, C)$ are satisfied:
     - (i) stand as a consequence of the precondition (i) for the current call, as the $n-2$ disks to be moved on C were on A before the original call, in the middle of α and β.
     - $(p)_{010}$: The number of disks on C is still even as we added two disks. The number of disks on A is still odd as we removed two disks.

     Therefore, because of $H(n-2)$, `move010`$(n-2, A, B, C)$ is correct.

  This finishes the proof that `move100`(n, A, B, C) is correct.
- Analysis of `move001`(n, A, B, C): This function, being similar to `move100`(n, A, B, C), for a task symmetric, has the same proof of correctness.
- Analysis of `move010`$(2, A, B, C)$: The two disks (let us call them α and β) are in position (given Fig. 11, (ii)) such that the `removal` order on A is (β, α) and the `insertion` order on C is $(\alpha, \beta,)$, as A and C have the same parity. β can be inserted on B and they can both be inserted on C because of requirement (i). So the two disks are correctly moved in three steps, using peg B to temporally dispose of the disk β. So `move010`$(2, A, B, C)$ is correct.
- Analysis of `move010`(n, A, B, C) if $n > 2$: Note that fixing two disks on the same peg does not change the parity of this peg.
  1. `move010`$(n-2, A, B, C)$ is correct as: from (i) for the initial call results (i) for the first recursive call; $(p)_{010}$ is a natural consequence of $(p)_{010}$ for the initial call (because the parity of the peg is conserved when two disks are fixed on it). So $H(n-1)$ implies that `move010`$(n-2, A, B, C)$ is correct.
  2. As A and B have different parities, we can move two consecutive disks in two consecutive calls, as for `move100`(n, A, B, C).
  3. The second recursive call to `move010`$(n-2, A, B, C)$ verifies conditions (i) and $(p)_{010}$ as only two extreme disks (the smallest and the largest) have been removed from A.

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
f_{010}	0	1	3	7	13	25	43	79	133	241	403	727	1213	2185	3643
f_{100}	0	1	2	4	7	13	22	40	67	121	202	364	607	1093	1822
f_{000}	0	1	3	5	9	15	27	45	81	135	243	405	729	1215	2187
$3^{\lceil n/2 \rceil}$	1	3	3	9	9	27	27	81	81	243	243	729	729	2187	2187

Figure 12 The first values of f_{010},f_{100} and f_{000}, computed automatically from the recursion. Those corroborate the intuition that $f_{100}(n) < f_{000}(n)$ for values of n larger than 1.

4. The two next steps are feasible because of the difference of parity between B and C (same argument as point 2).
5. The last recursive call is symmetric to the first call, as we move the $n-2$ disks back between the two extreme disks, but this time on C.

Therefore `move010`(n, A, B, C) is correct. ◀

We analyze the complexity of this solution in the next section.

3.3 Complexity of the algorithm

Let $f_{xyz}(n)$ be the complexity of the function `movexyz`(n, A, B, C), when $|A| \geq n$, $|A|-n \equiv x \mod 2$, $|B| \equiv y \mod 2$ and $|C| \equiv z \mod 2$. The algorithms from Figures 4 to 7 yield a recursive system of four equations.

$$\begin{cases} \forall x,y,z & f_{xyz}(0) & = 0 \\ \forall x,y,z & f_{xyz}(1) & = 1 \\ & f_{010}(2) & = 3 \\ & & \\ \forall n > 1, & f_{000}(n) & = f_{100}(n-1) + 1 + f_{001}(n-1) \\ \forall n > 1, & f_{100}(n) & = f_{100}(n-2) + 2 + f_{010}(n-2) \\ \forall n > 1, & f_{001}(n) & = f_{010}(n-2) + 2 + f_{001}(n-2) \\ \forall n > 2, & f_{010}(n) & = 3f_{010}(n-2) + 4 \end{cases}$$

As f_{001} is defined exactly as f_{100} (because of the symmetry between `move001`(n, A, B, C) and `move100`(n, A, B, C)), we can replace each occurrence of f_{001} by f_{100}, hence reducing the four equations to a system of three equations:

$$\begin{cases} \forall x,y,z & f_{xyz}(0) & = 0 \\ \forall x,y,z & f_{xyz}(1) & = 1 \\ & f_{010}(2) & = 3 \\ & & \\ \forall n > 1, & f_{000}(n) & = 2f_{100}(n-1) + 1 \\ \forall n > 1, & f_{100}(n) & = f_{100}(n-2) + 2 + f_{010}(n-2) \\ \forall n > 2, & f_{010}(n) & = 3f_{010}(n-2) + 4 \end{cases}$$

Lemmas 9 to 11 resolve the system function by function. The function $f_{010}(n)$ can be solved independently from the others:

▶ **Lemma 9.**

$$f_{010}(n) = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ 3 & \text{if } n = 2; \\ 3^{\frac{n+1}{2}} - 2 & \text{if } n \geq 3 \text{ is odd; and} \\ 5 \times 3^{\frac{n}{2}-1} - 2 & \text{if } n \geq 4 \text{ is even.} \end{cases}$$

Proof. Consider the recurrence $X_{k+1} = 3X_k + 4$ at the core of the definition of f_{010}: a mere extension yields the simple expression $X_k = 3^k(X_0 + 2) - 2$.

- When $n \geq 3$ is odd, set $k = \frac{n-1}{2} \geq 1$, $U_0 = 1$ and $U_{k+1} = 3U_k + 4$ so that $f(2k+1) = U_k = 3^k(1+2) - 2$. Then $f_{010}(n) = 3 \times 3^k - 2 = 3^{k+1} - 2$ for $n \geq 3$ and odd.
- When $n \geq 4$ is even, set $k = \frac{n}{2} \geq 1$, $V_0 = 3$ and $V_{k+1} = 3V_k + 4$ so that $f(2k) = V_k = 3^k(3+2) - 2$, so that $f_{010}(n) = 5 \times 3^k - 2$ for $n \geq 4$ and even.

Gathering all the results yields the final expression. ◀

The expression for the function f_{010} yields the expression for the function f_{100}:

▶ **Lemma 10.**

$$f_{100}(n) = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ 2 & \text{if } n = 2; \\ 4 & \text{if } n = 3; \\ \frac{5}{2} \times 3^{\frac{n}{2}-1} + 2 & \text{where } n \geq 4 \text{ is even; and} \\ \frac{3^{\frac{n+1}{2}} - 1}{2} & \text{where } n \geq 5 \text{ is odd.} \end{cases}$$

Proof. Consider the projection of the system to just f_{100}:

$$f_{100}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ f_{100}(n-2) + 2 + f_{010}(n-2) & \text{if } n \geq 2 \end{cases}$$

For any integer value of $k \geq 0$, we combine some change of variables with the results from Lemma 9 to yield two linear systems, which we solve separately:

- $V_k = f_{100}(2k)$ and $V_0 = f_{100}(0) = 0$ so that $f_{100}(n) = V_k$ if n is even and $k = \frac{n}{2}$; and
- $U_k = f_{100}(2k+1)$ and $U_0 = f_{100}(1) = 1$ so that $f_{100}(n) = U_k$ if n is odd and $k = \frac{n-1}{2}$.

On one hand, $U_k = U_{k-1} + 2 + f_{010}(2k+1-2)$ for $k > 0$ and $U_0 = 1$. This yields a linear recurrence which we develop as follows:

$$\begin{aligned} U_k &= U_{k-1} + 2 + f_{010}(2k-1) \text{ by definition;} \\ &= U_{k-1} + 2 + 3 \times 3^{\frac{(2k-1)-1}{2}} - 2 \text{ via Lemma 9 because } 2k-1 \text{ is odd;} \\ &= U_{k-1} + 3^k \text{ by mere simplification;} \\ &= U_0 + \frac{3}{2}(3^k - 1) \text{ by resolution of a geometric series;} \\ &= \frac{3^{k+1} - 1}{2} \text{ because } U_0 = 1. \end{aligned}$$

Since $f_{100}(n) = U_{\frac{n-1}{2}}$ when n is odd, the solution above yields $f_{100}(n) = \frac{3^{\frac{n+1}{2}} - 1}{2}$ if n is odd.

On the other hand, $V_k = V_{k-1} + 2 + f_{010}(2k-2)$ for $k > 0$ and $V_0 = 0$. The initial conditions of f_{010} for $n = 0, 1$ and 2 yield the three first values of V_k: $V_0 = 0$; $V_1 =$

$V_0 + 2 + f_{010}(0) = 0 + 2 + 0 = 2$; and $V_2 = V_1 + 2 + f_{010}(2) = 2 + 2 + 3 = 7$. We then develop the recursion for $k \geq 3$ similarly to U_k:

$$\begin{aligned}
V_k &= V_{k-1} + 2 + f_{010}(2k-2) \text{ by definition;}\\
&= V_{k-1} + 2 + 5 \times 3^{\frac{(2k-2)}{2}-1} - 2 \text{ for } 2k-2 \geq 4 \text{ even, or any } k \geq 3 \text{ via Lemma 9;}\\
&= V_{k-1} + 5 \times 3^{k-2} \text{ by mere simplification (still only for } k \geq 3);\\
&= V_2 + 5\left(3^1 + \cdots + 3^{k-2}\right) \text{ by propagation;}\\
&= V_2 + 5\frac{3^{k-1}-2}{2} \text{ by resolution of a geometric series;}\\
&= 7 + \frac{5}{2}(3^{k-1} - 2) \text{ because } V_2 = 7;\\
&= \frac{5}{2}3^{k-1} + 2 \text{ by simplification.}
\end{aligned}$$

Since $f_{100}(n) = V_{\frac{n}{2}}$ when n is even, the solution above yields $f_{100}(n) = \frac{5}{2}3^{\frac{n}{2}-1} + 2$ if n is even.

Reporting those results in the definition of f_{100} yields the final formula:

$$f_{100}(n) = \begin{cases} 0 & \text{if } n = 0;\\ 1 & \text{if } n = 1;\\ 2 & \text{if } n = 2;\\ 4 & \text{if } n = 3;\\ \frac{5}{2} \times 3^{\frac{n}{2}-1} + 2 & \text{where } n \geq 4 \text{ is even; and}\\ \frac{3^{\frac{n+1}{2}}-1}{2} & \text{where } n \geq 5 \text{ is odd.} \end{cases}$$

◀

Finally, the expression for the function f_{100} directly yields the expression for the function f_{000}:

▶ **Lemma 11.**

$$f_{000}(n) = \begin{cases} 1 & \textit{if } n = 1\\ 3 & \textit{if } n = 2\\ 5 & \textit{if } n = 3\\ 3^{\frac{n}{2}} & \textit{where } n \geq 4 \textit{ is even; and}\\ 5(3^{\frac{n-3}{2}} + 1) & \textit{where } n \geq 5 \textit{ is odd.} \end{cases}$$

Proof.

$$f_{100}(n) = \begin{cases} 1 & \text{if } n = 1;\\ 2 & \text{if } n = 2;\\ 4 & \text{if } n = 3;\\ \frac{5}{2}3^{\frac{n}{2}-1} + 2 & \text{where } n \geq 4 \text{ is even; and}\\ \frac{3^{\frac{n+1}{2}}-1}{2} & \text{where } n \geq 5 \text{ is odd.} \end{cases}$$

From these results, deduce the value of $f_{000}(n)$ using the fact that $f_{000}(n) = 2f_{100}(n-1) + 1$.

$$f_{000}(n) = \begin{cases} 1 & \text{if } n = 1 \\ 3 & \text{if } n = 2 \\ 5 & \text{if } n = 3 \\ 5 \times 3^{\frac{n-1}{2}-1} + 5 & \text{where } n \geq 5 \text{ is odd; and} \\ 3^{\frac{n}{2}} & \text{where } n \geq 6 \text{ is even.} \end{cases}$$

◀

As $\sqrt{3} \approx 1.73 < 2$, this value is smaller than the number $2^n - 1$ of steps required to move a Hanoï Tower. We prove that this is optimal in the next section.

4 Optimality

Each legal state of the Selenite Tower problem with three pegs and n disks can be uniquely described by a word of length n on the alphabet $\{A, B, C\}$, where the i-th letter indicates on which peg the i-th largest disk stands. Moreover, each word of $\{A, B, C\}^n$ corresponds to a legal state of the tower, so there are 3^n different legal states (even though not all of them are reachable from the initial state).

To prove the optimality of our algorithm, we prove that it moves the disks along the shortest path in the *configuration graph* (defined in Section 4.1) by a simple induction proof (in Section 4.2).

4.1 The configuration graph

The configuration graph of a Selenite Tower has 3^n vertices corresponding to the 3^n legal states, and two states s and t are connected by an edge if there is a legal move from state s to state t. The reversibility of moves (seen in Section 2.3) implies that the graph is undirected.

Consider the initial state $A \dots A$ ($= A^n$). The smallest disk 1 cannot be moved before the other disks are all moved to peg B or all moved to peg C: we cannot remove disk 1 from peg A if there is a disk under it, and we cannot put it on another peg if a larger disk is already there. This partitions G into three parts, each part being characterized by the position of disk 1; these parts are connected by edges representing a move of disk 1 (see the recursive decomposition of $G(n)$ in Figure 13).

Each part is an instance of the configuration graph $G'(n-1)$ defining all legal steps of $(n-1)$ disks $\{2, \dots, n\}$ given that disk 1 is fixed on its peg.

Let us consider this subgraph $G'(n-1)$, when disk 1 (the smallest) is fixed on one peg (say on peg A). Note each state of this graph $aX \dots Z$, where a stands for the disk 1 fixed on peg A, and $X \dots Z \in \{A, B, C\}^{n-1}$ describes the positions of the other disks. The removal order changes from those observed in G each time $|A|$ is odd.

To remove the two extreme disks 2 and n (not moving disk 1, since it is fixed), it is necessary to move all other disks to a single other peg (same argument as for $G(n)$), so we can divide our configuration graph in subsets of states corresponding to different positions where disks 2 and n are fixed.

This defines 9 parts, as each of the two fixed disks can be on one of the three pegs. Of those 9 parts, we need to focus only on 5:

- two parts of the graph cannot be accessed from the initial state $aA \dots A$, (see an illustration in Figure 14); and

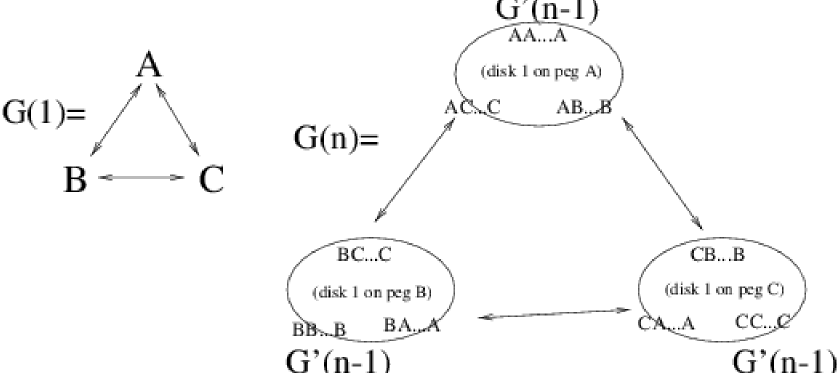


Figure 13 First decomposition of the configuration graph of the SELENITE TOWER problem.

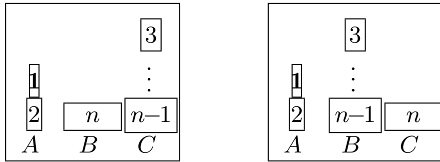


Figure 14 States where disk 2 is on A and disk n is on another peg (i.e. B or C) cannot be accessed from the initial state $A \ldots A$ for $n > 4$. No move is possible from these states as A cannot receive a larger disk than 2 (and all are), B cannot receive a smaller disk than n (and all disks are of size smaller or equal to n), and C cannot receive the disks 2 nor n if $n > 4$.

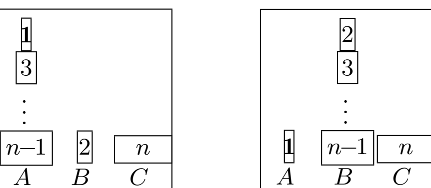


Figure 15 States $aBA \ldots AC$ and $aBB \ldots BC$ are not connected in the subgraph where disks 2 and n are fixed on B and C, and disk 1 is fixed on A: As no disk can be inserted under n, if $n > 4$ it is impossible to move the $n - 3 > 1$ unfixed disks from A to B (as to move more than one disk between two pegs of same parity require a third peg).

- the part of the graph where disk 2 is fixed on B and disk n is fixed on C contains two parts, which are not connected for $n > 4$ (see an illustration in Figure 15).

The five remaining parts are very similar. Three of them are of particular importance as each contains one key state, which are $aA \ldots A$, $aB \ldots B$ and $aC \ldots C$. Consider first the graphs $G'(n)$ for $n \in \{1, 2, 3\}$ ($n + 1$ disks in total if we count the fixed one): they are represented in Figure 16. When one disk is fixed on A, the task of moving disks from A to B is symmetric with moving them from A to C, but quite distinct from the task of moving disks from B to C.

Now, consider the part of the graph $G'(n-1)$ where the smallest and the largest disks (2 and n) are fixed on A. This part contains the initial state $A \ldots A$. The only way to free the smallest disk is to move the $n - 3$ other disks to another peg.

Once disks 2 and n are fixed on the same peg (in addition to disk 1), the situation is similar to the entire graph, with two fewer disks. It is the case each time two extreme disks

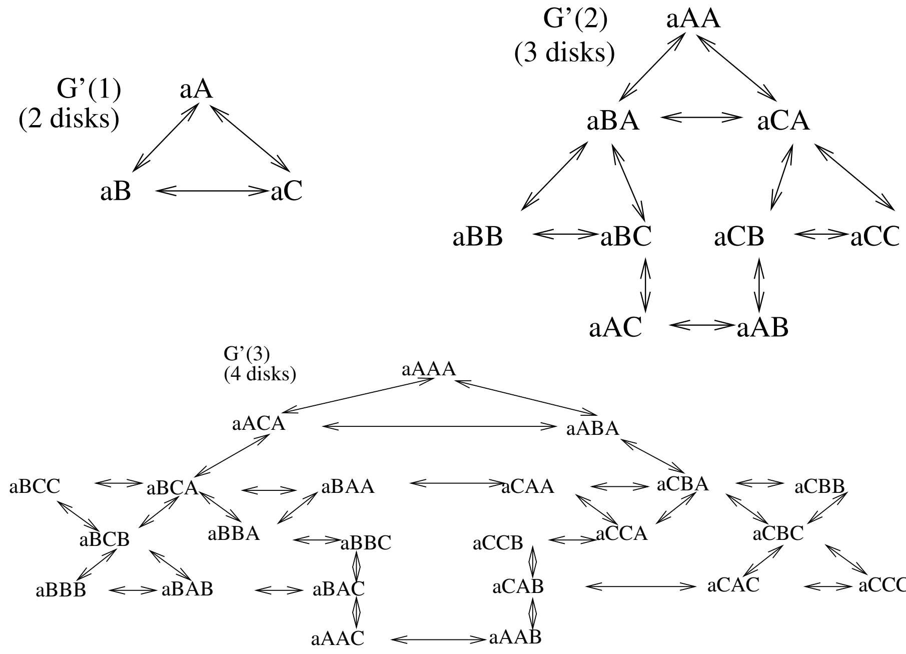


Figure 16 Subgraphs $G'(n)$ with one disk fixed on the peg A for $n \in \{1, 2, 3\}$.

are fixed on the same peg: when 2 and n are fixed on peg C or B, or when 1 and n are fixed on peg A; the process can then ignore the two fixed disks to move the $n-3$ remaining disks, as the parity of the peg is unchanged. See the definitions of the graph $G'(n)$ in Figure 16 for $n \in \{1, 2, 3\}$ and in Figure 17 for $n > 3$.

4.2 Proof of optimality

To prove the optimality of the solution described in Section 3, we prove that the algorithm is taking the shortest path in the configuration graph defined in the last section. A side result is that this is the unique shortest solution.

▶ **Theorem 12.** $\forall (x, y, z) \in \{0, 1\}^3$, $\forall n \geq 0$, $\texttt{movexyz}(n, A, B, C)$ *moves optimally* n *disks from* A *to* C.

Proof. Define the induction hypothesis $H(n)$ as "$\forall (x, y, z) \in \{0, 1\}^3 \, \texttt{movexyz}(n, A, B, C)$ moves optimally n disks from A to C". Trivially $H(0)$ and $H(1)$ are true. Suppose that there exists an integer $N > 1$ such that $\forall n < N$, the induction hypothesis $H(n)$ is true. We prove that $H(N)$ is then also true.

- $\texttt{move000}(N, A, B, C)$ is optimal: $\texttt{move000}(N, A, B, C)$ for $N > 0$ consists of one call to $\texttt{move100}(N, A, C, B)$, one unitary step, and one call to $\texttt{move001}(N, B, A, C)$.
  Therefore it moves optimally (by $H(N-1)$) from $aA \ldots A$ to $aB \ldots B$, and then to $cB \ldots B$, and after that to $cC \ldots C$. (In Figure 13 the right edge of the triangle.)
  A path not going through states $aB \ldots B$ or $cB \ldots B$ would take more steps:
  - if we do not go through the state $aB \ldots B$, then the state $aC \ldots C$ is necessary, with a cost of $f_{100}(N-1)$, and also the state $bC \ldots C$ (with a cost of 1), and at the end of the path we have to go through the state $cA \ldots A$, which optimal path to go to the final $cC \ldots C$ state is of length $f_{100}(N-1)$: this path is of length $f_{100}(N-1)+1+f_{100}(N-1)$ and is already as long as the one given by $\texttt{move000}(N, A, B, C)$.

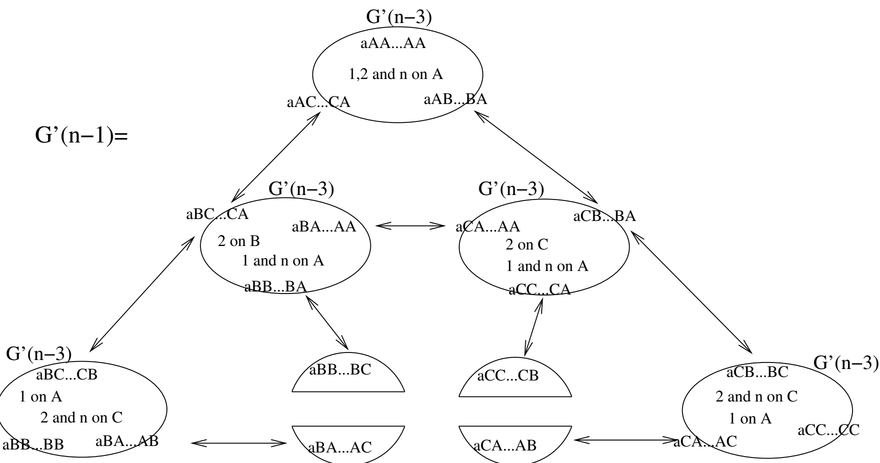


Figure 17 Recursive definition of $G'(n)$, the graph of all legal steps when one disk is fixed on the first peg, for $n > 3$. There is no way to connect the states $aBB....BC$, $aBA...AC$, $aCC...CB$ and $aCA...AB$ without moving some of the disks from $\{1, 2, n\}$.

- if we go through $aB \dots B$, but not through $cB \dots B$, then the path is not optimal as it must go through $aC \dots C$ and the optimal path from $aA \dots A$ to $aC \dots C$ does not go through $aB \dots B$.

So `move000`(N, A, B, C) is optimal.

- `move100`(N, A, B, C) is optimal:
  `move100`(N, A, B, C) for $N > 1$ consists of one call to `move100`$(N-2, A, C, B)$), two steps, and one call to `move010`$(N-2, B, A, C)$).
  As before, we shall consider these recursive calls of order smaller than N as optimal because of $H(N-2)$. So we know how to move optimally from $aAA \dots AA$ to $aAB \dots BA$, to $aCB \dots BA$, then to $aCB \dots BC$ and to $aCC \dots CC$ (in Figure 17), this corresponds to the left edge of the triangle).
  We must now prove that other paths take more steps:
  - We cannot avoid the state $aCB \dots BC$, neither $aCB \dots BA$, as there is no other way out of $aCC \dots CC$.
  - if we avoid the state $aAB \dots BA$ then the optimal path to $aCB \dots BA$ necessarily passes by $aBA \dots AA$ and $aCA \dots AA$, and is of length $f_{100}(N-2) + 1 + f_{010}(N-2) + 1 + f_{010}(N-2)$, which is longer than the whole solution given by the algorithm, of length $f_{100}(N) = f_{100}(N-2) + 2 + f_{010}(N-2)$.

  So `move100`(N, A, B, C) is optimal.
- `move010`(N, A, B, C) is optimal:
  `move010`$(1, A, B, C)$ and `move010`$(2, A, B, C)$ are special cases, we can see in graphs $G'(1)$ and $G'(2)$ on figure 16 page 14 that the optimal paths between $aB \dots B$ and $aC \dots C$ are of length 1 and 3, as the solutions produced by the algorithm. So `move010`$(1, A, B, C)$ and `move010`$(2, A, B, C)$ are proven optimal.
  `move010`(N, A, B, C) for $N > 2$ corresponds to a path going through the states (the first disk being fixed on b): (please report to Fig. 17 from $aCC \dots CC$ to $aBB \dots BB$ down left to down right.)

$$aCC \dots CC \xrightarrow{f_{010}(N-2)} aCB \dots BC \xrightarrow{2} aAB \dots BC$$
$$\xrightarrow{f_{010}(N-2)} aAC \dots CC \xrightarrow{2} aBC \dots CB \xrightarrow{f_{010}(N-2)} aBB \dots BB$$

We shall demonstrate that all other paths take more steps:

- The states $bAC \dots CA$ and $bCA \dots AC$ are mandatory, for connectivity, and so are $bAC \dots CB$ and $bCA \dots AB$.
- going through the state $bBC \dots CB$ makes the state $bBA \dots AB$ mandatory.
- going around the state $bBC \dots CB$ makes the states $bAB \dots BB$, $bCB \dots BB$ and $bCA \dots AB$ mandatory: the total path would be of length $3 + 4f_{010}(N-2)$, to be compared with $4 + 3f_{010}(N-2)$ (We trade one step with one recursive call). As $f_{010}(N-2) \geq 1$ for $N - 2 \geq q$ (i.e. $N \geq 3 > 2$), `move010`(N, A, B, C) is optimal for $N > 2$.

So `move010`(N, A, B, C) is optimal. ◀

We discuss further extensions of those results in the next section.

5 Discussion

All the usual research questions and extensions about the Hanoï Tower problem are still valid about the Selenite Tower problem. We discuss only a selection of them, such as the space complexity in Section 5.1, and the extension to other proportional `insertion` and `removal` points in Section 5.2.

5.1 Space Complexity

Allouche and Dress [1] showed that the optimal sequence of steps required to move a Hanoï Tower of n disks can be obtained by a simple function from the prefix of an infinite unique sequence, which itself can be produced by a finite automaton. This proves that the space complexity of the Hanoï Tower problem is constant.

The same technique does not seem to yield constant space for Selenite Towers: whereas the sequences of steps generated by each of the functions `move100`(n, A, B, C), `move010`(n, A, B, C) and `move001`(n, A, B, C) are prefixes of infinite sequences, extracting those suffixes and combining them in a sequence corresponding to `move000`(n, A, B, C) would require a counter using logarithmic space in the length of the sequences to be extracted, i.e. $\log_2(\sqrt{3}^n) \in \Theta(n)$, which would still be linear in the number of disks.

5.2 Levitating Towers

An extension of the Selenite Tower problem is to parametrize the `insertion` point, so that the `removal` point is at position $\lfloor \alpha n \rfloor + 1$ and the `insertion` point is under the disk at position $\lfloor \alpha(n+1) \rfloor$ in a tower of n disks, for $\alpha \in [0, \frac{1}{2}]$ fixed (the problem is symmetrical for $\alpha \in [\frac{1}{2}, 1]$). By analogy with Selenite Towers, we call this variant a α-Levitating Tower. This parametrization creates a continuous range of variants, of which the Hanoï Tower problem and the Selenite Tower problem are the two extremes:

- for $\alpha = 0$, the `removal`/`insertion` point is always at the top, which corresponds to a Hanoï Tower, while
- for $\alpha = \frac{1}{2}$ the problem corresponds to a Selenite Tower.

The complexity of moving a α-Levitating Tower cannot be smaller than the one of a Selenite Tower, as the key configuration permitting to move 2 disks in 2 steps between the same pegs is less often obtainable in a α-Levitating Tower.

Acknowledgements. We would like to thank Claire Mathieu, Jean-Paul Allouche and Srinivasa Rao for corrections and encouragements, and Javiel Rojas-Ledesma and Carlos Ochoa-Méndez for their comments on preliminary drafts.

Funding. Jérémy Barbay is partially funded by the Millennium Nucleus RC130003 “Information and Coordination in Networks”.

References

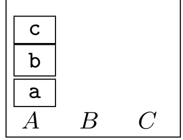
1 J.-P. Allouche and F. Dress. Tours de Hanoï et automates. *RAIRO, Informatique Théorique et applications*, 24(1):1–15, 1990.
2 M.D. Atkinson. The cyclic towers of Hanoï. *Information Processing Letters (IPL)*, 13(118-119), 1981.
3 W. R. Ball. *Mathematical Recreations and Essays*. McMillan, London, 1892.
4 J. S. Frame and B. M. Stewart. Solution of problem no 3918. *American Mathematics Monthly (AMM)*, 48:216–219, 1941.
5 Édouard Lucas. La tour d’Hanoï, véritable casse-tête annamite. In a puzzle game., Amiens, 1883. Jeu rapporté du Tonkin par le professeur N.Claus (De Siam).
6 Édouard Lucas. *Récréations Mathématiques*, volume II. Gauthers-Villars, Paris, quai des Augustins, 55, 1883.
7 D. Wood. The towers of Brahma and Hanoï revisited. *Journal of Recreational Mathematics (JRM)*, 14(1):17–24, 1981.

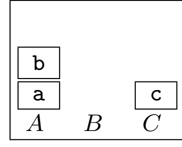
A Disk Pile Problem

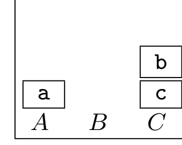
The Hanoï Tower problem is a classic example on recursivity, originally proposed by Édouard Lucas [5] in 1883. A recursive algorithm is known since 1892, moving the n disks of a Hanoï Tower in $2^n - 1$ unit moves, this value being proven optimal by a simple lower bound [3].

Consider the Disk Pile problem, a very simple variant where we allow some disks to be of the same size. This obviously introduces some much easier instances, including an extreme one where the disks are all the same size and the resulting tower can be moved in linear time (see Figure 18 for the sequence of steps moving such a tower of size 3 with a single size of disks).

1. Give a recursive algorithm to move a Disk Pile from one peg to the other, using only one extra peg, knowing that $\forall i \in \{1, \ldots, s\}$, n_i is the number of disks of size i. Your algorithm must be efficient for the cases where all the disks are the same size, and where all the disks are of distinct sizes.
   Solution: We present an algorithm in Figure 19. It is very similar to the algorithm for moving a Hanoï Tower, the only difference being that it moves the n_i disks of size i at the same time, in n_i consecutive moves. ◀







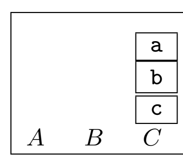


Figure 18 Moving a Disk Pile of size 3.

```
def diskPileMove(n,sizes,a,b,c):
  if n>0 :
    move(n-sizes[-1],sizes[0:-1],a,c,b)
    for i in range(0,sizes[-1]):
      move(a,c)
    move(n-sizes[-1],sizes[0:-1],b,a,c)
```

Figure 19 Python code to move a Disk Pile.

2. Give and prove the worst case performance of your algorithm over all instances of fixed s and vector $(n_1, \ldots, n_s)$.
Solution: By solving the recursive formula directly given by the recursion of the algorithm, one gets that the n_s largest disks are moved once, the n_{s-1} second largest disks are moved twice, the n_{s-2} third largest disks are moved four times, and so on to the n_1 smallest disks, which are each moved 2^{s-1} times. Summing all those moves gives the number of moves performed by the algorithm:
$$\sum_{i\in\{1,\ldots,s\}} n_i 2^{s-i}.$$
Note that for $s = n$ and $n_1 = \cdots = n_s = 1$, this yields $\sum_{i=1}^{s-1} 2^i = 2^n - 1$, the solution to the traditional Hanoï Tower problem. ◀
3. Prove that a performance of $\sum_{i\in\{1,\ldots,s\}} n_i 2^{s-i}$ is optimal.
Solution: We prove a lower bound of $\sum_{i\in\{1,\ldots,s\}} n_i 2^{s-i}$, for n disks of s distinct sizes, with n_i disks of size i by induction on the number of types of disks. We prove by induction on the number of types of disk s that any pile of disks of sizes $(n_1, \ldots, n_s)$ requires $\sum_{i\in\{1,\ldots,s\}} n_i 2^{s-i}$ disk moves to be moved to another peg.
   - *Initial Case*: for $s = 1$ the bound is n_1 and is obviously true, since each disk must be individually moved from one peg to the other.
   - *Inductive Hypothesis*: suppose there is some $\sigma \geq 1$ so that any pile of disks sizes $(n_1, \ldots, n_\sigma)$ requires $\sum_{i\in\{1,\ldots,\sigma\}} n_i 2^{\sigma-i}$ disk moves to be moved to another peg.
   - *Inductive Step*: consider a pile of disks of sizes $(n_1, \ldots, n_{\sigma+1})$: clearly all the disks of sizes smaller than $\sigma + 1$ need to be gathered on a unique peg before the largest disks can be moved, to allow those last ones to be moved in $n_{\sigma+1}$ disk moves, after which all the disks of sizes smaller than $\sigma + 1$ need to be stacked above the largest ones. By the inductive hypothesis, moving the smaller disks will require $\mathbf{2}\sum_{i\in\{1,\ldots,\sigma\}} n_i 2^{\sigma-i}$ disk moves, to be added to the $n_{\sigma+1}$ disk moves. Hence, any pile of disks of sizes $(n_1, \ldots, n_{\sigma+1})$ requires $\sum_{i\in\{1,\ldots,\sigma+1\}} n_i 2^{\sigma+1-i}$ disk moves to be moved to another peg.
   - *Conclusion*: The inductive hypothesis is verified for the initial case where $s = 1$, and propagates to any value of $s \geq 1$ through the inductive step. We conclude that any pile of disks of sizes $(n_1, \ldots, n_s)$ for $s \geq 1$ requires $\sum_{i\in\{1,\ldots,s\}} n_i 2^{s-i}$ disk moves to be moved to another peg.

   ◀
4. What is the worst case complexity of the Disk Pile problem over all instances of fixed value s and fixed total number of disks n? *Solution:* The worst case (of both the algorithms and the most precise lower bound with the number of disks of each size fixed) occurs when $n_1 = n - s + 1$ and $n_2 = \ldots = n_s = 1$. Using the previous results, it yields a complexity of $2^{s-1}(n - s + 1) + \sum_{i=1}^{s-1} 2^i = 2^{s-1}(n - s + 2) - 1$ steps in the worst case over all instances of fixed value s and fixed total number of disks n. This correctly yields $2^n - 1$ when $s = n$, in the worst case over all instances of fixed total number of disks n. ◀

B Code to generate the sequences of moves

See Figure 20 for the python code used to print the sequence of moves for a given SELENITE TOWER. For $n = 10$ it generates the following sequences:

```
move000(1,'a','b','c') =  (a,c)

move000(2,'a','b','c') = (a,b) (a,c) (b,c)

move000(3,'a','b','c') = (a,b) (a,b) (a,c) (b,c) (b,c)

move000(4,'a','b','c') = (a,c) (a,b) (a,b) (c,b) (a,c) (b,a) (b,c)
(b,c) (a,c)

move000(5,'a','b','c') = (a,c) (a,c) (a,b) (a,b) (c,a) (c,b) (a,b)
(a,c) (b,c) (b,a) (c,a) (b,c) (b,c) (a,c) (a,c)

move000(6,'a','b','c') = (a,b) (a,c) (a,c) (b,c) (a,b) (a,b) (c,b)
(c,a) (c,a) (b,c) (a,b) (a,b) (c,b) (a,c) (b,a) (b,c) (b,c) (a,b)
(c,a) (c,a) (b,a) (b,c) (b,c) (a,b) (a,c) (a,c) (b,c)

move000(7,'a','b','c') = (a,b) (a,b) (a,c) (a,c) (b,a) (b,c) (a,c)
(a,b) (a,b) (c,a) (c,b) (a,b) (c,a) (c,a) (b,a) (b,c) (a,c) (a,b)
(a,b) (c,a) (c,b) (a,b) (a,c) (b,c) (b,a) (c,a) (b,c) (b,c) (a,c)
(a,b) (c,b) (c,a) (c,a) (b,c) (b,a) (c,a) (b,c) (b,c) (a,c) (a,b)
(c,b) (a,c) (a,c) (b,c) (b,c)

move000(8,'a','b','c') = (a,c) (a,b) (a,b) (c,b) (a,c) (a,c) (b,c)
(b,a) (b,a) (c,b) (a,c) (a,c) (b,c) (a,b) (a,b) (c,b) (c,a) (c,a)
(b,c) (a,b) (a,b) (c,b) (c,a) (c,a) (b,c) (b,a) (b,a) (c,b) (a,c)
(a,c) (b,c) (a,b) (a,b) (c,b) (c,a) (c,a) (b,c) (a,b) (a,b) (c,b)
(a,c) (b,a) (b,c) (b,c) (a,b) (c,a) (c,a) (b,a) (b,c) (b,c) (a,b)
(a,c) (a,c) (b,a) (c,b) (c,b) (a,b) (c,a) (c,a) (b,a) (b,c) (b,c)
(a,b) (c,a) (c,a) (b,a) (b,c) (b,c) (a,b) (a,c) (a,c) (b,a) (c,b)
(c,b) (a,b) (a,c) (a,c) (b,a) (b,c) (b,c) (a,c)
```

C Code used to generate the values of the complexity

See Figure 21 for the python code used to experimentally generate the values of the complexity. For $n = 16$, it generates the following array:

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
f_{010}	0	1	3	7	13	25	43	79	133	241	403	727	1213	2185	3643	6559
f_{100}	0	1	2	4	7	13	22	40	67	121	202	364	607	1093	1822	3280
f_{000}	0	1	3	5	9	15	27	45	81	135	243	405	729	1215	2187	3645
$3^{\lceil n/2 \rceil}$	1	3	3	9	9	27	27	81	81	243	243	729	729	2187	2187	6561

```
def move(a,b):
  print "("+a+","+b+")",

def move010(n,a,b,c):
  if n == 1 :
    move(a,c)
  elif n == 2:
    move(a,b)
    move(a,c)
    move(b,c)
  elif n>2 :
    move010(n-2,a,b,c)
    move(a,b)
    move(a,b)
    move010(n-2,c,b,a)
    move(b,c)
    move(b,c)
    move010(n-2,a,b,c)

def move100(n,a,b,c):
  if n == 1 :
    move(a,c)
  elif n>1 :
    move100(n-2,a,c,b)
    move(a,c)
    move(a,c)
    move010(n-2,b,a,c)

def move001(n,a,b,c):
  if n == 1 :
    move(a,c)
  elif n>1:
    move010(n-2,a,c,b)
    move(a,c)
    move(a,c)
    move001(n-2,b,a,c)

def move000(n,a,b,c):
  if n == 1 :
    move(a,c)
  elif n>1 :
    move100(n-1,a,c,b)
    move(a,c)
    move001(n-1,b,a,c)

for i in range(1,9):
  print("move000("
   +str(i)+",'a'"
   +",'b'"+",'c') = "),
  move000(i,'a','b','c')
  print
  print
```

Figure 20 Python code to move a Selenite Tower.

```
def f010(n):
  if n < 2 :
    return n
  elif n == 2:
    return 3
  else:
    return 3*f010(n-2)+4

def f100(n):
  if n < 2 :
    return n
  else:
    return f100(n-2)+f010(n-2)+2

def f000(n):
  if n < 2 :
    return n
  else:
    return 2*f100(n-1)+1

def power(n):
  if n % 2 == 0 :
    return 3**(n/2)
  else:
    return 3**((n+1)/2)

def printArray(n):
  print("n\t "),
  for i in range (0,n):
    print(" &\t "+str(i) ),
  print("\\\\ \\hline")
  print("f_{010}\t "),
  for i in range (0,n):
    print(" &\t" + str(f010(i))),
  print("\\\\ \\hline")
  print("f_{100}\t "),
  for i in range (0,n):
    print(" &\t" + str(f100(i))),
  print("\\\\ \\hline")
  print("f_{000}\t "),
  for i in range (0,n):
    print(" &\t" + str(f000(i))),
  print("\\\\ \\hline")
  print("3^{\\lceil n/2\\rceil }\t "),
  for i in range (0,n):
    print(" &\t "+str(power(i)) ),
  print("\\\\")

printArray(16)
```

Figure 21 Python code to generate experimentally the values of the complexity.

Algorithms and Insights for RaceTrack

Michael A. Bekos[1], Till Bruckdorfer[2], Henry Förster[3], Michael Kaufmann[4], Simon Poschenrieder[5], and Thomas Stüber[6]

1 Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, Germany
2 Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, Germany
3 Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, Germany
4 Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, Germany
5 Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, Germany
6 Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, Germany

Abstract

We discuss algorithmic issues on the well-known paper-and-pencil game RaceTrack. On a very simple track called Indianapolis, we introduce the problem and simple approaches, that will be gradually refined. We present and experimentally evaluate efficient algorithms for single player scenarios. We also consider a variant where the parts of the track are known as soon as they become visible during the race.

1998 ACM Subject Classification G.2.2 Graph Theory

Keywords and phrases Racetrack, State-graph, complexity

Digital Object Identifier 10.4230/LIPIcs.FUN.2016.6

1 Introduction

RaceTrack is a paper-and-pencil game that simulates a car race. The game is played by two or more players on a squared sheet of paper, where pencil-lines keep track of cars' moves [8]. Cars move from one point to a new point of the underlying grid along a *track*, whose boundary is freehand drawn on the sheet of paper. Each move is subject to some *rules* that aim in simulating a car with a certain inertia and physical limits on traction. The rules are relatively simple and so the game is very popular even for pupils:

R.1: No two cars can simultaneously occupy the same grid point.

R.2: A grid point occupied by a car must lie within the track. In addition, the trajectory of a car must not intersect the boundaries of the track.

R.3: At each move, a car can change its speed by at most one unit of distance at the horizontal and/or at the vertical direction.

Initially, the cars are aligned along the so-called *start line*. At each turn, a player moves her car along the track according to rules R.1–3. The first car crossing the so-called *finish line* in a specific direction wins. Note that rule R.3 is also referred to as *eight-neighbours* rule. The reason is the following. Each move of a car can be represented by a 2-dimensional vector, e.g., a move two units to the right and four units downwards corresponds to vector $(2, -4)$. Hence, at each move each coordinate of this vector is allowed to change by ± 1 (simulating *acceleration* and *deceleration*, respectively), which gives rise to nine possible new grid points for the next move (eight of which are neighboring the previous one). Note that the game also serves as an educational tool for teaching vectors.

From an algorithmic point of view, the *state* of a car can be nicely encoded by a quadruple (x, y, s_x, s_y), where (x, y) denotes the position of the car on the track and (s_x, s_y) correspond

© Michael A. Bekos, Till Bruckdorfer, Henry Förster, Michael Kaufmann, Simon Poschenrieder, and Thomas Stüber;
licensed under Creative Commons License CC-BY
8th International Conference on Fun with Algorithms (FUN 2016).
Editors: Erik D. Demaine and Fabrizio Grandoni; Article No. 6; pp. 6:1–6:14
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

to its speed. By the eight-neighbours rule, it follows that subsequent states (x, y, s_x, s_y) and (x', y', s'_x, s'_y) must comply with the following two conditions: (i) $x' = x + s'_x$ and $y' = y + s'_y$, (ii) $|s_x - s'_x| \leq 1$ and $|s_y - s'_y| \leq 1$. The goal is to compute the minimum number of subsequent states from a certain starting position to the finish line.

The remainder of this paper is structured as follows: Section 2 overviews related work. In Section 3, we present a simple formulation of the problem as a shortest-path problem. We employ this simple formulation in order to develop a more efficient algorithm for a discretized version of the problem, where the boundaries of the track are rectilinear line segments (see Section 4). In Section 5, we consider a variant where the parts of the track are known as soon as they become visible during the race and we propose different strategies to drive safely but as fast as possible along the track. We experimentally evaluate our algorithms in Section 6. We conclude in Section 7 with open problems.

2 Previous work

To the best of our knowledge, the first reference to RaceTrack is due to Gardner [3] back in 1973, who seemed to learn about this game from a Swiss colleague. Note that the game is known under several names such as Vector Formula, Vector Rally, Vector Race, Graph Racers, PolyRace, Paper and pencil racing, or the Graph paper race game [8].

Next to Gardner, Erickson [2] discussed RaceTrack in his blog, where he considered the classic problem on the grid graph to be polynomial time solvable and conjectured the computational problem to be PSPACE-complete when the boundary of the tracks are given by sequences of line segments, which might make the size of the output exponential.

Although the running time for an algorithm is polynomial in the number of allowed grid points ($O(n^3)$ as stated by Erickson), Holzer and McKenzie [5] considered the decision problem for a winning strategy, distinguished variants where the boundary might be touched or not, and showed an NL-completeness result for the single-player variant. For the original 2-player variant, they show P-completeness, and summarize that "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".

Schmid [7] presents a BFS-based algorithm to find the fastest path of a car through a two-dimensional track. Olsson and Tarandi [6] describe an implementation of a genetic algorithm for RaceTrack. Ahlmann-Ohlsen [1] apply binary decisions diagrams to RaceTrack.

3 A first approach

The simplest scenario is to determine the minimum number of moves from a starting position along the track to the finish line, in the case where there exists only one car. Even though this problem sounds like a simple shortest-path problem, a closer look will reveal several problems that we will shortly analyze in more detail.

Following Erickson's approach, we assume that the underlying grid is of size $n \times n$. This implies that the maximum speed $s_{\max}$ of a car (either at the horizontal or at the vertical direction), which moves from one side of the track towards its opposite side cannot be more than $O(\sqrt{n})$, as the distance in one direction that is covered by the car with maximum speed $s_{\max}$ is at least $2 \sum_{i=0}^{s_{\max}-1} i + s_{\max}$, which is at most n.

Let $G = (V, E)$ be the so-called *state graph* which has a vertex for each possible state of a car and an edge between two states if and only if they are subsequent. By the eight-neighbors rule, G has out-degree at most 9. Hence, $|E| = O(|V|)$. Since the underlying grid is of size

$O(n) \times O(n)$ and each speed component ranges from 0 to $O(\sqrt{n})$ at most, the size of G is $O(n^3)$. To determine the minimum number of moves from a certain starting position to the finish line, we can simply perform a BFS algorithm on G. So, the complexity of this algorithm is $O(n^3)$, as independently observed by Erickson [2] and Schmid [7].

4 A more efficient approach

The algorithm described in the previous section shows that the problem of determining the minimum number of moves from a certain starting position to the finish line is polynomial time tractable, in the case of a single car (that is, rule R.1 is redundant). However, it is not difficult to observe that the size of the state graph can be too large even for relatively small tracks, which suggests that this first simple algorithm might not be really useful in practice. Motivated by this observation, in the following we will suggest an alternative approach that seeks to drastically reduce the size of the state graph (and therefore improve the usefulness of the corresponding shortest-path-based algorithm), assuming that the boundaries of the track are rectilinear line segments of particular lengths. Note that this is a quite reasonable assumption, as such tracks are quite common in RaceTrack.

In order to keep our presentation simple, we initially consider a simple rectangular-shaped track, that we call Indianapolis (see Figure 1). Later in this section, we show to which extend we can generalize our approach. Indianapolis track consists of four track segments of uniform width W that overlap exactly at the corner regions (dashed drawn in Figure 1). The horizontal track segments are of length L_1, while the vertical track segments are of height L_2, as illustrated in Figure 1. For simplicity, we assume that the parts of the track segments between the corner regions are at least of length W. Note that the time complexity of the algorithm of the previous section adjusted to Indianapolis is $O(W^{3/2}(L_1+L_2)^{3/2})$, because the underlying grid is of size $O(W(L_1+L_2))$ and speed s_x ranges between 0 and $O(\sqrt{L_1})$ when s_y ranges between 0 and $O(\sqrt{W})$ (horizontal track segment) and when s_y ranges between 0 and $O(\sqrt{L_2})$ then s_x ranges between 0 and $O(\sqrt{W})$ (vertical track segment).

4.1 Shrinking the size of the state graph for Indianapolis track

In order to shrink the size of the state graph, we will determine relatively small sets of points (and appropriate speed components for these points), so that a car must necessarily "land" on at least one of the points of each set in order to reach the finish line. We refer to such points as *landing points*. We also say that neighboring landing points form a so-called *landing region*. Intuitively, a landing point should be in or close to a corner region of the track. The idea underneath is that a car cannot reach the finish line without first passing through every corner of the track, in which its direction has to be changed (from horizontal to vertical or vice versa). Hence, our landing regions are nearly subsets of corner regions (which are of size $W \times W$; refer to the dashed-drawn squares of Figure 1).

Our approach is outlined as follows: First, we determine the set of potential landing points at each corner region of the track. Then, we determine the minimum number of moves between points of consecutive landing regions (which is a kind of all-pairs shortest-paths between consecutive landing regions). In a third step, we perform a weighted shortest-path search to compute the actual minimum number of moves from a starting position (through all computed landing regions) to the finish line.

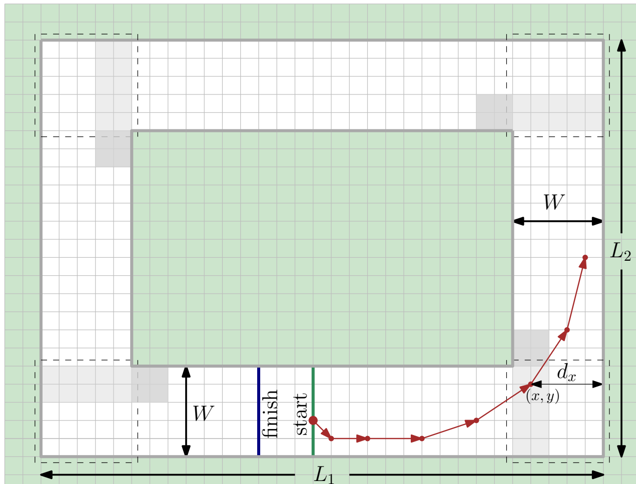


Figure 1 Illustration of the INDIANAPOLIS track: The corner regions are within the dashed-drawn squares. The landing regions are highlighted in gray.

Determining the landing points

Let $p = (x, y)$ be a potential landing point close to a corner of the track. If a car is at this point, then it must be able to change its direction, say w.l.o.g. from horizontal-right to vertical-up. Let $q = (x, y, s_x, s_y)$ be the state of the car at landing point p and let d_x be the distance of point p to the next (vertical) side of the track. Observe that if d_x is too small, then the car will crash. To avoid this scenario, the following relation must hold: $\sum_{i=1}^{s_x-1} i \leq d_x$. In the case where $d_x = W$, it follows that $s_x \leq \sqrt{2W}$. Consider now the set of points within the vertical strip of width $\sqrt{2W}$, whose left side coincides with the interior corner of the corresponding corner region (a part of it is highlighted in gray in Figure 1). From the above calculations, it follows that if a car passes through this area in the horizontal direction without landing on it, then it will crash on the next (vertical) side of the track. On the other hand, if a car has a speed $\leq \sqrt{2W}$ exactly before this vertical strip, then it will land on this strip and it will be able to avoid crashing.

So, this vertical strip is a good candidate for serving as a potential landing region. Next, we will further constrain its height. By the same arguments as before, s_y cannot be larger than $\sqrt{2W}$, if the car lands somewhere at the intersection of the vertical strip and the corner region and moves upwards. Alternatively, the car might avoid landing at the corner region, if it moves in one step from the left side of the corner region to some point above it. In this case, $s_y \leq \sqrt{2W} + 1$ and $s_x \leq \sqrt{2W}$ hold. It follows that the possible landing points for this case are in a rectangle of size $\sqrt{2W} \times (\sqrt{2W} + 1)$ whose bottom left corner coincides with the interior corner of the corresponding corner region. Hence, the landing region of this corner is formed by the union of this rectangle and the rectangle of size $\sqrt{2W} \times W$ below it. Note that some points associated with certain "low" speeds in the landing regions cannot be the first ones to land on. So, they can be neglected.

Moving from a landing point to a new landing point

We describe how to determine the minimum number of moves from one landing point p to another landing point p' of the next landing region. Let $q = (x, y, s_x, s_y)$ and $q' = (x', y', s'_x, s'_y)$ be the corresponding states of the car at points p and p', respectively. Since

p and p' belong to consecutive landing regions along the track, the movements in x- and y-directions can be handled independently. The only restriction is that the number of moves in one direction must be the same as the number of moves in the other direction. First, we determine the minimum numbers τ_x and τ_y of moves in x- and y-direction, respectively. Then, we "synchronize" them (by enlarging the smaller of the two), so to become same but still minimal.

In the following, we describe the computation for τ_x. The computation of τ_y is symmetric. Let $|x - x'| = \delta_x$ be the horizontal distance between p and p'. We assume w.l.o.g. that $s_x \leq s'_x$. The case where $s_x > s'_x$ is symmetric. Observe that τ_x is at least $s'_x - s_x$, because we have to perform at least $s'_x - s_x$ moves with speed increasing from s_x to s'_x. The distance α_x covered during theses moves is $(s_x + 1) + \ldots + (s_x + (s'_x - s_x)) = (s'_x - s_x)(s'_x + s_x + 1)/2$. If $\alpha_x = \delta_x$, then τ_x must be equal to $s'_x - s_x$. Otherwise, we consider the cases $\alpha_x < \delta_x$ and $\alpha_x > \delta_x$ separately. Before we proceed, we introduce the so-called *feasibility condition*, which determines whether distance δ_x can be covered within a certain number of moves.

Feasibility condition. For a fixed number t_x of moves, we can cover a maximum distance $\delta_{\max}(t_x)$ by accelerating as long as possible (namely for $(t_x + s'_x - s_x)/2$ steps) and then by decelerating enough to reach s'_x (namely for $(t_x - s'_x + s_x)/2$ steps). Analogously, the minimum distance $\delta_{\min}(t_x)$ that can be covered with a certain number t_x of moves is obtained by decelerating as long as possible (namely for $(t_x - s'_x + s_x)/2$ steps) and then by accelerating enough to reach s'_x (namely for $(t_x + s'_x - s_x)/2$ steps); see Figure 2a. Note that in our calculations we assume that the parities of the distances and the differences of the speeds are appropriate, otherwise we have to slightly adjust them, which can easily be done.

If $\delta_{\min}(t_x) \leq \delta_x \leq \delta_{\max}(t_x)$, then it is feasible to cover distance δ_x in t_x moves (we refer to this condition as *feasibility condition*). In particular, we know that the route realizing distance δ_x resides between the two routes realizing distances $\delta_{\min}(t_x)$ and $\delta_{\max}(t_x)$. So, in order to realize distance δ_x in t_x moves, we have either to accelerate or decelerate for a certain amount of steps, followed by some steps where the speed remains unchanged and finally decelerate or accelerate to reach speed s'_x. This gives rises to three different cases, which can be computed by solving the corresponding quadratic equations. Note that during the second period where the speed should remain unchanged, we might have to change it once by one unit, in order to reach precisely the desired distance δ_x; see Figure 2c.

Finding the smallest number of moves under the feasibility condition. In the following, we seek for the smallest t_x for which the feasibility condition holds. Note that $\delta_{\max}(s'_x - s_x) = \delta_{\min}(s'_x - s_x)$. Let t^*_x be the number of moves when the car brakes down to zero speed and then accelerates again to speed s'_x (observe that in this case the car moves only "forward"). We will consider two cases: (i) $\tau_x \leq t^*_x$ and (ii) $\tau_x > t^*_x$.

First consider Case (i). This case is possible, only if $\delta_{\max}(t^*_x) \geq \delta_x$ (which implies that $\alpha_x < \delta_x$ holds; if $\alpha_x > \delta_x$, then τ_x must be larger than t^*_x). Let $t^+_x \leq t^*_x$ be the smallest value for which $\delta_{\max}(t^+_x) \geq \delta_x$ holds (note that t^+_x can be computed by solving the corresponding quadratic equation). If additionally $\delta_{\min}(t^+_x) \leq \delta_x$ holds, then by the feasibility condition it follows that $\tau_x = t^+_x$. Otherwise, τ_x cannot be smaller than t^*_x (that is, Case (ii) applies).

From the above, it follows that in Case (ii) either $\delta_{\max}(t^*_x) < \delta_x$ or $\delta_{\min}(t^+_x) > \delta_x$ holds. To compute the minimum distance $\delta_{\min}(t_x)$ that can be covered by a car in t_x steps when $t_x > t^*_x$, we observe that the car has to reverse its direction. This subdivides the movement into four phases: The car has to brake from speed s_x to zero speed, further brake to a "negative" speed, then accelerate to zero speed and then to s'_x; see Figure 2b. Only in the

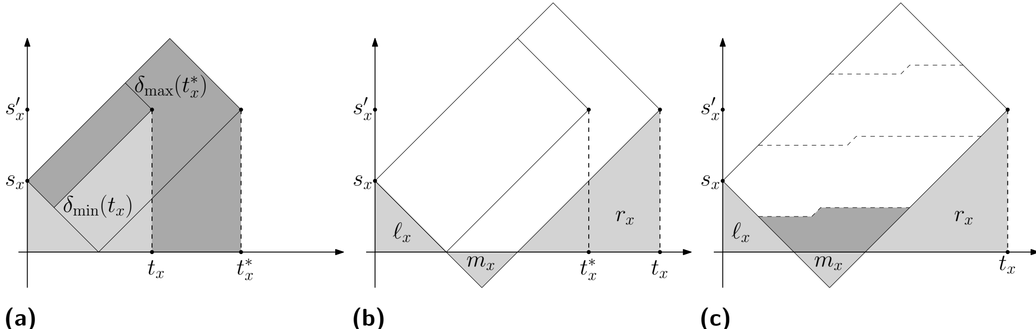


Figure 2 In the charts, the x-axis corresponds to the number of moves, the y-axis to the possible speeds. The light gray shaded areas indicate the corresponding distances for $\delta_{\min}(t_x)$. Figures (a) and (b) illustrate the configurations corresponding to $\delta_{\min}$ and $\delta_{\max}$ values for $t_x < t^*_x$ and $t_x > t^*_x$, respectively. Figure (c) illustrates the different cases that might arise when computing the route realizing δ_x when the feasibility condition holds for t_x. The dark gray region corresponds to the difference between δ_x and $\delta_{\min}(t_x)$ for one particular case.

first and the last phases the car moves from p towards p'. The distance covered in these two phases is fixed. In the first phase there are s_x steps of braking, which correspond to a distance of $s_x(s_x + 1)/2$. In the last phase there are s'_x steps of accelerating from zero, which correspond to a distance of $s'_x(s'_x + 1)/2$. Let ℓ_x and r_x be these two distances. To perform t_x step in total, we have to also perform $t_x - s_x - s'_x$ steps in negative, which corresponds to a distance of $(t_x - s_x - s'_x)((t_x - s_x - s'_x)/2 + 1)$ that we denote by m_x; see Figure 2b. It follows that: $\delta_{\min}(t_x) = \ell_x + r_x - m_x$.

Let $t^-_x \geq t^*_x$ be the smallest value for which $\delta_{\min}(t^-_x) \leq \delta_x$ holds (note that t^-_x can be computed by solving the corresponding quadratic equation). If $\delta_x \leq \delta_{\max}(t^-_x)$, then by the feasibility condition we have that $\tau_x = t^-_x$. If this is not the case (that is, $\delta_x > \delta_{\max}(t^-_x)$), then we have to increase the value of t^-_x until $\delta_x \leq \delta_{\max}(t^-_x)$. Note that for this particular value of t^-_x, it will also hold that $\delta_{\min}(t^-_x) \leq \delta_x$, as $\delta_{\min}$ is a decreasing function. In the following, we prove the correctness of our approach.

▶ **Lemma 1.** *The minimum number of moves computed under the feasibility condition ensures that the car does not crash.*

Proof. We prove the claim for the movement between two consecutive landing regions, which are connected, say w.l.o.g., by a vertical track segment as illustrated in Figures 1 and 2. Clearly, only the two vertical sides of the track are critical in this case. As discussed above, the movement might consist of one or three monotone parts, depending on Cases (i) or (ii). In Case (i), the movement consists of only one monotone part. Since there is no vertical side of the track between the two landing points of that part, the car will not crash. In Case (ii), the movement consists of three monotone parts. The first part consists of a braking phase ending with speed zero. Here, the choice of the landing point in the first landing region ensures that the car will not crash. Symmetrically the same holds for the third part, since it only consists of an acceleration phase from speed zero to the final speed. Since the endpoint of the first part and the starting point of the third part form the endpoints of the second part, and since the second part is also monotone, the car will not collide with vertical sides of the track. ◀

Coordinating the number of moves in both directions. Next, we discuss how to coordinate the moves in x- and y-direction so to become same. Assume w.l.o.g. that $\tau_y < \tau_x$. In this case, we set τ_y to be equal to τ_x and check whether the feasibility condition holds, that is, whether $\delta_{\min}(\tau_y) \leq \delta_y \leq \delta_{\max}(\tau_y)$. In the positive case, we have managed to coordinate the moves in x- and y-direction. In the negative case, $\delta_y < \delta_{\min}(\tau_y)$ must hold. So, we have to search for a new minimum value of τ_y to the right of t_y^* (which corresponds to Case (ii) in the description given above), that is, τ_y has to be at least as large as t_y^- (and larger than τ_x, as well). We do so and then we proceed by swapping the roles of τ_x and τ_y (to adjust τ_x to the new value of τ_y). In worst case, τ_x and τ_y will become equal, only when both are larger than the minimum of t_x^- and t_y^-, that is, after at most two steps.

From above it follows that we can compute in constant time the minimum number of movements between two landing points of consecutive landing regions (for a certain pair of speed components).

Computing the overall minimum number of moves

Finally, we compute the minimum number of moves from a starting position to the finish line, when moving from one landing region to another. This is a shortest path problem where the weights on the edges of the underlying state graph have been computed before in the second phase of our algorithm. Since the graph is acyclic, we can do this in linear time using topological sort: In every landing region we have $O(W^{3/2})$ points which are associated with $O(\sqrt{W}) \times O(\sqrt{W})$ of different values for the speed. Hence, in the state graph there exist $O(W^5)$ edges between any two adjacent landing regions. Since the number of landing regions of Indianapolis is constant, the state graph has $O(W^{5/2})$ vertices and $O(W^5)$ edges. The above analysis together with Lemma 1, yields the following theorem.

▶ **Theorem 2.** *The minimum number of moves for a single car in* Indianapolis *track can be computed in $O(W^5)$ time.*

Note that the time complexity of our algorithm is independent of the size of the track (that is, independent of L_1, L_2). It only depends on W which is usually a small constant. In particular, for $W = 7$, which is a typical value for the size of the track, the number of vertices of the state graph does not exceed 700. We provide more details in Section 6.

4.2 Extensions to more general tracks

In the general scenario, we define a track to consist of horizontal and vertical track-segments, which overlap only at their common corner regions. As in the previous subsection, we will assume that our track is of uniform width W. Hence, each corner region is of size $W \times W$. We will further assume that consecutive corner regions are separated by at least $2W$ units of distance in the horizontal or vertical direction, which also implies that no two corner regions overlap. In other words, we will assume that each track segment is of length at least $4W$.

Our goal is to construct appropriate landing regions close to the corner regions, as we did in the previous subsection. If we can guarantee that the maximal speed in each direction is $O(\sqrt{W})$, then the landing regions will be of approximately the same size as the corresponding ones of the Indianapolis track. Note that in general this is not the case, i.e., if track segments of length $3W$ are allowed. To see this, consider a staircase-shaped track, which consists of s horizontal and vertical track segments of length exactly $3W$, where s is a sufficiently large integer; see Figure 3a. Assume w.l.o.g. that W is an even integer and let the coordinates of the concave corners of this track be (iW, iW), $i = 1, 2, \ldots, s$. Then, a speed

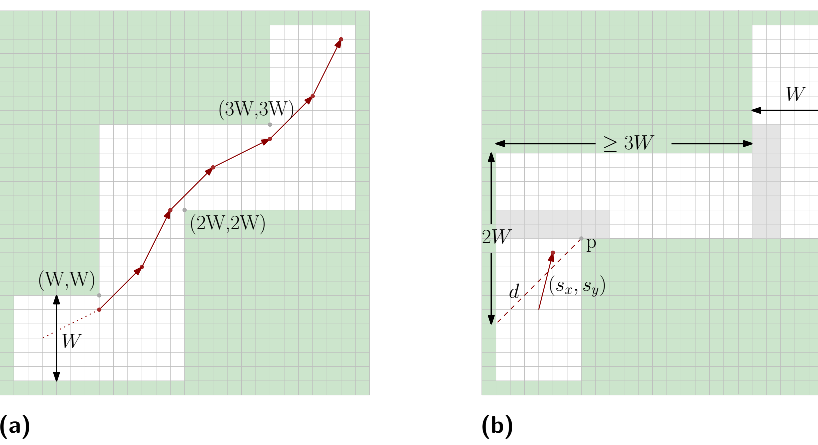


Figure 3 (a) Fast racing along a staircase-shaped track. (b) Entering a long track segment.

of $W/2$ in either directions is possible: We start at point $(W, W-1)$ with speed $(W/2, W/2)$. Then, we continue with speeds $(W/2+1, W/2-1)$, $(W/2, W/2)$, $(W/2-1, W/2+1)$ and $(W/2, W/2)$. As we show in Figure 3a, the car stays nicely in the track, which implies that indeed a speed of $W/2$ is possible in both directions. Note that such a high speed is not possible in the simple Indianapolis track.

Fortunately, if the segments of our track are of length at least $4W$, then we can limit the speed to $O(\sqrt{W})$ in both directions. To see this, consider a corner region which is open at the bottom and the right hand side, where a vertical segment ends and a horizontal segment starts. Let p be the concave corner at the corner region and let d be the diagonal passing through p with unit slope as shown in Figure 3b. When crossing this diagonal, the car will have a speed (s_x, s_y), such that $s_y > s_x$ holds.

We now claim that $s_x, s_y = O(\sqrt{W})$. To prove our claim we first consider the vertical speed and the number, say t, of steps that we can make in the vertical direction. There exist two scenarios. Either s_y is small enough to break before hitting the horizontal side of the track or s_y is not small enough to avoid hitting the horizontal side of the track. In the latter case we can prevent hitting the side of the track after t steps by reaching the next vertical segment of the track. Of course, in the first scenario it is reasonable to assume that the speed will be decreased as much as possible during these t steps. Since the distance to the horizontal side of the track is at most $2W$, it follows: $(s_y-1)+\ldots+(s_y-t) \leq 2W$ if and only if $s_y t - t(t+1)/2 \leq 2W$.

It follows that if $s_y \leq \sqrt{4W}$, then the car will be able to avoid hitting the horizontal side of the track. Since $s_y > s_x$, our claim follows. Next, we consider the second scenario. We can further assume w.l.o.g. that $s_y > \sqrt{4W}$. Now, we ask how many steps can we perform in the vertical direction before hitting the horizontal side of the track? In this case, that the maximum number of steps that we can perform is no more than $2W/s_y$. Hence, in this amount steps we have to be able to reach the next vertical segment of the track. W.l.o.g. we will suppose to accelerate in the horizontal direction for $t \leq 2W/s_y$ steps starting from speed s_x and see how far we can go. The distance that we will cover in the horizontal direction is:

$$(s_x+1)+\ldots+(s_x+t) = s_x t + t(t+1)/2 \leq \frac{s_x}{s_y}\cdot 2W + \frac{2W^2}{s_y^2} + \frac{W}{s_y} \leq 2W + \frac{W}{2} + \frac{\sqrt{W}}{2} < 3W\,.$$

Since we have assumed that all track segments are of length at least $4W$, we have obtained a contradiction. Therefore, $s_y \leq \sqrt{4W}$ which also implies that $s_x = O(\sqrt{W})$. Now, that

we know that the speeds are limited when crossing through the corner regions (provided that the track segments are long enough), we can determine the landing regions as described in Section 4.1. Since the distance from the diagonal defining (s_x, s_y) to the actual landing region is at most W, it follows that the speed in both directions cannot change by more than $O(\sqrt{W})$. Hence, the size of the landing region remains approximately the same as before. We conclude that the complexity of our algorithm to move from one landing region to the next landing region is the same as the one of Theorem 2.

▶ **Theorem 3.** *The minimum number of moves for a single car in a general track with rectilinear segments of length at least* $4W$ *can be computed in* $O(cW^5)$ *time, where c is the number of corners of the track.*

5 A competitive variant with limited view

In this section, we adopt the concepts that we developed in Section 4.2 and we use them in a scenario which reflects some real-world property, namely the "limited view". This scenario arises naturally in several computer games with the driver's view perspective, namely, with view limited to a certain portion of the track. Here, we assume that we can only see from one corner region to the next corner along one single track segment, and we additionally see where the next segment continues (left or right). We have no information about the sequence or the lengths of the track segments that follow. In this regard, we develop five heuristic strategies (without any approximation guarantee) that lead to considerably more efficient implementations.

The rule we face is the following: We assume that we are currently landed on a certain landing point. From there, we can compute the necessary moves to all points in the next landing region. This can be accomplished using our techniques from the Section 4.2. Based only on this information, we choose one single position in the next landing region, where we go next. So, the selection of the new landing point is the subject of a strategy that we discuss in the following. When choosing the strategy, we have to distinguish between long track segments, where we are able to move from one side of the track to the other and short segments, where the entering positions to the new track will play a major role.

To cope with the limited view scenario, we develop a list of possible strategies. For the sake of simplicity, we will assume w.l.o.g. that the current track segment is horizontal followed by a vertical one and that the next landing region resides along a left turn. Our strategies for the remaining cases are defined symmetrically.

S.1 Drive safely: According to this strategy, we choose the middle-position of each landing region, which can be reached with the minimum number of moves from our current position (formally the middle possition of a landing region is identified with the middle row of it). We also choose a speed equal to at most $2\sqrt{W}$ in both directions (*safe speed*), which ensures that the car will not collide, regardless of the sequence of track segments that follow. Since there exist $O(\sqrt{W})$ different middle positions in total, the time required to perform our choice is $O(W^{3/2})$.

S.2 Drive carefully: Again, we choose the middle-position of each landing region, which can be reached with the minimum number of moves from our current position. However, we choose the speed to be equal to at most $\sqrt{W}$ in both directions (*careful speed*), which is half of the maximum speed of the previous strategy. Although the strategy is inspired by German in-town speed regulations, it has the effect that the car might avoid time-consuming S-shaped movements within one track segment as it is the case of Figure 2b. Note that the time required to perform our choice is still $O(W^{3/2})$.

S.3 Topmost with highest y-speed: According to this strategy, we aim for one point of the topmost row of the next landing region, so that the speed in the vertical direction is maximized (but still safe to avoid collisions). Since there exist $O(\sqrt{W})$ different points in the topmost row and for each of them the maximum y-speed is uniquely defined, it follows that only the values for the x-speed may vary and so we can find the position, which also requires the smallest number of moves from our current position, in $O(W)$ time.

S.4 Be fast between landing regions: According to this strategy, we aim for any point of the next landing region, so that the distance between the current and the next landing region is covered as fast as possible. If there exist more than one alternatives, then we choose the one with the highest y-speed. This choice can be done in $O(W^{5/2})$ time.

S.5 Closest to the corner: According to this strategy, we aim for the point that is below and to the right of the internal corner of the turn, so that the x-speed at this point is one, while the y-speed is maximum. Since by construction the maximum speed at this point is uniquely defined, this choice can be done in $O(1)$ time.

Note that the running times that we gave above are restricted to pairs of landing regions. The total time complexity of our strategies is then subject to the total number of corners of the track. On the other hand, however, the theoretical performance of the strategies is a drastic improvement compared to the optimal algorithm (see the following proposition). We also note that the differences in the running times of the strategies might give a hint to the expected quality of the practical performance, which we discuss in Section 6.

▶ **Proposition 4.** *While the running time for computing the optimal solution is $O(cW^5)$ from Theorem 3, the corresponding running time for S.1 and S.2 is $O(cW^{3/2})$, for S.3 is $O(cW)$, for S.4 is $O(cW^{5/2})$, and for S.5 is $O(c)$, where c is the number of corners of the track.*

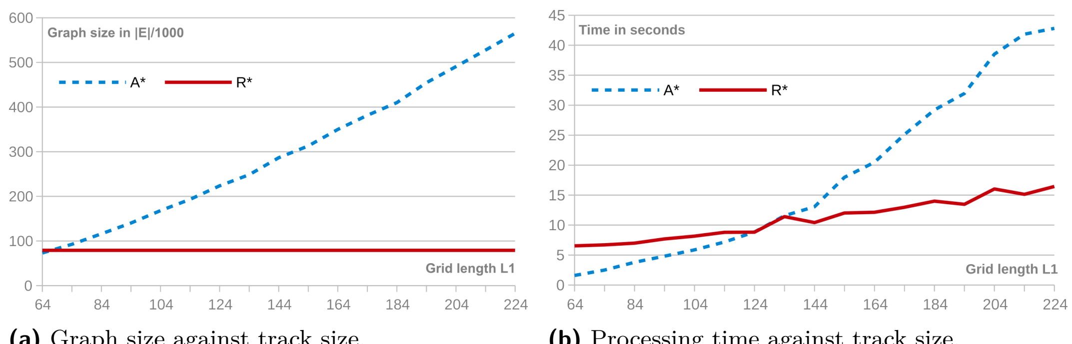
6 Evaluation

In this section, we present the results of the experimental evaluation of our algorithms. The experiment was performed on a Linux machine with four cores at 2,5 GHz and 8 GBs of RAM. The implementations were in Java. Apart from our algorithms, we have also implemented algorithm A* [4], which is common in path-finding and graph traversal problems. This algorithm is similar to Dijkstra's algorithm. It starts from a single source, but in contrast to Dijkstra's algorithm, only the vertices that are mainly in the direction to the target are processed assuming that this direction is somehow known. The vertices of the graph, which are too far away from the assumed direction, are ignored. The direction is usually estimated by a heuristic function, e.g., by the Euclidean distance to the target. In such a way, algorithm A* performs well in unknown search spaces.

Optimal algorithms

First, we experimentally compared our algorithm from Section 4 against algorithm A*. As a test set for our experiment, we used a simple (single-parameter) setting to get meaningful results. In particular, we used different instances of Indianapolis track, whose horizontal and vertical lengths were equal (that is, $L_1 = L_2$) ranging from 64 to 224 units of length. The width W of each track was set to 7 units of length.

Since the number of moves computed by both algorithms were equal (that is, both algorithms led to optimal solutions), we were mainly interested in comparing two aspects of these algorithms: (i) the total size of the underlying state graph (in terms of its number of edges) and (ii) the total time needed for computing the state graph together with finding the



(a) Graph size against track size.

(b) Processing time against track size.

Figure 4 Statistics on different instances of the Indianapolis track. Comparing algorithm A* against our algorithm from Section 4 denoted by R* w.r.t. graph size and processing time.

optimal route. The results of our experimental comparison are illustrated in Figure 4. In the plots, the curve denoted by R* stands for results of our algorithm from Section 4. The curve denoted by A* corresponds to the results obtained from algorithm A*.

It is eye-catching, that our algorithm uses a graph that is by far smaller than the corresponding graph used by algorithm A* (see Figure 4a). As expected, the size of the graph of algorithm A* increases as the size of the track increases (ranging from 73.205 to 565.212 edges in our experiment). On the other hand, the size of the state graph in our algorithm was constant in our experiment, because the width of the track was fixed (recall that the size of the state graph of our algorithm is independent of the dimensions of the track; it only depends on the width of the track). More precisely, its number of edges was constantly 79.056.

In Figure 4b, the computation time needed for computing the state graph together with finding the optimal route is plotted against the size of the tracks. More precisely, the computation time our algorithm is 6 seconds for a 64×64 track and increases slightly up to 16 seconds for larger tracks. On the hand, algorithm A* outperforms our algorithm for relatively small tracks of size at most 124×124. For larger tracks, however, it requires significantly more time (up to 43 seconds). Observe that the computation time of our algorithm increases as the size of the track increases, although the state graph remains of the same size. In our prototype, the calculations described in Section 4 are not optimally implemented and thus require more time for larger tracks.

Limited view scenario

In the second phase of our experimental evaluation, we compared strategies S.1,. . . , S.5 from Section 5 against the algorithm from Section 4, which leads to optimal solutions. As a test set for our experiment, we used different instances of Staircase track, which consists of alternating horizontal and vertical track segments (see, e.g., Figure 3). For our experiment each instance of the Staircase track had 10 stairs (that is, each track had 11 horizontal track segments and 10 vertical track segments). In addition, the horizontal and vertical track segments were of equal lengths ranging from $4W$ to $20W$ units of length, where the width W of the tracks was set again to 7 units of length. The results of our experimental comparison are illustrated in Figure 5, where the number of moves required by each strategy is plotted against the track size. In the plots, the curve denoted by R* stands for results of our algorithm from Section 4. The curves denoted by S.1,. . . , S.5 corresponds to the results obtained from the corresponding strategies from Section 5.

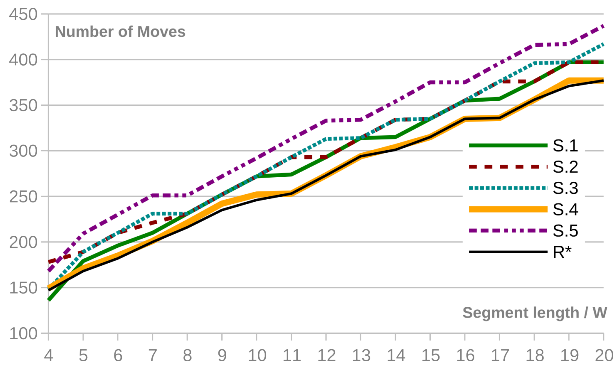


Figure 5 Statistics on different instances of the STAIRCASE track. Comparing our algorithm from Section 4 denoted by R* against strategies S.1,. . . ,S.5 w.r.t. the number of required moves.

It is clear that strategy S.4 (i.e., be fast between landing regions) outperforms the other ones as the length of the track segments increases. This is not surprising, since strategy S.4 seeks in covering the horizontal and the vertical track segments of the track as fast as possible. So, the longer these segments are, the more options this strategy has to cover the respective distances faster (and hence to win). Note also that strategy S.4 tends to have the same performance as the one of the optimal algorithm of Section 4 for tracks with long track segments. On the other hand, strategies S.1 and S.2 (i.e., drive safely and drive carefully, respectively) have, more or less, the same performance that is slightly better than the corresponding one of strategy S.3 (i.e., topmost with highest y-speed). Strategy S.5 (i.e., closest to the corner) is by far the one with worst performance, that is, it requires the maximum number of moves for the vast majority of the tracks of our experiment.

So, there seems to be a hierarchy between the strategies as the length of the track segments increases: S.4 → S.1 → S.2 → S.3 → S.5 from the best one to the worst one. This hierarchy is, up to a certain point, expected if one carefully observes the sizes of the graphs constructed by each strategy in order to compute the number of moves towards the next landing region (the higher a strategy is in the hierarchy, the larger is the graphs it uses), e.g., strategy S.4, which is the winning strategy, is the one that builds the largest graphs (and therefore takes into account more options). Strategies S.1 and S.2, that follow S.4 in the hierarchy, use graphs of comparable sizes. Hence, both have comparable performance, which is slightly better than the ones of S.3 and S.5 (but worse than the one of S.4).

An illustration of the routes of all strategies on a specific instance of the STAIRCASE track consisting of two stairs in total is given in Figure 6. The width W of this track is 7 (that is, equal to the one we used in our experimental evaluation; refer to Section 6). The length of the horizontal and vertical track segments of this track equal to $4W$ units of length. In Figure 6, we have highlighted by large dots the positions of the cars after their 10th and 20th move. It is eye-catching, that strategy S.4 (i.e., be fast between landing-regions; yellow-colored in Figure 6) outperforms all other ones. Also, observe that strategies S.1 and S.2 (i.e., drive safely and drive carefully; green- and red-colored, respectively) have, more or less, the same performance as stated earlier.

7 Conclusions

We developed and experimentally evaluated efficient algorithms for single player scenarios of RACETRACK. We considered variants where in one case the whole track is supposed to be

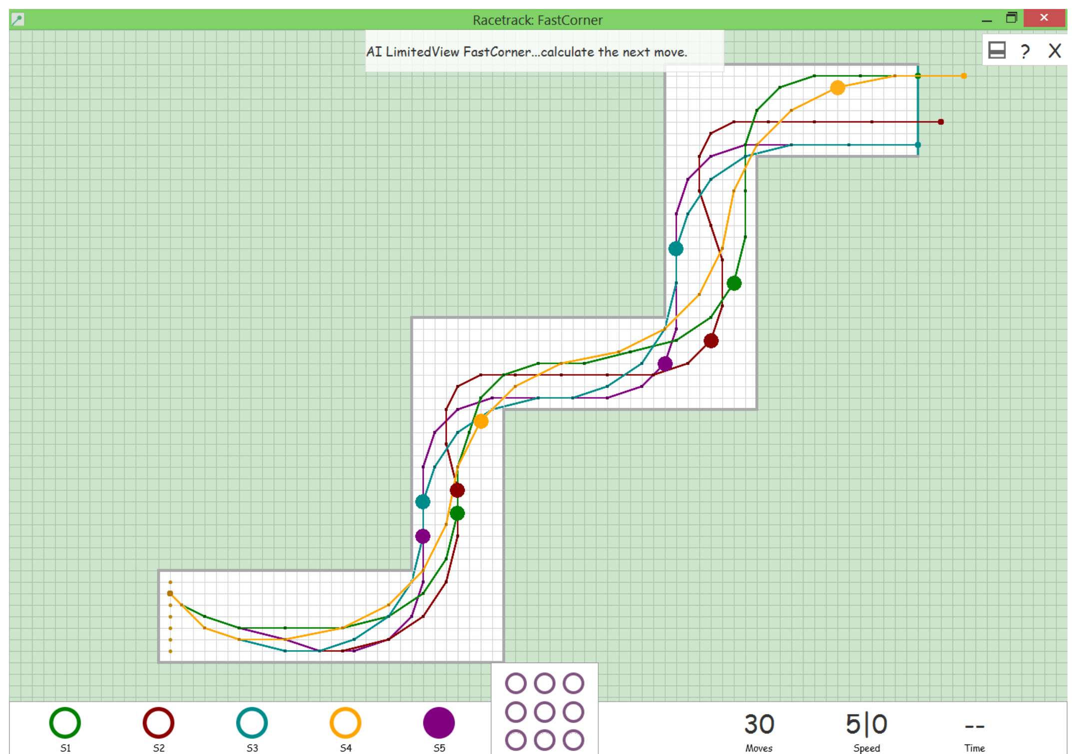


Figure 6 Illustration of the five strategies in the STAIRCASE track. The positions of the cars after their 10th and 20th move are highlighted by large dots.

known, or in another case the track is unknown except for these parts that are visible during the race (limited view). However, we restricted ourselves into discretized versions of the game and we used simple assumptions to make our algorithmic ideas applicable. In this regard, we conclude with the following open problems: (i) Apply the landing regions approach to scenarios with relatively short track segments (e.g., with overlapping corner regions) or to more challenging scenarios involving more than one player. (ii) Prove differences between overall optimal results and results from strategies under limited view. (iii) Prove quality bounds for different strategies. (iv) Develop efficient approaches for more general tracks.

Acknowledgments. This work has been supported by DFG grant Ka812/17-1. The authors would like to thank Stefan Feil, Denis Heid, Tobias Kaulich, and Sotirios Pavlidis for developing the user interface of our prototype.

References

1 Kristian Ahlmann-Ohlsen. Applying binary decision diagrams to solve the shortest path problem in vectorrace, 2005.

2 Jeff Erickson. How hard is optimal racing?, 2009. URL: `http://3dpancakes.typepad.com/ernie/2009/06/how-hard-is-optimal-racing.html`.

3 Martin Gardner. Mathematical games – Sim, chomp and race track: new games for the intellect (and not for lady luck). *Scientific American*, 228(1):108–115, 1973.

4 Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Systems, Science and Cybernetics*, SSC-4(2):100–107, 1968.

5 Markus Holzer and Pierre McKenzie. The computational complexity of racetrack. In Paolo Boldi and Luisa Gargano, editors, *FUN 2010*, volume 6099 of *LNCS*, pages 260–271. Springer, 2010.

6 Robert Olsson and Andreas Tarandi. A genetic algorithm in the game racetrack, 2011.

7 Jakob Schmid. VectorRace – finding the fastest path through a two-dimensional track, 2005. URL: `http://schmid.dk/articles/vectorRace.pdf`.

8 Wikipedia. Racetrack (game). URL: `https://en.wikipedia.org/wiki/Racetrack_game`.

Resource Optimization for Program Committee Members: A Subreview Article*

Michael A. Bender[1], Samuel McCauley[2], Bertrand Simon[3], Shikha Singh[4], and Frédéric Vivien[5]

1 **Stony Brook University, Stony Brook, NY 11794-4400 USA**
bender@cs.stonybrook.edu
2 **Stony Brook University, Stony Brook, NY 11794-4400 USA**
smccauley,shiksingh@cs.stonybrook.edu
3 **Université Lyon, LIP, CNRS – ENS de Lyon – INRIA, Lyon 69007 France**
bertrand.simon@inria.fr
4 **Stony Brook University, Stony Brook, NY 11794-4400 USA**
shiksingh@cs.stonybrook.edu
5 **Université Lyon, LIP, CNRS – ENS de Lyon – INRIA, Lyon 69007 France**
frederic.vivien@inria.fr

Abstract

This paper formalizes a resource-allocation problem that is all too familiar to the seasoned program-committee member. For each submission j that the PC member has the honor of reviewing, there is a choice. The PC member can spend the time to review submission j in detail on his/her own at a cost of C_i. Alternatively, the PC member can spend the time to identify and contact peers, hoping to recruit them as subreviewers, at a cost of 1 per subreviewer. These potential subreviewers have a certain probability of rejecting each review request, and this probability increases as time goes on. Once the PC member runs out of time or unasked experts, he/she is forced to review the paper without outside assistance.

This paper gives optimal solutions to several variations of the scheduling-reviewers problem. Most of the solutions from this paper are based on an iterated log function of C_i. In particular, with k rounds, the optimal solution sends the k-iterated log of C_i requests in the first round, the $(k-1)$-iterated log in the second round, and so forth. One of the contributions of this paper is solving this problem exactly, even when rejection probabilities may increase.

Naturally, PC members must make an integral number of subreview requests. This paper gives, as an intermediate result, a linear-time algorithm to transform the artificial problem in which one can send fractional requests into the less-artificial problem in which one sends an integral number of requests. Finally, this paper considers the case where the PC member knows nothing about the probability that a potential subreviewer agrees to review the paper. This paper gives an approximation algorithm for this case, whose bounds improve as the number of rounds increases.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Scheduling, Delegation, Subreviews

Digital Object Identifier 10.4230/LIPIcs.FUN.2016.7

* This work was partially supported by NSF Grants CNS 1408695, CCF 1439084, IIS 1247726, IIS 1251137, CCF 1217708, and Sandia National Laboratories. This research was conducted during Shikha Singh's stay at IBISC, University of Evry Val d'Essonne, Evry 91000, France supported by a Chateaubriand Fellowship, and Samuel McCauley's stay at ENS de Lyon, supported by INRIA and a Chateaubriand Fellowship.

© Michael A. Bender, Samuel McCauley, Bertrand Simon, Shikha Singh, and Frédéric Vivien;
licensed under Creative Commons License CC-BY
8th International Conference on Fun with Algorithms (FUN 2016).
Editors: Erik D. Demaine and Fabrizio Grandoni; Article No. 7; pp. 7:1–7:20
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Serving on the program committee (PC) for a flagship theory conference is rewarding for the PC members, but it is intense and time-consuming work. Each submitted manuscript is typically assigned to three or four PC members. As a result, when you serve on a PC, you are given somewhere between 10 and 60 manuscripts to review, depending on the committee. Reviews need to be completed rapidly—on the order of one month. For many of these papers, you have only a casual familiarity, meaning that doing a high-quality review will consume large amounts of time and may be fraught with uncertainty.

Consequently, most PC members rely on subreviewers to provide outside reviews. The subreviewer mechanism works as follows. When you serve on a PC, you decide which manuscripts require outside assistance. Then you scrape the Internet for subreviewers, sending request emails to friends, calling in favors, and searching out new domain experts.

Many of these subreview requests are denied. It is not uncommon that you need to send 3-8 review requests before you find a subreviewer that is not reading the paper for another PC member, already overburdened with reviews from the same conference, also serving on the PC, advising or being advised by one of the paper authors, having a baby, married to one of the paper authors, or just taking a well needed break from responding to email.

Thus, even the administrative process of collecting subreviews is work consuming and worth optimizing. One resource optimization, familiar to seasoned PC members, is the race to send out review requests the moment that the submissions have been meted out to the PC. The longer you wait to ask for a review from someone, the less likely it is that that person will comply. It is common that you miss out on collecting another subreviewer by just a few minutes—someone else's request showed up in the mail queue first.

Another optimization is the use of parallelism: send requests to several potential subreviewers in parallel to increase the probability of a positive answer. But every request takes time, and if multiple people say "yes," this leads to redundant work for the community and the PC member.

This paper studies this class of resource-allocation problem, which overburdened PC members naturally face while discharging their PC duties. Specifically, this paper studies the randomized resource-allocation problem of how to find subreviewers for the minimum expected cost. More generally, this paper explores how to delegate work to outside sources, when delegation has a cost, a nontrivial probability of failing, and the delegator must take this into account in his/her decisions.

Scheduling Model

There are N papers to review, denoted $1, \ldots, N$. Reviewing paper j yourself costs C_j. This cost is paper specific, because some submissions are easier for you to judge yourself, and others are harder. Each attempt to recruit a new subreviewer costs 1. The unit cost models the overhead of searching for email addresses, sending review requests, review reminders, thank you notes, requests for alternative reviewers, and all the other delegation overhead. Without loss of generality, $C_j > 1$; otherwise, it is not cost-effective to recruit subreviewers.

There is a (round-dependent and paper specific) probability of $p_{i,j}$ $(0 \leq p_{i,j} \leq 1)$ that at round i the candidate subreviewer declines to review paper j. Probability $p_{i,j}$ is monotonically increasing in i, since as time goes on, any candidate subreviewer is increasingly likely to be reviewing the paper for someone else or already has all of his/her time accounted for.

The reviewing process proceeds in $k+1$ rounds. In the first round, you email out a bolus of subreview requests for papers $1, 2, \ldots, N$. Some of these papers get successfully adopted

by subreviewers. For those that do not, you send a second bolus of review requests in the second round, and so forth. In the last round $k + 1$, you need to review those stray papers that did not find a subreviewer owner in the first k rounds.

The objective is to minimize the total expected cost to review all N papers. Because we are dealing with expected values, we can analyze the subreviewer-finding strategy for each submission in isolation. Henceforth, we focus on the case of a single paper, and the multiple-paper case follows by linearity of expectation. We simplify notation by dropping the subscripts that identify papers, writing C and p_i. The main exception is Section 3.3, where a request budget must be distributed among the N papers.

Modeling the process by a series of rounds faithfully captures how many PC members (including the authors on this paper) act when they serve on PCs. Round 1 opens with a flurry of energy and enthusiasm. It ends with a quiescent period filled with hope, despair, gratefulness, and occasional sleep. Round 2, 3, and so forth proceed similarly, as the PC members work up the gumption to renew soliciting reviews, after getting more declines from potential subreviewers.

Although we describe this problem using the colorful language of beating the bushes for subreviewers, this model similarly captures a natural parallel scheduling problem. There are N tasks (the papers). The objective is to schedule all tasks while minimizing the expected cost. A task j can be executed locally (you write the review) or remotely (you assign the review to a subreviewer). A local execution is expensive. A remote execution is cheaper, but even launching a job has a cost and fails with a certain probability.

Results

We first explore the case where the rejection probabilities are known.

- For the case of two rounds, we give a closed-form optimal solution for minimizing the expected reviewing cost when the number of requests is not required to be an integer (e.g., we are allowed to send 3/5th of a review request, if we want).
- We give a closed-form optimal solution for multiple rounds when the number of requests in each round is not required to be an integer. This closed-form solution gives intuition about how requests should be distributed over rounds, and is an important building block for our further results.
- Using the closed form solution, we construct an optimal algorithm for sending integral requests; the running time is linear in the number of rounds.
- We then consider the bounded-reviewers case. Specifically, we further generalize to the (common) case where a submission may have only a bounded number R of knowledgeable experts. Once all R experts decline to review the paper, all resources are exhausted and the beleaguered committee member has no choice but to review the submission without help. We give an optimal algorithm if the probabilities remain constant, an approximation algorithm when they are monotonically increasing, and a pseudo-polynomial algorithm when papers have to share a budget R.

We also study the scheduling subreviewer problem for the case where the probability that a subreviewer rejects the subreview request is constant but unknown to the PC member.

- We give a two round strategy which is a $(4\sqrt{C/\ln C}+2)$-approximation for any subreview cost $C \geq 2$.
- We further generalize to multiple rounds and obtain a k-round strategy that is a $k(C^{1/k}+1)$-approximation.

Related Work

The problem of optimizing the conference-review process through a careful assignment of papers to PC members has been studied in various guises [4, 28, 14, 25, 21]. Goldsmith and Sloan [13] address the problem of assigning papers to PC members in order to minimize "whingeing." Merelo-Guervós et al. [23] give evolutionary algorithms to optimize this assignment problem. Assigning conference papers to review has also been studied as a multi-agent fair-allocation problem [20]. Data mining and information-retrieval approaches have been used to model a reviewer's interest and build reviewer-recommender systems to assist paper assignment [24, 2, 11, 16]. Wang et al. [26, 27] present a survey on the various stages of the reviewer-assignment problem consolidating approaches from the areas of artificial intelligence, operations research, information retrieval, and algorithms.

Cormode [9] gives a fun guide on how *not* to review papers.

Probabilistic Scheduling. In many papers in scheduling and distributed computing, processors randomly choose which task to execute from a pool of tasks. Because multiple processors may choose the same task, there is some probability of wasted work. This is the case in the so-called do-all or write-all literature (among a very large literature, see for instance [22, 1, 8, 19]), as well as other contexts where processors randomly choose tasks to execute and may fail to find one that was not already chosen [17, 18, 3].

Fault tolerance. The subreviewer scheduling problem is related to the problem of executing tasks on failure-prone platforms, and particularly to the work on replication for fault-tolerance [7, 6]. There is an important difference, however. Most work on fault-tolerance for failure-prone HPC platforms assume that failures are rare [15]; typically the mean time between failures of a component is expressed in tens of years. This means that it is almost always better in practice only to duplicate failed tasks and not be proactive by replicating a priori. Furthermore, the low failure rate allows for first-order approximations [10, 5], which would be invalid in the current context of high rejection (i.e., failure) rates.

Preliminary Notions and Definitions

A *strategy* $\mathcal{S}$ is defined as the vector of requests sent in each round, that is, $\mathcal{S} = (R_1, R_2, \ldots, R_k)$. The probability of rejection is *constant* if $p_i = p$ for all rounds i, $1 \leq i \leq k$. The probability of rejection is *round-dependent* if p_i is monotonically increasing with respect to round i, that is, for any rounds i and j such that $1 \leq i < j \leq k$ we have $p_i \leq p_j$. We also consider the case of *unknown probability* where rejection probabilities are constant but not known to the PC member.

A round i *succeeds* if at least one request in round i is *accepted*, that is, some subreviewer in the round agrees to subreview the paper. If a round i succeeds, then no more requests are sent in the later rounds. A round i *fails* if all requests sent in round i are *rejected*, that is, none of the subreviewers asked want to subreview the paper. Round i fails with probability $p_i^{R_i}$. If rounds 1 through k fail, the $(k+1)$th round is reached. The PC member is then forced to review the paper on their own at a cost of C. We call C the *self-reviewing cost*.

We devise strategies that send nonnegative *integral number of requests*, that is, $R_i \in \mathbb{N}^0$ for $1 \leq i \leq k$. As an intermediate step, we construct strategies that send *nonintegral* or *real number of requests*, that is, $R_i \in \mathbb{R}^+$ for $1 \leq i \leq k$. For example, in round i the PC member could send 3/5ths of a request, meaning that the rejection probability is $p_i^{3/5}$. As a

shorthand, we call a strategy *integral* when the number of requests R_i for all rounds i is an integer. Otherwise, the strategy is a *nonintegral* or *real strategy*.

Finally, we consider bounds on the total number of requests a reviewer can make. First, we consider the setting where the PC member can make an *unbounded total number of requests* to an arbitrarily large pool of potential subreviewers. Second, we consider the situation where there is a *bounded total number R of requests* that the PC member can make, which models the situation that there are a limited number of experts capable of reviewing the submission.

2 Optimal Strategy to Schedule Subreviews

In this section, we give an optimal $(k+1)$-round strategy for sending out subreview requests. We consider the case where there is an arbitrary pool of potential subreviewers, that is, there is no upper limit on the total number of requests that the PC member is allowed to make.

As an intermediate step, we give an optimal nonintegral strategy to find a closed-form solution. Then we give a linear-time algorithm that uses the closed-form solution and finds the optimal integral solution.

We start by explaining the high-level idea behind the optimal strategy. Let R_i denote the number of requests we send out in round i. Then our total expected cost is the following:

$$E(R_1,...,R_k) = R_1 + p_1^{R_1}\left(R_2 + p_2^{R_2}\left(R_3 + ...\left(R_i + p_i^{R_i}\left(R_{i+1}... + p_{k-1}^{R_{k-1}}\left(R_k + p_k^{R_k}C\right)\right)\right)\right)\right).$$

Note that the expected reviewing cost *conditioned on reaching a particular round i* is independent of the past requests $R_1, R_2, \ldots, R_{i-1}$; call this cost E_i.

$$E_i(R_i, \ldots, R_k) = R_i + p_i^{R_i}\left(R_{i+1} + p_{i+1}^{R_{i+1}}\left(\ldots p_{k-1}^{R_{k-1}}\left(R_k + p_k^{R_k}C\right)\right)\right).$$

Thus, we can determine the request sequence $R_i, R_{i+1}, \ldots, R_k$ that minimizes E_i by working backwards from the last round.

2.1 Optimal Nonintegral Strategy

The expected reviewing cost conditioned on reaching round k only depends on the self-reviewing cost C. We first compute the optimal number of requests to be made in the last round R_k. This is equivalent to devising an optimal two-round strategy.

▶ **Lemma 1.** *A two-round optimal nonintegral strategy sends R requests, where,*

- $R = 0$ *if* $C \leq \frac{1}{\ln \frac{1}{p}}$*; then the total expected cost is* C*;*
- $R = \frac{1}{\ln \frac{1}{p}} \ln\left(C \ln \frac{1}{p}\right)$ *if* $C > \frac{1}{\ln \frac{1}{p}}$*; then the total expected cost is* $\dfrac{1}{\ln \frac{1}{p}}\left(1 + \ln\left(C \ln \dfrac{1}{p}\right)\right)$.

We generalize this notion to obtain a complete $(k+1)$-round strategy that sends real requests and is optimal. To simplify analysis, we first present the case where the probability of rejection is constant across all rounds.

▶ **Theorem 2.** *An optimal $(k+1)$-round strategy for sending real requests when the rejection probability p_i is constant, $p_i = p$, is to send R_i requests in round i for $1 \leq i \leq k$, where,*

- $R_i = 0$ *if* $C \leq \frac{1}{\ln(1/p)}$ *(no subreview requests) with a total expected cost of* $E = C$*.*

- *Otherwise,*

$$R_i = \frac{1}{\ln \frac{1}{p}} \underbrace{\ln\Big(1+\ln\Big(1+\ln\Big(1+\ldots\ln\Big(1+}_{(k-i)\ times}\ln\Big(C\ln\frac{1}{p}\Big)\Big)\ldots\Big)\Big)\Big)$$

with a total expected cost of $E = R_1 + \frac{1}{\ln(1/p)}$.

Theorem 2 generalizes to the case with monotonically increasing rejection probabilities.

▶ **Theorem 3.** *Let* κ *be the largest value in* $\{1,..,k\}$ *such that* $C \geq \frac{1}{\ln\frac{1}{p_\kappa}}$ *(with* $\kappa = 0$ *if* $C < \frac{1}{\ln\frac{1}{p_1}}$*). Then, when rejection probabilities are monotonically increasing, the optimal* $(k+1)$*-round strategy for sending real requests is to send* R_i *requests in round* i *where,*

$$\begin{cases} R_{\kappa+1} = \ldots = R_k = 0 \ , \\ R_\kappa = \dfrac{1}{\ln\frac{1}{p_\kappa}} \ln\left(C\ln\dfrac{1}{p_\kappa}\right), \\ R_i = \frac{1}{\ln\frac{1}{p_i}} \ln\left(E^{\mathrm{OPT}}_{i+1}\ln\frac{1}{p_i}\right) \ \text{where}\ E^{\mathrm{OPT}}_i = R_i + \dfrac{1}{\ln\frac{1}{p_i}} \ \text{for}\ 1 \leq i < \kappa\ . \end{cases}$$

The total expected cost $E = C$ *if* $C < \frac{1}{\ln\frac{1}{p_1}}$*. Otherwise,* $E = E^{\mathrm{OPT}}_1 = R_1 + 1/\ln(1/p_1)$*, where*

$$R_1 = \frac{1}{\ln\frac{1}{p_1}} \underbrace{\ln\left(\frac{\ln p_1}{\ln p_2}\left(1+\ln\left(\frac{\ln p_2}{\ln p_3}\left(1+\ldots\ln\left(\frac{\ln p_{\kappa-1}}{\ln p_\kappa}\left(1+\right.\right.\right.\right.\right.\right.}_{(\kappa-1)\ times}\ln\left(C\ln\frac{1}{p_\kappa}\right)\Big)\Big)\Big)\ldots\Big)\Big)\Big)\Big).$$

2.2 Optimal Integral Strategy

We use Theorem 3 to construct an algorithm to determine an optimal integral $(k+1)$-round strategy. We first calculate the real request R_i for round i, and then check whether rounding R_i up or down leads to a better expected review cost conditioned on having reached round i; see Algorithm 1. Then we work backwards to compute R_{i-1} through R_1.

▶ **Theorem 4.** *There exists a linear-time algorithm for finding an optimal* $(k+1)$*-round integral strategy for the case when the rejection probabilities are monotonically increasing.*

Algorithm 1: Computes an optimal $(k+1)$-round integral strategy.

1. $\kappa \leftarrow \max\left\{ i \in [1,k] \;\middle|\; C \geq \frac{1}{\ln\frac{1}{p_i}} \right\}$, or $\kappa \leftarrow 0$ if this set is empty
2. $R_i \leftarrow 0$ for all $i \in [1,k]$ `//` R_i `is the number of requests sent at round` i
3. $E \leftarrow C$ `//` E `is the current expected cost according to the values of` R_i
4. **for** *i from* κ *to* 1 **do**
5. $R_i \leftarrow \left\lfloor \ln_{\frac{1}{p_i}}(E\ln\frac{1}{p_i}) \right\rfloor$ `//` R_i `is rounded down by default`
6.   **if** $p_i^{R_i}E > 1 + p_i^{R_i+1}E$ **then** $R_i \leftarrow R_i + 1$ `// Round up` R_i `if beneficial`
7. $E \leftarrow R_i + p_i^{R_i}E$ `// Update` E `for the next iteration`
8. In each round i, send R_i requests. The expected cost is E.

Algorithm 2: Optimal strategy when the rejection probabilities are constant and there is no bound on the total number of rounds.

```
1  if C < 1/(1−p) then PC member reviews on their own (and does not send any request)
2  if C > 1/(1−p) then
3      while no request accepted and the number of requests sent is below the limit do
4          PC member sends a single request
5      if no request accepted then PC member reviews the paper
6  if C = 1/(1−p) then
7      while no request accepted and the number of requests sent is below the limit do
8          PC member chooses either to send a single request or review the paper
9      if no request accepted then
10         PC member reviews the paper
```

3 Bounded Number of Requests

A PC member may know only a limited number of experts to ask for a subreview. Let R denote the budget of total number of requests a PC member is allowed to make. In this section, we construct strategies that make a total of at most R requests.

3.1 Constant Rejection Probabilities

We give an optimal algorithm for the case when the probability of rejection is constant across rounds, that is, $p_i = p$ for all i $(1 \le i \le k)$.

Optimal Strategy for Unbounded Rounds

We first consider the case when there is no limit on the number of rounds. Regardless of the bound R on total number of requests, there exists an optimal strategy that never sends more than one request per round; see Algorithm 2. This strategy is also optimal for the case when the number of rounds is limited to k and the budget on total requests $R \le k$.

▶ **Theorem 5.** *When a maximum of R requests can be made over unlimited rounds and the rejection probabilities are constant ($p_i = p$), then the optimal integral strategy is to*
- *self-review immediately if $C \le 1/(1-p)$, or*
- *make exactly one request per round if $C > 1/(1-p)$.*

The optimal expected cost is $\min(C, \frac{1}{1-p})$.

Optimal Algorithm for a Bounded Number of Rounds

We now devise a $(k+1)$-round algorithm for the case when we are only allowed to send R requests in total where $R > k$. (If $R \le k$, then we can use Theorem 5.)

For a given self-reviewing cost C, we determine the optimal integral strategy $\mathcal{S}^*(C)$ that is allowed to make an unbounded number of requests. Let the total number of requests sent by $\mathcal{S}^*(C)$ be R^*. If $R^* \le R$, our budget is sufficient and we execute strategy $\mathcal{S}^*(C)$. If $R^* > R$, there exists an optimal algorithm with budget R that uses all R requests; see Lemma 6. Thus, the challenge is to determine how to distribute $R < R^*$ requests across k rounds to minimize the total expected review cost.

▶ **Lemma 6.** *For a given self-reviewing cost C, let R^* be the total number of requests sent by an optimal integral strategy $\mathcal{S}^*(C)$ without a budget. When a budget of only R requests is allowed, where $R < R^*$, there exists an optimal strategy $\mathcal{S}^*_R(C)$ that sends exactly R requests.*

Let $R_1, R_2, \ldots, R_k$ be a partition of the budget R across the k rounds, that is, $\sum_{i=1}^n R_i = R$. Then the total expected cost is

$$E(R_1, \ldots, R_k) = R_1 + p^{R_1} R_2 + \ldots p^{R_1 + \cdots + R_{k-1}} R_k + p^R C. \tag{1}$$

Note that the final term, $p^R C$ (the expected self-reviewing cost incurred when all requests fail), stays the same regardless of how requests are partitioned among rounds. As long as C is sufficiently large, further increasing C does not change the optimal partition of R among the k rounds. On the other hand, if C is sufficiently small that the unbounded-request solution does not need more than R requests, we already have a solution – Theorem 4.

With this observation, our goal is to find a self-reviewing cost C_R that uses exactly R requests. The partition determined by this solution is exactly what we are looking for.

The proof proceeds as follows. First, we assume that such a C_R exists and can be found efficiently. Given this assumption, we show that our algorithm computes an optimal strategy. Then, we complete the proof, that is, we show that an appropriate C_R always exists and can be computed efficiently.

▶ **Theorem 7.** *There exists an algorithm that computes an optimal $(k+1)$-round strategy for the case when the total number of requests allowed is bounded by R.*

Algorithm 3: Optimal strategy when the rejection probabilities are constant and when requests and rounds are limited.

```
1 S*(C) ← an optimal strategy for k+1 rounds and unlimited requests (use Algorithm 1)
2 R* ← number of requests required by S*(C)
3 if R* ≤ R then follow strategy S*(C)
4 else
5   | C_R ← self-reviewing cost for which an optimal strategy sends R requests
6   | S*(C_R) ← strategy computed using Algorithm 1 for cost C_R sending R requests
7   | follow strategy S*(C_R)
```

At first glance, it may seem immediate that such a C_R exists and can be found (perhaps using a binary search). However, the following example illustrates two complications. First, there may be several optimal strategies, each making different total numbers of requests. Second, arbitrarily small perturbations in C_R may change the optimal number of requests—in particular, there may only be a single value C_R that leads to exactly R requests.

▶ **Remark 8.** An example configuration where several strategies are optimal: let $p = 1/2$, $C = 8$ and consider 2 rounds plus the self-reviewing round. The strategies $(R_1, R_2) \in \{(1,2),(1,3),(2,2),(2,3)\}$ all achieve an expected cost of 3, and use between 3 and 5 requests.

In this example, $C = 8$ is the only value of C that uses exactly 4 requests. If $C = 8 - \varepsilon$, the optimal strategy uses 3 requests, and if $C = 8 + \varepsilon$ it uses 5.

Showing An Appropriate Self-Reviewing Cost Exists. If a strategy is allowed to make nonintegral requests, then showing that an appropriate self-review cost C_R exists is straightforward. The total number of requests as a function of the self-review cost C for nonintegral strategies is continuous and increasing; see Theorem 2. Thus, such a C_R always exists.

Now we show that an appropriate C_R exists even for integral strategies. We begin with a structural remark.

▶ **Lemma 9.** *When the self reviewing cost increases, the optimal expected cost does not decrease.*

The next two structural lemmas focus on a single round i.

First, we show that when the self-reviewing cost is increased, the number of requests sent in i (in any optimal solution) cannot decrease. For nonintegral requests, this statement follows directly from Theorem 2. The subtlety here is that the optimal strategy for integer requests, as described in Algorithm 1, performs a rounding to compute each value R_i. We prove that our result still holds after this rounding.

▶ **Lemma 10.** *Given two integral strategies $\mathcal{S}_C$ and $\mathcal{S}_D$ that are optimal for a self-reviewing cost of C and D respectively, with $C < D$, the number of requests sent in any round i by $\mathcal{S}_C$ is not larger than the number of requests sent in round i by $\mathcal{S}_D$.*

Proof. Let $R_i^*(x)$ denote the (not necessarily integral) optimal number of requests sent at round i with self-reviewing cost x. We know by Theorem 3 that $R_i^*(x)$ is monotonically increasing in x; in particular, $R_i^*(C) \leq R_i^*(D)$.

We show the result by induction on i.

First, for $i = k$, we proceed by contradiction. Assume to obtain a contradiction that $\mathcal{S}_C$ (resp. $\mathcal{S}_D$) sends A requests (resp. B) at round k, with $A > B$. By definition, either $A = \lfloor R_i^*(C) \rfloor$ or $A = \lceil R_i^*(C) \rceil$; similarly $B = \lfloor R_i^*(D) \rfloor$ or $B = \lceil R_i^*(D) \rceil$. Since $R_i^*(C) \leq R_i^*(D)$ and $A > B$, we must have $A = B + 1$. Recall that, as proved in Theorem 3, $R_k^*(\alpha)$ is equal to the value of x that minimizes $x + p_k^x \alpha$.

Then, due to the optimality of A and B with a self-reviewing cost of C and D, respectively, we have (first substituting $A = B + 1$, then rearranging):

$$A + p_k^A C \leq B + p_k^B C \quad \Leftrightarrow \quad 1 + p_k p_k^B C \leq p_k^B C \quad \Leftrightarrow \quad C \geq \frac{1}{p_k^B (1 - p_k)}.$$

Similarly,

$$B + p_k^B D \leq A + p_k^A D \quad \Leftrightarrow \quad p_k^B D \leq 1 + p_k p_k^B D \quad \Leftrightarrow \quad D \leq \frac{1}{p_k^B (1 - p_k)}.$$

Since by definition we have $C < D$, we obtain $D \leq C < D$; hence, a contradiction.

Now assume this result holds true for any j larger than some i; we will show that it also holds for i. As proved in Theorem 3, $R_i^*(C)$ minimizes $x + p_i^x E_{i+1}$, where E_{i+1} is the optimal expected cost for the rounds after round i. When the self-reviewing cost increases, we know by Lemma 9 that E_{i+1} is nondecreasing. Then, we can apply the same reasoning as above and complete the induction. ◀

Now we show that as C increases, the requests sent at i increase continuously as well.

▶ **Lemma 11.** *Given a round i and an integer q, there exists a self-reviewing cost for which an optimal integral strategy sends exactly q requests at round i.*

Proof. We begin with some definitions. We define $\bar{E}_i(C)$ as the optimal expected cost, with integral numbers of requests, generated by rounds i through $k + 1$, conditioned on reaching round i (i.e., all requests at rounds 1 through $i - 1$ failed). To begin, we let $\bar{E}_{k+1}(C) = C$ to account for the self-reviewing round. We then recursively define $E_i(x, C) = x + p_i^x \bar{E}_{i+1}(C)$

as the expected cost generated by rounds i through $k+1$ (conditioned on reaching round i) when x requests are sent at round i. Therefore, we have $\bar{E}_i(C) = \min_{x\in\mathbb{N}} E_i(x,C)$. We finally define $R_i(C) = \arg\min_{x\in\mathbb{N}} E_i(x,C)$: this is the smallest number of requests that can be sent at round i while retaining optimality.

We want to show that $R_i(C)$ spans all integers when C varies. Recall from Theorem 3 that the real value of x minimizing E_i is equal to $R_i^*(C) = \ln_{\frac{1}{p_i}}\left(\bar{E}_{i+1}(C)\ln\frac{1}{p_i}\right)$. Then if $\bar{E}_{i+1}$ is a continuous and strictly increasing function of C, R_i^* is also a continuous and strictly increasing function of C, and R_i^* spans all integers (for all $q\in\mathbb{N}$ there exists a value C_q such that $R_i^*(C_q) = q$). From Theorem 4, $R_i(C) \in \{\lfloor R_i^*(C)\rfloor, \lceil R_i^*(C)\rceil\}$. Thus, since R_i^* spans all integers, R_i spans all integers as well ($R_i(C_q) = q$).

Therefore, it suffices to show that $\bar{E}_i(C)$ is a continuous and (strictly) increasing function of C. We prove this property by induction on i from $k+1$ to 1. We begin with the base case $i = k+1$. Then $\bar{E}_{k+1}(C) = C$, which is continuous and (strictly) increasing.

Suppose now that the inductive hypothesis holds for all $j > i$: $\bar{E}_j(C)$ is continuous and strictly increasing and, thus, R_j spans all integers. We must have $\bar{E}_i(C) \le \bar{E}_{k+1}(C) = C$, therefore, $R_i(C) \le C$ and $\bar{E}_i(C) = \min_{0\le x\le C}(x + p_i^x \bar{E}_{i+1}(C))$. Then, $\bar{E}_i(C)$ is a continuous and strictly increasing function. ◀

With this structure in mind, we are ready to prove the existence of C_R.

▶ **Theorem 12.** *Given an integer number of requests R, there exists a self reviewing cost $C_R \in \mathbb{R}$ such that an optimal strategy uses R requests.*

Proof. As the self-reviewing cost increases from 1 to infinity, each R_i spans all the integers in increasing order, see Lemmas 10 and 11. Therefore, if C increases by a sufficiently small ε, the number of requests in each round either 1) stays the same, or 2) increases by 1. We refer to these increases as *jumps*.

If only one R_i jumps for each small increase in C, then the theorem is proved. However, it may be that for all $\varepsilon > 0$, there may be two rounds R_i and $R_{i'}$ that both jump. See Remark 8 for an example. The remainder of the proof handles this case.

Because the expectations E_i^* (see Lemma 11) are continuous and strictly increasing, for each value S of R_i, there exists a unique value C such that both $R_i = S$ and $R_i = S+1$ are optimal. In other words, when R_i jumps, there is a value C such that both values of R_i are optimal.

For a self-reviewing cost C, let $R(C)$ be the minimum number of requests sent in an optimal solution.

Let C be the largest self-reviewing cost such that $R(C) \le R$, and let $\mathcal{S}$ be an optimal strategy using $R(C)$ requests. If $R = R(C)$, we have the result. Otherwise, we let $\delta = \lim_{\varepsilon\to 0^+} R(C+\varepsilon) - R(C)$ be the minimum number of jumps when C increases by any ε. By definition of δ, we have $\delta \ge R - R(C)$.

Recall that at the point where R_i jumps, both R_i and $R_i + 1$ lead to the same expected cost. Then, there exist δ rounds in which we can add a request in $\mathcal{S}$ without modifying the expected cost. Choose $R - R(C)$ of these rounds, and increment the number of requests. We have shown that this retains the (optimal) total expected cost. Thus for any R, we get a C that has an optimal strategy with exactly R requests. ◀

Computing the Appropriate Self-Reviewing Cost. The main idea behind our algorithm for finding C_R is to use binary search. However, as mentioned in Remark 8, there may only be a single correct value of C_R. In fact, it may not be an integer, or even a rational number.

Algorithm 4: Computing a self-reviewing cost C_R given R.

```
1  L ← 1/(1−p); U ← C_0                          // Lower and upper bounds on C_R
2  while ∃i, R_i(U) − R_i(L) ≥ 2 do                          // binary search
3      x ← ½(L + U)
4      if R ∈ [R(x), R⁺(x)] then return x
5      else if R(x) < R then L ← x
6      else U ← x
7  while R⁺(L) < R do
8      I ← {i | R_i(U) = R_i(L) + 1}
       // Value of the expectation before the first discontinuity in [L, U]
9      E(R_1, ..., R_k, C) ← R_1 + p^{R_1} R_2 + ··· + p^{R_1+···+R_k} C
10     R_i ← R_i(L) for each i
11     foreach i ∈ I do
12         C_i ← solution of E(R_1, ..., R_i, ..., R_k, C) = E(R_1, ..., R_i + 1, ..., R_k, C)
       // The minimum value corresponds to the first discontinuity
13     L ← min_{i∈I} C_i
14 return L
```

Thus, once we are within 1 of the correct value of C_R, our algorithm jumps to the correct C_R more directly.

We begin with some definitions. We call a self reviewing cost C' a *point of discontinuity* at round i if there are two optimal strategies for cost C' that give a different number of requests in round i. (Note that these are similar to the "jumps" in the proof of Theorem 12.) Let $R_i(C)$ (resp. $R_i^+(C)$) be the minimum (resp. maximum) number of requests at round i that ensures optimality for a self-reviewing cost of C. Then we formally define $R(C)$ as the sum over all rounds of $R_i(C)$, and $R^+(C)$ likewise as the sum over $R_i^+(C)$. Note that these quantities can be computed by Theorem 7.

Our algorithm begins with a binary search, starting with the interval $[\frac{1}{1-p}, C]$, and ending on an interval $[L, U]$ which has $R_i(U) - R_i(L) \leq 1$ for all rounds i. In particular, we stop here because each round has at most one point of discontinuity in $[L, U]$.

Then, we refine the search within $[L, U]$. The algorithm finds, for each round i, the point of discontinuity within $[L, U]$ for round i; call it C_i. We then update L to be the smallest such C_i. We repeat until $R^+(L) = R$, at which point L is the desired value of C_R.

▶ **Theorem 13.** *Given a request target R, and a self-reviewing cost C (such that all optimal solutions for C require more than R requests) Algorithm 4 computes an appropriate self-reviewing cost C_R which has an optimal solution with exactly R requests.*

Proof. First, Theorem 12 ensures the existence of a self-reviewing cost C_R for which an optimal strategy with R requests exists.

By Lemma 10, we know that $R_i(C)$ and $R(C)$ are nondecreasing. Moreover, by Theorem 5, we know that $R(\frac{1}{1-p}) = 0$, and our theorem assumes that $R(C_0) > R$. Therefore, throughout the first loop of Algorithm 4, we know that there exists $C_R \in [L, U]$. Then, by Lemma 11, we know that each $R_i(C)$ spans every integer, so the first loop terminates. Thus, when the second loop is initiated, $[L, U]$ is in an interval containing C_R, and in which each round has at most one point of discontinuity.

Now, we study a given iteration of the second loop. Let j be the index of the round with the

first discontinuity in $[L, U]$. First, note that a round i encounters a discontinuity at the point C if and only if C meets the requirement of Line 12, where the optimal numbers of requests sent at each round for a self-reviewing cost of C are given by $R_1, \ldots, R_k$. Therefore, C_j is indeed the smallest point of discontinuity in $[L, U]$, i.e., we have $R(L) = R(C_j) < R^+(C_j)$, and for every other i, we have $R(L) = R(C_i)$. Thus, during the execution of the second loop, the value of L is increased without skipping any point of discontinuity. This proves the termination and the correctness of this loop.

Finally, we show that this algorithm is efficient, by bounding the number of iterations of the first loop. Note that if a round i has a discontinuity at a self-reviewing cost C_i^1, the next discontinuity C_i^2 satisfies $C_i^2 \geq C_i^1/p$, see below. Indeed, this inequality is actually an equality for the last round as the self-reviewing cost that has optimal strategies in which round k sends R_k or $R_k + 1$ requests is equal to $\frac{1}{p^{R_k}(1-p)}$. Then, as the previous rounds increase with a smaller rate, the inequality is proved.

Therefore, the first loop contains $O(\log_{1/p} C_0)$ iterations. ◀

3.2 Round-Dependent Rejection Probabilities

In this section we consider the case where rejection probabilities are not constant but are monotonically increasing.

The optimal strategy in Section 3.1 for constant rejection probabilities is based on the fact that when only an insufficient budget of R requests is available, then the optimal partition of R across the k rounds is independent of C. However, when the rejection probabilities are monotonically increasing this is no longer true. For rejection probabilities p_i for round i, the last term in the expression of expected review cost (Equation 1) becomes $p^{\Sigma_i R_i} C$ and thus the optimal review cost depends on the partition $R_1, \ldots, R_k$ of the budget R.

For the case of round-dependent rejection probabilities, we give a greedy strategy which achieves a approximation ratio dependent on the number of rounds. Roughly speaking, this strategy sends $R_i = \left\lfloor OPT \middle/ \prod_{j=1}^{i-1} p_j^{R_j} \right\rfloor$ requests in the ith round, finding the value of OPT through a binary search.

▶ **Theorem 14.** *There exists a greedy strategy whose expected cost is at most $1 + (k+1)OPT$ where OPT is the cost of the integral optimal strategy using at most R requests and k rounds.*

3.3 Multiple Papers Sharing a Common Request Budget

In this section we consider the case where the PC member can send a total of at most R requests and these requests have to be distributed among the N papers the PC member has been assigned to review. We first show that there exists a pseudo-polynomial-time algorithm to compute the optimal strategy for one paper using at most R requests. We then extend it to the optimal strategy for N papers, sharing a total budget R, with round-dependent rejection probabilities.

▶ **Theorem 15.** *There exists a pseudo-polynomial time algorithm computing an optimal strategy in $O(kR^2)$ time for any paper j such that the rejection probabilities with rounds are monotonically increasing and the total number of requests is bounded by R.*

Proof. The algorithm uses dynamic programming to build a table $E_j[i, r]$, for $1 \leq i \leq k+1$ and $0 \leq r \leq R$. $E_j[i, r]$ is the optimal expected cost when exactly r requests are spread among the rounds $R_i, R_{i+1} \ldots R_k$. $E_j[i, r]$ is constructed using the following equation:

$$\forall i \in [1, k],\ \forall r \in [0, R], \quad E_j[i, r] = \min_{0 \leq x \leq r} \left(x + p_{i,j}^x E_j[i+1, r-x] \right).$$

The algorithm proceeds by decreasing the value of round i, starting at round $i = k+1$. The table E_j is initialized by setting $E_j[k+1, r] = C_j$ for all r. For each round and each value of r, the algorithm has a cost of $O(R)$ to update the table E_j; hence, the overall complexity is $O(kR^2)$. The space used is $O(kR)$. The optimal strategy is determined by finding the value $\mathcal{R}$ that minimizes $E_j[1, \mathcal{R}]$ (for $0 \leq \mathcal{R} \leq R$). ◀

We use this result to build a solution for the general case for a total of N papers.

▶ **Theorem 16.** *There exists a pseudo-polynomial time algorithm computing an optimal strategy in $O(NkR^2)$ time for sending subreview requests for a total of N different papers such that the rejection probabilities are monotonically increasing with rounds and the total number of requests across the N papers is bounded by R.*

Proof. The algorithm uses dynamic programming to build a table $\mathbb{E}[j, r]$, for $0 \leq j \leq N$ and $0 \leq r \leq R$. $\mathbb{E}[j, r]$ is the optimal expected cost when exactly r requests are spread among the first j papers. $\mathbb{E}[j, r]$ is constructed using the following equation:

$$\forall r \in [0, R], \ \mathbb{E}[j, r] = \min_{0 \leq x \leq r} \left(E_j[1, x] + \mathbb{E}[j-1, r-x] \right),$$

where the value of $E_j[1, x]$ is given by Theorem 15. The algorithm proceeds by increasing values of j, from 1 to N. The overall complexity is $O(NkR^2)$. ◀

4 Unknown Probability of Rejection

In this section we consider the case in which the probability that a subreviewer rejects a subreview request is constant but unknown to the PC member.

We begin with an observation. Suppose we have an unbounded number of rounds for sending requests. Our result for unbounded rounds (see Algorithm 2) states that an optimal algorithm either takes the cost of C immediately, or performs one request per round indefinitely. Even when we do not know the value of p, we can mirror this idea using an approach similar to the classic *ski-rental problem* (see e.g. [12]).

▶ **Observation 17.** Consider a strategy $\mathcal{S}$ that makes one subreview request for C rounds then reviews for a self-reviewing cost C. Then, $\mathcal{S}$ is a 2-approximation (regardless of the value of the rejection probability p).

Proof. Strategy $\mathcal{S}$ has cost at most $2C$. If $C \leq 1/(1-p)$, the optimal strategy is to self-review at cost C and $\mathcal{S}$ is a 2-approximation. The cost of $\mathcal{S}$ is bounded by $1+p+\ldots+p^{C-1}+p^C C \leq 1/(1-p) + p^C C$. Note that $p^C C = p^C + p^C + \ldots p^C \leq 1 + p^2 + \ldots p^C \leq 1/(1-p)$. Thus the total cost is at most $2/(1-p)$. So for $C > 1/(1-p)$, $\mathcal{S}$ is also a 2-approximation. ◀

Now we consider the case when the total number of rounds used to make subreview requests is bounded. We first give a 2-round strategy (one round of making requests, followed by a self-review if all requests failed) and then generalize to a $(k+1)$-round strategy.

Two Rounds. We start with a 2-round strategy that achieves an approximation ratio of $(2\sqrt{C}+2)$. Then we improve the approximation ratio asymptotically; see Theorem 19.

To analyze our strategies we prove the following structural lemma. Roughly speaking, it shows that as long as the optimal cost is not too big, the cost of sending subreview requests is greater than the expected self-reviewing cost.

▶ **Lemma 18.** *Assume the cost of the optimal nonintegral strategy for two rounds with self-reviewing cost C is $\mathrm{OPT} \le \sqrt{C \ln C}$. Then if OPT sends R requests, $p^R C \le R$.*

Proof. By contradiction. Note that $\mathrm{OPT} \le \sqrt{C \ln C}$ implies that $\mathrm{OPT} < C$ since $C \ge 1$. Then by Theorem 2, we have $R = \log_{\frac{1}{p}}(C \ln \frac{1}{p})$, and $p^R C = \frac{1}{\ln \frac{1}{p}}$. With some algebra, if the lemma statement is false, then we must have $\ln \frac{1}{p} < e/C$. Note that $\ln(Cx)/x$ is minimized when x is maximized; thus $\ln \frac{1}{p} = e/C$ is a lower bound on the cost of OPT. Replacing back into the cost of OPT, we have

$$\sqrt{C \ln C} \ge \mathrm{OPT} = \frac{\ln C \ln \frac{1}{p}}{\ln \frac{1}{p}} + \frac{1}{\ln \frac{1}{p}} \ge \frac{2C}{e}.$$

Note that $\sqrt{C \ln C}/C$ is maximized at $C = e$ (for $C > 1$), at which point $\sqrt{C \ln C} =< 2C/e$. Thus $\sqrt{C \ln C} < 2C/e$ for any $C > 1$. ◀

Consider the 2-round strategy $\mathcal{S}$ that sends $\left\lceil \sqrt{C} \right\rceil$ requests in the subreview round. If all requests fail, a self-reviewing cost C is incurred in the next round.

If the optimal strategy sends more than $\sqrt{C}$ requests, or sends zero requests and incurs self-reviewing cost C, then $\mathcal{S}$ achieves an approximation ratio of $\sqrt{C} + 2$ since $\mathcal{S}$ has total review cost at most $\left\lceil \sqrt{C} \right\rceil + C$. If the optimal strategy sends $R < \sqrt{C}$ requests, then $p^R C \le R < \sqrt{C}$ by Lemma 18, and we have $p^{\sqrt{C}} C < \sqrt{C}$. Thus, $\mathcal{S}$ has total review cost at most $2\left\lceil \sqrt{C} \right\rceil$ and the optimal strategy has cost at least 1.

Next, we improve the approximation ratio. Note that for small C ($C \le e^4 < 55$), the warm-up strategy leads to better guarantees.

▶ **Theorem 19.** *Consider a 2-round strategy $\mathcal{S}$ that sends $\left\lceil \sqrt{C/\ln C} \right\rceil$ requests. This strategy is a $(4\sqrt{C/\ln C} + 2)$-approximation for any $C \ge 2$.*

Proof. We compare our algorithm's performance to an optimal strategy $\mathcal{S}^*$. To simplify analysis we do not require $\mathcal{S}^*$ to be integral; however, we do require that $\mathcal{S}^*$ either sends zero requests, or sends at least one request. This lower bounds the cost of an optimal integral strategy. Let OPT be the cost of $\mathcal{S}^*$. We divide the proof into several cases based on the value of OPT.

Case 1 (Large OPT): Assume $\mathrm{OPT} \ge \sqrt{C \ln C}$. Since $\mathcal{S}$ has cost at most $\left\lceil \sqrt{C/\ln C} \right\rceil + C$, the approximation ratio is at most

$$\frac{\left\lceil \sqrt{C/\ln C} \right\rceil + C}{\sqrt{C \ln C}} \le \frac{\sqrt{C/\ln C} + 1 + C}{\sqrt{C \ln C}} \le \frac{1}{\ln C} + \frac{1}{\sqrt{C \ln C}} + \frac{\sqrt{C}}{\sqrt{\ln C}} \le \frac{2\sqrt{C}}{\sqrt{\ln C}} + 1 \text{ for } C > 2.$$

Thus, in this case $\mathcal{S}$ is $(2\sqrt{C/\ln C} + 1)$-competitive. Note that $\sqrt{C/\ln C} > 1$ for all $C > 1$ and $1/\sqrt{C \ln C} < 1$ when $C^C > e$, which holds for all $C \ge 2$.

Case 2 (Small OPT): Assume $\mathrm{OPT} \le \sqrt{C/\ln C}$. If $\mathcal{S}^*$ sends R requests, its cost is $R + p^R C$. Since $R \le \mathrm{OPT} \le \sqrt{C/\ln C}$, $p^R \ge p^{\sqrt{C/\ln C}}$. Note that $p^R C \le \mathrm{OPT}$. Then the cost of our algorithm is $\left\lceil \sqrt{C/\ln C} \right\rceil + p^{\sqrt{C/\ln C}} C \le \left\lceil \sqrt{C/\ln C} \right\rceil + \mathrm{OPT}$. Since the optimal algorithm has cost at least 1, this results in a $\left(2\sqrt{C/\ln C} + 1\right)$-approximation algorithm.

Case 3 (Medium OPT): Assume that $\mathcal{S}^*$ has cost OPT, where $\sqrt{C/\ln C} < \text{OPT} < \sqrt{C \ln C}$. By Lemma 18, if $\mathcal{S}^*$ makes R requests, then $p^R C \le R$. Rearranging, we obtain $p \le e^{(\ln R)/R - (\ln C)/R}$; taking the derivative with respect to R we get $(R/C)^{1/R}((\ln C)/R^2 + (1 - \ln R)/R^2) \ge 0$ since $R \le C$. Thus $(R/C)^{1/R}$ is increasing with respect to R; since $R < \text{OPT}$, we obtain $p \le (R/C)^{1/R} < (\text{OPT}/C)^{1/\text{OPT}}$.

Substituting, the approximation ratio is at most

$$\frac{\left\lceil\sqrt{C/\ln C}\right\rceil + p^{\left\lceil\sqrt{C/\ln C}\right\rceil} C}{\text{OPT}} \le \frac{1 + \sqrt{\frac{C}{\ln C}} + \left(\frac{\text{OPT}}{C}\right)^{\frac{\sqrt{C/\ln C}}{\text{OPT}}} C}{\text{OPT}} \le 2 + \left(\frac{C}{\text{OPT}}\right)^{1 - \frac{\sqrt{C/\ln C}}{\text{OPT}}}.$$

Let $x = \sqrt{C/\ln C}/\text{OPT}$, so $x \in (1/\ln C, 1)$. Then the ratio is

$$2 + \left(x\sqrt{C \ln C}\right)^{1-x} = 2 + e^{(1-x)\ln(x\sqrt{C \ln C})}$$

Taking the derivative, we get

$$e^{(1-x)\ln(x\sqrt{C \ln C})}\left(\frac{1-x}{x} - \ln(x\sqrt{C \ln C})\right).$$

This is equal to zero only when $1/x = \ln(x\sqrt{C \ln C}) + 1$. Substituting into the original equation, we are bounded by:

$$2 + \left(x\sqrt{C \ln C}\right)^{1-x} = 2 + \left(\frac{\sqrt{C \ln C}}{\ln(x\sqrt{C \ln C}) + 1}\right)^{1-x} \le 2 + \left(\frac{\sqrt{C \ln C}}{\ln\sqrt{C/\ln C}}\right)^{1-x}.$$

Note that $\ln C \le C^{1/e}$ for all C. Then $\sqrt{C/\ln C} \ge C^{1/4}$, so

$$2 + \left(\frac{\sqrt{C \ln C}}{\ln\sqrt{C/\ln C}}\right)^{1-x} \le 2 + 4\sqrt{\frac{C}{\ln C}}.$$

Finally, we test the boundary cases. When $x = 1$, $2 + \left(x\sqrt{C \ln C}\right)^{1-x} = 2 < 2 + 4\sqrt{C/\ln C}$. When $x = 1/\ln C$, then $2 + \left(x\sqrt{C \ln C}\right)^{1-x} \le 2 + \sqrt{C/\ln C}$. Thus, the maximum is reached when the derivative is zero and we have the theorem. ◀

Multiple Rounds. We generalize to k rounds and give a $kC^{1/k}$-competitive algorithm. We begin with a useful lemma.

▶ **Lemma 20.** *Assume the optimal nonintegral algorithm has total expected cost* $\text{OPT} < C$. *Then* $p^i i \le \text{OPT}$ *for all* $i \ge \text{OPT}$.

Proof. The lemma is trivially satisfied if $i = \text{OPT}$, since $p^{\text{OPT}}\text{OPT} \le \text{OPT}$. Secondly, note that by Theorem 2, $\text{OPT} \ge 1/\ln\frac{1}{p}$.

Taking the derivative, we have

$$\frac{d}{di}p^i i = p^i\left(1 - i\ln\frac{1}{p}\right) < 0,$$

where the second inequality follows from the fact that $i > \text{OPT} \ge 1/\ln\frac{1}{p}$. Thus, $p^i i$ decreases as i increases. In other words, $p^i i \le p^{\text{OPT}}\text{OPT} \le \text{OPT}$. ◀

▶ **Theorem 21.** *Let $\mathcal{S}$ be a $(k+1)$-round strategy with unknown rejection probability p where:*

- *self-review immediately if $\lceil C^{(i-1)/k} \rceil > C$, or*
- *send $\lceil C^{(i-1)/k} \rceil$ requests at round i.*

Then $\mathcal{S}$ is a $k(C^{1/k}+1)$-approximation strategy.

Proof. If $\mathrm{OPT} = C$, the strategy $\mathcal{S}$ sends at most kC requests, and is k-competitive. Now assume $\mathrm{OPT} < C$. We define a variable a which roughly characterizes the performance of OPT in terms of $\mathcal{S}$. Let a be such that $\lceil C^{a/k} \rceil \leq \mathrm{OPT} < \lceil C^{(a+1)/k} \rceil$; in other words, $a+1$ is the last round where $\mathcal{S}$ makes fewer requests than the total cost of OPT. Note that $0 \leq a \leq k$. The cost of $\mathcal{S}$ in rounds 1 to a is bounded by

$$\sum_{i=1}^{a} \left\lceil C^{i/k} \right\rceil \leq a \left\lceil C^{a/k} \right\rceil \leq a \cdot \mathrm{OPT}.$$

Consider round $i \geq a+1$. The cost of $\mathcal{S}$ at round i is

$$p^{\sum_{j=1}^{i-1} C^{j/k}} \left\lceil C^{i/k} \right\rceil \leq p^{C^{i/k}} C^{i/k} C^{1/k} + p^{C^{i/k}} \leq \mathrm{OPT} \cdot C^{1/k} + 1,$$

where the final inequality follows from Lemma 20. Thus our algorithm has total cost at most $a \cdot \mathrm{OPT} + (k-a)(\mathrm{OPT} \cdot C^{1/k} + 1) \leq \mathrm{OPT}(kC^{1/k} + k)$. ◀

5 Conclusion

Many of our structural lemmas in this paper compartmentalize the subreviewer problem into independent tractable subproblems, which are much easier to analyze. For example, we can optimize the subreviewing for each paper independently, thanks to the comforting magic of linearity of expectation. Similarly, we can analyze the later rounds in the reviewing process independently of preceding rounds, thanks to the magic of doing probability theory correctly.

Natural open problems abound, generalizing the subreviewer optimization problem from this paper. But in many of these generalizations, this independence and comfortable compartmentalizing gets thrown out the window.

For example, an assiduous PC member may want to obtain *two* independent subreviews for some controversial submission. Now what happens in later rounds depends on previous rounds, and we cannot immediately apply the “work-backward method” given in Section 2.

Similarly, different papers may share potential subreviewers—a PC member may have ten experts to ask about paper 1 and twelve to ask about paper 2, but five of these experts are the same. Now optimizing the subreviewing for these two papers probably cannot be done independently. We also lose independence if we have a limited pool of subreviewers, and each subreviewer has a paper-dependent and time-dependent rejection probability.

References

1. Dan Alistarh, Michael A. Bender, Seth Gilbert, and Rachi Guerraoui. How to allocate tasks asynchronously. In *Proc. of the 53nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 331–340, 2012.
2. Chumki Basu, Haym Hirsh, William W. Cohen, and Craig Nevill-Manning. Recommending papers by mining the web. In *Proc. of the IJCAI Workshop on Learning about Users*, 1999.
3. Michael A. Bender and Cynthia A. Phillips. Scheduling DAGs on asynchronous processors. In *Proc. of the 19th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 35–45, 2007.

4 Salem Benferhat and Jérôme Lang. Conference paper assignment. *International Journal of Intelligent Systems*, 16(10):1183–1192, 2001.

5 George Bosilca, Aurélien Bouteiller, Élisabeth Brunet, Franck Cappello, Jack Dongarra, Amina Guermouche, Thomas Hérault, Yves Robert, Frédéric Vivien, and Dounia Zaidouni. Unified Model for Assessing Checkpointing Protocols at Extreme-Scale. *Concurrency and Computation: Practice and Experience*, 26(17):2727–2810, 2013.

6 Henri Casanova, Fanny Dufossé, Yves Robert, and Frédéric Vivien. Mapping applications on volatile resources. *International Journal of High Performance Computing Applications*, 29(1):19, 2015. doi:10.1177/1094342013518806.

7 Henri Casanova, Dounia Zaidouni, and Frédéric Vivien. Using replication for resilience on exascale systems. In Thomas Hérault and Yves Robert, editors, *Fault-Tolerance Techniques for High-Performance Computing*, page 50. Springer, 2015.

8 Bogdan S. Chlebus and Dariusz R. Kowalski. Cooperative asynchronous update of shared memory. In *Proc. of the 37th Annual ACM Symposium on Theory of Computing (STOC)*, pages 733–739, 2005.

9 Graham Cormode. How not to review a paper: The tools and techniques of the adversarial reviewer. *ACM SIGMOD Record*, 37(4):100–104, 2009.

10 John T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems*, 22(3):303–312, 2004.

11 Susan T. Dumais and Jakob Nielsen. Automating the assignment of submitted manuscripts to reviewers. In *Proc. of the 15th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 233–244, 1992.

12 Hiroshi Fujiwara and Kazuo Iwama. Average-case competitive analyses for ski-rental problems. *Algorithmica*, 42(1):95–107, 2005.

13 Judy Goldsmith and Robert H. Sloan. The AI conference paper assignment problem. In *Proc. of the AAAI Workshop on Preference Handling for Artificial Intelligence, Vancouver*, pages 53–57, 2007.

14 David Hartvigsen, Jerry C. Wei, and Richard Czuchlewski. The conference paper-reviewer assignment problem. *Decision Sciences*, 30(3):865–876, 1999.

15 Thomas Hérault and Yves Robert. *Fault-Tolerance Techniques for High-Performance Computing*. Springer, 2015. doi:10.1007/978-3-319-20943-2.

16 Seth Hettich and Michael J. Pazzani. Mining for proposal reviewers: lessons learned at the national science foundation. In *Proc. of the 12th Annual ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 862–871, 2006.

17 Z. M. Kedem, K. V. Palem, M. O. Rabin, and A. Raghunathan. Efficient program transformation for resilient parallel computation via randomization. In *Proc. of the 24th Annual ACM Symposium on the Theory of Computing (STOC)*, pages 306–317, 1992.

18 Zvi M. Kedem, Krishna V. Palem, and Paul G. Spirakis. Efficient robust parallel computations. In *Proc. of the 22rd Annual ACM Symposium on Theory of Computing (STOC)*, pages 138–148, 1990.

19 Dariusz R. Kowalski and Alexander A. Shvartsman. Writing-all deterministically and optimally using a nontrivial number of asynchronous processors. *ACM Transactions on Algorithms*, 4:33:1–33:22, 2008.

20 Julien Lesca and Patrice Perny. LP solvable models for multiagent fair allocation problems. In *Proc. of the 19th European Conference on Artificial Intelligence*, pages 393–398. IOS Press, 2010.

21 Xinlian Li and Toyohide Watanabe. Automatic paper-to-reviewer assignment, based on the matching degree of the reviewers. *Procedia Computer Science*, 22:633–642, 2013.

22 Charles Martel and Ramesh Subramonian. On the complexity of certified write-all algorithms. *Journal of Algorithms*, 16:361–387, 1994.

23 Juan Julián Merelo-Guervós and Pedro Castillo-Valdivieso. Conference paper assignment using a combined greedy/evolutionary algorithm. In *Parallel Problem Solving from Nature-PPSN VIII*, pages 602–611, 2004.

24 David Mimno and Andrew McCallum. Expertise modeling for matching papers with reviewers. In *Proc. of the 13th Annual ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 500–509, 2007.

25 Jaroslaw Protasiewicz. A support system for selection of reviewers. In *IEEE International Conference on Systems, Man and Cybernetics (SMC)*, pages 3062–3065, 2014.

26 Fan Wang, Ben Chen, and Zhaowei Miao. A survey on reviewer assignment problem. In *New frontiers in applied artificial intelligence*, pages 718–727. Springer, 2008.

27 Fan Wang, Ning Shi, and Ben Chen. A comprehensive survey of the reviewer assignment problem. *International Journal of Information Technology & Decision Making*, 9(04):645–668, 2010.

28 David Yarowsky and Radu Florian. Taking the load off the conference chairs: towards a digital paper-routing assistant. In *Proc. of the Joint SIGDAT Conference on Empirical Methods in NLP and Very-Large Corpora*, 1999.

A Omitted Proofs

Proof of Lemma 1. The expected cost when sending a single round of R requests is $E(R) = R + p^R C = R + e^{R\ln p}C$. Taking the first and second derivatives, we obtain $E'(R) = 1 + (\ln p)p^R C$ and $E''(R) = (\ln p)^2 p^R C$. Thus, E'' is (strictly) positive, and E' is (strictly) increasing. Then, E' is zero when $R = \frac{1}{\ln\frac{1}{p}}\ln\left(C\ln\frac{1}{p}\right)$. If $C\ln\frac{1}{p} \leq 1$, then $R \leq 0$. Because R must be nonnegative and because E' is (strictly) increasing then in that case the optimal solution is to send no requests ($R = 0$) for an expected cost of C. Otherwise, the optimal expected cost is $R + p^R C = R + \frac{1}{\ln\frac{1}{p}} = \frac{1}{\ln\frac{1}{p}}\ln\left(C\ln\frac{1}{p}\right) + \frac{1}{\ln\frac{1}{p}}$. ◀

Proof of Theorem 2. The expression for the total expected review cost can be written as:

$$E(R_1,\ldots,R_k) = \left(R_1 + p^{R_1}R_2 + \cdots + p^{R_1+\cdots+R_{k-2}}R_{k-1}\right) \; + p^{R_1+\cdots+R_{k-1}}(R_k + p^{R_k}C).$$

Let $E_k(R_k) = R_k + p^{R_k}C$. Using Lemma 1, find the optimal value of R_k and E_k.

If $C \leq \frac{1}{\ln\frac{1}{p}}$, then the optimal value for R_k is 0 and thus all the R_is must be 0 and the optimal strategy is to self-review immediately.

For $C > \frac{1}{\ln\frac{1}{p}}$, we do an induction on the round i, starting from $i = k$. The optimal value of R_k is the base case. That is,

$$R_k = \frac{1}{\ln\frac{1}{p}}\ln\left(C\ln\frac{1}{p}\right) \quad \text{and} \quad E_k^{\text{OPT}} = \frac{1}{\ln\frac{1}{p}}\left(C\ln\frac{1}{p}\right) + \frac{1}{\ln\frac{1}{p}}.$$

Let E_j denote the expected cost conditioned on reaching round j, then

$$E_j(R_j,...,R_k) = R_j + p^{R_j}R_{j+1} + \cdots + p^{R_j+\cdots+R_{k-1}}R_k + p^{R_j+\cdots+R_k}C.$$

Suppose we have determined the optimal values of R_{i+1} to R_k for some value of i. And suppose that for $i+1 \leq j \leq k$, the expected cost $E_j(R_j,...,R_k)$ of an optimal solution is equal to $E_j^{\text{OPT}} = R_j + \frac{1}{\ln\frac{1}{p}}$. Now consider R_i. We rewrite the expression of the overall expected cost:

$$\begin{aligned} E(R_1,\ldots,R_k) = &\left(R_1 + \cdots + p^{R_1+\cdots+R_{i-2}}R_{i-1}\right) \\ &+ p^{R_1+\cdots+R_{i-1}}\left(R_i + p^{R_i}\left(R_{i+1} + \cdots + p^{R_{i+1}+\cdots+R_{k-1}}R_k + p^{R_{i+1}+\cdots+R_k}C\right)\right). \end{aligned}$$

$$= \left(R_1 + \cdots + p^{R_1+\cdots+R_{i-2}} R_{i-1}\right) + p^{R_1+\cdots+R_{i-1}} \left(R_i + p^{R_i} E_{i+1}^{\mathrm{OPT}}\right).$$

Thus, the optimal value for R_i does not depend on the values of R_1 to R_{i-1} and can be determined by Lemma 1 when substituting E_{i+1}^{OPT} to C. That is,

$$R_i = \frac{1}{\ln \frac{1}{p}} \ln \left(E_{i+1}^{Opt} \ln \frac{1}{p}\right) = \frac{1}{\ln \frac{1}{p}} \ln \left(\left(R_{i+1} + \frac{1}{\ln \frac{1}{p}}\right) \ln \frac{1}{p}\right) = \frac{1}{\ln \frac{1}{p}} \ln \left(1 + \left(R_{i+1} \ln \frac{1}{p}\right)\right).$$

To complete the proof, we use $E_i^{\mathrm{OPT}} = R_i + p^{R_i} E_{i+1}^{\mathrm{OPT}} = R_i + \frac{1}{\ln \frac{1}{p}}$ from Lemma 1. Finally, note that the total expected cost of the strategy is $E = E_1^{\mathrm{OPT}}$. ◀

Proof of Theorem 3. We prove the result by induction on i, starting from $i = k$. The expression for the total expected cost can be written as:

$$E(R_1, \ldots, R_k) = \left(R_1 + p_1^{R_1} R_2 + \cdots + \left(\prod_{j=1}^{k-2} p_j^{R_j}\right) R_{k-1}\right) + \left(\prod_{j=1}^{k-1} p_j^{R_j}\right) \left(R_k + p_k^{R_k} C\right).$$

Let $E_k(R_k) = R_k + p_k^{R_k} C$. The optimal value for R_k and E_k from Lemma 1 is:

$$\begin{cases} R_k = 0 \text{ and } E_k = C & \text{if } C < \frac{1}{\ln \frac{1}{p_k}} \\ R_k = \frac{1}{\ln \frac{1}{p_k}} \ln \left(C \ln \frac{1}{p_k}\right) \text{ and } E_k = E_k^{Opt} = R_k + \frac{1}{\ln \frac{1}{p_k}}, & \text{otherwise.} \end{cases}$$

If $C \leq 1/(\ln \frac{1}{p_k})$, then $R_k = 0$ and using induction the theorem follows for the rounds κ through k (recall that the p_i's are non-decreasing).

Let E_j denote the expected cost conditioned on reaching round j. Then

$$E_j(R_j, ..., R_k) = R_j + p_j^{R_j} R_{j+1} + \cdots + \left(\prod_{l=j}^{k-1} p_l^{R_l}\right) R_k + \left(\prod_{l=j}^{k} p_l^{R_l}\right) C.$$

Similar to the proof of Theorem 2, assume that we have optimal values of R_l and E_l, for $i + 1 \leq l \leq \kappa$ (and, thus, $R_{\kappa+1} = ... = R_k = 0$). Consider R_i. Rewriting the expression of the total expected cost we have:

$$\begin{aligned} E(R_1, \ldots, R_k) = & \left(R_1 + \cdots + \left(\prod_{j=1}^{i-2} p_j^{R_j}\right) R_{i-1}\right) \\ & + \prod_{j=1}^{i-1} p_j^{R_j} \left(R_i + p_i^{R_i} \left(R_{i+1} + \cdots + \left(\prod_{j=i+1}^{k-1} p_j^{R_j}\right) R_k + \left(\prod_{j=i+1}^{k} p_j^{R_j}\right) C\right)\right) \\ & = R_1 + \cdots + \left(\prod_{j=1}^{i-2} p_j^{R_j}\right) R_{i-1} + \left(\prod_{j=1}^{i-1} p_j^{R_j}\right) \left(R_i + p_i^{R_i} E_{i+1}(R_{i+1}, ..., R_k)\right). \end{aligned}$$

Since the values of R_{i+1} to R_κ are optimal, and the optimal value for R_i can be determined using Lemma 1 by substituting E_{i+1}^{OPT} to C. Therefore, we obtain the optimal value of R_i:

$$\begin{aligned} R_i = \frac{1}{\ln \frac{1}{p_i}} \ln \left(E_{i+1}^{\mathrm{OPT}} \ln \frac{1}{p_i}\right) = \frac{1}{\ln \frac{1}{p_i}} \ln \left(\left(R_{i+1} + \frac{1}{\ln \frac{1}{p_{i+1}}}\right) \ln \frac{1}{p_i}\right) = \\ \frac{1}{\ln \frac{1}{p_i}} \ln \left(\frac{\ln p_i}{\ln p_{i+1}} + R_{i+1} \ln \frac{1}{p_i}\right). \end{aligned}$$

We have $E_i^{\text{OPT}} = R_i + p^{R_i} E_{i+1}^{\text{OPT}} = R_i + \frac{1}{\ln \frac{1}{p_i}}$ using Lemma 1. Finally, note that the total expected cost of the strategy is $E = E_1^{\text{OPT}}$. ◀

Proof of Theorem 4. For $i > \kappa$, an optimal solution has $R_i = 0$ as in Theorem 3.

We prove that Algorithm 1 is correct and that at the end of Step 7 E equals $\bar{E}_i$ (at iteration i), the optimal expected cost conditioned on reaching round i.

We perform induction on i starting at $i = \kappa$ similar to the proof of Theorem 3. Let $E_\kappa(R_\kappa) = R_\kappa + p_\kappa^{R_\kappa} C$. We know that the nonintegral optimal value of R_κ also minimizes E_κ, i.e., $R_\kappa^* = \ln_{\frac{1}{p_\kappa}}(C \ln \frac{1}{p_\kappa})$. As E_κ decreases then increases, the integral optimal value of R_κ is either $\lfloor R_\kappa^* \rfloor$ or $\lceil R_\kappa^* \rceil$ (or both). This proves the result for $i = \kappa$.

Now, we assume the induction hypothesis for each round $l \in [i+1, \kappa]$. Consider R_i. We rewrite the expression of E when R_l is assigned to its optimal value, for $i + 1 \leq l \leq k$:

$$E(R_1, \dots, R_i) = R_1 + \dots + \left(\prod_{j=1}^{i-2} p_j^{R_j} \right) R_{i-1} + \left(\prod_{j=1}^{i-1} p_j^{R_j} \right) \left(R_i + p_i^{R_i} \bar{E}_{i+1} \right).$$

Using Lemma 1 the nonintegral optimal value of R_i is $\ln_{(1/p_i)}(\bar{E}_{i+1} \ln \frac{1}{p_i})$. The integral optimal value of R_i is one of its closest integers. Thus by induction, E has the value of $\bar{E}_{i+1}$ at the of Step 7 Algorithm 1 (at iteration $i + 1$). This proves the correctness for round i. Then, since E is updated to $R_i + p_i^{R_i} \bar{E}_{i+1} = \bar{E}_i$, the induction is complete. ◀

Physical Zero-Knowledge Proofs for Akari, Takuzu, Kakuro and KenKen*

Xavier Bultel[1], Jannik Dreier[2], Jean-Guillaume Dumas[3], and Pascal Lafourcade[4]

1 **LIMOS, University Clermont Auvergne, Campus des Cézeaux, Aubière, France**
xavier.bultel@udamail.fr
2 **Université de Lorraine, Loria, UMR 7503, Vandoeuvre-lès-Nancy, France; and Inria, Villers-lès-Nancy, France; and CNRS, Loria, UMR 7503, Vandoeuvre-lès-Nancy, France**
jannik.dreier@loria.fr
3 **LJK, Université Grenoble Alpes, CNRS umr 5224, 51, av. des Mathématiques, BP53, 38041 Grenoble, France.** Jean-Guillaume.Dumas@imag.fr
4 **LIMOS, University Clermont Auvergne, Campus des Cézeaux, Aubière, France**
pascal.lafourcade@udamail.fr

Abstract

Akari, Takuzu, Kakuro and KenKen are logic games similar to Sudoku. In Akari, a labyrinth on a grid has to be lit by placing lanterns, respecting various constraints. In Takuzu a grid has to be filled with 0's and 1's, while respecting certain constraints. In Kakuro a grid has to be filled with numbers such that the sums per row and column match given values; similarly in KenKen a grid has to be filled with numbers such that in given areas the product, sum, difference or quotient equals a given value. We give physical algorithms to realize zero-knowledge proofs for these games which allow a player to show that he knows a solution without revealing it. These interactive proofs can be realized with simple office material as they only rely on cards and envelopes. Moreover, we formalize our algorithms and prove their security.

1998 ACM Subject Classification D.4.6 Security and Protection

Keywords and phrases Physical Cryptography, ZKP, Games, Akari, Kakuro, KenKen, Takuzu

Digital Object Identifier 10.4230/LIPIcs.FUN.2016.8

1 Introduction

Akira and Totoro, close friends and both great fans of logical games, decide to challenge each other on their favorite games. Akira is an expert in Akari, and Totoro is a specialist of Takuzu. Each one proposes a grid of his favorite game to the other one as a challenge. However, as both are extremely competitive, they choose grids that are so difficult that the other is not able to solve them, as they are less experienced in the other game. Totoro, working in security, immediately supposes that something went wrong, and wants a proof from Akira that his grid actually has a solution. Akira, feeling hurt by this distrust, directly asks Totoro the same question. Obviously both of them do not want to reveal the solution, as this would render the challenge pointless. So they decide to ask Ken, a common friend,

* This work was partially supported by "Digital trust" Chair from the University of Auvergne Foundation, by the HPAC project of the French Agence Nationale de la Recherche (ANR 11 BS02 013), and by the CNRS PEPS JCJC VESPA. The authors would like to thank the anonymous reviewers for their helpful comments.

© Xavier Bultel, Jannik Dreier, Jean-Guillaume Dumas, and Pascal Lafourcade;
licensed under Creative Commons License CC-BY

for help. However, Ken, a grand master of KenKen, has the same problem. He and his best friend Kakarotto exchanged challenges for KenKen and Kakuro, and want to show each other that they know the solution without revealing it. But then Kakarotto has the idea to ask Cobra, a computer scientist and cryptographer and common friend, for help. Cobra directly sees a solution: a zero knowledge proof of knowledge of the solution (ZKP). A ZKP is a protocol that allows a prover P to convince a verifier V that he knows some solution s to the instance $\mathcal{I}$ of a problem $\mathcal{P}$, without revealing any information about s. Such a protocol satisfies three properties[1]:

Completeness: If P knows s, then he is able to convince V.

Soundness: If P does not know s, then he is not able to convince V except with some *"small"* probability[2].

Zero-knowledge: V learns *nothing* about s except $\mathcal{I}$, *i.e.* there exists a probabilistic polynomial time algorithm $\texttt{Sim}(\mathcal{I})$ (called the simulator) such that outputs of the real protocol and outputs of $\texttt{Sim}(\mathcal{I})$ follow the same probability distribution.

Yet, ZKPs are usually executed by computers, which Totoro (a very skeptical security expert) does not trust. However, Cobra is still able to help them, by inventing a ZPK that relies only on physical objects such as cards and envelopes. In this paper, we present Cobra's solution to Akira's, Totoro's, Ken's and Kakarotto's dilemma.

Contributions: We construct physical ZKP for Akari, Kakuro, KenKen and Takuzu.

- For Akari, our ZKP construction uses special cards and envelopes. Moreover, the prover needs to interact with the verifier to construct the proof.
- For Takuzu, we propose an interactive construction using cards, paper and envelopes.
- For Kakuro, we use red and black cards and envelopes to implement an addition operation.
- For KenKen, we also rely on red and black cards and envelopes, but the interactive proof is more complex due to the different operations (sum, difference, product, quotient).

We also prove the security of our constructions.

Related Work: Sudoku, introduced under this name in 1986 by the Japanese puzzle company Nikoli, and similar games such as Akari, Takuzu, Kakuro and Ken-Ken have gained immense popularity in recent years. Following the success of Sudoku, generalizations such as Mojidoku which uses letters instead of digits, and other similar logic puzzles like Hitori, Masyu, Futoshiki, Hashiwokakero, or Nurikabe were developed. Many of them have been proved to be NP-complete [12, 5].

Interactive ZKPs were introduced by Goldwasser *et al.* [8], and it was then shown that for any NP-complete problem there exists an interactive ZKP protocol [7]. An extension by Ben-Or *et al.* [1] showed that every provable statement can be proved in zero-knowledge. They also showed that physical protocols using envelopes exist, yet their construction is – due to its generality – rather involved an often impractical for real problem instances. Proofs can also be non-interactive in the sense that the prover and verifier do not need to interact during the protocol [3]. For more background on ZKPs see for example [15].

As ZKPs have always been difficult to explain, there are works on how to explain the concepts to non experts, partly using physical protocols as illustrations. For example, in

---

[1] Moreover, if $\mathcal{P}$ is NP-complete, then the ZKP should be polynomial. Otherwise it might be easier to find a solution than proving that a solution is a correct solution, making the proof pointless.

[2] More precisely, we want a negligible probability, *i.e.*, the probability should be a function f of a security parameter $\mathfrak{K}$ (for example the number of repetitions of the protocol) such that f is negligible, that is for every polynomial P, there exists $n_0 > 0$ such that $\forall\ x > n_0, f(x) < 1/P(x)$.

their famous paper [18], Quisquater *et al.* propose "Ali Bababa's cave" as a tool to explain Zero-Knowledge Proof to kids. In [20], ZKP's are illustrated using a magician that can count the number of sand grains in a heap of sand. Naor *et al.* used the well-known "Where's Waldo?" cartoons to explain the concept of ZKP to kids, and also proposed an efficient physical protocol for the problem in [17].

In 2007, the same authors proposed a ZKP for Sudoku using cards [9], which partly inspired Cobra's solution in our paper. This was later extended for Hanjie [4].

As in [9], we here also assume an abstract *shuffle functionality* which is essentially an indistinguishable shuffle of a set of sealed envelopes or of face down cards. This functionality is necessary to prevent information leakage, and cannot be realized neither by the verifier nor the prover. The verifier cannot perform this action, as otherwise he could be perform a shuffle that is not random, and break zero-knowledge. Moreover, in a physical protocol the prover cannot perform this action either, as he might be able to tamper with the packets at this step, similar to a magician performing a card trick. A possible realization would be to rely on a trusted third party (Cobra for instance), or trying to ensure that the prover cannot mess with the cards.

Moreover, there is work on implementing cryptographic protocols using physical objects, for example in [16], where the authors use cards to perform multi-party computations, or in [6] where a physical secure auction protocol is proposed.

Outline: For all games, we first present the rules followed by our ZKP construction and then the security analysis. In Section 2, we present Akari, in Section 3, Takuzu, in Section 4 Kakuro, and in Section 5 KenKen. In Section 6, we conclude the paper.

2 Akari

Akari is a logic puzzle first published in 2001 by *Nikoli*, a Japanese games and logic puzzles publisher. It is also called *Light up*[3]. The goal of the game is to illuminate all the cells of a rectangular grid by placing some lights. The lights illuminate horizontally and vertically all adjacent white cells. The grid also contains black cells, which represent walls that the light cannot cross. Some black cells can contain an integer between 0 and 4. To illuminate all the cells, lights have to be added in the white cells according to the following constraints:

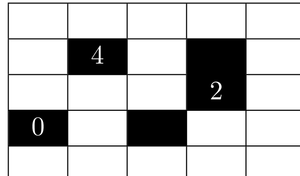
1. Two lights cannot illuminate each other.
2. The number in a black cell indicates how many lights are present in the adjacent white cells, *i.e.*, in the white cells directly above, below, left and right of the number.
3. All the white cells are illuminated by at least one light.

In Figure 1, we give a simple example of an Akari game. It is easy to verify that the three constraints are satisfied, but solving Akari was shown to be NP-complete *via* a reduction from Circuit-SAT [14].

2.1 ZKP Construction for Akari

Our aim is to give a ZKP proof such that Akira, the prover denoted by P, proves to Totoro, the verifier denoted by V, that he knows a solution of a given Akari grid G. We assume that the grid is printed on a sheet of paper, and that we can use a *shuffle functionality*. We use

[3] `https://en.wikipedia.org/wiki/Light_Up_(puzzle)`



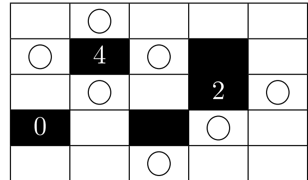


Figure 1 Example of an Akari grid and its solution.

2	2	2	2	2
2	4	2		2
2	2	2	2	2
0	2		2	2
2	2	2	2	2

	1			
1	4	1		
	1		2	
0				

1	1	1	1	1
1	4			
1			2	
0				

Figure 2 Illustrations for steps 1 to 3 in the Akari ZKP construction. From left to right: **Step 1:** two cards added on all white cells. **Step 2:** cards added around black cells with number. **Step 3:** cards added per cell as a function of adjacent cells.

two kinds of cards to model the fact that there is one light or not in one cell of the solution. The symbol ◯ represents the presence of a light, and an empty card the absence of a light.

Setup: According to his solution, the prover P places, face down on each white cell, ◯ cards when there is a light, and empty cards when there is none. Note that all the cards in the packet on the same cell are thus identical. Placing the cards happens in three phases:

1. In all white cells, P adds 2 cards face down (first illustration in Figure 2).
2. P adds an additional card for all white cells adjacent to a black cell that contains a number. For instance, in the second illustration in Figure 2, for the black cell containing the number 4, P adds one card in all positions with a grey background, and similarly for the black cells containing 0 and 2.
3. Finally, for each white cell in the grid, P places a card on the cell and all horizontally and vertically adjacent white cells. The third illustration in Figure 2 shows this for the top left cell: P adds a card to the cell itself, and to the six adjacent cells (all marked in grey, two below and four to the right). This has to be done for all white cells.
4. The same three steps are repeated once more to build a dual second grid, *i.e.*, a grid where all empty cards are replaced by light cards and *vice-versa*.

At the end of the algorithm, P has added multiple cards face down on each cell of the grid G and its dual grid $\hat{G}$.

Verification: To check the correctness of the solution, the verifier picks a random bit c in $\{0, 1\}$ and executes the following verifications:

if $c = 0$: For each cell of the grid G the verifier places the packet of cards in an envelope, then he does the same for each cell of grid $\hat{G}$. He shuffles the envelopes using the shuffle functionality, and then opens each envelope to check whether it contains only light cards or only empty cards. With this he is sure that all the cards on a given cell were identical.

if $c = 1$: The verifier discards all the cards on the dual grid $\hat{G}$ and checks whether the 3 constraints that a solution should satisfy are respected using the cards on the grid G:

1. **No two lights see each other:** For each row or column of adjacent white cells, V randomly picks one card per cell to form a packet. Then V passes the cards to the *shuffle functionality* who shuffles them, and passes them on to P. If all cards are empty cards then P adds a ◯ card, otherwise he adds an empty card. P returns the cards, once again shuffled, to V, who checks that there is exactly one light card in the packet.
2. **Black cells:** For each black cell that contains a number l, V picks one card in all adjacent cells and gives them to the shuffle functionality. He then looks at the cards and checks that the number l corresponds to the number of ◯ cards. In second illustration of Figure 2, a card from all grey cells has to be collected to verify the black cell containing the number 4.
3. **All cells are illuminated:** For each cell, V picks one card from the cell plus one card from all horizontally and vertically adjacent white cells (for example, for the top left cell in Figure 2, he takes one card from all grey cells). V passes all collected cards face down to P. If there are two light cards in the received cards, then P adds one empty card, otherwise he adds one ◯ card. P passes the cards to the shuffle functionality, and returns them to V. Then V opens the cards and checks that there is exactly two light cards in the packet.

This protocol is repeated $\mathfrak{K}$ times where $\mathfrak{K}$ is a chosen security parameter. Note that the number of cards to place on the grid and the number of verification steps is polynomial in the size of the grid, making our ZKP polynomial both for the prover and the verifier.

2.2 Security Proofs for Akari

We now prove the security of our construction.

▶ **Lemma 1** (Akari Completeness). *If P knows a solution of a given Akari grid, then he is able to convince V.*

Proof. If P knows the solution of an Akari grid, he can place the cards on the grids (G and $\hat{G}$) following our ZKP algorithm. Then on each cell of both grids there is a packet of identical cards, showing either nothing or a light, depending on whether there is a light in the solution. The verifier picks c among $\{0, 1\}$.

if $c = 0$: The verifier V puts each packet on each cell for the two grids into envelopes and shuffles them. Since all the packets on cells on both grids contain the same kind of card, the verifier accepts.

if $c = 1$: In the following, when taking a card from a cell, the verifier V randomly picks one card among all the cards in that cell. This has no effect when the prover is honest since all of them are identical. Using these cards, the verifier checks the three properties that a solution should satisfy.

- To check that two lights cannot see each other, for all lines and columns the verifier takes one card from each cell inside the line or column and asks the prover to add one extra card. The resulting selection is then shuffled. As the solution is correct, there can be either no light or one light, but not more, as otherwise two lights would see each other. If there is none, P adds one, otherwise he adds an empty card. Thus the cards contain exactly one light, which V will successfully verify.
- For all black cells that contain a number, the verifier picks one card in all adjacent white cells, uses the shuffle functionality on the selected cards, and then verifies that they contain the same number of lights as is written in the black cell. As P's solution is correct, the verification will succeed.

- To verify that each cell of the grid is illuminated by a light, for each cell the verifier picks one card from all white cells which are horizontally and vertically adjacent. He asks P to add exactly one card according to the algorithm and uses the shuffle functionality on the remaining cards. As the solution is correct, there will be either one or two lamps in the packet since each cell is illuminated, so after adding the right card there will be exactly two. Then V verifies that the set of cards contains exactly two lights, which will succeed. ◀

▶ **Lemma 2** (Akari Soundness). *If P does not know a solution of a given Akari's grid of size n then he is not able to convince V except with a negligible probability lower than $p = (\frac{1}{2})^{\mathfrak{K}}$ when the protocol is repeated $\mathfrak{K}$ times.*

Proof. We call a *cell-packet* the set of cards that are placed in one cell. When the verifier is collecting some cards during the verification phase for one cell, we also call a *row-packet* the set of cards that are collected in an horizontal line, a *column-packet* the set of cards that are collected in a vertical line and *cross-packet* the set of cards that are collected in horizontally and vertically adjacent white cells of a given cell.

Given a grid of size n, we show that if P is able to perform the proof for any challenge c, then he has a solution of the grid. This implies by contraposition that if the prover does not have a solution, then he fails the procedure for either $c = 0$, or $c = 1$, or both. In that case the probability that the verifier asks him a challenge for which he does not have a solution is at least $\frac{1}{2}$, which results in the probability of $p = (\frac{1}{2})^{\mathfrak{K}}$ as the protocol is repeated $\mathfrak{K}$ times.

Thus it suffices to show that if P is able to perform the proof for $c = 0$ and $c = 1$, then he has a solution of the grid.

If P is able to perform the proof for $c = 0$, then his commitment is well-formed, *i.e.*, all the cards in each cell-packet are the same.

If P is able to perform the proof for $c = 1$, then his solution passes all the verifier's checks. Moreover, we show that if the commitment of the prover is well-formed (which we know from the above), and this commitment does not correspond to a solution, then the verifier detects it with our ZKP algorithm. This implies that the prover has a solution to the grid, which is what we need to show to conclude the proof.

Suppose now that the commitment is well-formed, and that it does not correspond to a solution. Then we can distinguish three cases for the commitment, each corresponding to one constraint of the game that is not respected:

- There are two lights that can see each other. According to our ZKP, the verifier picks a row-packet or a column-packet for each row or column and passes it to the prover. The prover adds one card and gives it back to the verifier. Then the verifier checks that there is exactly one light in the packet. If there are two lights that can see each other, the prover cannot cheat and the verifier will see at least two lights.
- There is not the right number of lights close to a black cell containing a number. Then the verifier discovers that the number of lights does not match when he performs the second point of our ZKP algorithm.
- A cell is not illuminated. According to our algorithm, the verifier picks a cross-packet for this cell. Then he gives the packet to the prover who adds one extra card. Then the verifier checks that there are exactly two lights in the packet. If the cell is not illuminated, he will have at most one light in the packet and thus detect the error.

Thus, if the commitment is well-formed and does not correspond to a solution, the verifier will detect it. Hence, if the prover passes all the verifier's checks, he has a solution, which concludes the proof. ◀

Note that an optimal soundness of 0 can be obtained using a stronger model, namely the *triplicate functionality* of [9, § 4.2]. Instead of the challenge $c = 0$, one uses cards that can be cut in k equal parts (a "*k-plicate functionality*"), thus guaranteeing identical parts.

▶ **Lemma 3** (Akari Zero-Knowledge). *V learns nothing about P's solution of a given Akari grid.*

Proof. We use the proof technique described in [9, Protocol 3]: to show zero-knowledge, we have to describe an efficient simulator that simulates any interaction between a cheating verifier and a real prover. The simulator does not have a correct solution, but it does have an advantage over the prover: when shuffling decks, it is allowed to swap the packets for different ones. We thus show how to construct a simulator for each challenge c in $\{0, 1\}$ in the Akari ZKP.

if $c = 0$: The simulator simulates the prover P as follows: it fills the grid G only with empty cards and the dual grid $\hat{G}$ only with light cards. If the verifier chooses the challenge $c = 0$, the verifier puts each packet of cards in an envelope of both grids and shuffles them. Then he checks that all packets contain only the same card. This perfectly simulates an interaction with the real prover because there are two packets for each cell: one with only light cards and one with only empty cards.

if $c = 1$: The simulator first prepares the following packets:

- For each white cell in a row of r white cells and in a column of c white cells, the simulator sets:
  - A packet containing (c) empty cards (denoted type 1).
  - A packet containing (r) empty cards (denoted type 2).
  - A packet containing $(c + r - 3)$ empty cards and 2 ◯ cards (denoted type 3).
- For each black cell with number l, the simulator sets a packet containing l ◯ cards and $4 - l$ empty cards. Such a packet is of type 4.

Then, still when the verifier has chosen the challenge $c = 1$, the verifier discards the dual grid $\hat{G}$ and execute the following checks:

- **No two lights see each other:** for each horizontal (resp. vertical) line of adjacent white cells, V randomly picks one card per cell to form a packet. Then the simulator, acting as the the shuffle functionality, replaces this packet by the type 1 (resp. 2) packet corresponding to the same cell. The simulator, now acting as P, adds one ◯ card to the packet and returns the cards to V, who can check that there is exactly one light card in the packet.
- **Black cells:** for each black cell that contains a number l, V picks one card in all adjacent cells to form a packet. Then the simulator, acting as the shuffle functionality, replaces this packet by the type 4 packets corresponding to the same cell. V looks at the cards and can check that l corresponds to the number of ◯ cards.
- **All cells are illuminated:** for each cell, V picks one card from the cell plus one card from all horizontally and vertically adjacent white cells. Then the simulator, acting as the shuffle functionality, replaces this packet by the type 3 packet corresponding to the same cell. The simulator, now acting as P, adds one empty card to the packet and returns the cards to V, who can check that there are exactly two light cards in the packet.

For each cell, all cards are face down before the last shuffle of all packets and the simulated proofs and the real proofs are indistinguishable. Therefore, V learns nothing from the verification phases and the protocol is zero-knowledge. ◀

	1		**0**
		0	
	0		
1	**1**		**0**

0	**1**	1	**0**
1	0	**0**	1
0	**0**	1	1
1	**1**	0	**0**

Figure 3 Example of a Takuzu grid and its solution.

3 Takuzu

Takuzu is a puzzle invented by Frank Coussement and Peter De Schepper in 2009[4]. It was also called *Binero*, *Bineiro*, *Binary Puzzle*, *Brain Snacks* or *Zernero*. A similar game (using crosses and circles) was proposed in 2012 by Aldolfo Zanellat under the name of *Tic-Tac-Logic*. The goal of Takuzu is to fill a rectangular grid of even size with 0's and 1's. An initial Takuzu grid already contains a few filled cases. A grid is solved when it is full (*i.e.*, no empty cases) and respects the following constraints:

1. Each row and each column contains exactly the same number of 1's and 0's.
2. Each row is unique among all rows, and each column is unique among all columns.
3. In each row and each column there can be no more than two same numbers adjacent to each other; for example 110010 is possible, but 110001 is impossible.

Figure 3 contains a simple 4×4 Takuzu grid and its solution. We can verify that each row and column is unique, contains the same number of 0's and 1's, and there are never three consecutive 1's or 0's. The problem of solving a Takuzu grid was proven to be NP complete in [2, 23].

3.1 ZKP Construction for Takuzu

Similar to Akari, our aim is to give a ZKP proof such that the prover P, now Totoro, proves to the verifier V, now Akira, that he knows a solution of a given Takuzu grid. We assume that the grid is printed on a sheet of paper, and use two kinds of cards: cards with a 0 or a 1 printed on them. Moreover, we also use a second grid, a piece of paper, and an envelope.

Setup: Let G be the $n \times n$ Takuzu grid and S its solution known by P. The prover chooses uniformly at random two permutations: π_R for the rows, and π_C for the columns. He then computes $S' = \pi_R(\pi_C(S))$, and writes the two permutations π_C and π_R on a paper, and inserts it into an envelope E. Then P places cards face down on the second grid according to S'. We denote these cards $\tilde{S}'$.

Verification: The verifier V picks c randomly among $\{0, 1, 2, 3\}$.

If $c = 0$: P opens the envelope E. Then V takes the initial grid G and computes $G' = \pi_R(\pi_C(G))$. V reveals those cards in $\tilde{S}'$ that are in places of initial values of the grid in G'. He checks that the all revealed cards are equal to the initial values given in G'.

If $c = 1$: The verifier V picks d randomly among $\{0, 1\}$. If $d = 0$ (resp. $d = 1$), for each row (resp. column), the verifier takes all the cards in the row (resp. in the column) without looking at them. Each one of these n decks is shuffled by the shuffle functionality and

[4] https://en.wikipedia.org/wiki/Takuzu

then the verifier opens all cards and checks that each deck contains exactly the same number of 1's and 0's.

If $c = 2$: The verifier checks that a randomly chosen row or column is different from all other rows or columns. For this, the verifier picks randomly one row or one column. The verifier opens all the cards of his chosen row (or column). For each of the $n-1$ other rows (or columns) the verifier takes all the cards that are in the same column (or row) as a 0 in the revealed row (or column), without looking at them. Each one of these $n-1$ sets of cards is shuffled by the shuffle functionality and given back to the prover, who reveals one card per set that is a 1. Thus each one of the other $n-1$ rows (or columns) has a 1 where the revealed row (or column) has a 0, they are thus different from the revealed row. If there are several 1's in a deck, the prover randomly chooses which one to reveal.

If $c = 3$: P opens E. Then V permutes (face down) the cards of $\tilde{S}'$ to obtain $\tilde{S} = \pi_c^{-1}(\pi_R^{-1}(\tilde{S}'))$. Then, V picks d randomly among $\{0, 1\}$ and e randomly among $\{1, 2, 3\}$.

If $d = 0$: For each row, V sets $x = \lfloor \frac{n-e}{3} \rfloor$ decks of three cards $\{(e+3\cdot i+1, e+3\cdot i+2, e+3\cdot i+3)\}_{\{0\leq i<x\}}$ where the triplet (i, j, k) denotes a deck containing the ith, the jth and the kth cards of the row.

If $d = 1$: For each column, V sets $x = \lfloor \frac{n-e}{3} \rfloor$ decks of three cards $\{(e+3\cdot i+1, e+3\cdot i+2, e+3\cdot i+3)\}_{\{0\leq i<x\}}$ where the triplet (i, j, k) denotes a deck containing the ith, the jth and the kth cards of the column.

Then, V gives the decks to P one by one. For each deck, the prover discards one of the two identical cards (*e.g.*, a 1 if he has 101, and a 0 in case of *e.g.*, 001). Then P reveals the cards to V, who accepts only if he sees two different cards. Note that it is possible to do this verification step for several sequences of three cards.

To have the best security guarantees, the verifier should choose his challenges c, d, etc. such that each combination of challenges at the end has the same probability. This protocol is repeated $\mathfrak{K}$ times where $\mathfrak{K}$ is a chosen security parameter. Note that the ZKP is again polynomial in the size of the grid, as shown next.

3.2 Security Proofs for Takuzu

We now prove the security of our construction.

▶ **Lemma 4** (Takuzu Completeness). *If P knows a solution of a given Takuzu grid, then he is able to convince V.*

Proof. Suppose that P, knowing the solution S of the grid G, runs the setup algorithm as described in Section 3.1. Then we show that P is able to perform the proof for any challenge $c \in \{0, 1, 2, 3\}$.

If $c = 0$: Since S is the solution of G, $G' = \pi_R(\pi_C(G))$ and $S' = \pi_R(\pi_C(S))$, the values of the cards in $\tilde{S}'$ that are in places of non empty cells of G' are equal to the corresponding values of the grid in G'.

If $c = 1$: S has $(n/2)$ occurrences of 1 and $(n/2)$ occurrences of 0 on each row and column. Column and row permutations do not change this property. Therefore, since $S' = \pi_R(\pi_C(S))$, S' has $(n/2)$ occurrences of 1 and $(n/2)$ occurrences of 0 on each row and column.

If $c = 2$: If a row (resp. column) of S is unique and all rows (resp. columns) have the same number of 0's, then no other row of S can have its 0's at the exact same places. Column and row permutations in S' do not change this property.

If $c = 3$: Since $S' = \pi_R(\pi_C(S))$, $\tilde{S} = \pi_C^{-1}(\pi_R^{-1}(\tilde{S'}))$ is the solution S hidden with cards face down, and three consecutive cells of S are never the same since S is a valid solution. Then, using three consecutive cards, the prover is always able to discard one out of three cards such that two different cards remain. ◀

▶ **Lemma 5** (Takuzu Soundness). *If P does not provide a solution of a given $n \times n$ Takuzu grid G, then he is not able to convince V except with a negligible probability $\left(1 - \frac{1}{2n+9}\right)^{\mathfrak{K}}$ when the protocol is repeated $\mathfrak{K}$ times.*

Proof. We show that if P is able to perform the proof of a solution of G for any challenge (c and the sub-challenges d, e, etc. depending on c), then he knows a solution to the Takuzu grid. During the setup phase, P commits:

- An envelope E containing two permutations π_C and π_R.
- A grid of face down cards $\tilde{S'}$.

We set $S = \pi_C^{-1}(\pi_R^{-1}(S'))$. Since P is able to perform the proof for any challenge, we observe that:

- Non empty cells of $G' = \pi_R(\pi_C(G))$ are equal to corresponding cells of S'. Then **non empty cells of G are equal to corresponding cells of S.**
- Rows and columns of S' have the same number of 0s and 1s, and each row and each column of S' is unique. Then, **rows and columns of S have the same number of 0s and 1s, and each row and each column of S is unique**.
- **Three consecutive cells of S do not contain the same value**.

We deduce that S is a solution of G. Hence, if P does not provide a solution of G, then he fails the proof for at least one of the challenges. To compute the probability, we enumerate the possible challenges for all values of c:

If $c = 0$: In this case there is only one possible challenge.

If $c = 1$: There are two possibilities, verifying the rows, or verifying the columns.

If $c = 2$: There are $2n$ choices for the verifier, n rows and n columns.

If $c = 3$: There are two possible values for the challenge d and three possible values for the challenge e, then, there is $2 \times 3 = 6$ combinations for the pair (d, e).

Overall, P receives a challenge out of $1 + 2 + 2n + 6 = 2n + 9$ possibilities. We suppose that the verifier selects any one of these challenges uniformly at random. If the prover gives a wrong grid, then at least one of the checks will fail, and this check will have been selected with probability $\frac{1}{2n+9}$. As the protocol is repeated $\mathfrak{K}$ times, the probability that P convinces V without the solution is at least $\left(1 - \frac{1}{2n+9}\right)^{\mathfrak{K}}$. ◀

▶ **Lemma 6** (Takuzu Zero-Knowledge). *During the verification phase, V learns nothing about P's solution for a given Takuzu grid.*

Proof. Similar to the proof for Akari, we show how to construct a simulator for each challenge $c \in \{0, 1, 2, 3\}$:

$c = 0$: The simulator completes the grid G with random bits to obtain the grid S, and randomly chooses the two permutations π_R and π_C. It then puts π_R and π_C in an envelope E and it commits $S' = \pi_R(\pi_C(S))$ using cards face down (we denote this commitment $\tilde{S'}$). Then it simulates the prover and the verifier as follows: the prover commits $(E, \tilde{S'})$. Then the verifier can open E and use π_R and π_C to compute $G' = \pi_R(\pi_C(G))$. V can then return those cards in $\tilde{S'}$ that are in places of initial values of the grid in G' and check that all returned cards are equal to the initial values given in G'. Since π_R and π_C are randomly chosen, and since the values of cards in $\tilde{S'}$ that are in places of non empty

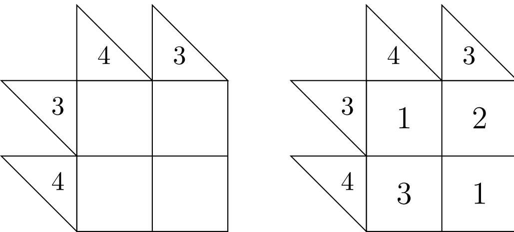


Figure 4 Simple example of a Kakuro grid and its solution.

cells $G' = \pi_R(\pi_C(G))$ are equal to the corresponding values of the grid in G' (the other cards of $\tilde{S}'$ remain face down), the simulated proofs and real proofs are indistinguishable.

$c = 1$: When the verifier shuffles the decks of cards taken in a single row or column using the shuffle functionality, the simulator returns $(n/2)$ cards with 1's and $(n/2)$ cards with 0's, shuffled. This is indistinguishable from a shuffled deck with the same number of 1's and 0's.

$c = 2$: When the verifier selects a row or column, the simulator randomly chooses to position $(n/2)$ 1's and $(n/2)$ 0's. Given the random permutations π_R and π_C, the permuted chosen row is indistinguishable from a randomly chosen one. Then, when the verifier shuffles the deck corresponding to selected cards in the other rows (or columns), the simulator places one card with a 1, and $(n/2 - 1)$ randomly selected other cards. Once again this is indistinguishable from the given decks, as the verifier only sees one card with a 1.

$c = 3$: The simulator chooses randomly S such that three cells of S never contain the same bit. It randomly chooses the two permutations π_R and π_C and puts π_R and π_C in an envelope E. It commits $S' = \pi_R(\pi_C(S))$ using cards face down (we denote this commitment $\tilde{S}'$). Then it simulates the interaction between the prover and the verifier as follows: the prover commits $(E, \tilde{S}')$. The verifier opens E and uses π_R and π_C to compute $\tilde{S} = \pi_C^{-1}(\pi_R^{-1}(\tilde{S}'))$, where $\tilde{S}$ is the commitment of S using face down cards. The verifier then randomly chooses d and e and collects the cards according to the verification algorithm. V then gives each deck of three cards to the prover. For each deck, the prover thus obtains three cards such that (exactly) two cards are identical. He discards one of these two cards and returns the two different cards. Since π_R and π_C are randomly chosen, then simulated proofs and real proofs are indistinguishable. ◀

Therefore, our protocol for Takuzu is complete, sound and zero-knowledge.

4 Kakuro

Kakuro can been seen as a numerical version of crosswords. According to [21], it was proposed for the first time in the 1950's as a logic puzzle in "Official Crossword Puzzles" by Dell Publishing Company. It is also known by its the original English name *Cross Sums*.

A Kakuro grid contains square and triangular white cells, and sometimes also black cells. The goal is to fill in the white cells on the grid with digits from 1 to 9, such that the sum of each line and each column corresponds to the total given in each triangular cell. Moreover, in each line and in each column a number can appear only once. An example of the game with its the solution is given in Figure 4. This game has been proven NP-complete in [12, 22].

4.1 ZKP Construction for Kakuro

In this game, the main challenge is to be able to verify that a sum of some numbers is correct without revealing any other information.

Setup: In this construction, we use black and red playing cards. To represent a number l, for instance 3, we put 9 cards into an envelope: $l = 3$ black cards and $9 - l = 6$ red cards. Using this trick, we can construct a ZKP as follows:

1. In each cell, we place 4 identical envelopes containing some cards that correspond to the number contained in the cell of the solution. The number is encoded using some cards as explained above.
2. Next to each triangular cell, we place envelopes containing cards for all missing numbers in this line or column. For instance, for the first line of Figure 4, we place 7 envelopes containing black and red cards corresponding to the numbers 3, 4, 5, 6, 7, 8, 9.

Verification: We proceed as follows:

- To verify that a number appears only once per line and per column, the verifier proceeds as follows: For each line and each column, he randomly picks an envelope per cell plus all the envelopes next to the triangular cell. The packet of envelopes is shuffled by the shuffle functionality, and then the verifier opens all the envelopes and verifies that all numbers appear only once, and that all numbers between 1 and 9 are present.
- To verify that the sum per line and per column corresponds to the number in the triangular cell, the verifier randomly picks one envelope per cell in the line (resp. in the column) and opens (face down) the content of each envelope. All the cards are shuffled by the shuffle functionality and then the verifier returns them. He can then check that the number of black cards corresponds to the number given in the triangular cell.

This protocol is repeated $\mathfrak{K}$ times where $\mathfrak{K}$ is a chosen security parameter. It is easy to see that the ZKP is polynomial in the size of the grid as we only need a polynomial number of cards and verification steps.

4.2 Security Proofs for Kakuro

We prove the security of our ZKP protocol for Kakuro.

▶ **Lemma 7** (Kakuro Completeness). *If P knows a solution of a given Kakuro grid, then he is able to convince V.*

Proof. After a correct setup each cells contains identical envelopes.

- The unicity of numbers per row or column is given by the fact that all n numbers are present exactly once between the envelopes within the cell and next to the triangle of the row/column.
- The correctness of the sum is given by the fact that when mixed the number of black cards is exactly the value within the triangle. ◀

▶ **Lemma 8** (Kakuro Soundness). *If P does not provide a solution of a given Kakuro grid, then he is not able to convince V except with a negligible probability lower than $p = (1/4)^{\mathfrak{K}}$ when the protocol is repeated $\mathfrak{K}$ times.*

Proof. The proof follows the same line as in [9, Lemma 1]. Each rule is separately verified (unicity in row/column, correct sum in row/column).

Assume that the prover does not know a valid solution for the puzzle. Then he is always caught by the protocol as a liar if he places the cards such that each cell has four cards of identical value. The only way a prover can cheat is by placing on a cell, say cell a, four envelopes that do not all contain the same value. This means that in this cell at least one envelope contains a value y different from at least 2 of the other 3 numbers. Given any assignment of envelopes to the unicities and sums for all other cells, there is either only one envelope with value y in the cell and thus for the (cheating) prover exactly one of the unicities/sums rules that requires y in this cell, or there are two envelopes with value y in the cell and exactly two of the unicities/sums rules require y in this cell. In the first case, the probability that for cell a the verifier assigns y to the one rule needing it is $1/4$. When there are two envelopes encoding y in the cell, the probability to assign y to the first rule needing it is $1/2$ and then the probability to assign the second y to the second rule needing it is $1/3$, overall this is $1/6 < 1/4$. As the protocol is repeated $\mathfrak{K}$ times, the probability that P convinces V without the solution is bounded by $(1/4)^{\mathfrak{K}}$, which is negligible. ◀

▶ **Lemma 9** (Kakuro Zero-Knowledge). *V learns nothing about P's solution of a given Kakuro grid.*

Proof. As in the proof for Akari (Lemma 3) we use the advantage of the simulator over the prover: when shuffling packets (of selected envelopes to show unicity; or collected cards for sums) it is allowed to swap the packets for different ones. The simulator acts as follows:

- The simulator places four arbitrary envelopes on each cell.
- The verifier randomly picks the envelopes for the corresponding packets.
- Then there are two cases:
  - When verifying the unicity of a number, the envelopes in a packet are shuffled. The simulator swaps the packets with a randomly shuffled packet of envelopes, in which each value appears once.
  - When verifying the sums, the envelopes are opened and mixed by the verifier. Then all the cards are shuffled. The simulator swaps the set of cards with a randomly shuffled packet with the same number of cards, in which the number of black cards corresponds to the sum.

The final packets are indistinguishable from those provided by an honest prover assuming that the shuffle functionality guarantees that the packets each contain randomly shuffled sets. ◀

5 Ken-Ken

KenKen is a Japanese game invented by Tetsuya Miyamoto, also known as *Calcudoku*, *Mathdoku* or *Kendoku*. It combines ideas from Sudoku and Kakuro. A KenKen grid is a square grid of size $n \times n$. To solve a KenKen grid, like in Sudoku, each row and each column must contain exactly once all numbers between 1 and n. Moreover, a KenKen grid is divided in groups of cells called *cages*. The example given in Figure 5 contains 3 cages: one vertical line of 3 cells, one horizontal line of 2 cells and a square of 4 cells. Each cage contains a *target* number that has to be produced using the numbers in the cells (in any order) and the mathematical operation (addition, subtraction, multiplication or division) given before the target number. For example in Figure 5, the vertical three-cell cage with the addition operator and a target number of 6 is satisfied with the digits 1, 2, and 3, as $3 + 1 + 2 = 6$. The target number and the operator are given in the upper left corner of a cage. The target must be a positive integer. KenKen is known to be NP-complete [13, 11, 10].

+ 6	− 1	
	∗ 18	

+ 6 3	− 1 1	2
1	∗ 18 2	3
2	3	1

Figure 5 Simple example of a KenKen grid and one possible solution.

In most KenKen grids, division and subtraction operators are restricted to cages of only two cells. For instance in the grid given in Figure 5, the cage with the subtraction operator and the target of 1 has four possible solutions when analyzed in isolation: $(1, 2)$, $(2, 1)$, $(2, 3)$ or $(3, 2)$. In general grids do not need to respect this hypothesis, and there are puzzles that use more than two cells for these operations. In any case, for each subtraction or division cage, there is at least one maximal element of which the other elements in the cage are subtracted or divided.

5.1 ZKP Construction for KenKen

We use the same idea as in Sudoku [9] to verify that all numbers appear only once per row and column. We also use the same representation of numbers as in Kakuro, and the same technique to verify the addition cages. For multiplications, the idea is to check the sum of the exponents of the prime factors of the target. Indeed, in a $n \times n$ grid, all prime factors are below n and there are at most $\mathcal{O}(n/\log(n))$ of those (see, *e.g.*, [19]), so we need at most that number of parallel exponent addition protocols. There we need to encode each integer by its prime factor exponents: for this we use small envelopes marked with the prime factor p, called p-envelopes, and containing the black/red cards encodings for the associated exponent. The maximum possible exponents have to be found among the factors of the integers between 1 and n. For instance, with $n = 9$, all the integers are encoded with the primes 2, 3, 5, 7, with respective exponents between 0 and 3, between 0 and 2, and between 0 and 1 for both 5 and 7. Therefore the 2-envelopes contain exactly 3 cards, the 3-envelopes 2 cards and the 5- and 7-envelopes, 1 card.

To deal with subtractions and divisions, we need an extra interactive round to identify the largest integer in a cage, and then we reduce to either an addition or a multiplication verification. To remain zero-knowledge even after the identification of the maximal element, the solution of the cage is mixed with other solutions, with all the other possible maximal elements, before the verifier checks them. Finally, to be able to deal with both additions/subtractions and multiplications/divisions, we need larger envelopes containing both kind of encodings per cell. The encodings must match. For instance, a 6 in a grid of size 9 is encoded by a large envelope containing: 6 black cards and 3 red cards (just like for Kakuro); but also a small 2-envelope marked with a 2 and containing 1 black card and 2 red cards; similarly, a small 3-envelope with 1 black card and 1 red card; a small 5-envelope with 1 red card and a small 7-envelope with 1 red card. This works also for the value 1, encoded with p-envelopes containing only red cards.

Setup: Our ZKP scheme works as follows for a grid of size n:

- In each cell, we place three identical envelopes encoding the number of the solution, in both encodings.
- For each subtraction cage with $c \geq 2$ cells of target t, let max be the maximal value in the c cells and $c_i \geq 1$, $i = 1, \ldots, c-1$, the other values in any order. Then $t = max - \sum_{i=1}^{c-1} c_i$ and $n \geq max = t + \sum_{i=1}^{c-1} c_i \geq t + (c-1)$. We thus place at most $(n-t-c+1)$ large envelopes next to the grid. Each of these envelopes contains: one marked small envelope (itself containing n cards) and $n(c-1)$ other black or red cards. The number of black cards in the small envelope minus the number of other black cards must match the target. These $(n-t-c+1)$ large envelopes contain all combinations (corresponding to distinct maximal elements) except the one from the solution.
- Each division cage with c cells of target t must contain a maximal element divisible by t and by the $c-1$ other elements. This maximal element must be less or equal than n and larger or equal than t. We denote by u the number of possible maximal elements, $u = |\{m, n \geq m \geq t \text{ and } m/t \in \mathbb{N}\}| \leq (n-t+1)$. Then we place $u-1$ large envelopes next to the grid. Each of these large envelopes contains: a full set of p-envelopes and another envelope, marked, itself containing a full set of p-envelopes, for all primes $p \leq n$. For each prime p, the number of black cards in the p-envelopes of the marked envelope minus the number of black cards in the other p-envelope equals the exponent of p within the factorization of the target t. For instance if the target is 4 and the 2-envelope within the marked envelope contains 3 black cards, then the other 2-envelope in the large envelope must contain exactly 1 black card. A complete example for a division cage is given below.

Verification: Once all placements are done, the verifier randomly picks envelopes placed on each cell and perform the following verifications:

- To verify that each number appears only once per row and column, the verifier randomly picks for each line and for each column one envelope per cell. This packet is shuffled by the shuffle functionality, and then the verifier opens all the envelopes and check that all numbers appear exactly once. Moreover, the verifier checks that both encodings coincide, that is that the exponents within the p-envelopes coincide with the factorization of the number of black cards.
- For each addition cage, the verifier randomly picks one envelope per cell. Then he opens all the envelopes and discards (or gives back to the prover) all its p-envelopes without opening them. Finally, as in Kakuro, the verifier mixes the black and red card from all envelopes face down, asks the shuffle functionality to shuffle them and then verifies that the number of black cards corresponds to the target number.
- For each subtraction cage, there is an extra interactive round:
  1. The verifier randomly picks one envelope per cell in the cage, and asks the shuffle functionality to shuffle them;
  2. The verifier gives to the prover these envelopes, one at a time, after discarding (or giving back to the prover);
  3. The prover looks inside, and has two possibilities:
     - If the envelope does not contain the maximum of the cage, the prover gives back the envelope unmodified to the verifier.
     - Otherwise the prover marks the envelope (in view of the verifier, for instance with a pencil) and gives the marked envelope back to the verifier.

  If there are multiple copies of a maximum element in the cage, the prover randomly chooses which one to mark.

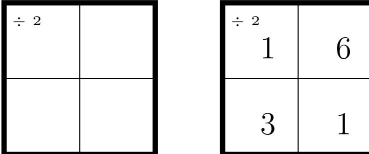


Figure 6 A 2×2 division cage and one solution within a 9×9 KenKen grid.

4. The verifier empties all the envelopes, but the one marked by the prover, into a larger envelope (all these cards are shuffled using the functionality) and discard all the p-envelopes.
5. The verifier adds the marked envelope, still sealed, to the same larger envelope.
6. The verifier asks the shuffle functionality to shuffle this larger envelope with the $(n-t)$ other large envelopes associated to the subtraction cage. Then the verifier opens all large envelopes, checks that each large envelope satisfies the target (black cards in the marked envelope minus the free black cards in the large envelope equals the target) and checks that the $n-t+1$ possible combinations are present exactly once.

- For each multiplication cage of target t, the verifier randomly picks one envelope per cell and opens them all. He discards (or gives back to the prover) the free black and red cards without returning them. Then, one prime p at a time, he empties all the associated small p-envelopes, mixes all the black and red cards, asks the shuffle functionality to shuffle them and then verifies that the number of black cards is the exponent of the prime factor p in the factorization of t.
- For each division cage, we mix the subtraction and multiplication protocols: as in the subtraction, the prover and the verifier enter an interactive extra round to mark an envelope containing a maximal element; then for each of the u possible maximal elements, there is a set of possible multiplicative solutions. There is a complete example below.

This protocol is repeated $\mathfrak{K}$ times where $\mathfrak{K}$ is a chosen security parameter. The protocol can be verified in *polynomial time*. This stems from the fact that the prime factors are all lower than n. Therefore, even factoring the target numbers is just looking at greatest common divisors between t and values from 2 to n.

5.2 Example of a division cage setup for KenKen

To illustrate our construction, we use the division cage with $c = 4$ cells given in Figure 6.

Suppose the cage in the figure is part of an 9×9 grid and that the solution is the one given, that is $2 = 6/3/1/1$. As $9 = 3^2$ and $8 = 2^3$, the maximal exponents for 2, 3, 5 and 7, will be bounded by 3, 2, 1 and 1, respectively, and denoted by e_2, e_3, e_5 and e_7, respectively. There are $u = 4$ possible maximal elements (2, 4, 6, and 8) because $n = 9$ and the target number is 2. Moreover, as this cage contains $c = 4$ cells, the maximal elements are divided by 3 numbers. For instance, not counting orders, the target can be obtained with the following solutions (one per possible maximum): $2 = 8/2/2/1$; $2 = 6/3/1/1$; $2 = 4/2/1/1$; $2 = 2/1/1/1$.

Setup: For each cell of the initial cage the prover places, according to his solution, the following envelopes, where the number of cards contained in a p-envelope is e_p:

1. For each one of the two 1's he places three identical envelopes containing each:
   - To verify addition and subtraction: 1 black card and 8 red cards,

- To verify multiplication and division: a 2-envelope with $e_2 = 3$ red cards, a 3-envelope with $e_3 = 2$ red cards, a 5-envelope with $e_5 = 1$ red card, a 7-envelope with $e_7 = 1$ red cards;

2. For the 3 he places three identical envelopes containing each:
   - To verify addition and subtraction verification: 3 black cards and 6 red cards,
   - To verify multiplication and division: a 2-envelope with $e_2 = 3$ red cards, a 3-envelope with 1 black card and $e_3 - 1 = 1$ red card, a 5-envelope with $e_5 = 1$ red card, a 7-envelope with $e_7 = 1$ red card;
3. For the 6: three identical envelopes containing each:
   - To verify addition and subtraction: 6 black cards and 3 red cards,
   - To verify multiplication and division: a 2-envelope with 1 black card and $e_2 - 1 = 2$ red cards, a 3-envelope with 1 black card and $e_3 - 1 = 1$ red card, a 5-envelope with $e_5 = 1$ red card, a 7-envelope with $e_7 = 1$ red card;

Furthermore, the prover prepares three large envelopes next to the grid:

1. One for the maximal element 8, containing the encoding of $2^1 \cdot 2^1 \cdot 1$:
   - A 2-envelope that contains $3 \cdot e_2 = 9$ cards including 2 black and $3 \cdot e_2 - 2 = 7$ red cards.
   - A 3-envelope that contains $3 \cdot e_3 = 6$ cards including 0 black card and $3 \cdot e_3$ red cards.
   - A 5-envelope that contains $3 \cdot e_5 = 3$ cards including 0 black card and $3 \cdot e_5$ red cards.
   - A 7-envelope that contains $3 \cdot e_7 = 3$ cards including 0 black card and $3 \cdot e_7$ red cards.
   - A marked envelope containing the encoding of $8 = 2^3 \cdot 3^0 \cdot 5^0 \cdot 7^0$:
     - A 2-envelope that contains $e_2 = 3$ cards including 3 black cards and 0 red card.
     - A 3-envelope that contains $e_3 = 2$ cards including 0 black card and 2 red cards.
     - A 5-envelope that contains $e_5 = 1$ cards including 0 black card and 1 red card.
     - A 7-envelope that contains $e_7 = 1$ cards including 0 black card and 1 red card.
2. One for the maximal element 4, containing the encoding of $2^1 \cdot 1 \cdot 1$:
   - A 2-envelope that contains $3 \cdot e_2 = 9$ cards including 1 black card and $3 \cdot e_2 - 1 = 8$ red cards.
   - A 3-envelope that contains $3 \cdot e_3 = 6$ cards including 0 black card and $3 \cdot e_3$ red cards.
   - A 5-envelope that contains $3 \cdot e_5 = 3$ cards including 0 black card and $3 \cdot e_5$ red cards.
   - A 7-envelope that contains $3 \cdot e_7 = 3$ cards including 0 black card and $3 \cdot e_7$ red cards.
   - A marked envelope containing the encoding of $4 = 2^2 \cdot 3^0 \cdot 5^0 \cdot 7^0$:
     - A 2-envelope that contains $e_2 = 3$ cards including 2 black cards and 1 red card.
     - A 3-envelope that contains $e_3 = 2$ cards including 0 black card and 2 red cards.
     - A 5-envelope that contains $e_5 = 1$ cards including 0 black card and 1 red card.
     - A 7-envelope that contains $e_7 = 1$ cards including 0 black card and 1 red card.
3. One for the maximal element 2, containing the encoding of $1 \cdot 1 \cdot 1$:
   - A 2-envelope that contains $3 \cdot e_2 = 9$ cards including 0 black card and $3 \cdot e_2$ red cards.
   - A 3-envelope that contains $3 \cdot e_3 = 6$ cards including 0 black card and $3 \cdot e_3$ red cards.
   - A 5-envelope that contains $3 \cdot e_5 = 3$ cards including 0 black card and $3 \cdot e_5$ red cards.
   - A 7-envelope that contains $3 \cdot e_7 = 3$ cards including 0 black card and $3 \cdot e_7$ red cards.
   - A marked envelope containing the encoding of $2 = 2^1 \cdot 3^0 \cdot 5^0 \cdot 7^0$:
     - A 2-envelope that contains $e_2 = 3$ cards including 1 black card and 2 red cards.
     - A 3-envelope that contains $e_3 = 2$ cards including 0 black card and 2 red cards.
     - A 5-envelope that contains $e_5 = 1$ cards including 0 black card and 1 red card.
     - A 7-envelope that contains $e_7 = 1$ cards including 0 black card and 1 red card.

Verification: The cards that are intended for addition or subtraction are discarded. The verifier and the prover start the round to mark the envelope of the 6. Then the verifier merges the remaining p-envelopes. Thus, the large envelope contains at the end:

- A 2-envelope that contains $3 \cdot e_2 = 9$ cards including 0 black card and 9 red cards, encoding the sum of exponents of 2 for $3^1 \cdot 1 \cdot 1$;
- A 3-envelope that contains $3 \cdot e_3 = 6$ cards including 1 black card and 5 red cards, encoding the sum of exponents of 3 for $3^1 \cdot 1 \cdot 1$;
- A 5-envelope that contains $3 \cdot e_5 = 3$ cards including 0 black card and 3 red cards, encoding the sum of exponents of 5 for $3^1 \cdot 1 \cdot 1$;
- A 7-envelope that contains $3 \cdot e_7 = 3$ cards including 0 black card and 3 red cards, encoding the sum of exponents of 7 for $3^1 \cdot 1 \cdot 1$;
- A marked envelope containing the encoding of $6 = 2^1 \cdot 3^1 \cdot 5^0 \cdot 7^0$:
  - A 2-envelope that contains $e_2 = 3$ cards including 1 black card and 2 red cards.
  - A 3-envelope that contains $e_3 = 2$ cards including 1 black card and 1 red card.
  - A 5-envelope that contains $e_5 = 1$ cards including 0 black card and 1 red card.
  - A 7-envelope that contains $e_7 = 1$ cards including 0 black card and 1 red card.

This large envelope is then shuffled by the shuffle functionality with the 3 other large envelopes prepared in the setup phase. Then the verifier checks that those 4 envelopes encode one possible cage solution for each maximum number, namely, 8, 6, 4 and 2. That is, in each of those four large envelopes, for each prime p, the differences in terms of black cards between both p-envelopes always gives the prime factor decomposition of the target, 2.

5.3 Security Proofs for KenKen

Now we prove the security of our algorithm.

▶ **Lemma 10** (KenKen Completeness). *If P knows a solution of a given KenKen grid, then he is able to convince V.*

Proof. Unicity in rows/columns as well as correctness of addition cages follows from the completeness of the Kakuro protocol in Lemma 7. Correctness for the subtraction comes from the fact that if k is a maximal element in a cage, then the sum of all the remaining elements in the cage is equal to $k - t$ (note that this implies that k must be larger than t). Correctness for the multiplication of target t is guaranteed for each prime p: when mixed, the number of black cards in all the p-envelopes is exactly the exponent of p in the factorization of the target t. Similarly, for the division, for each possible maximal element k, the verifier checks by multiplications that the set of factors always yield k/t. ◀

In order for the protocol to be acceptable its verification phase should at least remain polynomial with the size of the grid. We show that this is indeed the case in Lemma 11.

▶ **Lemma 11** (KenKen Complexity). *The number of operations to verify a KenKen grid is polynomial in the size n of the grid.*

Proof. It is easy to see that the addition and subtraction protocols are linearly checked. For the unicity checks, as well as for the multiplication and division cages, we have to consider the number of distinct possible prime factors. By classical number theory (see for instance, [19, (2.18)]) the number of prime factors below n is $\mathcal{O}(n/\log(n))$. So checking a multiplication cage can be done with that number of prime exponent checks, just like checking the correspondence of the encodings in each cell, and checking a division is performed with at most n multiplication checks (one for each possible maximal element in a cage). Then

each target t is at most $(n!)^n$, so each exponent is at most $\log_2(t) \leq n \log_2(n!) \leq n^2 \log_2(n)$. Finally the number of cages in a grid is at most n^2, so a very rough bound on the number of operations for the verifier is $n^{5+o(1)}$. ◀

▶ **Lemma 12** (KenKen Soundness). *If P does not provide a solution of a given KenKen grid, then he is not able to convince V except with a negligible probability $p = (1/3)^{\mathfrak{K}}$ when the protocol is repeated $\mathfrak{K}$ times.*

Proof. As for Kakuro, separately checking unicity rules, addition or multiplication is perfectly sound. For subtraction, the prover could mark an element of the cage which is not maximal. But then the subtraction would yield a negative result, necessarily different from the target. Therefore checking the subtraction alone is also perfectly sound. A similar argument works for the division. Therefore, in a similar way as for Kakuro (Lemma 8) the prover is always caught by the protocol as a liar if he places the large envelopes such that each cell has three identical envelopes. Thus again, the only way a cheating prover can cheat is by placing on a cell, say cell a, three envelopes that do not all contains the same value. Then at least one value is distinct form the other two, and the probability to randomly pick it for the rule needing it is lower than 1/3. As the protocol is repeated $\mathfrak{K}$ times, the probability that P convinces V without the solution is bounded by $(1/3)^{\mathfrak{K}}$, which is negligible. ◀

▶ **Lemma 13** (KenKen Zero-Knowledge). *V learns nothing about P's solution of a given KenKen grid.*

Proof. The same kind of simulator as for Lemma 9 can be used: for unicity and addition rules the simulator is directly that of Kakuro. Similarly, for multiplication, subtraction and division cages, the verifiers look at the colors of the cards only during the final step of the verification phase, and only after a last shuffle by the prover. Therefore, in this penultimate step, the simulator can use its ability to replace shuffles by his choice of envelopes that satisfy the expected rule. Once again, this indistinguishable from those provided by an honest prover. ◀

6 Conclusion

In this paper, we devised a solution that allows Cobra to solve his friend's dilemma for both games. Akira can now prove to Totoro that she knows a solution to his Akari problem without revealing his solution, and Totoro can prove to Akira that he knows a solution to his Takuzu problem without revealing it either. The same is true for Ken and Kakarotto playing Kakuro and KenKen.

We showed that our solutions are secure, that is complete, sound and zero-knowledge. Moreover, we do not use any cryptographic primitives, but only cards, paper and envelopes.

As future work, we would like to investigate other similar games. For example, we would like to analyse Futoshiki, which can be seen has a variation of Sudoku with additional constraints on the order of the numbers, or Hitori, which has the constraint that all unmarked cells need to be connected, unlike any constraint in the games analysed in this paper.

References

1 Michael Ben-Or, Oded Goldreich, Shafi Goldwasser, Johan Håstad, Joe Kilian, Silvio Micali, and Phillipe Rogaway. Everything provable is provable in zero-knowledge. In *CRYPTO'88*, pages 37–56. Springer, 1990.

2 Marzio De Biasi. Binary puzzle is NP-complete. 2012. URL: http://www.nearly42.org/vdisk/cstheory/binaryp.pdf.

3 Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications. In *STOC'88*, pages 103–112. ACM, 1988.

4 Yu-Feng Chien and Wing-Kai Hon. Cryptographic and physical zero-knowledge proof: From sudoku to nonogram. In *Fun with Algorithms, 5th International Conference, FUN'10*, volume 6099 of *LNCS*, pages 102–112, 2010.

5 Erik D. Demaine. Playing games with algorithms: Algorithmic combinatorial game theory. In *Mathematical Foundations of Computer Science 2001*, volume 2136 of *LNCS*, pages 18–32, 2001.

6 Jannik Dreier, Hugo Jonker, and Pascal Lafourcade. Secure auctions without cryptography. In *Fun with Algorithms, 7th International Conference, FUN'14*, pages 158–170, 2014.

7 Oded Goldreich and Ariel Kahan. How to construct constant-round zero-knowledge proof systems for NP. *Journal of Cryptology*, 9(3):167–189, 1991.

8 Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems. In *STOC'85*, pages 291–304. ACM, 1985.

9 Ronen Gradwohl, Moni Naor, Benny Pinkas, and Guy N. Rothblum. Cryptographic and physical zero-knowledge proof systems for solutions of sudoku puzzles. In *Fun with Algorithms, 4th International Conference*, FUN'07, pages 166–182. Springer-Verlag, 2007.

10 Kazuya Haraguchi and Hirotaka Ono. BLOCKSUM is NP-complete. *IEICE Transactions*, 96-D(3):481–488, 2013.

11 Kazuya Haraguchi and Hirotaka Ono. How simple algorithms can solve latin square completion-type puzzles approximately. *JIP*, 23(3):276–283, 2015.

12 Graham Kendall, Andrew J. Parkes, and Kristian Spoerer. A survey of NP-complete puzzles. *ICGA Journal*, 31(1):13–34, 2008.

13 Jonas Kölker. *I/O-Efficient Multiparty Computation, Formulaic Secret Sharing and NP-Complete Puzzles*. PhD thesis, Aarhus University, 2012.

14 Brandon McPhail. Light up is NP-complete. feb 2005. URL: http://www.mountainvistasoft.com/docs/lightup-is-np-complete.pdf.

15 Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.

16 Takaaki Mizuki and Hiroki Shizuya. Practical card-based cryptography. In *Fun with Algorithms, 7th International Conference, FUN'14*, pages 313–324, 2014.

17 Moni Naor, Yael Naor, and Omer Reingold. Applied kid cryptography or how to convince your children you are not cheating. In *Eurocrypt'94*, pages 1–12, 1999.

18 Jean-Jacques Quisquater, Myriam Quisquater, Muriel Quisquater, Michaël Quisquater, Louis Guillou, Marie Annick Guillou, Gaïd Guillou, Anna Guillou, Gwenolé Guillou, Soazig Guillou, and Thomas A. Berson. How to explain zero-knowledge protocols to your children. In *CRYPTO'89*, pages 628–631. Springer-Verlag, 1990.

19 J. Barkley Rosser and Lowell Schoenfeld. Approximate formulas for some functions of prime numbers. *Illinois Journal of Mathematics*, 6:64–94, 1962.

20 Dennis Shasha. Upstart puzzles: Proving without teaching/teaching without proving. *Commun. ACM*, 57(11):120–120, 2014.

21 Will Shortz. *Easy Kakuro: 100 Addictive Logic Puzzles*. St. Martin's Griffin, 2006.

22 Seta Takahiro. The complexities of puzzles, cross sum and their another solution problems(asp). Master's thesis, University of Tokyo, 2001.

23 Putranto Hadi Utomo and Ruud Pellikaan. Binary puzzles as an erasure decoding problem. In *Proceedings of the 36th WIC Symposium on Information Theory in the Benelux*, pages 129–134, 2015.

Analyzing and Comparing On-Line News Sources via (Two-Layer) Incremental Clustering

Francesco Cambi[1], Pierluigi Crescenzi[2], and Linda Pagli[3]

1 **Bridge Consulting S.r.l., Via L. Rosellini, 50127 Firenze, Italy**
`fcambi@bridgeconsulting.it`

2 **Dipartimento di Ingegneria dell'Informazione, Università degli Studi di Firenze, Viale Morgagni 65, 50134 Firenze, Italy**
`pierluigi.crescenzi@unifi.it`

3 **Dipartimento di Informatica, Università degli Studi di Pisa, Largo B. Pontecorvo 3, 56127 Pisa, Italy**
`linda.pagli@unipi.it`

Abstract

In this paper, we analyse the contents of the web site of two Italian news agencies and of four of the most popular Italian newspapers, in order to answer questions such as what are the most relevant news, what is the average life of news, and how much different are different sites. To this aim, we have developed a web-based application which hourly collects the articles in the main column of the six web sites, implements an incremental clustering algorithm for grouping the articles into news, and finally allows the user to see the answer to the above questions. We have also designed and implemented a two-layer modification of the incremental clustering algorithm and executed some preliminary experimental evaluation of this modification: it turns out that the two-layer clustering is extremely efficient in terms of time performances, and it has quite good performances in terms of precision and recall.

1998 ACM Subject Classification H.2.8 Database Applications: Data mining, H.3.3 Information Search and Retrieval: Clustering, H.3.5 Online Information Services: Web-based services

Keywords and phrases text mining, incremental clustering, on-line news

Digital Object Identifier 10.4230/LIPIcs.FUN.2016.9

1 Introduction

The web is a huge source of data, which are produced by companies, institutions and individuals, and, most of the times, are available for free. The use we can make of all this information is limited just by our imagination. In the last few years, for example, there has been a quite significant amount of research devoted to the analysis of the so-called on-line social networks. One of the most recent examples of such analysis is the adaptation of the well-known "six degrees of separations" phenomenon to the Facebook network: it has, indeed, has been observed that, in the case of this on-line social network, the degrees of separation are less than four [5]. In this paper, instead, we show how the information available on the web sites of news agencies and newspapers can be used in order to answer several questions about the news on-line system, such as the following ones.

- *What are the most relevant news*? Clearly, answering to this question in an automatic way implies defining the notion of relevance, that is, a news score function. In this paper, we propose one possible definition (based on the position occupied by the news in the web sites) and apply it to all the news collected from mid October until mid December of

© Francesco Cambi, Pierluigi Crescenzi, and Linda Pagli;
licensed under Creative Commons License CC-BY
8th International Conference on Fun with Algorithms (FUN 2016).
Editors: Erik D. Demaine and Fabrizio Grandoni; Article No. 9; pp. 9:1–9:14
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

the last year. It turns out that *the news with the maximum score is the shooting down of an American drone in Turkey.* Among the news with the highest score, there is one concerning a case of homicide. The score pattern of this latter news is quite interesting since it has three main peaks: one peak corresponds to the disappearance of the victims, the second peak corresponds to the rising of the hypothesis of homicide, and the third peak corresponds to the identification of possible suspects. We believe that it is interesting to analyse this kind of patterns because they help us to better understand the evolution of a news.

- *What is the average lifespan of a news*? Similar questions have already been posed and answered in different contexts. For instance, a network of hundreds of media sites have been recently looked into, and it has been found that most posts had a shelf life of 2 days, with the median stretching out to about 2.5 days [11]. Closer to our analysis are the results presented in [2], where the authors study the diffusion patterns of news articles from several popular news sources, determine the lifespan of a news article on Twitter by the time difference between the last and first tweet posted containing the URL to that article, and show that for most of the news media companies, about 45% of their articles survive beyond 18 hours. In this paper, we analyse almost 10000 articles, grouped into approximately 7000 news, and we observe that *the average lifespan of a news is a little bit less than one day.* The news with the longest lifespan concerns vaccination campaigns in Italy (with a lifespan greater than 19 days), while one of the many news with the shortest lifespan concerns the rescue of two bear cubs in Laos (with a lifespan of at most one hour).
- *Are different news sources really different?* Once again, answering this question implies defining a notion of similarity among the on-line news sources. In this paper, we propose such a definition, and we apply it to the six on-line sources we have analysed. Interestingly enough, it turns out that *the two news agencies are quite similar*, *one newspaper seems to echo the two news agencies*, while the other *thre newspapers are equally and significantly distant from the two news agencies*, but in opposite directions.

In order to answer the above questions, we have developed a web-based tool which allows us to perform the following operations.

- *Web scraping.* The tool can download the articles (formed by the title, the abstract, and the text) from the web site of two Italian news agencies, and of four popular Italian on-line newspapers. Almost 10000 articles have been downloaded from mid October 2015 to mid December 2015. These articles have been appropriately processed in order to be subsequently analysed in terms of their similarity.
- *Clustering.* As we said, we have defined a similarity measure between two articles, which is based on quite standard text analysis methods. By using these similarity values, a simple incremental clustering algorithm has been implemented: this algorithm basically insert a new article into the cluster with the highest average similarity, if this average similarity is greater than a specific threshold (otherwise a new cluster is created containing only the new article). This incremental clustering algorithm (which is different from previous incremental clustering approaches applied to news detection such as the one analysed in [1, 3]), performs very well in terms of precision and recall results. However, the time performance of the algorithm degrades as the number of articles (and, hence, of clusters) increases. To improve the time performances, we have designed and implemented a two-layer variation of the incremental clustering algorithm (which is also different from similar two-layer approaches such as the one used while analysing Twitter messages in [12]). In this variation, a centroid is associated with each cluster, and the centroids are

clustered by making use of the same approach (that is, by using the average similarity values). Once the best cluster of centroids has been determined, the original incremental clustering is applied to the corresponding clusters. As far as we know, this two-layer approach is different from all previous techniques, which have been used to cluster news. Even if its performances in terms of precision degrade of about 10% the time performance of the new approach are drastically better: indeed, clustering approximately 3000 articles requires less than two seconds, while the original approach required more than 2 minutes.

- *Statistical analysis.* In order to answer the first two questions listed above, we have defined a score function of news, which is based on the position of the articles, included in the same news, in the front page of the web site of the on-line news sources. The position in the front page (along with variation of the Kendall tau measure) has been used also to define similarity measure between different newspapers, and, hence, to answer the third question.

Apart from the fact that our clustering algorithms are different from previous approaches used in news detection (even if, due to the huge literature in the field of incremental clustering, they are similar to several other clustering approaches), as far as we know, this is the first time that such techniques are used to analyse and explicitly compare on-line news sources, on the ground of the order in which news appear in the web sites of the news sources themselves.

The paper is structured as follows. In Section 2, we describe the web scraping performed by our tool, and we present some preliminary descriptive statistics. In Section 3 we introduce our incremental clustering algorithm in order to detect news from articles. In Section 4 we present the results about the score and the lifespan of news, and about the comparison of the six analysed news on-line sources. In Section 5 we describe the two-layer variation of the clustering algorithm. Finally, in Section 6 we conclude and propose some possible directions for future research.

2 The Web Scraping Process

Web scraping is a technique to extract data from web sites, by making use of software libraries that simulates surfing on the web. Fortunately, most of the on-line news sources use well-defined XML schemata and tag systems, which makes the content independent of the graphical rendering of the web site. This feature (which is usually part of what is called the "web semantic") allowed us to quite easily download the contents of the articles which were present on the main web page of the analysed news sources. For instance, in the left part of Figure 1 it is shown part of the main page of one of the most popular Italian on-line newspaper. The XML schema used by this page easily allowed us to detect the main article column (rounded in green), and, within this column, the main article sub-column (rounded in red). In the figure, we have three articles, whose content can then be downloaded by using the hypertext link associated with the title of the article. In the current implementation of our tool, we have then decided to ignore the articles appearing in other sub-columns, since in the vast majority of the cases these articles seem to be of relatively smaller importance. However, it would not be too difficult to change the web scraping module of our tool in order to let it download also these articles.

In the vast majority of the cases, the content of an article consists of three components: the title, the abstract, and the text. In the example shown in the right part of Figure 1, the title is rounded in orange, the abstract is rounded in red, and the text is rounded in green. Once again, the XML schema used by the web site of the on-line news source allows us to easily detect these three components (by also eliminating all types of content we are not

Figure 1 The structure of the main page of a news source (left) and of an article (right).

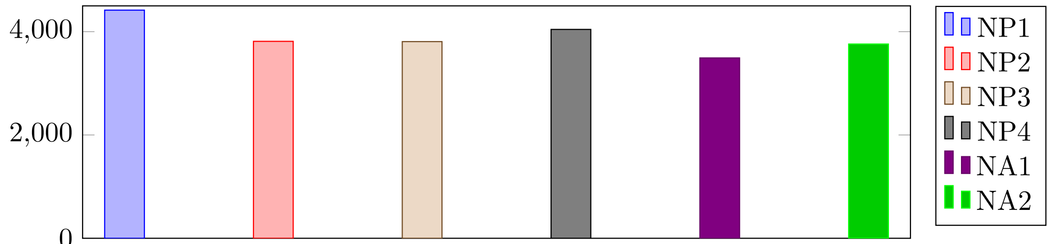


Figure 2 The number of articles downloaded from mid October 2015 until mid January 2016.

interested in, such as video and pictures). After having downloaded the title, the abstract, and the text of an article, this information is stored in a MySQL database, in order to be used for the news detection process that we are going to describe in the next section. This web scraping process has been started at the beginning of October 2015, and it is still in action: currently, we have downloaded more than 23000 articles.

2.1 Some descriptive statistics

As we just said, we have downloaded 23335 articles (during approximately five months) from the web sites of two news agencies NA1 and NA2, and of four newspapers NP1, NP2, NP3, and NP4. These articles are quite uniformly distributed among the six news sources, as it is shown in Figure 2. Note that the download of these articles has been done hourly (almost every day), but, clearly, we have avoided to download again articles that had been previously downloaded. In other words, the number of daily downloaded articles is not just 24 times the number of articles included in the home page of the web site of a news source, since the same articles can remain on the home page itself for several hours. Indeed, in Figure 3 we show the minimum, maximum, and average numbers of distinct daily downloaded articles from each of the six news sources. Once again, these numbers seem to uniformly distributed:

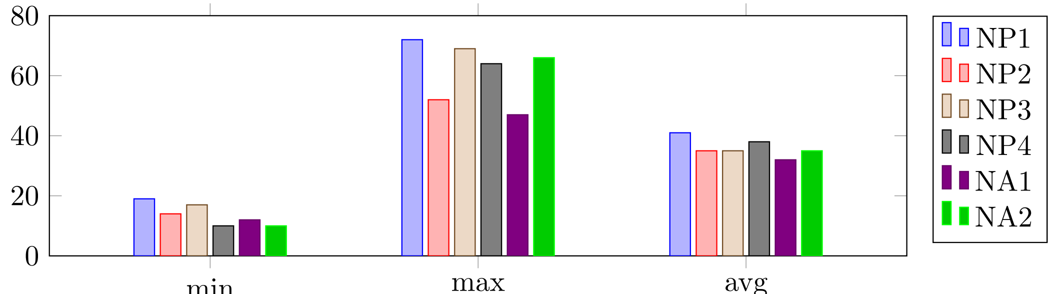


Figure 3 The minimum, maximum, and average number of daily downloaded articles from mid October 2015 until mid January 2016.

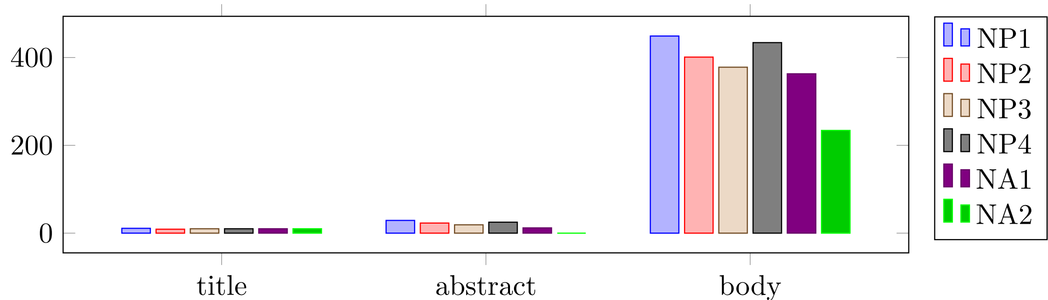


Figure 4 The average lengths of titles, abstracts, and bodies of the six news sources.

this suggest that, on the average, approximately 40 new articles are produced every day in the main area of each home page. The activity of a news source will be better analysed in Section 4.1, in which we will try to measure the degree of variability of the home page of a news source in terms of news (that is, clusters of articles), instead that in terms of articles.

Another simple statistics that can be immediately computed, once a sufficiently large dataset of articles has been downloaded, is the word length of the articles themselves. This number does not really differs among the six news sources (apart form the second news agency that seems to produce, on average, shorter articles). This, indeed, can be deduced from Figure 4, where the average lengths of the titles, the abstracts, and the bodies of the six news sources are shown. More precisely, it seems that, on average, a title (respectively, an abstract, and a body) contains 10 (respectively, 20, and 400) words. The news agency NA2, however, uses shorter body lengths and no abstract at all. This uniformity among the article lengths can be, maybe, justified because of the standards normally used within the printed news system (mainly due to the limited number of pages of a newspaper): however, it is quite surprising that similar standards are also used within the on-line news system, where, in theory, no limit is a-priori existing on the number of pages an article should utilise. It is also worth noting that, if we consider the maximum word length (instead of the average one), the first newspaper NP1 is clearly producing much longer articles: indeed, its maximum article word length is 7216 which is almost twice the second longest maximum length (that is, 3730 in the case of the fourth newspaper NP4).

We have also computed the frequency of the words used in the title or in the abstract of an article. In particular, in Figure 5 we show the frequencies of the names of the leaders of the main Italian parties. As expected, Renzi (with almost 1200 occurrences), leader of *Partito Democratico* and Italian Prime Minister, is by far the most frequent name, followed by Berlusconi (283 occurrences), leader of *Forza Italia*, and Alfano (244 occurrences), leader

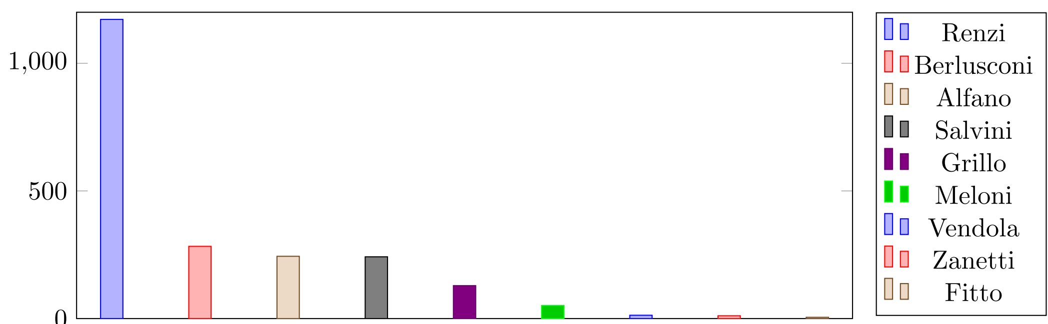


Figure 5 The number of occurrences of the name of the leaders of the Italian parties in the title or in the abstract of an article.

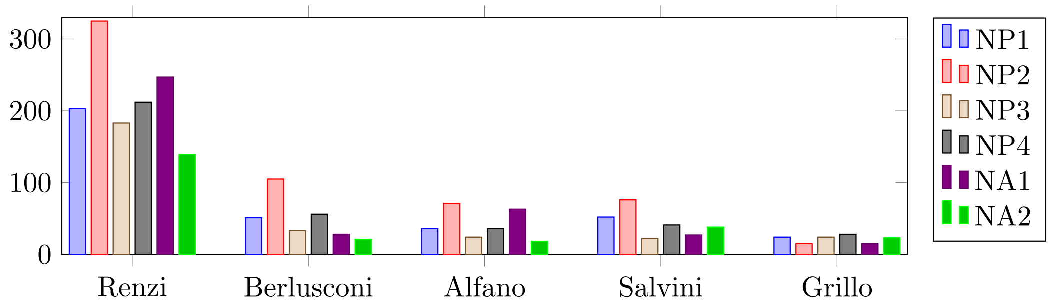


Figure 6 The number of occurrences of the name of five leaders of Italian parties in the title or in the abstarct of an article, depending on the news source.

of *Nuovo Centro-Destra*. More surprisingly, Grillo (129 occurrences), a popular comedian and blogger who is the leader of *Movimento 5 Stelle*, is only fifth in this ranking, preceded also by Salvini (242 occurrences), leader of *Lega Nord*. Interestingly enough, the second newspaper NP2 is systematically the one that most frequently includes the names of the first five leaders into the title or the abstract of an article, apart from the case of Grillo (as it is shown in Figure 6). On the other hand, the second news agency NA2 is the news source which almost always includes the name of the leaders less than the other news sources: in this case, however, we should again observe that this agency does not use abstracts at all.

Finally, we also tried to verify somehow whether the following statement taken from Wikipedia corresponds to reality: “the major news agencies generally prepare hard news stories and feature articles that can be used by other news organisations with little or no modification” [13]. To this aim, we made use of the Levenshtein distance between two strings (that is, the minimum number of insertions, deletions and substitutions required to change one string into the other) [9] in order to define a similarity measure between two string as follows: the difference between the length of the longest string and the Levenshtein distance, divided by the length of the longest string. We then computed for a (small) sample set of articles the similarity of their texts with the texts of the articles of the first news agency NA1. As a result of this comparison, we found a quite high similarity between the articles. The maximum found similarity has been, indeed, equal to 0.96. The two articles reaching this value reported about the first American democratic party debate, and their length was 4527 and 4555, respectively. This implies that the two articles were almost identical.

3 The Clustering Algorithm

As we said in the introduction, we focused our attention on news, instead of articles. Indeed, several different articles can refer to the same news, either because they are present on different web sites, or because the news evolved and new articles concerning the news itself have been produced. For this reason, we had to design a clustering of the articles into clusters corresponding to news. To this aim, we first have to define a similarity measure between articles, and subsequently apply a clustering algorithm on the ground of the similarity values.

3.1 The similarity measure

In order to define a similarity measure between articles, we made use of quite standard text mining techniques (see, for example, the first chapter of [8]). First of all, we cleaned the contents of the articles by eliminating the so-called "stop words", that is, the several hundred most common words in Italian that do not carry any significance by themselves (such as, for example, the articles). Successively, we applied a stemming algorithm for reducing derived words to their word root so that related words map to the same root, even if this root is not in itself a valid word (note that algorithms for stemming have been studied in computer science since the late sixties [10]). Finally, we identified the *keywords* of an article a, that is, the words to be used in order to measure the similarity of a with other articles, as the words appearing in its title $\texttt{title}(a)$ and in its abstract $\texttt{abstract}(a)$, and the ones appearing in its text $\texttt{text}(a)$ starting with a capital letter. In the following, we will denote by $\texttt{key}(a)$ the set of keywords of article a. For each set of articles A and for each word w, the *inverse document frequency* of w with respect to A is defined as

$$\texttt{idf}(w, A) = \log\left(\frac{|A|}{|\{a \in A : w \in \texttt{text}(a)\}|}\right).$$

Moreover, the *term frequency* of w with respect to an article $a \in A$ is defined as

$$\texttt{tf}(w, a) = \frac{\texttt{occ}(w, a)}{|a|}$$

where $\texttt{occ}(w, a)$ denotes the number of occurrences of w in $\texttt{text}(a)$, and $|a|$ denotes the number of words in $\texttt{text}(a)$. For each pair of articles a_1 and a_2 in the set A, we can then define their TF-IDF vectors as follows. Let $\texttt{key}(a_1, a_2) = \texttt{key}(a_1) \cup \texttt{key}(a_2) = \{k_1, \ldots, k_m\}$ be the ordered set of keywords of either a_1 or a_2. Then, for any i with $1 \leq i \leq m$, we define

$$\texttt{tfidf}_{a_1,a_2}[i] = \texttt{tf}(k_i, a_1) \cdot \texttt{idf}(k_i, A)$$

and

$$\texttt{tfidf}_{a_2,a_1}[i] = \texttt{tf}(k_i, a_2) \cdot \texttt{idf}(k_i, A).$$

Finally, the *cosine similarity* $\texttt{cosim}(a_1, a_2)$ between a_1 and a_2 is the cosine value of the angle formed by the two vectors $\texttt{tfidf}_{a_1,a_2}$ and $\texttt{tfidf}_{a_2,a_1}$. Formally, it is defined as

$$\texttt{cosim}(a_1, a_2) = \frac{\sum_{i=1}^{m} \texttt{tfidf}_{a_1,a_2}[i]\texttt{tfidf}_{a_2,a_1}[i]}{\sqrt{\sum_{i=1}^{m} \texttt{tfidf}_{a_1,a_2}[i]^2}\sqrt{\sum_{i=1}^{m} \texttt{tfidf}_{a_2,a_1}[i]^2}}.$$

Note that this value is as close to 1 as the two vectors are similar, that is, as the two articles have similar TF-IDF vectors.

3.2 The incremental clustering

By using the above defined similarity measure, we have designed and implemented the following quite simple incremental clustering algorithm. Assume that a set A of articles has already been clustered into a set N of n news or clusters. Indeed, N can be seen as a function $N : A \to [n]$, where $[n]$ denotes the set of integers between 1 and n. Assume also that a new article x has been downloaded and, hence, has to be classified in either one of the already existing n clusters or in a new one. For each cluster c with $c \in [n]$, let $A_c = \{a \in A : N(a) = c\}$. We then define the similarity between x and c as the average of the similarities between x and all articles in c. More formally,

$$\mathtt{sim}(x, c) = \frac{1}{|A_c|} \sum_{a \in A_c} \mathtt{cosim}(x, a).$$

Let c^* be the cluster c for which $\mathtt{sim}(x, c)$ is maximum: if $\mathtt{sim}(x, c^*)$ is greater than or equal to a given threshold τ, then $N(x) = c^*$ (and x is assigned to cluster c^*), otherwise $N(x) = n + 1$ (and a new cluster is created).

3.2.1 Threshold estimation

In order to apply the above clustering algorithm, we have to determine the threshold τ. To this aim, we have manually clustered the set T containing the first 3000 downloaded articles, and we then have determined the threshold value that produced the best results. In particular, given two clustering N_1 and N_2 with n_1 and n_2 clusters, respectively, we can define the matching function $M_{N_1,N_2} : [n_1] \to [n_2]$ as follows:

$$M_{N_1,N_2}(c_1) = \max_{c_2 \in [n_2]} \mathtt{jac}(c_1, c_2)$$

where $\mathtt{jac}(c_1, c_2)$ denotes the Jaccard index of c_1 and c_2, defined as $\frac{|N_1(c_1) \cap N_2(c_2)|}{|N_1(c_1) \cup N_2(c_2)|}$. In order to evaluate the clustering N_τ obtained by using threshold τ and applied to T, we have first computed the function M_{N_τ,N^*}, where N^* is the manual cluster, and subsequently computed the recall and precision values of each cluster in N_τ. The *precision* of a cluster c is defined as

$$\mathtt{pre}(c) = \frac{|\{a \in T : a \in M_{N_\tau,N^*}(c)\} \cap T_c|}{|T_c|},$$

while the *recall* of a cluster c is defined as

$$\mathtt{rec}(c) = \frac{|\{a \in T : a \in M_{N_\tau,N^*}(c)\} \cap T_c|}{|\{a \in T : a \in M_{N_\tau,N^*}(c)\}|}.$$

The recall and precision values can be combined by obtaining the F-measure, which is defined as follows:

$$\mathtt{F}(c) = 2\frac{p(c) \cdot r(c)}{p(c) + r(c)}.$$

The *average weighted F-measure* of a clustering N_τ is then defined as the average value of the F-measure weighted with respect to the sizes of the clusters. Formally,

$$\mathtt{F}(N_\tau) = \frac{\sum_{i=1}^{n_\tau} \mathtt{F}(c_i)|c_i|}{\sum_{i=1}^{n_\tau} |c_i|}$$

where n_τ denotes the number of clusters in N_τ and $|c_i|$ denotes the size of the i-th cluster (that is, $|c_i| = |\{a \in T : N_\tau(a) = i\}|$). The threshold τ has then been determined by selecting the one who produced the higher average weighted F-measure. It turned out that the best value of τ is equal to 0.35: with this value of τ the average weighted F-measure is equal to 0.89.

4 Analyzing and Comparing the On-Line News Sources

Once the articles have been clustered, and, hence, the news have been detected, we can now answer to several interesting questions concerning the on-line news system.

4.1 Activity of a News Source

A *front page* $f = (t, L)$ of a news source is a time stamped ordered list L of news, which are the news appearing in the home page of the web site of the news source at time t. We would like to measure what is the degree of variability of the front pages of the same news source during the day. To this aim, we have to compare two ordered lists of news, which is equivalent to comparing two top k lists for similarity/dissimilarity. We then decided to make use of the averaging Kendall distance analysed in [6], which is a modification of the Kendall's tau metric between permutations (see the textbook [7]) for the case when we only have the top k members of the ordering. Given two ordered lists L_1 and L_2 of distinct numbers taken from a domain D, for each pair $(x, y) \in D \times D$, the contribution $K_{x,y}(L_1, L_2)$ of x and y to the averaging Kendall distance of L_1 and L_2 is defined as follows (in the following we denote by $a <_{L_i} b$ the fact that a precedes b in L_i).

1. If x and y belong to both L_1 and L_2, then $K_{x,y}(L_1, L_2) = 0$ if they are in the same order in both lists, otherwise $K_{x,y}(L_1, L_2) = 1$.
2. Else if x and y belong to L_i, for $i \in \{1, 2\}$, and either x or y belongs to L_{3-i}, then $K_{x,y}(L_1, L_2) = 0$ if $x <_{L_i} y$ and x belongs to L_{3-i} or if $y <_{L_i} x$ and y belongs to L_{3-i}, otherwise $K_{x,y}(L_1, L_2) = 1$.
3. Else if x belongs to L_i, for $i \in \{1, 2\}$, and y belongs to L_{3-i}, then $K_{x,y}(L_1, L_2) = 1$.
4. Else if x and y belong to L_i, for $i \in \{1, 2\}$, and neither x nor y belongs to L_{3-i}, then $K_{x,y}(L_1, L_2) = \frac{1}{2}$.
5. Else $K_{x,y}(L_1, L_2) = 0$.

The *averaging Kendall distance* $K(L_1, L_2)$ of L_1 and L_2 is then equal to the sum of the contributions of all possible pairs $(x, y) \in D \times D$. Given two front pages $f_1 = (t_1, L_1)$ and $f_2 = (t_2, L_2)$ we then define their distance as the averaging Kendall distance of the corresponding lists L_1 and L_2 of news. Since we have downloaded articles every hour, we have then computed the distance between the front page taken at a given time and the front page taken one hour later (that is, t_2 is equal to t_1 plus 3600 seconds). In Figure 7 we show the plot of these distances computed for the front pages of one news agency (upper part of the figure) and of one newspaper (lower part of the figure). In both cases, it is quite evident that there is a periodic behaviour in updating the web site: however, it is also clear that while the web site of the news agency is updated quite uniformly, the updating process of the web site of the newspaper is significantly concentrated in a given moment of the day (which is, quite obviously, the morning).

4.2 Distances between Different News Sources

By using the averaging Kendall distance, we can also define a distance between two different news sources (note that, in the previous case, we have used the averaging Kendall distance to define a distance between two different front pages of the same news source). Indeed, let $t_1, \ldots, t_n$ be the time instants in which we have downloaded the articles and let $f_i^S = (t_i, L_i^S)$ be the front page of the news source S taken at time t_i. The distance between two news

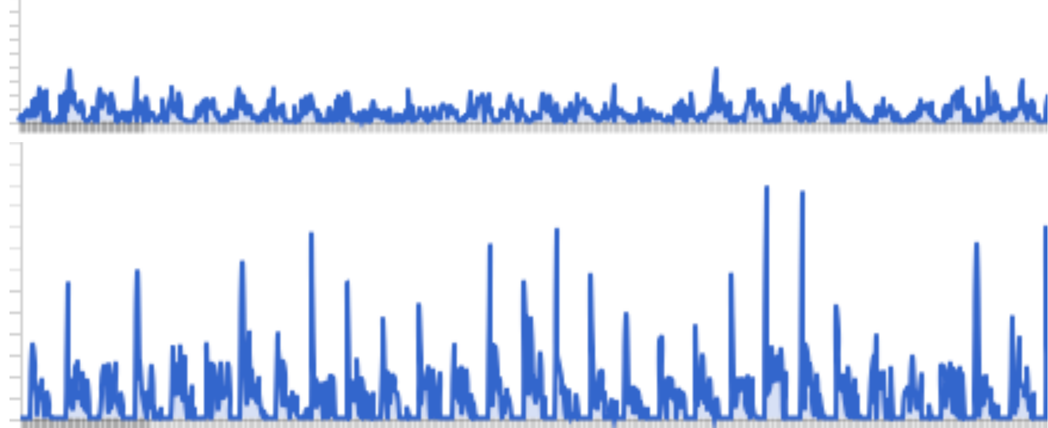


Figure 7 Analysing and comparing the activity of two different news sources.

sources S_1 and S_2 at time t_i is then defined as $K(L_i^{S_1}, L_i^{S_2})$, and the *distance* $d(S_1, S_2)$ the two news sources is the average of all such distances, that is,

$$d(S_1, S_2) = \frac{1}{n} \sum_{i=1}^{n} K(L_i^{S_1}, L_i^{S_2}).$$

Once we have computed these distances, we have then applied the well-known multi-dimensional technique [4] in order to plot the news sources on a two-dimensional plane. The result is shown in Figure 8. As it can be seen from the figure, the two news agencies (in black and grey) are in the centre of the plot and quite close each to the other. One of the four newspapers (in green) is also in the centre of the plot and quite close to both the two news agencies: this suggest that this newspaper acts more as an echo of the news agencies. The other three newspapers, instead, are quite far from the two news agencies and almost equidistant from them (even if at opposite sides): this might imply that these newspapers do indeed elaborate the new produced by the news agencies and present them in the front pages in different ways.

4.3 News Score

The front pages can also be used to determine the score of a news. Indeed, it is well known that a web reader usually reads a web page from top to bottom and from left to right. Hence, we can assume that an article, which appears first in the list of a front page, should receive a higher score than an article which appears after. We have decided to assign to each article a penalty proportional to its position in the front page. More formally, if an article a appears in position i of a front page f formed by n articles, then its *penalty* is equal to $\frac{i-1}{n}$. In other words, the score of this article is

$$\mathtt{rel}(a, f) = \frac{n - i + 1}{n}.$$

We can agglomerate the score of the single articles and obtain the score of a news c with respect to a front page, as follows:

$$\mathtt{rel}(c, f) = \sum_{a:N(a)=c} \mathtt{rel}(a, f).$$

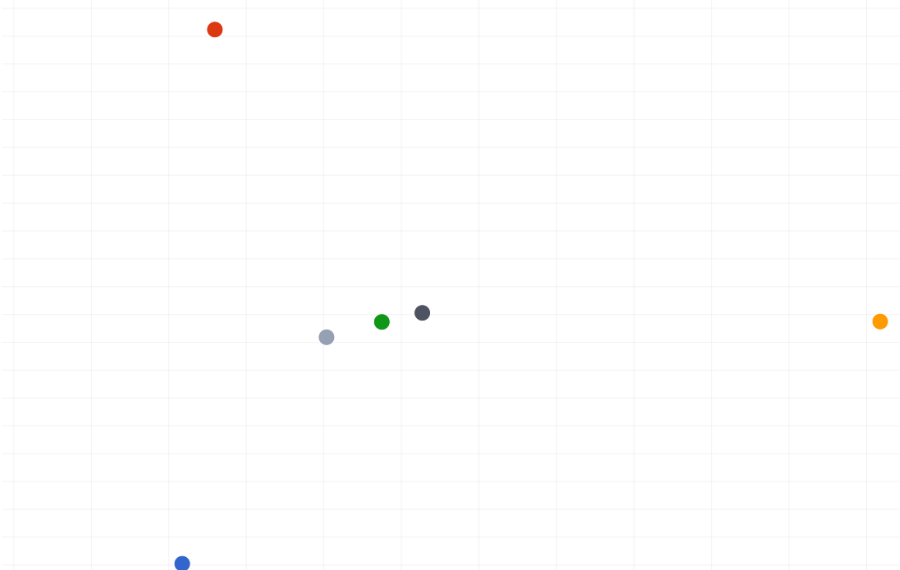


Figure 8 Plotting the news sources on the plane (the two news agencies are in black and in grey).

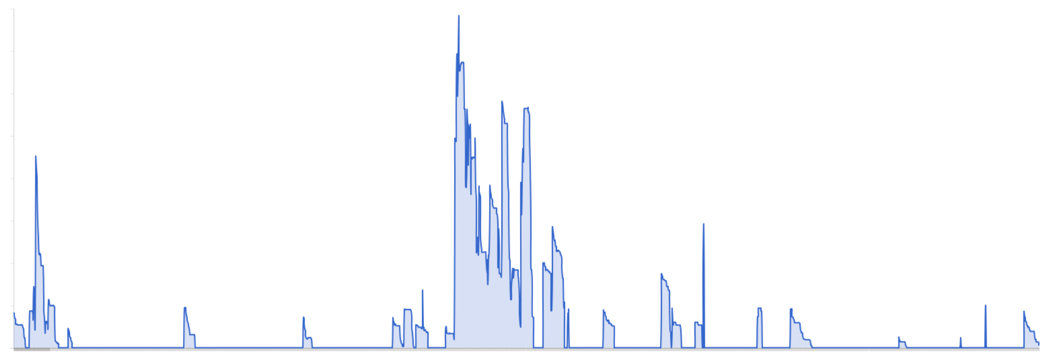


Figure 9 The score of the news concerning the shooting dwon of an American drone.

The total Score of a news is just the sum of its score with respect to all front pages. In Figure 9 we show the score values, during the last four months, of the most relevant news among the almost 7000 news obtained by clustering approximately 10000 articles. This news concerns the shooting down of an American drone in Turkey. As we can see from the figure, the score of this news has two main peaks: one peak corresponds to the news of a shooting down, while the second peak corresponds to the discover that the object shot down was an American drone. It is interesting to analyse this kind of figures because they help us to better understand the evolution of a news. For example, in Figure 10 the evolution of a news concerning a case of homicide in Italy is shown: in this case we have several peaks corresponding, more or less, to the disappearance of the victims, to the rising of the hypothesis of homicide, and to the identification of possible suspects.

4.4 Lifespan of a News

Finally, the news detection, performed by means of the clustering algorithm, allows us to estimate the lifespan of a news. In this case, we have first to define what the lifespan is.

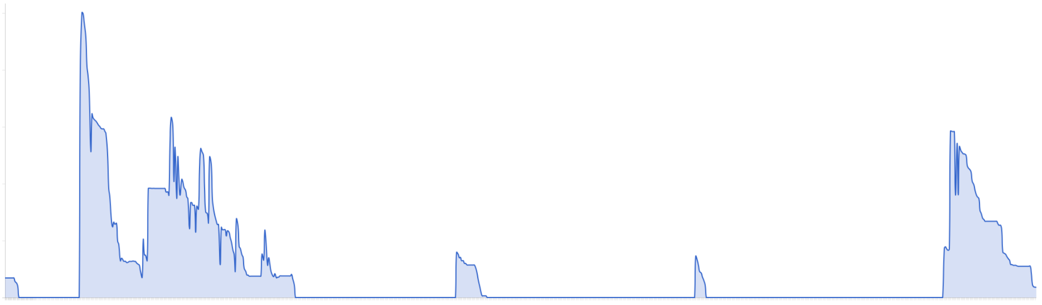


Figure 10 The score of the news concerning a case of homicide in Italy.

Indeed, it does not seem to be correct to consider the lifespan as the temporal interval between the first time the news has appeared and the last time in which it is still present in some news source, since during this interval there might have been long sub-intervals in which the news was not present at all. A typical example of this phenomenon is given by a news concerning a homicide. Typically, at the beginning the news is present for few hours, and then disappears. When some evolution in the investigation takes place, the news appears again and very fast disappears. Finally, when the killer is found, the news appears again and definitively disappears. All this can happen during a very long interval time, which cannot really be interpreted as the lifespan of the news, since during the vast majority of this interval the news was not present at all. We then decided to consider as the *lifespan* of a news as the effective time in which at least one article included in the news was present in at least one source of news. By using this definition, we computed the lifespan of the 7000 news produced by the algorithm with input the 10000 downloaded articles. It turned out that on average the lifespan of a news is approximately twenty hours: the maximum lifespan is a little bit more than 19 days, and it is reached by a news concerning vaccination campaigns inItaly, while the minimum lifespan is one hour, and it is reached, for example, by a news concerning the rescue of two bear cubs in Laos.

5 Improving the Clustering Time Performance

Despite of the very good performances in terms of precision and recall values, the incremental clustering algorithm described above has the disadvantage of degrading its performances while increasing the number of articles to be clustered. Indeed, any new article has to be compared with any other article already clustered in order to compute its average similarity with all the existing clusters. In order to improve the time performances of the clustering algorithm, we have then designed and implemented a two-layer variation of the previously described incremental algorithm. In this variation, each cluster of the first level is “represented” by its *centroid*, which is a non-existing article whose keyword set is the union of all the keyword sets of the articles in the cluster, and whose occurrence function is the average of all the occurrence functions. The centroids are themselves clustered by still using the same incremental approach (with maybe a different threshold value). More formally, the *centroid* of a set of articles A is defined as a “dummy” article m such that

$$\texttt{key}(m) = \bigcup_{a \in A} k(a)$$

and, for each $w \in \texttt{key}(m)$,

$$\texttt{tf}(w,m) = \frac{1}{|A|}\sum_{a\in A}\texttt{tf}(w,a).$$

A *two-layer clustering* is defined as a pair (N_1, N_2) of two functions $N_1 : A \to [n_1]$, where n_1 is the number of clusters in the first level, and $N_2 : C_1 \to [n_2]$, where C_1 is the set of centroids of the n_1 clusters at the first level and n_2 is the cardinality of the set C_2 of clusters in the second level. Given a two-layer clustering (N_1, N_2) and given a new article $x \notin A$, let c_2^* be the cluster $c_2 \in C_2$ for which $\texttt{sim}(x, c_2)$ is maximum: if $\texttt{sim}(x, c_2^*)$ is smaller than a given threshold τ_2, then $N_2(x) = n_2 + 1$ and $N_1(x) = n_1 + 1$ (that is, a new cluster is created both in the first and in the second level). Otherwise, the incremental clustering algorithm described in the previous section is applied to the sub-clustering of N_1 determined by the centroids included in c_2^*. If x is inserted in a cluster c_1^*, then a new centroid of c_1^* is computed and substituted inside the cluster in C_2 containing the previous centroid. Otherwise, a new cluster is created both in the first and in the second level (that is, $N_2(x) = n_2 + 1$ and $N_1(x) = n_1 + 1$).

5.1 Threshold estimation

In order to apply the above two-layer incremental clustering algorithm, we have to determine the two thresholds τ_1 and τ_2. To this aim, we have used again the manual clustering T of the first 3000 downloaded articles, and we have chosen the two values of τ_1 and τ_2 which produced the higher average F-measure. It turned out that the best value of τ_1 and τ_2 is equal to 0.19 and 0.48, respectively: with these values of τ_1 and τ_2 the average F-measure is equal to 0.8. As expected, the obtained average F-measure is lower than the one obtained in the one-layer clustering algorithm. However, the time performance improvement is quite impressive: the clustering of the 3000 articles was executed in less than 2 seconds, while the original approach required more than 2 minutes. Since the number of articles which are downloaded is increasing quite rapidly (approximately 200 articles every day), we believe that a little loss in precision is a reasonable price to be paid in order to make the clustering really efficient.

6 Conclusions

In this paper we have proposed a new tool for analysing and comparing on-line news sources. The main ingredients of this tool are a web scraping module, a news detection module based on a (two-layer) incremental clustering algorithm, and a statistical analysis module, which allowed us to answer several questions concerning the Italian on-line news system. Although the study is at the beginning and it is limited to the Italian news system, the results are quite encouraging: some of them are surprising (for instance, the average lifespan of a news), others can be seen just as curiosities, as for instance the news with minimal lifespan (that is, the rescue of two bear cubs in Laos) or which are the most relevant news according to their position in the site, compared with the truly most relevant ones. Another curiosity is to know that some of the online newspapers add very little information to their news, that result almost identical to that of the news agencies.

Apart from integrating our tool with other statistical analysis, we think that the most interesting possible future research directions concerns the possibility of improving our incremental clustering algorithm. Indeed, this can be done either by considering a k-level clustering algorithm with $k > 2$, or by using the estimated average lifespan of the news

in order to eliminate from the clustering execution all clusters (that is, news) which are sufficiently old.

Acknowledgements. We would like to thank Alice Marchetti and Andrea Marino to suggest us to include some descriptive statistics of the collection of downloaded articles. The second and third authors received additional support from the Italian Ministry of Education, University and Research, under PRIN 2012C4E3KT National research project AMANDA.

References

1. J. Azzopardi and C. Staff. Incremental Clustering of News Reports. *Algorithms*, 5:364–378, 2012.
2. D. Bhattacharya and S. Ram. Sharing News Articles Using 140 Characters: A Diffusion Analysis on Twitter. In *IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, pages 966–971, 2012.
3. Jon Borglund. Event-Centric Clustering of News Articles. Technical report, Department of Information Technology, University of Uppsala, 2013.
4. T.F. Cox and M.A.A. Cox. *Multidimensional Scaling (2nd ed.)*. Chapman and Hall, 2000.
5. S. Edunov, C.G. Diuk, I.O. Filiz, S. Bhagat, and M. Burke. Three and a half degrees of separation, 2016. URL: `http://research.facebook.com/blog/`.
6. R. Fagin, R. Kumar, and D. Sivakumar. Comparing Top K Lists. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 28–36, 2003.
7. M. Kendall and J. D. Gibbons. *Rank Correlation Methods*. Edward Arnold, 1990.
8. J. Leskovec, A. Rajaraman, and J.D. Ullman. *Mining of Massive Datasets*. Cambridge University Press, 2014.
9. Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10:707–710, 1966.
10. J.B. Lovins. Development of a Stemming Algorithm. *Mechanical Translation and Computational Linguistics*, 11:22–31, 1968.
11. Parse.ly. What is the Lifespan of an Article?, 2015. URL: `http://parsely.com`.
12. G. Petkos, S. Papadopoulos, and Y. Kompatsiaris. Two-level Message Clustering for Topic Detection in Twitter. In *SNOW 2014 Data Challenge co-located with 23rd International World Wide Web Conference*, pages 49–56, 2014.
13. Wikipedia – News Agency. URL: `https://en.wikipedia.org/wiki/News_agency`.

Spy-Game on Graphs*

Nathann Cohen[1], Mathieu Hilaire[2], Nícolas A. Martins[3], Nicolas Nisse[4], and Stéphane Pérennes[5]

1 CNRS, Université Paris Sud, LRI, Orsay, France
2 ENS Cachan, France
3 Universidade Federal do Ceará, Fortaleza, Brazil
4 Inria, France; and
Université Nice Sophia Antipolis, CNRS, I3S, UMR 7271, Sophia Antipolis, France
5 Université Nice Sophia Antipolis, CNRS, I3S, UMR 7271, Sophia Antipolis, France; and
Inria, France

Abstract

We define and study the following two-player game on a graph G. Let $k \in \mathbb{N}^*$. A set of k *guards* is occupying some vertices of G while one *spy* is standing at some node. At each turn, first the spy may move along at most s edges, where $s \in \mathbb{N}^*$ is his *speed*. Then, each guard may move along one edge. The spy and the guards may occupy same vertices. The spy has to escape the surveillance of the guards, i.e., must reach a vertex at distance more than $d \in \mathbb{N}$ (a predefined distance) from every guard. Can the spy win against k guards? Similarly, what is the minimum distance d such that k guards may ensure that at least one of them remains at distance at most d from the spy? This game generalizes two well-studied games: Cops and robber games (when $s = 1$) and Eternal Dominating Set (when s is unbounded).

We consider the computational complexity of the problem, showing that it is NP-hard and that it is PSPACE-hard in DAGs. Then, we establish tight tradeoffs between the number of guards and the required distance d when G is a path or a cycle. Our main result is that there exists $\beta > 0$ such that $\Omega(n^{1+\beta})$ guards are required to win in any $n \times n$ grid.

1998 ACM Subject Classification G.2.2 Graph Theory

Keywords and phrases graph, two-player games, cops and robber games, complexity

Digital Object Identifier 10.4230/LIPIcs.FUN.2016.10

1 Introduction

We consider the following two-player game on a graph G, called *Spy-game*. Let $k, d, s \in \mathbb{N}$ be three integers such that $k > 0$ and $s > 0$. One player uses a set of k *guards* occupying some vertices of G while the other player plays with one *spy* initially standing at some node. This is a full information game, thus any player has full knowledge of the positions and previous moves of the other player. Note that several guards and even the spy could occupy a same vertex.

Initially, the spy is placed at some vertex of G. Then, the k guards are placed at some vertices of G. Then, the game proceeds turn-by-turn. At each turn, first the spy may move

* This work has been partially supported by ANR project Stint under reference ANR-13-BS02-0007, ANR program "Investments for the Future" under reference ANR-11-LABX-0031-01, the associated Inria team AlDyNet, the project ECOS-Sud Chile and the project GAIATO, International Cooperation FUNCAP/FAPs/Inria/INS2i-CNRS, no. INC-0083-00047.01.00/13, with Federal Univ. of Ceara, Brasil.

© Nathann Cohen, Mathieu Hilaire, Nícolas A. Martins, Nicolas Nisse, and Stéphane Pérennes;
licensed under Creative Commons License CC-BY

along at most s edges (s is the *speed* of the spy). Then, each guard may move along one edge. The spy wins if, after a finite number of turns (after the guards' move), it reaches a vertex at distance greater than d from every guard. The guards win otherwise, in which case we say that the guards *control* the spy at distance d, i.e. that there is always at least one guard at distance at most d from the spy.

Given a graph G and two integers $d, s \in \mathbb{N}$, $s > 0$, let the *guard-number*, denoted by $gn_{s,d}(G)$, be the minimum number of guards required to control a spy with speed s at distance d, against any strategy from the spy. We also define the following dual notion. Given a graph G and two integers $k, s \in \mathbb{N}$, $s > 0$, $k > 0$, let $d_{s,k}(G)$, be the minimum distance d such that k guards can control a spy with speed s at distance d, whatever be the strategy of the spy.

1.1 Preliminary remarks

We could define the game by placing the guards first. In that case, since the spy could choose its initial vertex at distance greater than d from any guard, we need to slightly modify the rules of the game. If the guards are placed first, they win if, after a finite number of turns, they ensure that the spy always remains at distance at most d from at least one guard. Equivalently, the spy wins if it can reach infinitely often a vertex at distance greater than d from every guard. We show that both versions of the game are closely related. In what follows, we consider the spy-game against a spy with speed s that must be controlled at distance d for any fixed integers $s > 0$ and d.

► **Claim 1.** *If the spy wins in the game when it starts first, then it wins in the game when it is placed after the guards.*

Proof of the claim. Assume that the spy has a winning strategy $\mathcal{S}$ when it is placed first. In particular, there is a vertex $v_0 \in V(G)$ such that, starting from v_0 and whatever be the strategy of the guards, the spy can reach a vertex at distance $> d$ from every guard. If the spy is placed after the guards, its strategy first consists in reaching v_0, and then in applying the strategy $\mathcal{S}$ until it is at distance $> d$ from every guard. The spy repeats this process infinitively often. ◄

The converse is not necessary true, however we can prove a slightly weaker result which is actually tight. For this purpose, let us recall the definition of the well known *Cops and robber* game [15, 4]. In this game, first k cops occupy some vertices of the graph. Then, one robber occupies a vertex. Turn-by-turn, each player may move its token (the cops first and then the robber) along an edge. The cops win if one of them reach the same vertex as the robber after a finite number of turns. The robber wins otherwise. The *cop-number* $cn(G)$ of a graph G is the minimum number of cops required to win in G [1].

► **Claim 2.** *If k guards win in the game when the spy is placed first in a graph G, then $k + cn(G) - 1$ guards win the game when they are placed first.*

Proof of the claim. Assume that k guards have a winning strategy when the spy is placed first. Such a strategy $\mathcal{S}$ is defined as follows. For any position $v \in V(G)$ of the spy, each guard g_i ($1 \leq i \leq k$) is assigned a vertex $pos(i, v)$, such that, for any vertex $w \in V(G)$ at distance at most s from v and for any $i \leq k$, $pos(i, w) \in N[pos(i, v)]$ where $N[x]$ denote the set of vertices at distance at most one from $x \in V$. Moreover, for any $v \in V(G)$, there exists $i \leq k$ such that the distance between v and $pos(i, v)$ is at most d.

Now, let us assume that $k + cn(G) - 1$ guards are placed first. We show that after a finite number of turns, when the spy occupies some vertex v, the vertices $pos(i, v)$ are occupied for all $1 \leq i \leq k$ and then the guards occupying these vertices can follow $\mathcal{S}$ and so win.

Let $0 \leq j < k$ and assume that the vertices $pos(i, v)$ are occupied for all $1 \leq i \leq j$ ($j = 0$ means no such vertex is occupied). The guards occupying the vertices $pos(1, v), \cdots, pos(j, v)$ follow the strategy $\mathcal{S}$. There remains $k + cn(G) - 1 - j \geq cn(G)$ "free" guards. A team of $cn(G)$ of free guards will target the position $pos(j + 1, v)$ (which acts as a robber moving at speed one in G). Therefore, after a finite number of steps, one free guard reaches $pos(j+1, v)$ (where v is the position of the spy at this step). Continuing this way, the vertices $pos(i, v)$ are occupied for all $1 \leq i \leq k$ after a finite number of steps which concludes the proof. ◀

The bound of the previous claim is tight. Indeed, for any graph G, $gn_{1,0}(G) = 1$ since one guard can be placed at the initial position of the spy and then follows it. On the other hand, if the guards are placed first, the game (for $s = 1$ and $d = 0$) is equivalent to the classical Cops and robber game and, therefore, $cn(G)$ guards are required.

1.2 Related work

Further relationship with Cops and robber games

The Cops and robber game has been generalized in many ways [3, 8, 2, 5, 9]. In [3], Bonato *et al.* proposed a variant with *radius of capture*. That is , the cops win if one of them reaches a vertex at distance at most d (a fixed integer) from the robber. The version of our game when the guards are placed first and for $s = 1$ is equivalent to Cops and robber with radius of capture. Indeed, when the spy is not faster than the guards, capturing the spy (at any distance d) is equivalent to controling it at such distance: once a guard is at distance at most d from the spy, it can always maintain this distance (by following a shortest path toward the spy).

This equivalence is not true anymore as soon as $s > 1$. Indeed, one cop is always sufficient to capture one robber in any tree, whatever be the speed of the robber or the radius of capture. On the other hand, we prove below that $\Theta(n)$ cops are necessary to control a spy with speed at least 2 at some distance d in any n-node path. This is mainly due to the fact that, in the spy-game, the spy may cross (or even occupy) a vertex occupied by a guard. Therefore, in what follows, we only consider the case $s \geq 2$.

Note that the Cops and robber games when the robber is faster than the cops is far from being well understood. For instance, the exact number of cops with speed one required to capture a robber with speed two is unknown in 2-dimensional grids [7]. One of our hopes when introducing the Spy-game is that it will lead us to a new approach to tackle this problem.

Generalization of Eternal Domination

A d-dominating set of a graph G is a set $D \subseteq V(G)$ of vertices such that any vertex $v \in V(G)$ is at distance at most d from a vertex in D. Let $\gamma_d(G)$ be the minimum size of a d-dominating set in G. Clearly, $gn_{s,d}(G) \leq \gamma_d(G)$ for any $s, d \in \mathbb{N}$. However these two parameters may differ arbitrary as shown by the following example. Let G be the graph obtained from a cycle C on n-vertices by adding a node x and, for any $v \in C$, adding a path of length $d + 1$ between v and x. It is easy to check that $\gamma_d(G) = \Omega(n/d)$ while $gn_{s,d}(G) = 2$ (the two guards moving on x and its neighbors).

In the *eternal domination* game [10, 11, 13, 14], a set of k *defenders* occupy some vertices of a graph G. At each turn, an *attacker* chooses a vertex $v \in V$ and the defenders may move to adjacent vertices in such a way that at least one defender is at distance at most d (a fixed predefined value) from v. Several variants of this game exist depending on whether exactly one or more defenders may move at each turn [11, 13, 14]. It is easy to see that the spy-game, when the spy has unbounded speed (equivalently, speed at least the diameter of the graph) is equivalent to the Eternal Domination game when all defenders may move at each turn.

1.3 Our contributions

In this paper, we initiate the study of the spy-game for $s \geq 2$. In Section 2, we study the computational complexity of the problem of deciding the guard-number of a graph. We prove that computing $gn_{3,1}(G)$ is NP-hard in the class of graph G with diameter at most 5. Then, we show the problem is PSPACE-complete in the case of DAGs (where guards and spy have to follow the orientation of arcs, but distances are in the underlying graph). Then, we consider particular graph classes. In Section 3, we precisely characterize the cases of paths and cycles. Precisely, for any $k \geq 1$, $s \geq 2$, we prove that

$$\left\lfloor \frac{n(s-1)}{2ks} \right\rfloor \leq d_{s,k}(P_n) \leq \left\lceil \frac{(n+1)(s-1)}{2ks} \right\rceil$$

for any path P_n on n vertices, and

$$\left\lfloor \frac{(n-1)(s-1)}{k(2s+2)-4} \right\rfloor \leq d_{s,k}(C_n) \leq \left\lfloor \frac{(n+1)(s-1)}{k(2s+2)-4} \right\rfloor$$

for any cycle C_n on n vertices. Our most interesting result concerns the case of grids. In Section 4, we prove that there exists $\beta > 0$ such that $gn_{s,d}(G_{n\times n}) = \Omega(n^{1+\beta})$ in any $n\times n$ grid $G_{n\times n}$. For this purpose, we actually prove a lower bound on the number of guards required in a *fractional relaxation* of the game (the formal definition is given in the corresponding section).

Notations

As usual, we consider connected simple graphs. Given a graph $G = (V, E)$ and $v \in V$, let $N(v) = \{w \mid vw \in E\}$ denote the set of neighbors of v and let $N[v] = N(v) \cup \{v\}$.

2 Complexity

2.1 NP-hardness

▶ **Theorem 3.** *Given a graph G with diameter at most 5 and an integer k as inputs, deciding whether $gn_{3,1}(G) \leq k$ is NP-hard.*

Proof. The result is obtained by reducing the classical Set Cover Problem. In the Set Cover Problem the input is a set of elements $\mathcal{U}$, a family $\mathcal{S}$ of subsets of $\mathcal{U}$ such that $\cup_{S\in\mathcal{S}} S = \mathcal{U}$ and an integer k. The question is whether there exists a set $C \subseteq \mathcal{S}$ such that $|C| \leq k$ and $\cup_{S\in C} S = \mathcal{U}$, the set C is called a cover of $\mathcal{U}$.

Let $(\mathcal{U} = \{u_1, \ldots, u_n\}, \mathcal{S} = \{S_1, \ldots, S_m\}, k)$ be an instance of the Set Cover Problem. Note that, for any $u_i \in \mathcal{U}$, there exists $S_j \in \mathcal{S}$ such that $u_i \in S_j$ (since $\cup_{S\in\mathcal{S}} S = \mathcal{U}$). We create a graph G such that there is a cover $C \subseteq \mathcal{S}$ of $\mathcal{U}$ with size at most k if and only if $g^3_1(G) \leq k$.

The graph G is constructed in the following way. Abusing the notation, let us identify the elements in $\mathcal{U} \cup \mathcal{S}$ with some vertices of G. Let $V(G) = \mathcal{S} \cup \mathcal{U} \cup \mathcal{V}$ with $\mathcal{V} = \{v_1, \cdots, v_n\}$. Start with a complete graph with set of vertices $\mathcal{S} = \{S_1, \cdots, S_m\}$ and, for any $1 \leq i \leq n$, add an edge $\{u_i, v_i\}$. Finally, for every $u_i \in \mathcal{U}$ and $S_j \in \mathcal{S}$ such that $u_i \in S_j$, let us add an edge $\{u_i, S_j\}$.

First, let us prove that, if $\mathcal{U}$ admits a cover C of size at most k, then $g_1^3(G) \leq k$. For this purpose, we give a strategy for the guards that ensure that the spy is always at distance at most 1 from at least one guard. When the spy occupies a vertex in $\mathcal{S} \cup \mathcal{U}$, the guards occupy all the vertices of C. When the spy occupies a vertex v_i for some $i \leq n$, let $j(i)$ be such that $u_i \in S_{j(i)} \in C$, then one guard occupies u_i and the other guards occupy the vertices of $C \setminus \{S_{j(i)}\}$. Because the speed of the spy is 3, from a vertex v_i, the spy can only reach a vertex in $\mathcal{S} \cup \mathcal{U}$. Therefore, whatever be the initial position of the spy and its moves, the guards can always ensure the previously defined positions.

Suppose now that there is no cover C of $\mathcal{U}$ with size k, we show that $g_1^3(G) > k$. Let us assume at most k guards are occupying vertices in G, let us consider the following strategy for the spy. The spy starts at S_1. If there exists $i \leq n$ such that no guards dominate u_i, i.e., no guards occupy a vertex of $N[u_i]$, the spy goes at v_i (note that any vertex in $\{v_1, \cdots, v_n\}$ is at distance at most 3 from S_1). Then, no guard can reach a vertex at distance at most 1 from v_i (since u_i is the only neighbor of v_i) and the spy wins.

Let us show that such a vertex u_i exists by reverse induction on the number ℓ of guards occupying vertices in $\{S_1, \cdots, S_m\}$. That is, let $\mathcal{O}$ be the set of vertices occupied by the guards (note that $|\mathcal{O}| = k$) and let $\ell = |\mathcal{O} \cap \mathcal{S}|$. We show that there exists $i \leq n$ such that $\mathcal{O} \cap N[u_i] = \emptyset$. If $\ell = k$, i.e., $\mathcal{O} \subseteq \mathcal{S}$, then the result holds since there is no cover of $\mathcal{U}$ of size at most k. If $\ell < k$, there exists $j \leq n$ such that a guard is occupying u_j or v_j, i.e., there exists $x \in \{u_j, v_j\}$ such that $x \in \mathcal{O}$. Let $z \leq m$ such that $u_j \in S_z$ and let $\mathcal{O}' = \mathcal{O} \cup \{S_z\} \setminus \{x\}$. By induction and because $|\mathcal{O}' \cap \mathcal{S}| = \ell + 1$, there exists $i \leq n$ such that $\mathcal{O}' \cap N[u_i] = \emptyset$. Since $\mathcal{O} \cap N[u_p] \subseteq \mathcal{O}' \cap N[u_p]$ for any $p \leq n$, the result follows. ◀

Note that the previous proof could be easily adapted for a speed $s > 2$ and distance $d = s - 2$ simply adjusting the size of the paths to $s - 1$. The question to generalize this result to any s and d is open. Moreover, since the set cover problem is not approximable within a factor of $(1 - o(1)) \ln n$ [6], our proof also implies the same result to the spy game.

2.2 PSPACE-hardness in the directed case

Then, we consider a variant of our game played on digraphs. In this variant, both the guards and the spy can move only by following the orientation of the arcs. However, the distances are the ones of the underlying undirected graph.

▶ **Theorem 4.** *The problem of computing $gn_{s,2}$ is PSPACE-hard in the class of DAGs, when the guards are placed first.*

The result is obtained by reducing the PSPACE-complete Quantified Boolean Formula in Conjunctive Normal Form (QBF) problem. Given a set of boolean variables $x_1, \ldots, x_n$ and a boolean formula $F = C_1 \wedge C_2 \wedge \ldots \wedge C_m$ where C_j is a disjunction of literals, the QBF problem asks whether the expression $\phi = Q_1 x_1 Q_2 x_2 \ldots Q_n x_n F$ is true, where every Q_i is either $\forall$ or $\exists$.

Proof. For ease of readability, the proof below is given for $d = 2$ but can easily be adapted for any distance d.

Let ϕ be quantified boolean formula with n boolean variables. We construct a DAG D_ϕ such that ϕ is true if and only if n guards control a spy at distance 2 in D_ϕ after a finite number of turns.

For each $Q_i x_i$ of ϕ we construct a gadget digraph D_i. If $Q_i = \exists$ then $V(D_i) = \{w_{i-1}, z_i^1, z_i^2, z_i^3, z_i^4, x_i, x_i^*, \overline{x}_i, \overline{x}_i^*, y_i, v_i, v_i', w_i\}$, the arcs between the vertices are shown in Figure 1a. If $Q_i = \forall$ then $V(D_i) = \{w_{i-1}, z_i^1, z_i^2, z_i^3, z_i^4, x_i, x_i^*, \overline{x}_i, \overline{x}_i^*, y_i, \overline{y}_i, v_i, \overline{v}_i, v_i', w_i\}$ the arcs between the vertices are shown in Figure 1b.

Observe that the vertex w_i appears in both D_i and D_{i+1}. It remains to establish a relationship between each clause and the variables it contains. For each clause C_i we create a vertex c_i in D_ϕ and add an arc from w_n to c_i. We also add an arc from c_i to $x_i(\overline{x}_i)$ if clause C_i contains the literal $x_i(\overline{x}_i)$.

An example of the digraph D_ϕ for $\phi = \exists x_1 \forall x_2 (x_1 \vee \overline{x}_2) \wedge (\overline{x}_1 \vee x_2)$ is shown on Figure 1c.

It remains to prove that ϕ is true if and only if $\vec{g}_2(D_\phi) = n$.

First note that, for each gadget D_i, at least one guard have to pick a vertex from $S_i = \{z_i^1, z_i^2, z_i^3\}$ as his initial position, otherwise the spy would pick z_i^1 as his initial position and no guard could ever reach distance 2 from such vertex, therefore the spy would win. We will refer to the guard initially in S_i as p_i. Since D_ϕ has n such gadgets, then $\vec{g}_2(D_\phi) \geq n$. Furthermore, assuming that each guard p_i starts on z_i^1 he can only occupy the vertices on the set $R_i = \{z_i^1, z_i^2, z_i^3, z_i^4, x_i, x_i^*, \overline{x}_i, \overline{x}_i^*\}$ during the rest of the game.

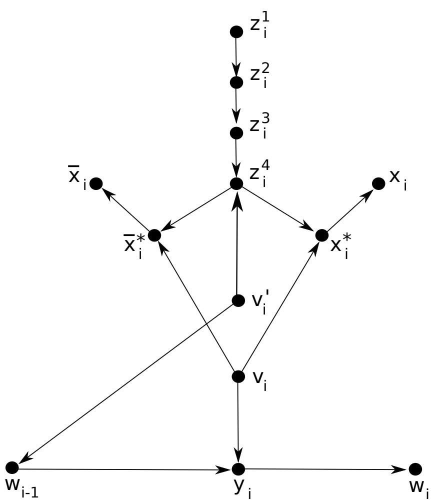
Suppose that $\phi = false$. We describe a winning strategy for the spy playing against n guards. Lets assume that there is exactly one guard in each set S_i, that is, the spy cannot win just initially positioning himself in one unprotected z_i^1. The spy starts on the vertex w_0.

Now, suppose that the spy is in some w_{i-1} of $D_i(\forall)$, then the only guard that can reach a vertex at distance at most 2 from w_{i-1} is p_i when he occupies the vertex z_i^4. The spy waits until the guard p_i moves to z_i^4, if the guard never do so the spy stays on w_{i-1} and wins the game. Therefore suppose that p_i eventually moves to z_i^4, then the spy chooses between moving to y_i or $\overline{y}_i$, depending the choice of the spy the guard p_i is then forced to move to x_i^* or to $\overline{x}_i^*$, because these are the only vertices that are reachable for any guard that are at distance at most 2 from y_i and $\overline{y}_i$ respectively. If p_i moves to x_i^* the corresponding variable x_i is set to $true$. Otherwise, if p_i moves to $\overline{x}_i^*$ then $x_i = false$. It means that for a quantified variable $\forall x_i$ the spy chooses the value of x_i.

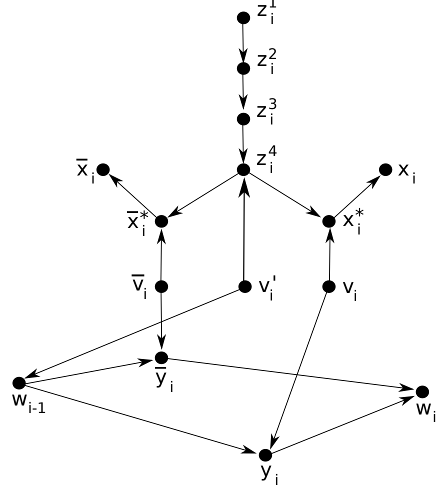
If the spy is in some w_{i-1} of $D_i(\exists)$, again, the only guard that can reach a vertex at distance at most 2 from the spy is p_i when he occupies the vertex z_i^4. The spy then waits until the guard p_i moves to z_i^4 and then moves to y_i, this time p_i is not forced to move to specifically x_i^* or to $\overline{x}_i^*$, but he still must choose one of them. Again, if p_i moves to x_i^* the corresponding variable x_i is set to $true$, otherwise, if p_i moves to $\overline{x}_i^*$ then $x_i = false$. It means that for a quantified variable $\exists x_i$ the guards choose the value of x_i.

When p_n moves to x_n^* or $\overline{x}_n^*$ each guard is on $x_i^*(\overline{x}_i^*)$ or $x_i(\overline{x}_i)$. Observe that each guard can only reach a safe distance from the vertices c_j corresponding to the clauses that contains the literal he set true. Since $\phi = false$ then the spy can choose between y_i and $\overline{y}_i$ on gadgets $D_i(\forall)$ in such a way that no matter how the guards choose x_i^* or $\overline{x}_i^*$ on gadgets $D_i(\exists)$ there is at least one vertex c_j that cannot be protected by any guard. Then the spy moves to such vertex, stays there and wins the game.

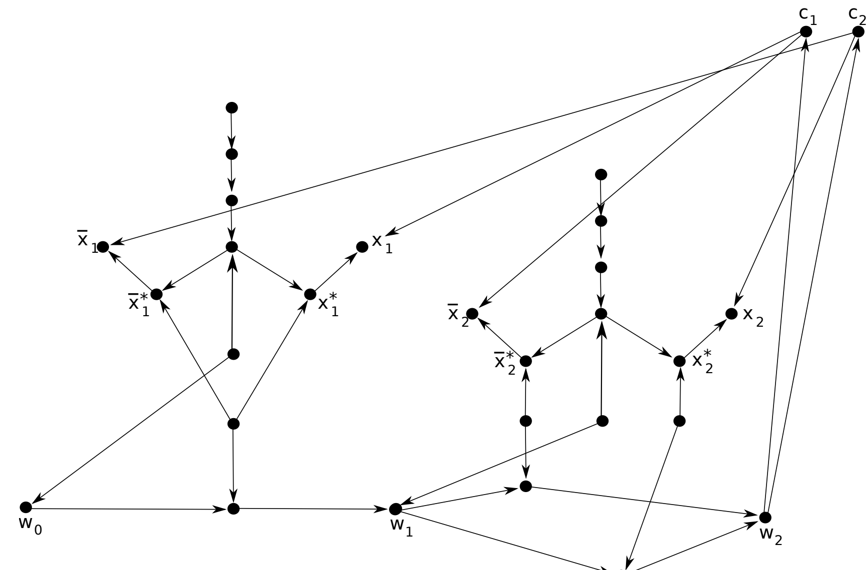
Suppose that $\phi = true$. Each guard p_i, $i = 1, ..., n$, will choose z_i^3 as his initial position. If the spy choose as his initial position $z_i^1, z_i^2, z_i^3, z_i^4, x_i^*$ or $\overline{x}_i^*$ the guard p_i do not need to move since the spy is at distance at most 2 from z_i^3. The only vertices that the spy can go from these initial positions that are not under the protection of p_i are x_i or $\overline{x}_i$. If he goes to any of them the guard p_i just moves to z_i^4. Since the spy cannot move anymore and is at



(a) gadget $D_i(\exists)$ for existential quantifier.



(b) Gadget $D_i(\forall)$ for universal quantifier.



(c) Example of the graph D_ϕ for the formula $\phi = \exists x_1 \forall x_2 (x_1 \vee \overline{x}_2) \wedge (\overline{x}_1 \vee x_2)$.

Figure 1

distance at most 2 from a guard, the guards win the game. If the spy starts on some $v_i, \overline{v}_i$ or v'_i then p_i moves to z_i^4, after that, if the spy goes to $x_i^*, \overline{x}_i^*$ or z_i^4 then p_i follows the same strategy from above. Therefore the spy, independent of his initial position, must eventually move to a vertex $w_i, y_i, \overline{y}_i$ or some clause vertex c_j, otherwise he loses.

Suppose that the spy is in some vertex w_{i-1} of $D_i(\forall)$ then the guard p_i moves to z_i^4 and prevents the spy from communicating. The spy must move to y_i or $\overline{y}_i$ forcing p_i to move to x_i^* or $\overline{x}_i^*$ accordingly. Again, for a quantified variable $\forall x_i$ the spy chooses the value of x_i. After the spy moves from $y_i(\overline{y}_i)$ the cop moves to $x_i(\overline{x}_i)$ and stays there forever.

Similarly, if the spy is in some vertex w_{i-1} of $D_i(\exists)$ then the guard p_i moves to z_i^4 and prevents the spy from communicating. The spy must move to y_i, this time p_i is not forced to move to specifically x_i^* or to $\overline{x}_i^*$, but he still must choose one of them. Therefore, for a quantified variable $\exists x_i$ the guards choose the value of x_i. After the spy moves from y_i the cop moves to x_i or $\overline{x}_i$ depending of his previous movement and stays on that vertex forever.

Observe that after the spy moves from y_n or $\overline{y}_n$ every guard is at distance 2 from w_n at distance 1 from each clause vertex that contains the literal he chose to set true and at distance 2 from each of the other literals of these clauses. Since $\phi = true$ then the guards can choose between y_i and $\overline{y}_i$ on gadgets $D_i(\exists)$ in such a way that no matter how the spy chooses x_i^* or $\overline{x}_i^*$ on gadgets $D_i(\forall)$ all clause vertices are at distance 1 from at least one guard. Therefore the only vertices reachable for the spy are at distance at most 2 from the guards. ◀

The question of the complexity of the spy game in undirected graphs is left open. Is it PSPACE-hard, or more probably EXPTIME-complete as Cops and Robber games [12]? The question of parameterized complexity is also open.

3 Case of paths and rings

In this section, we characterize optimal strategies in the case of two simple topologies: the path and the ring. For ease of readability, some proofs are given in the case $s = 2$. The general proofs (for any $s \geq 2$) are similar.

3.1 Paths

The following theorem directly follows from Lemmas 6 and 7.

▶ **Theorem 5.** *For any path P with $n+1$ nodes and for any $k \geq 1$ and $s \geq 2$,*

$$\left\lfloor \frac{n(s-1)}{2ks} \right\rfloor \leq d_{s,k}(P_n) \leq \left\lceil \frac{(n+1)(s-1)}{2ks} \right\rceil .$$

▶ **Lemma 6.** *For any path P with $n+1$ nodes and for any $k \geq 1$ and $s \geq 2$,*

$$d_{s,k}(P) \geq \left\lfloor \frac{n(s-1)}{2ks} \right\rfloor .$$

Proof. For ease of readability, we prove the lemma in the case $\frac{2d-1}{s-1} \in \mathbb{N}$.

Let $P = (v_0, v_1, \cdots, v_n)$. Let $d = \lfloor \frac{n(s-1)}{2ks} \rfloor$. We show that a spy with speed s playing against at most k guards can reach a vertex at distance at least d from any guard. Intuitively, the strategy of the spy simply consists in starting from one end of P and running at full speed toward the other end. We show that there must be a turn when the spy is at distance at least d from every guard and therefore $d_{s,k}(P) \geq d$.

More formally, let the strategy for the spy be the following. Initially, the spy is occupying an end of the path, say vertex v_0. Then, at each turn $i \geq 1$, the spy moves from $v_{i(s-1)}$ to v_{is}.

We prove by induction on $1 \leq i \leq k$, after turn $i\frac{2d-1}{s-1}$ (when the spy occupies $v_{si\frac{2d-1}{s-1}}$), either at least i guards are occupying vertices in $\{v_0, \cdots, v_{si\frac{2d-1}{s-1}-d}\}$, or there is turn $0 \leq j < i\frac{2d-1}{s-1}$ such that, after Turn j, the distance between the spy and all guards was at least d.

Initially, there must be at least one guard, call g_1, occupying some vertex in $\{v_0, \cdots, v_{d-1}\}$ because otherwise all guards are at distance at least d from the spy at Turn 0. Therefore, after Turn $\frac{2d-1}{s-1}$, Guard g_1 is occupying a vertex in $\{v_0, \cdots, v_{\frac{2d-1}{s-1}+d-1}\} = \{v_0, \cdots, v_{s\frac{2d-1}{s-1}-d}\}$ and the spy is occupying $v_{s\frac{2d-1}{s-1}}$. Hence, the induction hypothesis holds for $i = 1$. Note that the spy is at distance at least d from g_1.

Let $1 \leq i \leq k$ and let us assume by induction that, after Turn $i\frac{2d-1}{s-1}$, there are at least i guards occupying vertices in $\{v_0, \cdots, v_{si\frac{2d-1}{s-1}-d}\}$. Moreover, by definition of the spy's strategy, the spy is occupying $v_{si\frac{2d-1}{s-1}}$. Note that, all these i guards are at distance at least d from the spy.

Then, after Turn $i\frac{2d-1}{s-1}$, there must be at least one guard, call it g_{i+1}, occupying some vertex in $\{v_{si\frac{2d-1}{s-1}-d+1}, \cdots, v_{si\frac{2d-1}{s-1}+d-1}\}$ because otherwise all guards are at distance at least d from the spy at Turn i. Therefore, after Turn $(i+1)\frac{2d-1}{s-1}$, Guard g_{i+1} is occupying a vertex in $\{v_0, \cdots, v_{(si+1)\frac{2d-1}{s-1}+d-1}\}$, that is in $\{v_0, \cdots, v_{s(i+1)\frac{2d-1}{s-1}-d}\}$, and the spy is occupying $v_{(i+1)s\frac{2d-1}{s-1}}$. Similarly, all the i guards that were occupying some vertices in $\{v_0, \cdots, v_{si\frac{2d-1}{s-1}}\}$ after Turn $i\frac{2d-1}{s-1}$ must occupy vertices in $\{v_0, \cdots, v_{s(i+1)\frac{2d-1}{s-1}-d}\}$ after Turn $(i+1)\frac{2d-1}{s-1}$. Hence, the induction hypothesis holds for $i+1$.

Therefore, after Turn $k\frac{2d-1}{s-1}$, either there has been a previous turn when the spy was at distance at least d from all guards, or all the k guards are occupying vertices in $\{v_0, \cdots, v_{sk\frac{2d-1}{s-1}-d}\}$ while the spy occupies $v_{ks\frac{2d-1}{s-1}}$ (note that this vertex exists since $ks\frac{2d-1}{s-1} \leq n$ by definition of d). In the latter case, the spy is at distance at least d from all guards at this turn. ◀

▶ **Lemma 7.** *For any path P with $n+1$ nodes and any $k \geq 1$, $s \geq 2$,*

$$d_{s,k}(P) \leq \left\lceil \frac{(n+1)(s-1)}{2ks} \right\rceil.$$

Proof. For ease of readability, we prove the lemma for $s = 2$.

It is clearly sufficient to prove the result in the case $d = \frac{n+1}{4k} \in \mathbb{N}$. Let $P = (v_0, \cdots, v_n)$ and, for any $1 \leq i \leq k$, let $P_i = (v_{4(i-1)d}, \cdots, v_{4di})$.

We design a strategy ensuring that k guards may maintain the spy at distance at most d from at least one guard. The ith guard is assigned to the subpath P_i (it moves only in P_i). Moreover, a guard i will move at some turn only if the move of the spy at this turn is along an edge of P_i (note that the subpaths P_i are edge-disjoint).

Let $i \leq k$ be such that the spy occupies the node $x = v_{(4i-2)d+\ell}$ with $-2d \leq \ell \leq 2d$. That is, $x \in P_i$. Let us assume that

- for any $1 \leq j < i$, the jth guard occupies $v_{(4j-1)d}$;
- for any $i < j \leq k$, the jth guard occupies $v_{(4j-3)d}$;
- the ith guard occupies $v_{(4i-2)d+\lfloor \ell/2 \rfloor}$ if $\ell \geq 0$ and $v_{(4i-2)d+\lceil \ell/2 \rceil}$ if $\ell \leq 0$.

Clearly, if these conditions are satisfied, the spy is at distance at most $\lceil |\ell|/2 \rceil \leq d$ from the ith guard. Moreover, such positions can be chosen by the guards once the spy has chosen its initial position.

We next show that, whatever be the move of the spy, we can maintain these conditions. Let y be the next vertex to be occupied by the spy. Note that $y = v_{(4i-2)d+\ell+a}$ with $a \in \{-2, -1, 0, +1, +2\}$.

We start with the case when x and y are not in the same subpath P_i. It may happen in only two cases: either $x = v_{4id-1}$ and $y = v_{4id+1}$ ($\ell = 2d-1$ and $a = +2$) or $x = v_{4(i-1)d+1}$ and $y = v_{4(i-1)d-1}$ ($\ell = -2d+1$ and $a = -2$). In the first case, the ith guard goes from $v_{(4i-1)d-1}$ to $v_{(4i-1)d}$ and the $(i+1)^{th}$ guard goes from $v_{(4(i+1)-3)d} = v_{(4i+1)d}$ to $v_{(4i+1)d+1}$. In the latter case, the ith guard goes from $v_{(4i-3)d+1}$ to $v_{(4i-3)d}$ and the $(i-1)^{th}$ guard goes from $v_{(4(i-1)-1)d}$ to $v_{(4(i-1)-1)d-1}$. In both cases, the conditions remain valid.

From now on, let us assume that x and y belong to P_i. In that case, only the ith guard may move. There are several cases depending on the value of $a \in \{-2, -1, 0, +1, +2\}$ and ℓ,

- if $\ell \geq 0$ and $\ell + a \geq 0$, then
  $v_{(4i-2)d+\lfloor(\ell+a)/2\rfloor} \in \{v_{(4i-2)d+\lfloor \ell/2\rfloor-1}, v_{(4i-2)d+\lfloor \ell/2\rfloor}; v_{(4i-2)d+\lfloor \ell/2\rfloor+1}\}$.
  Hence, whatever be the move of the spy, the ith guard can go from $v_{(4i-2)d+\lfloor \ell/2\rfloor}$ to $v_{(4i-2)d+\lfloor(\ell+a)/2\rfloor}$ either moving to one of its neighbor or staying idle.
- if $\ell \leq 0$ and $\ell + a \leq 0$ then
  $v_{(4i-2)d+\lceil(\ell+a)/2\rceil} \in \{v_{(4i-2)d+\lceil \ell/2\rceil-1}, v_{(4i-2)d+\lceil \ell/2\rceil}; v_{(4i-2)d+\lceil \ell/2\rceil+1}\}$.
  Hence, whatever be the move of the spy, the ith guard can go from $v_{(4i-2)d+\lceil \ell/2\rceil}$ to $v_{(4i-2)d+\lceil(\ell+a)/2\rceil}$ either moving to one of its neighbor or staying idle.
- finally, if $\ell * (\ell + a) < 0$, then $(\ell, a) = (-1, 2)$ or $(\ell, a) = (1, -2)$. In that case, the ith guard remains on $v_{(4i-2)d}$.

In all cases, all properties are satisfied after the move of the guards. ◀

3.2 Cycles

We then consider the case of cycles. The following theorem directly follows from Lemmas 9 and 10.

▶ **Theorem 8.** *For any cycle C with $n+1$ nodes and any $k \geq 1$,*

$$\left\lfloor \frac{(n-1)(s-1)}{k(2s+2)-4} \right\rfloor \leq d_{s,k}(C_n) \leq \left\lfloor \frac{(n+1)(s-1)}{k(2s+2)-4} \right\rfloor .$$

▶ **Lemma 9.** *For any cycle C with $n+1$ nodes and any $k \geq 1$, $s \geq 2$,*

$$d_{s,k}(C) \geq \left\lfloor \frac{(n-1)(s-1)}{k(2s+2)-4} \right\rfloor .$$

Proof. Again, the proof is given in the case $s = 2$ for ease of readability.

Let $C = (v_0, v_1, \cdots, v_n)$. Let $d = \lfloor \frac{n-1}{6k-4} \rfloor$. Let the strategy for the spy be the following. Initially, the spy is occupying v_0 and one guard, denoted by g_0, occupies v_{-d} or v_{-d-1} or v_{-d-2} after the guard's turn (the indices of the vertices must be understood modulo $n+1$). Note that such intial position can always be achieved (up to renaming the nodes): the spy goes at distance $d+1$ from the guard g_0 and after the guards' turn, g_0 is at distance $d, d+1$ or $d+2$ from the spy. Then, at each turn $i \geq 1$, the spy moves from v_{2i-2} to v_{2i}.

We prove by induction on $1 \leq i < k$, after Turn $2id$, either at least $i+1$ guards are occupying vertices in $\{v_{-d-2id-2}, \cdots, v_{(4i-1)d-1}\}$, or there is turn $0 \leq j \leq i$ such that, after Turn j, the distance between the spy and all guards was at least d.

Initially, some vertex in $\{v_{-d+1}, \cdots, v_{d-1}\}$ must be occupied by at least one guard, call it g_1, because otherwise the spy is at distance at least d from each guard. Note that g_0 and g_1 are different guards.

Therefore, after Turn $2d$, some vertices in $\{v_{-3d-2}, \cdots, v_{3d-1}\}$ are occupied by Guards g_0 and g_1, and the spy is occupying v_{4d}. Hence, the induction hypothesis holds for $i = 1$.

Let $1 \leq i < k-1$ and let us assume by induction that, after Turn $2id$, there are at least $i+1$ guards occupying vertices in $\{v_{-d-2id-2}, \cdots, v_{(4i-1)d-1}\}$. Moreover, by definition of the spy's strategy, the spy is occupying v_{4id}.

Then, after Turn $2id$, there must be at least one guard, call it g_{i+1}, occupying some vertex in $\{v_{(4i-1)d+1}, \cdots, v_{(4i+1)d-1}\}$ because otherwise all guards are at distance at least d from the spy at Turn i. Therefore, after Turn $2(i+1)d$, Guard g_{i+1} is occupying a vertex in $\{v_{(4i-3)*d+1}, \cdots, v_{(4i+3)d-1}\}$ and the spy is occupying $v_{4(i+1)d}$. Similarly, all the $i+1$ guards that were occupying some vertices in $\{v_{-d-2id-2}, \cdots, v_{(4i-1)d-1}\}$ after Turn $2id$ can only occupy vertices in $\{v_{-d-2(i+1)d-2}, \cdots, v_{(4i+1)d-1}\}$ after Turn $2(i+1)d$. Hence, the induction hypothesis holds for $i+1$: the guards $g_0, \cdots, g_{i+1}$ are occupying nodes in $\{v_{-d-2(i+1)d-2}, \cdots, v_{(4i+3)d-1}\}$.

Therefore, after Turn $2(k-1)d$, either there has been a previous turn when the spy was at distance at least d from all guards, or all the k guards are occupying vertices in $\{v_{-d-2(k-1)d-2}, \cdots, v_{(4k-5)d-1}\}$ while the spy occupies $v_{4(k-1)d}$.

In the latter case, if $v_{-d-2(k-1)d-2}$ is at distance at least d from $v_{4(k-1)d}$ and $v_{4(j-1)d} \notin \{v_{-d-2(k-1)d-2}, \cdots, v_{(4k-5)d-1}\}$ (in other words, if $4(k-1)d + d \leq -d - 2(k-1)d - 2 \mod (n+1)$), then the spy is at distance at least d from all guards at this turn. This is actually the case since $(6k-4)d < n$. ◀

▶ **Lemma 10.** *For any cycle C with $n+1$ nodes and any $k \geq 1$ and $s \geq 2$,*

$$d_{s,k}(C) \leq \left\lfloor \frac{(n+1)(s-1)}{k(2s+2)-4} \right\rfloor.$$

Proof. Again, the proof is given in the case $s = 2$.

It is clearly sufficient to prove the result in the case $d = \frac{n+1}{6k-4} \in \mathbb{N}$. Let $C = (v_0, \cdots, v_n)$. Note that, the indices of the vertices must be understood modulo $n+1$. We design a strategy ensuring that k guards may maintain the spy at distance at most d from at least one guard (note that, in the following strategy, the guard g_1 is at distance $\ell \leq d$ from the spy).

Initially, the spy is in v_h for some $0 \leq h \leq n$. We want to maintain the property that there exists $0 \leq \ell \leq d$ such that the configuration is the following. A guard g_1 is in $v_{\ell+h}$, a guard g_2 is in $v_{4d+3\ell+h}$, and a guard v_{-1} is in $v_{-4d+3\ell+h}$. Then, for any $3 \leq i \leq k-1$, a guard g_i is in $v_{4d+3\ell+6d(i-2)+h} = v_{6di+3\ell-8d+h}$. Note that, for $3 \leq i \leq k-1$, the guard g_i is at distance $6d$ from the guard g_{i-1}, and the guard g_{k-1} is at distance $6d$ from g_{-1}. We show how to maintain such a configuration whatever be the move of the spy.

Obviously, if the spy does not move, no guards move and we are done. If the spy moves along one edge clockwise (resp. anti-clockwise), all guards do the same move and the configuration is maintained. Hence, we only have to consider the cases when the spy moves along 2 edges.

Roughly, in each remaining case, the guard g_1 executes the same move as the spy, and all other guards do the opposite move.

- Case when the spy moves to v_{h+2} (i.e., clockwise) and $\ell \geq 1$. Then, g_1 moves clockwise and all other guards move anti-clockwise. We show that the properties hold for $0 \leq \ell' = \ell - 1 \leq d$ and $h' = h + 2 \mod n+1$. Indeed, g_1 moves from $v_{\ell+h}$ to $v_{\ell+h+1} = v_{\ell'+h'}$. The guard g_2 moves from $v_{4d+3\ell+h}$ to $v_{4d+3\ell+h-1} = v_{4d+3\ell'+h'}$. The guard g_{-1} moves

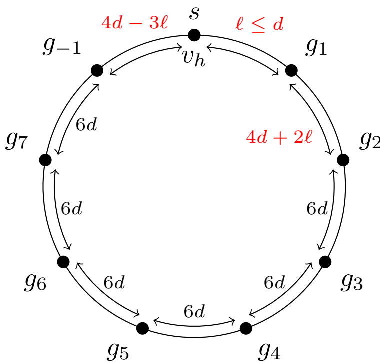


Figure 2 General position in the case $k = 8$, $s = 2$.

from $v_{-4d+3\ell+h}$ to $v_{-4d+3\ell+h-1} = v_{-4d+3\ell'+h'}$. Finally, for any $3 \leq i \leq k-1$, the guard g_i moves from $v_{6di+3\ell-8d+h}$ to $v_{6di+3\ell-8d+h-1} = v_{6di+3\ell'-8d+h'}$. Hence, the property is still valid after the guards' turn.

- Case when the spy moves to v_{h-2} (i.e., anti-clockwise) and $\ell \leq d-1$. Then, g_1 moves anti-clockwise and all other guards move clockwise. Similarly as the previous item, it can be checked that the property holds for $0 \leq \ell' = \ell + 1 \leq d$ and $h' = h - 2 \mod n+1$.
- Case $\ell = 0$. Let us assume that the spy goes anti-clockwise from v_h to v_{h-2} (the case when it goes to v_{h+2} is symmetric). Then, g_1 goes anti-clockwise to v_{-1}, and all other guards go clockwise. Similarly as the previous items, it can be checked that the property holds for $\ell' = 1$ and $h' = h - 2$.
- Case $\ell = d$. Let us assume that the spy goes clockwise from v_h to v_{h+2} (the case when it goes to v_{h-2} is symmetric, the guard g_{-1} playing the role of the guard g_1). Then, g_1 goes clockwise to v_{h+d+1}, and all other guards go anti-clockwise. Similarly as the previous items, it can be checked that the property holds for $\ell' = d-1$ and $h' = h+2$. ◀

4 Case of Grids

It is clear that, for any $n \times n$ grid G, $gn_{s,d}(G) = O(n^2)$. However, the exact order of magnitude of $gn_{s,d}(G)$ is not known. In this section, we prove that there exists $\beta > 0$, such that $\Omega(n^{1+\beta})$ guards are necessary to win against one spy in an $n \times n$-grid. Our lower bound actually holds for a relaxation of the game that we now define.

Fractional relaxation

In the *fractional relaxation* of the game, each guard can be *split* at any time, i.e., the guards are not required to be integral entities at any time but can be "fractions" of guards. More formally, let us assume that some *amount* $\alpha \in \mathbb{R}^+$ of guards occupies some vertex v at some step t, and let $N(v) = \{v_1, \cdots, v_{deg(v)}\}$. Then, at their turn, the guards can choose any $deg(v)+1$ non-negative reals $\alpha_0, \cdots, \alpha_{deg(v)} \in \mathbb{R}^+$ such that $\sum_i \alpha_i = \alpha$, and move an amount α_i of guards toward v_i, for any $0 \leq i \leq deg(v)$ (where $v = v_0$). Then, the guards must ensure that, at any step, the sum of the amount of guards occupying the nodes at distance at most d from the spy is at least one. That is, let $c_t(v) \in \mathbb{R}^+$ be the amount of guards occupying vertex v at step t. The guards wins if, for any step t, $\sum_{v \in B(R_t,d)} c_t(v) \geq 1$, where $B(R_t, d)$ denotes the ball of radius d centered into the position R_t of the spy at step t.

Let $g^{frac}_{s,d}(G)$ be the infimum total amount of guards (i.e., $\sum_{v \in V} c_0(v)$) required to win the fractional game at distance d and against a spy with speed s. Since any *integral strategy* (i.e. when guards cannot be split) is a fractional strategy, we get:

▶ **Proposition 11.** *For any graph G and any integers d, s, $g^{frac}_{s,d}(G) \leq gn_{s,d}(G)$.*

Conversely, a fractional strategy can be represented to some extent by a variation of an integral strategy. Let G be a graph and d, s be two integers. Let also t, k be any two integers. In what follows, t and k will be arbitrary large and can be some function of n, the number of vertices of G. Let $g^{k,t}_{s,d}(G)$ be the minimum number of (integral) guards necessary to maintain at least k guards at distance $\leq d$ from a spy with speed s in G, during t turns. The next lemma will be used below to give a lower bound on $g^{frac}_{s,d}$.

▶ **Lemma 12.** *Let G be a graph with n vertices and $d, s, t, k \in \mathbb{N}$ (t and k may be given by any function of n). Then,*

$$g^{k,t}_{s,d}(G) \leq k g^{frac}_{s,d}(G) + tn^2 .$$

Asymptotically, this yields a useful bound on $g^{frac}_{s,d}$: $\limsup_{k\to\infty} \frac{g^{k,t}_{s,d}(G)}{k} \leq g^{frac}_{s,d}(G)$.

Proof. From a fractional strategy using an amount c of guards, we produce an integer strategy keeping $\geq k$ guards around the spy. Initially, each vertex which has an amount x of guards receives $\lfloor xk \rfloor + tn$ guards, for total number of $\leq ck + tn^2$ guards.

We then ensure that, at step $i \in \{1, ..., t\}$, a vertex having an amount of x guards in the fractional strategy has $\geq xk + (t - i)n$ guards in the integer strategy. To this aim, whenever an amount x_{uv} of guards is to be transferred from u to v in the fractional strategy, we move $\lfloor x_{uv} k \rfloor + 1$ in the integer strategy.

As our invariant is preserved throughout the t steps, the spy which had an amount of ≥ 1 guards within distance d in the fractional strategy now has $\geq k$ guards around it, which proves the result. ◀

In what follows, we prove that $g^{frac}_{s,d}(G) = \Omega(n^{1+\beta})$ for some $\beta > 0$ in any $n \times n$-grid G. The next lemma is a key argument for this purpose.

▶ **Lemma 13.** *Let $G = (V, E)$ be a graph and $d, s \in \mathbb{N}$ ($s \geq 2$), with $g^{frac}_{s,d}(G) > c \in \mathbb{Q}^*$ and the spy wins in at most t steps against c guards starting from $v \in V(G)$. For any strategy using a total amount $k > 0$ of guards, there exists a strategy for the spy (with speed $\leq s$) starting from $v \in V(G)$ such that after at most t steps, the amount of guards at distance at most d from the spy is less than k/c.*

Proof. For purpose of contradiction, assume that there is a strategy $\mathcal{S}$ using $k > 0$ guards that contradicts the lemma. Then consider the strategy $\mathcal{S}'$ obtained from $\mathcal{S}$ by multiplying the number of guards by c/k. That is, if $v \in V$ is initially occupied by $q > 0$ guards in $\mathcal{S}$, then $\mathcal{S}'$ places qc/k guards at v initially (note that $\mathcal{S}'$ uses a total amount of kc/k=c guards). Then, when $\mathcal{S}$ moves an amount q of guards along an edge $e \in E$, $\mathcal{S}'$ moves qc/k guards along e. Since $\mathcal{S}$ contradicts the lemma, at any step $\leq t$, at least an amount k/c of guards is at distance at most d from the spy, whatever be the strategy of the spy. Therefore, $\mathcal{S}'$ ensures that an amount of at least 1 cop is at distance at most d from the spy during at least t steps. This contradicts that $g^{frac}_{s,d}(G) > c$ and that the spy wins after at most t steps. ◀

While it holds for any graph and its proof is very simple, we have not been able to prove a similar lemma in the classical (i.e., non-fractional) case.

The main technical lemma is the following. To prove it, we actually prove Lemma 18 which gives a lower bound on $g^{k,t}_{s,d}(G)$ in any grid G (this technical lemma is postponed at the end of the section). Then, it is sufficient to apply Lemmas 12 and 18 to obtain the following result.

▶ **Lemma 14.** *Let G be a $n \times n$-grid and $a \in \mathbb{N}^*$ such that $d = 2n/a \in \mathbb{N}$. There exists $\gamma > 0$ such that $g^{frac}_{s,d}(G) \geq \gamma a H(a)$, where H is the harmonic function. Moreover, the spy wins after at most $2n$ steps starting from a corner of G.*

From Lemmas 13 and 14, we get

▶ **Corollary 15.** *Let G be a $n \times n$-grid and $a \in \mathbb{N}^*$. For any strategy using a total amount of $k > 0$ guards, there exists a strategy for the spy (with speed $\leq s$) starting from a corner of G such that after at most $2n$ steps, the amount of guards at distance at most $2n/a$ from the spy is less than $k * (aH(a))^{-1}$.*

▶ **Theorem 16.** *$\exists \beta, \gamma > 0$ such that, for any $n \times n$-grid $G_{n\times n}$ and $s, d \in \mathbb{N}$ ($s \geq 2$), the spy (with speed $\leq s$) can win (for distance d) in at most $2n$ steps against $< \gamma n^{1+\beta}$ guards.*

Proof. We actually prove that $\exists \beta > 0$ such that $\Omega(n^{1+\beta}) = g^{frac}_{s,d}(G_{n\times n})$ in any $n \times n$-grid $G_{n\times n}$ and the result follows from Proposition 11.

Let $a_0 \in \mathbb{N}$ be such that $H(a_0)^{-1} \leq 1/2$. Since $g^{frac}_{s,d}(G_{n\times n})$ is non-decreasing as a function of n, it is sufficient to prove the lemma for $n = (a_0)^i$ for any $i \in \mathbb{N}^*$.

We prove the result by induction on i. It is clearly true for $i = 1$ since a_0 is a constant. Assume by induction that there exists $\gamma, \beta > 0$, such that, for $i \geq 1$ with $n = (a_0)^i$, the spy (with speed $\leq s$) can win (for distance d) in at most $2n$ steps against $\gamma a_0^{i(1+\beta)}$ guards in any $n \times n$ grid.

Let G be a $n \times n$-grid with $n = (a_0)^{i+1}$. Let $k \leq \gamma n^{1+\beta}$. By Corollary 15, there exists a strategy for the spy (with speed $\leq s$) starting from a corner of G such that after $t \leq 2n$ steps, the amount of guards at distance at most $2n/a_0$ from the spy is less than $k * (a_0 H(a_0))^{-1} \leq k/(2a_0) \leq \gamma n^{1+\beta}/(2a_0)$.

Let v be the vertex reached by the spy at the step t of strategy $\mathcal{S}$. Let G' be any subgrid of G with side n/a_0 and corner G. By previous paragraph at most $\gamma n^{1+\beta}/(2a_0)$ can occupy the nodes at distance at most d from any node of G' during the next $2n/a_0$ steps of the strategy. So, by the induction hypothesis, the spy playing an optimal strategy in G' against at most $\gamma n^{1+\beta}/(2a_0)$ guards will win. ◀

▶ **Corollary 17.** *$\exists \beta > 0$ such that, for any $n \times n$-grid $G_{n\times n}$ and $s, d \in \mathbb{N}$ ($s \geq 2$),*

$$g_{s,d}(G_{n\times n}) = \Omega(n^{1+\beta}).$$

To conclude, it remains to prove Lemma 14. As announced above, we actually prove a lower bound on $g^{k,t}_{s,d}(G)$. Since $g^{k,t}_{s,d}(G)$ is an nondecreasing function of s, it is sufficient to prove it for $s = 2$.

▶ **Lemma 18.** *Let G be a $n\times n$ grid. $\exists \beta > 0$ such that for any $d, k > 0$, $g^{k,2n}_{2,d}(G) \geq \beta k \frac{n}{d} H(\frac{n}{d})$.*

Proof. Let G be a $n \times n$ grid and let us identify its vertices by their natural coordinates. That is, for any $(i_1, j_1), (i_2, j_2) \in [n]^2$, vertex (i_1, j_1) is adjacent to vertex (i_2, j_2) if $|i_1 - i_2| + |j_1 - j_2| = 1$.

In order to prove the result, we will consider a *family* of strategies for the spy. For every $r \in [n]$, the spy starts at position $(0, 0)$ and runs at full speed toward $(r, 0)$. Once there, it

continues at full speed toward $(r, n-1)$. We name P_r the path it follows during this strategy, which is completed in $\lceil \frac{1}{2}(r+n-1) \rceil$ tops.

Let us assume that there exists a strategy using an amount q of guards that maintains at least k guards at distance at most d from the spy during at least $2n$ turns. Moreover, the spy only plays the strategies described above.

Assuming that the guards are labelled with integers in $[q]$, we can name at any time of strategy P_r the labels of k guards that are at distance $\leq d$ of the spy. In this way, we write $c(2r, 2j)$ this set of k guards that are at distance $\leq d$ from the spy, when the spy is at position $(2r, 2j)$.

▶ **Claim 19.** *If $|j_2 - j_1| > 2d$, then $c(2r, 2j_1)$ and $c(2r, 2j_2)$ are disjoint.*

Proof of the claim. Assuming $j_1 < j_2$, it takes $j_2 - j_1$ tops for the spy in strategy P_r to go from $(2r, 2j_1)$ to $c(2r, 2j_2)$. A cop cannot be at distance $\leq d$ from $(2r, 2j_1)$ and, $j_2 - j_1$ tops later, at distance $\leq d$ from $(2r, 2j_2)$. Indeed, to do so its speed must be $\geq 2(j_2 - j_1 - d)/(j_2 - j_1) > 1$, a contradiction. ◀

▶ **Claim 20.** *If $|r_2 - r_1| > 2d + 2\min(j_1, j_2)$, then $c(2r_1, 2j_1)$ and $c(2r_2, 2j_2)$ are disjoint.*

Proof of the claim. Assuming $r_1 < r_2$, note that strategies P_{2r_1} and P_{2r_2} are identical for the first r_1 tops. By that time, the spy is at position $(2r_1, 0)$. If $c(2r_1, 2j_1)$ intersects $c(2r_2, 2j_2)$, it means that at this instant some cop is simultaneously at distance $\leq d + j_1$ from $(2r_1, 2j_1)$ (strategy P_{2r_1}) and at distance $\leq d + |r_2 - r_1| + j_2$ from $(2r_2, 2j_2)$ (strategy P_{2r_2}). As those two points are at distance $2|r_2 - r_1| + 2|j_2 - j_1|$ from each other, we have:

$$\begin{aligned} 2|r_2 - r_1| + 2|j_2 - j_1| &\leq (d + j_1) + (d + |r_2 - r_1| + j_2) \\ |r_2 - r_1| + 2|j_2 - j_1| &\leq 2d + j_1 + j_2 \\ |r_2 - r_1| &\leq 2d + 2\min(j_1, j_2) \end{aligned}$$ ◀

We can now proceed to prove that the number of guards is sufficiently large. To do so, we define a graph H on a subset of $V(G)$ and relate the distribution of the guards (as captured by c) with the independent sets of H. It is defined over $V(H) = \{(2r, 4dj) : 2r \in [n], 4dj \in [n]\}$, where:

- $(2r, 4dj_1)$ is adjacent with $(2r, 4dj_2)$ for $j_1 \neq j_2$ (see Claim 19).
- $(2r_1, 4dj_1)$ is adjacent with $(2r_2, 4dj_2)$ if $|r_2 - r_1| > 4d(1 + \min(j_1, j_2))$ (see Claim 20).

By definition, c gives k colors to each vertex of H, and any set of vertices of H receiving a common color is an independent set of H. If we denote by $\#c^{-1}(x)$ the number of vertices which received color x, and by $\alpha_{(2r_1, 4dj_1)}(H)$ the maximum size of an independent set of H containing $(2r_1, 4dj_1)$, we have:

$$\begin{aligned} q &= \sum_{(2r_1, 4dj_1) \in V(H)} \sum_{x \in c(2r_1, 4dj_1)} \frac{1}{\#c^{-1}(x)} \\ &\geq \sum_{(2r_1, 4dj_1) \in V(H)} \frac{k}{\alpha_{((2r_1, 4dj_1))}(H)} \end{aligned}$$

It is easy, however, to approximate this lower bound.

▶ **Claim 21.** $\alpha_{((2r_1, 4dj_1))}(H) \leq 4d(j_1 + 1) + 1$.

Proof of the claim. An independent set $S \subseteq V(H)$ containing $(2r_1, 4dj_1)$ cannot contain two vertices with the same first coordinate. Furthermore, $(2r_1, 4dj_1)$ is adjacent with any vertex $(2r_2, 4dj_2)$ if $|r_2 - r_1| > 4d(1 + j_1)$. ◀

We can now finish the proof:

$$\begin{aligned}
q &\geq \sum_{(2r_1,4dj_1)\in V(H)} \frac{k}{\alpha_{((2r_1,4dj_1))}(H)} \\
&\geq \sum_{(2r_1,4dj_1)\in V(H)} \frac{k}{4d(j_1+1)+1} \\
&\geq \frac{n}{2} \sum_{j_1\in\{0,\dots,n/4d\}} \frac{k}{4d(j_1+1)+1} \\
&\geq \frac{kn}{16d} \sum_{j_1\in\{1,\dots,n/4d+1\}} \frac{1}{j_1} \\
&\geq \frac{kn}{16d} H(n/4d)
\end{aligned}$$
◀

We leave the exact value of $gn_{s,d}$ in grids as an intriguing open problem.

References

1 M. Aigner and M. Fromme. A game of cops and robbers. *Discrete Applied Mathematics*, 8:1–12, 1984.

2 N. Alon and A. Mehrabian. On a generalization of Meyniel's conjecture on the cops and robbers game. *Electr. J. Comb.*, 18(1), 2011.

3 A. Bonato, E. Chiniforooshan, and P. Pralat. Cops and robbers from a distance. *Theor. Comput. Sci.*, 411(43):3834–3844, 2010.

4 A. Bonato and R. Nowakovski. *The game of Cops and Robber on Graphs.* American Math. Soc., 2011.

5 J. Chalopin, V. Chepoi, N. Nisse, and Y. Vaxès. Cop and robber games when the robber can hide and ride. *SIAM J. Discrete Math.*, 25(1):333–359, 2011.

6 Uriel Feige. A threshold of $\log n$ for approximating set cover. *J. ACM*, 45(4):634–652, 1998.

7 F. V. Fomin, P. A. Golovach, J. Kratochvíl, N. Nisse, and K. Suchan. Pursuing a fast robber on a graph. *Theor. Comput. Sci.*, 411(7-9):1167–1181, 2010.

8 F. V. Fomin, P. A. Golovach, and D. Lokshtanov. Cops and robber game without recharging. In *12th Scandinavian Symp. and Workshops on Algorithm Theory (SWAT)*, volume 6139 of *LNCS*, pages 273–284. Springer, 2010.

9 F.V. Fomin, P. A. Golovach, and P. Pralat. Cops and robber with constraints. *SIAM J. Discrete Math.*, 26(2):571–590, 2012.

10 W. Goddard, S.M. Hedetniemi, and S.T. Hedetniemi. Eternal security in graphs. *J. Combin.Math.Combin.Comput.*, 52, 2005.

11 John L. Goldwasser and William Klostermeyer. Tight bounds for eternal dominating sets in graphs. *Discrete Mathematics*, 308(12):2589–2593, 2008.

12 William B. Kinnersley. Cops and robbers is exptime-complete. *J. Comb. Theory, Ser. B*, 111:201–220, 2015.

13 W.F. Klostermeyer and G MacGillivray. Eternal dominating sets in graphs. *J. Combin.Math.Combin.Comput.*, 68, 2009.

14 W.F. Klostermeyer and C.M. Mynhardt. Graphs with equal eternal vertex cover and eternal domination numbers. *Discrete Mathematics*, 311(14):1371–1379, 2011.

15 R. J. Nowakowski and P. Winkler. Vertex-to-vertex pursuit in a graph. *Discrete Maths*, 43:235–239, 1983.

The Complexity of Snake

Marzio De Biasi[1] and Tim Ophelders[*2]

1 **Vittorio Veneto, Italy**
marziodebiasi@gmail.com
2 **Department of Mathematics and Computer Science, TU Eindhoven, The Netherlands**
t.a.e.ophelders@tue.nl

Abstract

Snake and Nibbler are two well-known video games in which a snake slithers through a maze and grows as it collects food. During this process, the snake must avoid any collision with its tail. Various goals can be associated with these video games, such as avoiding the tail as long as possible, or collecting a certain amount of food, or reaching some target location. Unfortunately, like many other motion-planning problems, even very restricted variants are computationally intractable. In particular, we prove the NP–hardness of collecting all food on solid grid graphs; as well as its PSPACE–completeness on general grid graphs. Moreover, given an initial and a target configuration of the snake, moving from one configuration to the other is PSPACE–complete, even on grid graphs without food, or with an initially short snake.

Our results make use of the nondeterministic constraint logic framework by Hearn and Demaine, which has been used to analyze the computational complexity of many games and puzzles. We extend this framework for the analysis of puzzles whose initial state is chosen by the player.

1998 ACM Subject Classification F.1.1 Models of Computation

Keywords and phrases Games, Puzzles, Motion Planning, Nondeterministic Constraint Logic, PSPACE

Digital Object Identifier 10.4230/LIPIcs.FUN.2016.11

1 Introduction

Recently the study of the complexity of puzzles and video games has gained a lot of popularity [2, 5, 13]. These puzzles are often based on motion planning problems. We will consider puzzles that can be modeled using paths or entities that move on planar graphs. A few such *motion planning* problems are the train marshalling problem [1], the robot and multi–robot path planning problems [7, 14], and the self–reconfiguring robot problem [8]. As a real–world example we can consider a set of linked wagons towed by a locomotor that must reach a target configuration by moving through a narrow environment. More geometric variants have also been studied, such as motion planning of deformable snake-like paths [6] in the Euclidean plane with obstacles. The problems we will consider arise from the popular games Snake and Nibbler.

Snake is a well-known video game with simple rules that dates back to 1978. It was inspired by the 1976 game Blockade. Since its original release, many variants of Snake have been created, implemented over a wide range of platforms. The simplicity of Snake has led to implementations for graphing calculators and cellphones in the late 90's. Despite its age,

* Tim Ophelders is supported by the Netherlands Organisation for Scientific Research (NWO) under project no. 639.023.208.

© Marzio De Biasi and Tim Ophelders;
licensed under Creative Commons License CC-BY
8th International Conference on Fun with Algorithms (FUN 2016).
Editors: Erik D. Demaine and Fabrizio Grandoni; Article No. 11; pp. 11:1–11:13
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

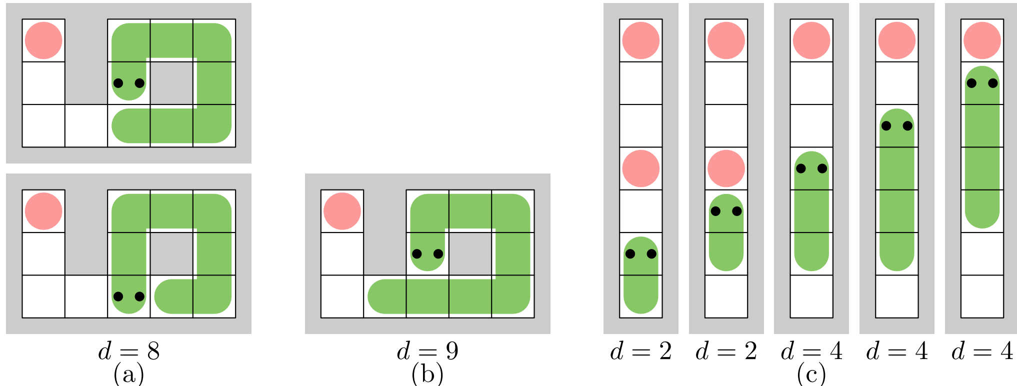


Figure 1 A valid move (a). A snake that cannot move (b). Collecting food with $g = 2$ (c).

the popularity of Snake has hardly decreased, as new variants still appear to this day. A variant we focus our analysis on in this paper is the 1982 arcade game Nibbler.

The objective of Nibbler is to collect a set of items (food), placed on vertices of a graph, by maneuvering a simple path (a snake) through that graph. This path grows by a constant number of vertices per collected item, and the path can move only by extending the front of the path (the head) or removing vertices from the end of the path (the tail). So once a vertex is part of the snake, it is removed only after all vertices towards the tail have been removed. As a result, such vertices may trap the head of the snake, preventing it from reaching particular items. The challenge of Nibbler is to route the head without trapping it before collecting all food. We define moves between states of Nibbler in Definition 1, and the food collection problem in Definition 2. In our analysis of this problem, the growth rate g, the initial length $|P|$ of the snake, and the amount of food $|F|$ are treated as parameters.

▶ **Definition 1** (Valid moves between Nibbler states)**.** Consider a graph G and a *growth rate* $g \in \mathbb{N}$. A *snake* is a sequence P of vertices forming a simple path in G. A Nibbler *state* (P, F, d) is a snake P, a set $F \subseteq V_G$ of *food* vertices, and an integer $d \geq |P|$ representing the target length of the snake, see also Figure 1. A *move* $(P, F, d) \to (P', F', d')$ between Nibbler states is *valid* if and only if $F' = F \setminus P'$, and $d' = d + g \cdot |F \setminus F'|$, and $t +\!\!\!+ P' = P +\!\!\!+ h$, with $|h| = 1$ and $|t| = 1$ unless $|P| < d'$, in which case $|t| = 0$. Here, $+\!\!\!+$ denotes sequence concatenation and h and t capture the movement of the head and tail of the snake.

▶ **Definition 2** (Nibbler food collection problem (NIBBLER))**.**
Input. A graph G, growth rate g, and Nibbler state (P, F, d) with $F \cap P = \emptyset$ and $d = |P|$.
Output. Is there a sequence of valid moves that reaches a state (P', F', d') with $|F'| = 0$?

Figure 1 illustrates valid moves (or the absence thereof) for small instances of Nibbler. Generally, Nibbler takes place on a rectangular grid, possibly containing walls. We distinguish the variant without walls (solid grid graphs) from the one with walls (grid graphs), and discuss them separately in Sections 2 and 4. For the latter, we use reductions from PSPACE–complete problems introduced in Section 3. These problems may be of independent interest for proving PSPACE–hardness of puzzles in which no initial position can be enforced.
A *grid graph* is a finite node–induced subgraph of the infinite two–dimensional integer grid, see Definition 3. The Hamiltonian cycle and path problems are NP–complete even when restricted to grid graphs [9]. A *solid grid graph* is a grid graph without holes. Formally, all points $p \in \mathbb{Z} \times \mathbb{Z}$ that are not vertices of a solid grid graph lie in its outer face. The

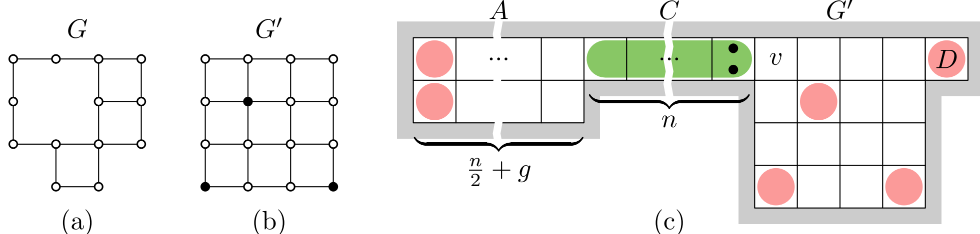


Figure 2 An instance of Nibbler (c) derived from a Hamiltonian cycle problem instance (a).

Hamiltonian cycle problem on solid grid graphs is solvable in polynomial time [11]. In contrast, the complexity of the Hamiltonian path problem on solid grid graphs is still open.

▶ **Definition 3** (Grid graph). A finite undirected graph (V, E) with $V \subseteq \mathbb{Z} \times \mathbb{Z}$ and $(u, v) \in E$ if and only if $\|u - v\| = 1$. So all edges are of the form $((x, y), (x + 1, y))$ or $((x, y), (x, y + 1))$.

An *orthogonal grid embedding* of a planar graph is a drawing in which each vertex is a distinct vertex of a grid graph, and each edge is represented as a path of edges of that grid graph. Given a planar graph whose vertices have degree at most 4, the algorithm of Tamassia [10] computes an orthogonal grid embedding with area $O(n^2)$ in polynomial time.

2 Nibbler without walls

In Theorem 4, we show that it is NP–hard to decide whether a snake with growth rate $g \geq 1$, moving on a solid grid graph can consume all food.

▶ **Theorem 4.** Nibbler *on solid grid graphs is NP–hard for any constant growth rate* $g \geq 1$.

Proof. We reduce from the NP–complete Hamiltonian cycle problem on grid graphs (with holes) [9]. Let the graph G be an instance of this problem, see Figure 2 (a). We include G as a subgraph of a rectangular grid graph G', see Figure 2 (b).

The two leftmost vertices in the topmost row of G must be part of any Hamiltonian cycle (if there is only one vertex, then there is no Hamiltonian cycle), so we can attach a path (that is two vertices wide) on top of these vertices, and route it to the top left corner of G'. The resulting graph has a Hamiltonian cycle if and only if G has one. Hence, without loss of generality, we assume that the two topmost vertices in the leftmost column of G' are vertices of G and denote the topmost one by v. We may also assume that the rectangular grid graph G' has even width. We place food on all vertices of G' except those of G, and attach an extra vertex D with food to the right of the top–right corner of G. Because D is a dead end, the snake must collect the food in D last.

Let n be the (even) number of vertices of G, and attach a path C containing an initial snake P of length n to the left of v, such that the head of P faces v, see Figure 2 (c). On the opposite end of path C, we attach a rectangular area A of height 2 and width $n/2 + g > 2g$ with two vertices with food in the leftmost column of A.

The snake P is forced to enter G', and it must find a way to turn around in order to reach the food in A. If the snake consumes any food in G' before consuming the food in A, the snake will be trapped in A, and hence unable to consume the food in D. The reason the snake gets trapped is that the snake will be of length greater than n when entering A, and after eating the food in A, the tail of P will still be be blocking the exit of A when the head reaches the exit. Hence, the snake can reach A with length at most n if and only if it uses a

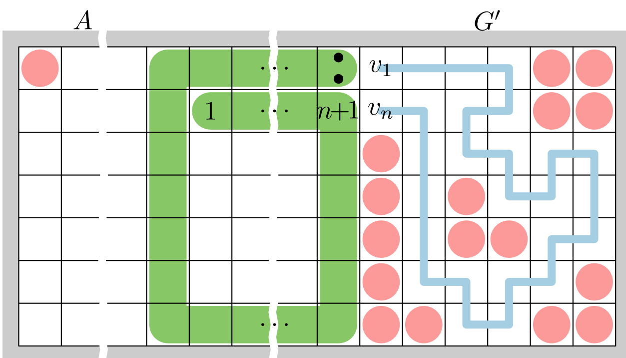


Figure 3 Snake on rectangular grid graphs.

Hamiltonian cycle in G. If and only if it reaches A with length at most n, it can consume the food in A and return to G'. It can then consume all remaining food in G' using a zig–zag motion (going up and down in columns of G' from left to right). Eventually, the head of the snake will be able to consume the food in D after consuming all other food. An analogous argument shows that if $g \geq 2$, a snake of (odd) length $n-1$ must use a Hamiltonian cycle to exit G' to consume all food. The case where $g = 1$ and the snake has odd length is handled using 3 food vertices in A. Hence, NIBBLER is NP–hard on solid grid-graphs for $g \geq 1$. ◀

Although the above construction shows hardness for snakes of arbitrarily large initial (even) length $|P| = n$ and odd length $|P| = n-1$, the construction extends to a setting in which the initial snake is short ($|P| \geq 3$). For this, the short snake is placed at the start of path C, and we place $\lfloor (n - |P|)/g \rfloor$ pieces of food in front of the snake, which the snake is then forced to consume. This grows the snake to length n or $n-1$, and inevitably results in the initial position of Theorem 4, so Corollary 5 follows.

▶ **Corollary 5.** NIBBLER *on solid grid graphs is NP–hard for any $|P| \geq 3$ and any constant growth rate $g \geq 1$.*

A *rectangular grid graph* is a (solid) grid graph whose vertex set is a complete rectangle $[1, \ldots, w] \times [1, \ldots, h]$. A second extension shows that consuming all food on a rectangular solid grid graphs is NP–hard for $g \geq 2$.

▶ **Theorem 6.** *It is NP–hard to decide if a snake with growth rate $g \geq 2$ can consume all food on a rectangular grid graph.*

Proof. As in Theorem 4, let grid graph G be an instance of the Hamiltonian cycle problem. Let G be a subgraph of a rectangular grid graph G' of width w and height h, with food on all vertices except those of G. We denote to the two topmost vertices of the leftmost row of G' as v_1 and v_n, and assume without loss of generality they are vertices of G, and thus empty. Let n be the number of vertices in G. If G has a Hamiltonian cycle, then it has a Hamiltonian cycle that starts at v_1, and v_n is visited last before returning to v_1.

We lay out the initial snake P of length $3n + 2h$ in a spiral of height h and width $n + 2$ next to G', such that its head lies next to v_1 and its last $n + 1$ vertices lie next to v_n; so if the snake does not consume food during the first n moves, the tail will lie next to v_n. Finally, we place a single food in the top-left corner of an area A of height h and width $3n + 2h$ that is placed to the left of the snake, see Figure 3.

We claim that if G' does not contain a Hamiltonian cycle, the snake cannot exit G'. Indeed, the first opportunity for the snake to exit occurs next to v_n, and only after $n + g \cdot k \geq n + 2k$

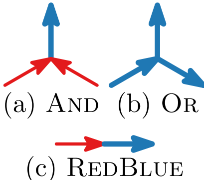


Figure 4 Three NCL vertex types.

moves, where k is the number of items consumed. Hence, to exit G', the snake must occupy at least $n + 2k$ vertices of G', which it can only do if $k = 0$ since there will only be $n + k$ vertices without food in G'. So to consume the food in A, the snake head must be on v_n after n moves without consuming food. This is possible only if G' contains a Hamiltonian cycle, which we may assume is $\langle v_1, \ldots, v_n, v_1 \rangle$. In that case, the snake can follow this cycle up to v_n; chase its tail until it can enter area A and consume the food contained in it and finally re–enter G' to collect the remaining food using a zig-zag motion as in Theorem 4. In the opposite direction, if G has a Hamiltonian cycle, then the snake can follow the same pattern described above to consume all food. ◀

Corollary 7 follows from extending area A and placing more food in it.

▶ **Corollary 7.** *For any positive constant fraction, it is NP–hard to decide if a snake with growth rate $g \geq 2$ can consume at least that fraction of food on a rectangular grid graph.*

3 Nondeterministic Constraint Logic

Nondeterministic constraint logic (NCL) is a framework by Hearn and Demaine [4] for proving the complexity of reconfiguration problems. An *NCL graph* is a graph whose edges all have weight 1 or 2 and all vertices have an *inflow constraint*, which is either 1 or 2. When drawing an NCL graph, we color the (thin) edges of weight 1 red, and the (fat) edges of weight 2 blue. We will use two types of vertices with an inflow constraint of 2, namely AND and OR vertices. An AND *vertex* has two red edges (of weight 1), and one blue edge (weight 2), whereas an OR *vertex* has three blue edges. We sometimes use an additional type of vertex, called a REDBLUE vertex, see Figure 4. A REDBLUE vertex has inflow constraint 1, and one red and one blue edge. An *oriented* NCL graph is a directed version of an NCL graph.

The *inflow* of a vertex is the sum of weights of inward directed edge, and an oriented NCL graph is *valid* if and only if each vertex has as inflow at least its inflow constraint. So for AND vertices, the blue edge can be directed outward only if both red edges are directed inward. For OR and REDBLUE vertices, at least one edge is directed inward. The NCL graphs we use in this paper consist of only AND, OR and REDBLUE vertices. We say that an NCL graph is in *normal form* if it uses only AND and OR vertices.

A *move* is an operation on a valid NCL graph that reverses the direction of one edge and a move is *valid* if it results in a valid NCL graph. Since reversing an edge twice does not change the graph, any valid move can be executed twice in succession to return to the original graph. Define the *configuration graph* of an NCL graph to be the graph of valid oriented NCL graphs, with an edge between two graphs if and only if a valid move between them exists; that is, if exactly one edge direction is different.

Given an initial NCL graph, the problem of reversing the direction of a target edge through a sequence of valid moves (see Definition 8) is PSPACE–complete as proven by Hearn

and Demaine [4] using a reduction from the quantified Boolean formula problem (QBF); the result holds even if the NCL graph is planar (see Theorem 9).

▶ **Definition 8** (NCL edge reversal problem: NclRev(G, e^*)).
Input. A valid NCL graph G with orientation o and an edge $e^* \in E_G$.
Output. Is there a sequence of valid moves, starting from o, that eventually reverses edge e^*?

▶ **Theorem 9.** NclRev *is PSPACE–complete, even for planar graphs in normal form.*

We generalize Theorem 9 to a setting where the edge directions of the initial graph can be chosen arbitrarily. In that case, finding an initial graph from which a given edge can be reversed is only NP–complete by reduction from the Boolean satisfiability problem (SAT). In contrast, we prove that reversing two edges in this setting (Definition 10) is PSPACE–complete in Theorem 12. We remark that this theorem was already known to hold for the instances resulting from the reduction from QBF [3]. In contrast, our alternative proof makes no assumptions on the NCL graphs used, and can hence generalize to other reductions, such as NCL graphs whose bandwidth is bounded by a constant, for which it was recently shown [12] that the NCL problem remains PSPACE–complete. If we do not require G to be in normal form, Theorem 13 shows that reversing *all* edges at least once (Definition 11) is also PSPACE–complete. Because the initial orientation can be guessed, this also holds if the initial graph is fixed (Corollary 14).

▶ **Definition 10** (Free NCL edge reversal problem: FreeNclRev(G, e^*, f^*)).
Input. An NCL graph G without orientation and two edges $e^*, f^* \in E_G$.
Output. Does there exist a valid initial orientation for the edges of G for which a sequence of valid moves reverses both e^* and f^* at least once (but not necessarily simultaneously)?

▶ **Definition 11** (Free NCL complete reversal problem: FreeNclRevAll(G)).
Input. An NCL graph G without orientation.
Output. Does there exist a valid initial orientation for the edges of G for which a sequence of valid moves reverses all edges of G at least once (but not necessarily simultaneously)?

▶ **Theorem 12.** FreeNclRev *is PSPACE–complete, even for planar graphs in normal form.*

▶ **Theorem 13.** FreeNclRevAll *is PSPACE–complete, even for planar graphs.*

▶ **Corollary 14.** FreeNclRevAll *is PSPACE–complete if the initial orientation is known.*

Before we prove Theorem 12, we construct gadgets that enforce the directions of a subset of edges in an NCL graph if a given edge is directed outward with respect to such gadget. We use these gadgets to enforce the initial edge directions of NclRev whenever a specific edge f^* is directed outward. Because the configuration graph has strongly connected components only, it is then PSPACE–complete to reverse both f^* and e^*.

3.1 Enforcing edge directions

Given an instance (G, e^*) of NclRev with NCL graph G in normal form with orientation o, we derive an instance (H, e^*, f^*) of FreeNclRev that has a solution if and only if the instance to NclRev has a solution. For any edge e of G, we create a copy $\phi(e)$ of that edge in H using the transformation of Figure 5. If $e \in E'$ is red (a), we first transform it into a blue edge (b), after which we transform that blue edge (c) into $\phi(e)$ (d). In the resulting graph, we have an edge f_e for each edge e of the original graph. If f_e is directed outward, then the direction of $\phi(e)$ must correspond to the direction of e in orientation o of G. If on

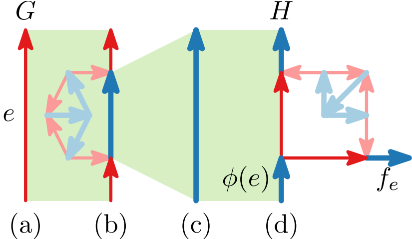


Figure 5 An edge e of weight 1 (a) or weight 2 (c) is transformed into edge $\phi(e)$ whose direction is fixed if edge f_e is directed outward (d). Light edges ensure that H is in normal form.

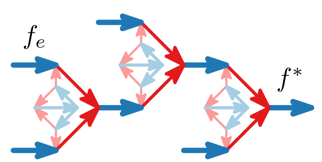


Figure 6 Three BRANCH gadgets can be used to connect four edges to f^*.

the other hand f_e is directed inward, then $\phi(e)$ behaves exactly as e does in G. We connect all f_e to a single edge f^* using $|E'| - 1$ BRANCH gadgets (see Figure 6), such that if f^* is directed outward (to the right in the figure), all f_e are directed outward; and if f^* is directed inward (to the left in the figure), then all f_e can be directed inward as well.

We ensure that the left endpoint of f^* can always be directed inward by connecting it to a so-called *free edge terminator* gadget. Planarity is ensured using *crossover* gadgets. Both the crossover and the free edge terminator gadgets [4] were introduced by Hearn and Demaine. Since their details are irrelevant for the proof of Theorem 12, we do not illustrate these gadgets. These gadgets ensure that H is planar and uses only AND and OR vertices, without changing the behavior of the NCL graph, or the edges representing $\phi(e^*)$ or f^*.

Proof of Theorem 12. We use a reduction from the PSPACE–complete NCLREV to prove that reversing two edges is PSPACE–complete if the initial orientation of an NCL graph can be chosen arbitrarily. Consider an NCL graph G with initial orientation o and edge e^* given by an instance of NCLREV. Construct the graph H as described above with edge f^* forcing for each e of G the direction of $\phi(e)$ of H to be that of e as given by o. We show that it is PSPACE–hard to reverse both f^* and $\phi(e^*)$ from an arbitrary initial orientation of H. Indeed, if f^* is reversed at some point, then the orientation of H at some point corresponds to the orientation of o for G. The sequence of moves connecting this orientation in H and the reversal of $\phi(e^*)$ corresponds to a sequence of valid moves in G that reverse e^* in G and vice-versa. Hence, deciding whether two edges can be reversed from an arbitrary initial orientation is PSPACE–hard. The problem is clearly in NPSPACE, and since PSPACE = NPSPACE by Savitch's theorem, FREENCLREV is PSPACE–complete. ◀

▶ Remark. Theorem 12 remains true for graphs with bandwidth bounded by a constant. To see this, observe that each edge of G is locally replaced by gadgets (transformed edges and crossovers) of constant size in H. Additionally, each BRANCH gadget can be placed near a distinct such transformed edge, and BRANCH gadgets can be connected in a linear fashion, so that the bandwidth of H grows by at most a constant factor with respect to that of G.

3.2 Relaxing edge directions

From now on, we base our constructions on the graph H constructed above, and refer to $\phi(e^*)$ simply as e^*. We use a second construction to allow arbitrary orientations for certain subsets of edges if a given edge is directed inward, and use this to prove Theorem 13. Because REDBLUE vertices are used in this construction, the theorem does not immediately extend to NCL graphs in normal form. The main tool used in the construction is the RELAX gadget, which has an input and an output edge, see Figure 7 (a). The input or output

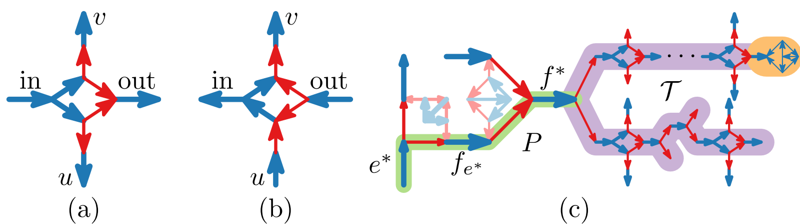


Figure 7 A Relax gadget with its input and output enabled (a) or disabled (b). Connecting f^* to a tree (c) of Relax gadgets using And vertices to connect outputs to inputs of children.

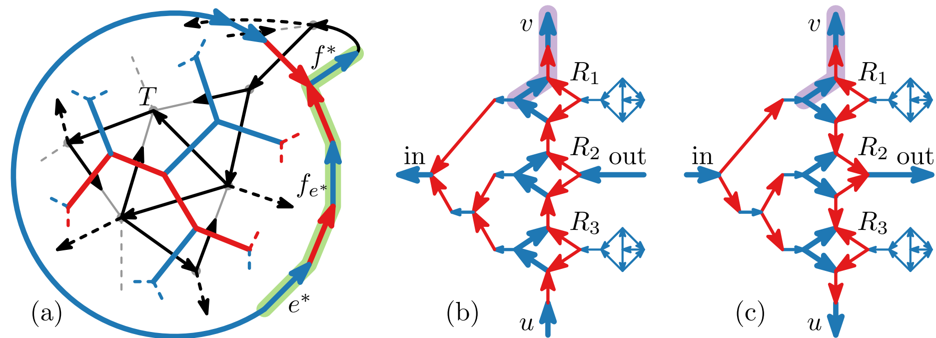


Figure 8 Schematically, tree T in black (a). A black arrow of (a) that crosses an edge (u, v) represents three Relax gadgets of $\mathcal{T}$, shown in detail in (b) and (c).

edge of a Relax gadget is *enabled*, respectively *disabled*, if it is directed as in Figure 7 (a), respectively (b). Any edge between u and v can be replaced by a Relax gadget between u and v (when replacing a red edge, we omit the top and bottom blue edges).

If the input of a Relax gadget is disabled, edges of the gadget cannot direct towards u and v simultaneously, and its output is also disabled. So the Relax gadget acts like a normal edge with respect to u and v if its input is disabled. On the other hand, if it is enabled, the output can be enabled and edges of the Relax gadget can direct into both u and v simultaneously. Multiple Relax gadgets can be connected to form a tree, illustrated in purple in Figure 7 (c). Inputs of gadgets in the tree can all be enabled if (and only if) the first is enabled. We connect the outputs of leaves of this tree to free edge terminators (highlighted in orange), such that these outputs can always be put in a disabled state.

Consider the graph H obtained from G in the construction for Theorem 12. We may assume that e^* and f^* lie on the outer face of the planar graph, and that e^* is connected to the Branch gadget closest to f^*, as illustrated in Figure 7 (c). We construct a graph H' by replacing the free edge terminator gadget connected to f^* in H by a tree $\mathcal{T}$ of Relax gadgets, such that only if f^* is directed into $\mathcal{T}$, inputs of Relax gadgets in $\mathcal{T}$ can be enabled. Let $\mathcal{T}$ denote the tree of Relax gadgets including the free edge terminators at its leaves. Let P be the path on Branch gadgets from f^* to e^* (highlighted in green). We will replace each edge of H', except those of P and $\mathcal{T}$, by three Relax gadgets of $\mathcal{T}$ (see Figure 8 (b)). We route the tree $\mathcal{T}$ such that planarity is preserved. To do this, we construct an ordinary tree T that acts as a backbone of the tree $\mathcal{T}$ of Relax gadgets. Take any spanning subtree T of the dual graph of a planar drawing of $H' \setminus \mathcal{T}$, such that T does not use any (dual) edges of P. Observe that such a tree exists since P is a strict subpath of the boundary of one face.

For each edge of $H' \setminus (P \cup \mathcal{T})$ that is not yet crossed by T, add an edge from either of the two incident vertices (of the dual graph) to a new leaf to T, so that the tree T is planar and

crosses exactly the edges of $H' \setminus (P \cup \mathcal{T})$. Let the root of T be in the (primal) face adjacent to (both sides of) f^*, such that f^* can be connected to the root of T. See Figure 8 (a). To obtain $\mathcal{T}$ from T, consider an edge t of T that crosses an edge (u, v) of H'. Replace the edge (u, v) by three Relax gadgets, and combine their inputs into one using two And vertices, see Figure 8 (c). The output of the middle Relax gadget along (u, v) is used to reach descendants of edge t of T (using And vertices if the target of t has multiple children). The other two Relax gadgets (and also the middle one if t ends in a leaf) are leaves of $\mathcal{T}$, so their outputs are connected to free edge terminators. Call the resulting graph H''.

Proof of Theorem 13. We claim that all edges of H'' can be reversed if and only if e^* and f^* can be reversed in H. If not both e^* and f^* can be reversed in H, then e^* and f^* in H'' can also not both be reversed in H'' because they are directly connected through P in H'', and no edge of P was replaced by a Relax gadget. To prove FreeNclRevAll is PSPACE–complete, we show that if both e^* and f^* can be reversed, so can all edges of H''.

Consider a sequence of moves in H that reverses both e^* and f^*, and apply the equivalent sequence to H''. This reverses e^* and f^* and therefore all edges of P in H''. Because all edges of H'' except those of P lie on $\mathcal{T}$, it suffices to show that all edges of $\mathcal{T}$ can be reversed. Because f^* was reversed, the first input of $\mathcal{T}$ can be reversed and hence enabled.

We show how the first edges of $\mathcal{T}$ can be reversed, and how they allow descendants to be reversed. For this, consider Figure 8 (b) and (c), which show in detail how an edge (u, v) was replaced in (a). Assume without loss of generality that the edges are directed as in (b), and that the leftmost edge can be reversed. In that case, we can reverse all edges but the three highlighted ones, and then direct the edges as in (c), such that the output of Relax gadget R_2 is enabled, and the descendants in $\mathcal{T}$ can be reversed in a similar way[1].

Doing so, we have for all of the edges (u, v) of H that were replaced in H'', that all but the highlighted edges near the target vertex (without loss of generality, v) have reversed in H''. If v does not lie on P, then it is incident to only Relax gadgets, that are now all directed into v, so v has sufficient inflow to reverse the highlighted edges in incident Relax gadgets and return them to state (c). If on the other hand, v lies on P, then the blue edge incident to v has reversed at some point, so the highlighted edges could also be reversed. ◀

4 Nibbler with walls

On (non-solid) grid graphs, we prove that several variants of Nibbler are PSPACE–hard using reductions from the problems of Section 3. Let G_{NCL} be a planar NCL graph (of n vertices) that we reduce from, and consider an orthogonal grid embedding G'_{NCL} of G_{NCL}. Recall that each edge of G_{NCL} is represented by a path of edges in G'_{NCL}, and we may assume that the embedding uses only $O(n^2)$ edges [10]. We ensure that paths of G'_{NCL} incident to any RedBlue vertex in G_{NCL} meet at 90 degrees by scaling G'_{NCL} by a constant factor, and rerouting an incident path whenever they meet at a RedBlue vertex at 180 degrees. Similarly, we ensure that the two red edges incident to an And vertex meet at 180 degrees.

We ensure that all paths (representing edges of G_{NCL}) between adjacent And, Or and RedBlue vertices of G_{NCL} have similar lengths by subdividing each path (representing an edge of G_{NCL}) by an even number of RedBlue vertices to obtain paths whose lengths differ by at most a constant factor. This graph of $O(n^2)$ edges can be scaled by a factor w to

[1] Note that if we had only replaced edges by a single Relax gadget instead of three, descendants would not always be able to reverse since we may need the edges of u and v to be reversible for descendants. Using three Relax gadgets allows the outputs of the R_1 and R_3 to be disabled during this propagation.

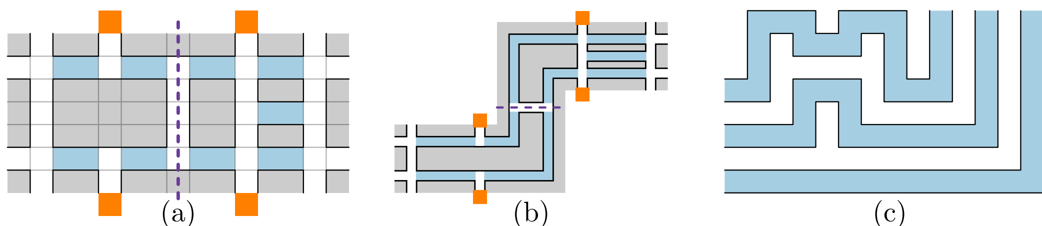


Figure 9 An edge gadget (a). Bundling blue paths (b). Routing bundles around corners ensures that bundled paths have equal length (c).

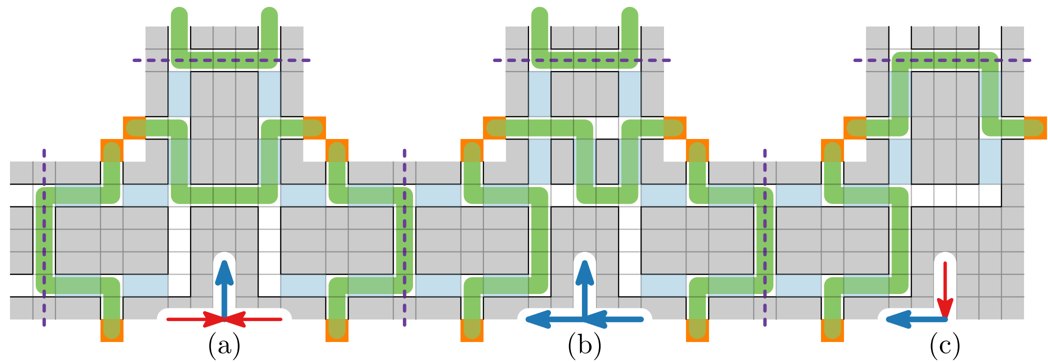


Figure 10 AND (a), OR (b) and REDBLUE (c) gadgets in NIBBLER, separated by dashed lines.

obtain a graph with $O(wn^2)$ edges, whose faces all contain $\Omega(w^2)$ grid points. By introducing detours on short paths, we obtain paths that differ in length by at most a constant.

We will use this embedding of G_{NCL} to obtain an instance G of NIBBLER as follows. First, we scale G'_{NCL} and replace all paths (edges of G_{NCL}) by *edge* gadgets, see Figure 9 (a), each consisting of two *half-edges* (separated by a dashed line). Groups of parallel blue shaded paths are routed along the desired paths in bundles of similar length (b). By routing paths around corners as in (c), we ensure that paths of the same bundle have equal length.

Figure 10 shows AND, OR, and REDBLUE gadgets of G, corresponding to vertices of G_{NCL}, and illustrates how these gadgets connect half-edges in G at these vertices. The figure also shows green paths that connect the orange *connector port pair* of each half-edge.

We denote the graph of half-edges and AND, OR, and REDBLUE gadgets as $\mathcal{G}$. In addition to $\mathcal{G}$, the graph G contains so-called *connector paths* (which we place in the faces of G'_{NCL}). These connector paths are long compared to the size of $\mathcal{G}$, and each of them connects two (orange) connector ports. In our constructions, the key idea is to lay out the snake P (green) to form a long cycle alternating between all the half-edges and connector paths of G, forcing the head of the snake to chase its tail in order to not get stuck. The snake's head may reroute the segments of the cycle inside the two half-edges of an edge gadget to reverse edges.

We place the connector paths in such a way that they, together with the segments of P inside the half-edges, form a simple closed path. By Lemma 15, a simple closed path in the plane exists that crosses each path of G'_{NCL} twice. This path can be drawn on the grid such that each crossing coincides with the connector port pair of a half-edge of $\mathcal{G}$, and the connector paths are routed along segments between two such crossings, see Figure 11. We introduce detours on connector paths, so that they all have length $\Omega(\frac{w^2}{wn^2}) = \Omega(\frac{w}{n^2})$, where w is the scaling factor of G'_{NCL}. We take $w = \Theta(n^4)$, so that all connector paths can be made significantly longer than the size of $\mathcal{G}$.

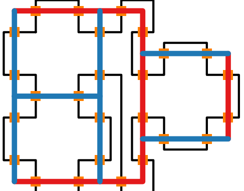


Figure 11 Black connector paths.

▶ **Lemma 15.** *For a planar embedding of a graph G, and a subset E' of its edges that are connected in the dual graph of G, there exists a simple closed path in the plane that intersects the edges of E' exactly twice, and no other edges.*

Proof. By induction on $|E'|$. The case where E' has at most one edge is trivial. In the case where $|E'| > 1$, there is an edge $e \in E'$ for which $E' \setminus \{e\}$ is connected in the dual graph. So by hypothesis, a simple closed curve intersects all edges of E' except e twice. This curve has a segment in a face adjacent to e, so we can deform the curve to also cross e twice. ◀

We are now ready to consider the problem of collecting two items ($|F| = 2$) with an initially long snake. Assume the growth rate g is any constant. Let S_P be the set of segments of P in $\mathcal{G}$. Call a path P *stable* if its head and tail lie outside $\mathcal{G}$, so all segments of S_P connect two connector ports. If the head and tail of a stable path P lie on the same connector path C, and $|S_P| \geq 1$, define the *head-tail distance* of P as the number of vertices in $C \setminus P$. Let h be the number of half-edges in $\mathcal{G}$. We call a stable path P *valid* if S_P contains h segments, each of which connects two connector ports of a single half-edge and each of which contains exactly two blue paths.

Recall that all blue paths differ in length by at most a constant, and denote by W the length of the shortest blue path, and let the longest blue path have length $W + c$ for some constant c. We derive a valid initial path P from the NCL instance, and ensure that the head-tail distance is $D = g \cdot |F| + b \cdot c$, where b is the number of blue paths in $\mathcal{G}$. The head-tail distance for any valid path reachable from P will then be between 0 and $D + b \cdot c \leq 2D$.

Observe that for some segment $s \in S_P$ of a valid path P, any connected component of $(\mathcal{G} \setminus S_P) \cup s$ consists of at most $R = 11(W + c) + 25$ vertices. We will ensure that connector paths are all longer than $R + 2D$ vertices, so if the head of P moves into a half-edge (through a connector port), then P can be reconfigured into stable a path only by reaching the other connector port of the same half-edge. If it does so by moving through fewer than two blue paths, then it cannot use any blue paths, so the head-tail distance will have decreased by at least $2W$ when the head reaches the other connector port.

We will ensure that $2W > 2D$, so the snake cannot reach a stable path before using at least two blue paths, since the head-tail distance cannot be negative. Observe that there are no simple paths (connecting the two connector ports at the ends of $s \in S$) that use more than two blue paths in $(\mathcal{G} \setminus S_P) \cup s$. Hence, for any stable path P' reachable from a valid path P, the segments of $S_{P'}$ all use exactly two blue paths. So we conclude that all such P' are valid based on the following assumptions, both of which can be satisfied by our choice of $w = \Theta(n^4)$, which scales G'_{NCL} sufficiently.

- The initial valid path P has a head-tail distance of $D = g \cdot F + b \cdot c < W$.
- Each connector path is longer than $R + 2D = 11(W + c) + 25 + 2D < 13W + 11c + 25$.

We refer back to Figure 10 and argue that all moves leading from one valid path to another correspond to valid edge reversals of the NCL graph. The direction of an edge of the

Figure 12 (a) Placing food in half-edges. (b) Valid connections. (c) Invalid connections.

NCL graph is encoded in P by the routes taken by the segments of S_P in the corresponding half-edges. We remark that whenever an edge of the NCL graph is directed into a vertex, the corresponding segment is routed away from the corresponding vertex in G, and vice-versa. It suffices to observe that for an Or or RedBlue gadget of G, the only constraint is that the segment of one of the incident half-edges is always routed away from the gadget. For an And gadget, the segments of both red half-edges must be routed away from the gadget whenever the blue half-edge is not routed away from it (as in the figure). Lemma 16 follows.

▶ **Lemma 16.** *It is PSPACE–complete to reach a valid path in which the segment of a half-edge (of the target edge of the NCL graph) is rerouted from a valid initial path in* Nibbler.

▶ **Corollary 17.** *Reaching a path P' from P is PSPACE–hard on grid graphs, even if $|F| = 0$.*

Using this lemma, we prove that collecting two items is PSPACE–complete, so let $|F| = 2$ and let $|P| \gg 1$ be large. Place one item in a blue path of the target half-edge, such that this item is collected when the segment of the half-edge changes its route. We use the second item to verify that a valid position was reached after collecting the first item. To do so, we place it in a dead end branching from a connector path, so that the second item must be collected last. Then both items can be collected if and only if a valid position is reached after reversing the target edge. Theorem 18 follows, but requires the initial snake to be long.

▶ **Theorem 18.** Nibbler *is PSPACE–complete for any constant growth rate $g \geq 0$ if $|F| = 2$.*

We wish to extend this theorem to a setting in which the initial snake is short, say $|P| = 1$. The idea is to use a lot of food (items) to grow the snake, and force it into a valid position, from which it must collect remaining food. As before, an item in a dead end branching from a connector path must be the final item consumed in any solution to Nibbler.

Let C and $C+c'$ (for some constant c') be the lengths of the shortest and longest connector paths, respectively, and assume without loss of generality that C is greater than $4|\mathcal{G}|$. We place the initial snake in a long path (with one dead end) that leads to the middle of a connector path, and fill this long path with $(h - \frac{1}{4})(C + c')/g$ food items (where h is the number of connector paths). When the tail of the snake leaves this path, the snake, call it P', must use part of each connector path, since otherwise more than $|\mathcal{G}|$ vertices of P' would not lie in a connector path, which is impossible. Moreover, the head and tail of P' lie in the same connector path. There is at least one blue path that is not used by P' since each segment of $S_{P'}$ blockades an endpoint of a blue path that is not used by that segment.

We place food items in half-edges as illustrated in Figure 12 (a), such that the snake cannot have consumed all items of $\mathcal{G}$ before its tail exits the initial long path. For a reduction from FreeNclRevAll, we claim that P' can only consume all items if P' is a valid path, and all edges of the corresponding NCL graph can be reversed from the corresponding valid orientation. Observe that valid paths reachable from P' can collect the food of exactly the edges reversible in FreeNclRevAll. This covers the case where P' is a valid path.

If P' is not a valid path, then a segment $s \in S_{P'}$ connects ports of different half-edges. In this case, we claim that no stable path can be reached after collecting all food of $\mathcal{G}$. We may assume that all edges of the NCL graph connect distinct pairs of vertices. If s connects ports of different half-edges h_1 and h_2, then h_1 and h_2 are half-edges of the same edge, or half-edges of the same AND, OR, or REDBLUE gadget. In general, if the food in the first and last blue path adjacent to s but not used by s is consumed, the snake can no longer reach a stable path. Figure 12 (c) shows these items in black. So all food can be consumed only if P' is a valid path (and if and only if there is a solution to FREENCLREVALL).

▶ **Theorem 19.** NIBBLER *is PSPACE–complete for any growth rate* $g \geq 1$*, even if* $|P| = 1$.

References

1. Elias Dahlhaus, Peter Horák, Mirka Miller, and Joseph F. Ryan. The train marshalling problem. *Discrete Applied Mathematics*, 103(1-3):41–54, 2000. doi:10.1016/S0166-218X(99)00219-X.
2. Erik D. Demaine and Robert A. Hearn. Playing games with algorithms: Algorithmic combinatorial game theory. In *Games of No Chance 3*, volume 56 of *Mathematical Sciences Research Institute Publications*, pages 3–56. Cambridge University Press, 2009.
3. Robert A Hearn and Erik D Demaine. PSPACE-completeness of sliding-block puzzles and other problems through the nondeterministic constraint logic model of computation. *Theoretical Computer Science*, 343(1):72–96, 2005.
4. Robert A. Hearn and Erik D. Demaine. *Games, puzzles and computation.* A K Peters, 2009.
5. Graham Kendall, Andrew J. Parkes, and Kristian Spoerer. A survey of NP-complete puzzles. *ICGA Journal*, 31(1):13–34, 2008.
6. Irina Kostitsyna and Valentin Polishchuk. Simple wriggling is hard unless you are a fat hippo. *Theory of Computing Systems*, 50(1):93–110, 2012. doi:10.1007/s00224-011-9337-4.
7. J.-C. Latombe. *Robot Motion Planning.* Kluwer, Boston, MA, 1991.
8. Amit Pamecha, Imme Ebert-Uphoff, and Gregory S. Chirikjian. Useful metrics for modular robot motion planning. *IEEE Transactions on Robotics and Automation*, 13(4):531–545, 1997. doi:10.1109/70.611311.
9. Christos H. Papadimitriou and Umesh V. Vazirani. On two geometric problems related to the traveling salesman problem. *Journal of Algorithms*, 5(2):231–246, 1984. doi:10.1016/0196-6774(84)90029-4.
10. Roberto Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM Journal on Computing*, 16(3):421–444, 1987. doi:10.1137/0216030.
11. Christopher Umans and William Lenhart. Hamiltonian cycles in solid grid graphs. In *FOCS*, pages 496–505, 1997. doi:10.1109/SFCS.1997.646138.
12. Tom C. van der Zanden. Parameterized Complexity of Graph Constraint Logic. In *IPEC*, pages 282–293, 2015. doi:10.4230/LIPIcs.IPEC.2015.282.
13. Giovanni Viglietta. Gaming is a hard job, but someone has to do it! *Theory of Computing Systems*, 54(4):595–621, 2014. doi:10.1007/s00224-013-9497-5.
14. Zhi Yan, Nicolas Jouandeau, and Arab Ali Cherif. A survey and analysis of multi-robot coordination. *International Journal of Advanced Robotic Systems*, 10:399, 2013.

The Fewest Clues Problem

Erik D. Demaine[1], Fermi Ma[2], Ariel Schvartzman[3], Erik Waingarten[4], and Scott Aaronson[5]

1 **MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar St, Cambridge, MA 02139, USA**
edemaine@mit.edu

2 **Department of Computer Science, Princeton University, 35 Olden St, Princeton, NJ 08544, USA**
fermim@princeton.edu

3 **Department of Computer Science, Princeton University, 35 Olden St, Princeton, NJ 08544, USA**
acohenca@cs.princeton.edu

4 **Department of Computer Science, Columbia University 1214 Amsterdam Ave, New York, NY 10027, USA**
eaw@cs.columbia.edu

5 **MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar St, Cambridge, MA 02139, USA**
aaronson@csail.mit.edu

Abstract

When analyzing the computational complexity of well-known puzzles, most papers consider the algorithmic challenge of *solving* a given instance of (a generalized form of) the puzzle. We take a different approach by analyzing the computational complexity of designing a "good" puzzle. We assume a puzzle maker designs part of an instance, but before publishing it, wants to ensure that the puzzle has a unique solution. Given a puzzle, we introduce the FCP (fewest clues problem) version of the problem:

> Given an instance to a puzzle, what is the minimum number of clues we must add in order to make the instance uniquely solvable?

We analyze this question for the Nikoli puzzles Sudoku, Shakashaka, and Akari. Solving these puzzles is NP-complete, and we show their FCP versions are Σ_2^P-complete. Along the way, we show that the FCP versions of 3SAT, 1-in-3 SAT, Triangle Partition, Planar 3SAT, and Latin Square are all Σ_2^P-complete. We show that even problems in P have difficult FCP versions, sometimes even Σ_2^P-complete, though *"closed under cluing"* problems are in the (presumably) smaller class NP; for example, FCP 2SAT is NP-complete.

1998 ACM Subject Classification F.1.3 Complexity Measures and Classes

Keywords and phrases computational complexity, pencil-and-paper puzzles, hardness reductions

Digital Object Identifier 10.4230/LIPIcs.FUN.2016.12

1 Introduction

A natural aesthetic for designing pencil-and-paper puzzles is to provide as few starting hints as possible, yet have the resulting solution be unique. For example, in the well-known Sudoku puzzle, the starting configuration is a partially filled 9×9 grid where the goal is to fill in the remaining entries. Hard Sudoku puzzles tend to give very few numbers in the starting

© Erik D. Demaine, Fermi Ma, Ariel Schvartzman, Erik Waingarten, and Scott Aaronson; licensed under Creative Commons License CC-BY
8th International Conference on Fun with Algorithms (FUN 2016).
Editors: Erik D. Demaine and Fabrizio Grandoni; Article No. 12; pp. 12:1–12:12
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

configuration, but must still give enough numbers to ensure that the puzzle has a unique solution. The natural question here is, how small can we make the starting configuration?

In this paper, we formalize this question by the family of "Fewest Clues Problem" (FCP). Given an NP search problem where the certificate is written as a string, we ask: is there a setting of at most k characters of the certificate such that there exists exactly one way to complete the certificate (fill in the remaining characters)?

FCP seems superficially related to two other classes of problems, the "Another Solution Problem" (ASP) and counting (#) problems [7, 10, 11, 12, 13]. The ASP problem asks: given an instance as well as one solution to a search problem, is there another solution? Counting problems ask: given an instance, how many solutions are there? All three of these problem types are transformations of an original NP search problem.

ASP versions of NP-hard problems are in NP by definition, and interestingly, there are NP-hard problems whose ASP versions are NP-hard as well. The counting version of a problem can be significantly harder the original problem. It is well known that some problems in P (such as 2SAT) have #P-hard counting versions, and Toda's theorem states that such problems are as hard as the polynomial hierarchy [9].

Our results. This paper has three main contributions:

1. We introduce the FCP framework for analyzing puzzles, and we develop simple techniques to adapt existing hardness reductions to the FCP setting (Section 2). With these techniques, we show that the FCP versions of common NP-complete problems are Σ_2^P-complete. These include FCP 3SAT, FCP 1-in-3 SAT, FCP Triangle Partition, FCP Planar 3SAT, and FCP Latin Square (Section 3.2).
2. We show that the FCP versions of three common Nikoli puzzles (Sudoku, Shakashaka, and Akari) are Σ_2^P-complete by reducing from the above problems (Section 4). Figure 1 shows a chain of reductions which allow us to conclude these problems are Σ_2^P-complete.
3. We analyze how the computational complexity of problems in P changes when we consider their FCP versions (Section 5). For a class of problems *closed under cluing*, their FCP versions are in NP. In addition, we show FCP 2SAT, which naturally falls in the class of problems closed under cluing, is NP-complete. We give an example of a problem in P whose FCP version is Σ_2^P-complete.

2 Definitions and Overview

2.1 FCP Framework

▶ **Definition 1.** For each $A \in \mathsf{NP}$, we associate an NP relation, R_A the set of instance-certificate pairs.

For example, the NP relation for the Boolean satisfiability problem (SAT) consists of pairs (ϕ, x) where ϕ is a Boolean formula and x is an assignment to the variables where $\phi(x)$ evaluates to true. We assume that instances and certificates are strings from some alphabet Σ. From now on, when referring to some $A \in \mathsf{NP}$, we implicitly consider the NP relation, R_A.

If the string x is a solution to a SAT problem, we call a "partially filled" version of x (constructed by blocking out certain characters of x with a $\perp$ symbol) a *clue* for that problem. We define this notion for all NP problems as follows.

▶ **Definition 2.** For any instance I of a problem $A \in \mathsf{NP}$, a clue is a string $c = (c_i) \in (\Sigma \cup \{\perp\})^*$ if there exists some certificate string $y = (y_i) \in \Sigma^*$ with $(I, y) \in A$ such that if

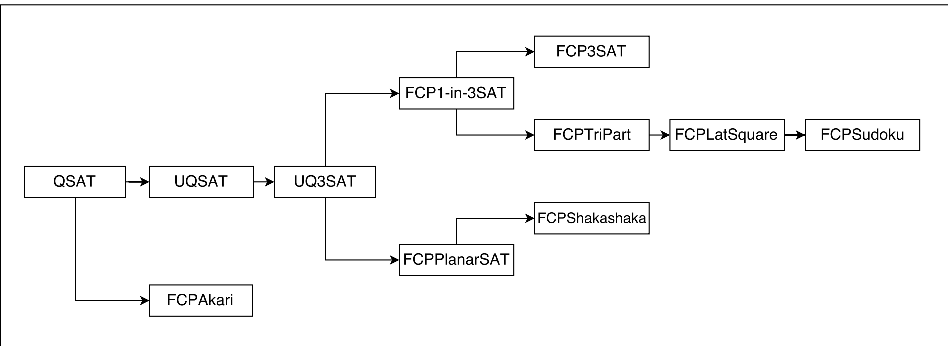


Figure 1 Chain of reductions from QSAT to FCP versions of satisfiability problems and FCP versions of pencil-and-paper puzzles.

$c_i \neq \perp$, then $c_i = y_i$. The size of a clue is the number of non-$\perp$ characters. We refer to this y as a solution which *satisfies* the clue c, and denote this as $c \subset y$.

▶ **Definition 3.** For $A \in$ NP, FCP A is the following decision problem:

Given $x \in \Sigma^*$ and an integer k, does there exist a clue c of size at most k with only one satisfying solution to x?

2.2 FCP Versions of Classic Problems

In Section 3.1, we show that the FCP version of any NP problem is in Σ_2^P. Our more compelling result is that the FCP versions of many NP-complete problems are themselves Σ_2^P-complete. We show this by using a variety of techniques that modify existing NP-completeness reductions to give Σ_2^P-completeness results for the corresponding FCP problems.

We first prove Σ_2^P-completeness of the FCP versions of many common SAT variants. This will be useful since many known NP-completeness reductions are from variants of SAT. We show that FCP 3SAT, FCP 1-in-3 SAT, and FCP Planar 3SAT are all Σ_2^P-complete. We obtain these hardness results with a chain of reductions, starting with the canonical Σ_2^P-complete problem, quantified satisfiability with one pair of alternating quantifiers, $\exists$ and $\forall$ (QSAT). The chain of reductions for all problems defined in this subsection and Section 2.3 are shown in Figure 1.

A slight issue with constructing this chain of reductions is that QSAT does not have any notion of uniqueness, which is essential in FCP problems. We overcome this by defining the unique quantified satisfiability problem (UQSAT), a problem we show is Σ_2^P-complete. Unfortunately, the reduction will introduce clauses which are longer than three literals. For our hardness reductions, we often want to work with 3-CNF formulae. Thus we introduce UQ3SAT, another provably Σ_2^P-complete problem.

For the following three problems, $\phi(x, y)$ is a Boolean function in CNF form on the sets of variables x and y.

QSAT: Given $\phi(x, y)$, does there exist an assignment of the variables in x, such that for all assignments of variables in y, $\phi(x, y) = 1$?

UQSAT: Given $\phi(x, y)$, does there exist an assignment of the variables in x, such that there exists a unique assignment of the variables in y with $\phi(x, y) = 1$?

UQ3SAT: Given $\phi(x, y)$ in 3-CNF form, does there exist an assignment of the variables in x, such that there exists a unique assignment of the variables in y with $\phi(x, y) = 1$?

The transition to FCP problems relies on the following observation: while UQ3SAT specifies which variables can be assigned, FCP 3SAT allows any k variables to be assigned.

We address this distinction by building *assign gadgets*; these gadgets are built so that specifying some part of them fixes in place other aspects of the reduction. These assign gadgets also have the property of having inherent ambiguity. Thus, any clue must specify something within the assign gadget to resolve this ambiguity, or else a unique solution will not be possible. If we place exactly k assign gadgets, we will need a clue of size at least k. This technique will be useful for reducing from non-FCP problems to FCP problems.

For the following three problems, $\phi(x)$ is a Boolean formula in 3-CNF form.

FCP 1-in-3 SAT: Given $\phi(x)$ and a number k, does there exist a partial assignment of at most k variables such that the remaining formula ϕ' has only one satisfying assignment, where each clause has exactly one true literal?

FCP Planar 3SAT: Given a planar $\phi(x)$ and a number k, does there exist a partial assignment of at most k variables such that the remaining formula ϕ' has only one satisfying assignment?

FCP 3SAT: Given $\phi(x)$ and a number k, does there exist a partial assignment of at most k variables such that the remaining formula ϕ' has only one satisfying assignment?

We use assign gadgets to show FCP 1-in-3 SAT is Σ_2^P-complete. The same technique will show that FCP Planar 3SAT is also Σ_2^P-complete.

In some cases, an NP-hardness reduction from problem A to problem B may already preserve clue structure without needing any modifications. If FCP A is Σ_2^P-hard, then the same reduction implies that FCP B is Σ_2^P-hard. This method will show that FCP 3SAT is Σ_2^P-complete.

2.3 FCP Versions of NP-hard Puzzles

In this section, we introduce a number of NP-hard pencil-and-paper puzzles. We modify NP-hardness reductions for these problems to show that their FCP versions are Σ_2^P-hard. These problems were chosen because their NP-hardness reductions mostly preserve clue structure. Figure 1 summarizes the chain of reductions.

FCP Latin Square: Given a partially filled Latin Square and a number k, do there exist k additional squares to fill in such that the remaining puzzle has a unique solution?

FCP Sudoku: Given a partially filled Sudoku board and a number k, do there exist k additional squares to fill in such that the remaining puzzle has a unique solution?

FCP Shakashaka: Given a Shakashaka board and a number k, do there exist k clues for the board such that the remaining board has a unique solution?

FCP Akari: Given an Akari board and a number k, do there exist k clues to the board such that the remaining board has a unique solution?

There is a fundamental difference between Shakashaka and Akari, and Sudoku. In the former, there is a clear distinction between problem instance and solution, whereas in the latter, this distinction is less clear. In particular, filling in a clue in Sudoku and Latin Squares simply produces a new instance of the problem, whereas filing in clues for Shakashaka and Akari do not create new problem instances.

2.4 FCP Versions of Easy Problems

It is also interesting to consider the FCP transformation applied to problems in P. Here, we do not find many general hardness bounds; instead, the results are problem-dependent.

We provide an upper bound for the class of problems in P where filling in a clue results in another instance of the problem. The FCP versions of such problems are always in NP. As an example of such a problem, consider 2SAT.

FCP 2SAT: Given a Boolean formula $\phi(x)$ written in 2-CNF form, and a number k, does there exist a partial assignment of at most k variables such that the remaining formula ϕ' has only one satisfying assignment?

In this problem, filling in a clue means assigning truth values to certain variables of ϕ, which results in another instance of 2SAT. Thus, our upper bound will imply that FCP 2SAT is in NP. In fact, this bound is tight as we show that FCP 2SAT is NP-complete by reduction from Minimum Independent Dominating Set.

However, for problems in P that do not have this property, the hardness of their FCP versions cannot be so easily characterized. This happens because we can modify hard problems by planting solutions to make them "easy", while maintaining the hardness of their FCP versions. We give a simple example of a problem in P whose FCP version is Σ_2^P-hard.

3 The FCP Transformation

3.1 Upper Bounds

▶ **Theorem 4.** *If $L \in$ NP, then FCP $L \in \Sigma_2^P$.*

Proof. We write FCP L as an instance of QSAT, a problem in Σ_2^P. The original problem of deciding membership in L can be written as

$$x \in L \leftrightarrow \exists y : A(x, y) = 1,$$

where A is a polynomial time algorithm and $|y|$ is polynomial in $|x|$.

FCP L asks: for a given instance x and positive integer k, does there exist a clue c of size at most k such that there is only one solution y with $c \subset y$ and $(x, y) \in R$? We write this in logical notation as

$$(x, k) \in \text{FCP } L \leftrightarrow \exists c, y : \forall y' : A'(x, c, y, y') = 1.$$

A' simply checks that $c \subset y$, c is of size at most k (c is a valid clue of the correct size), $c \subset y' \neq y$ (y' is another possible solution), $A(x, y) = 1$ (y is a solution), and $A(x, y') = 0$ (all other possible solutions do not work). ◀

3.2 Lower Bounds

In this subsection we present a chain of Σ_2^P-hardness reductions starting with UQSAT and UQ3SAT. These results allow us to transition to Σ_2^P-hardness proofs of various FCP problems.

▶ **Lemma 5.** UQSAT *is Σ_2^P-hard.*

Proof. We reduce from QSAT. We define a function f from instances of QSAT to instances of UQSAT which maps $\phi(x, y)$ to $\phi'(x, y, z)$, where z is one additional variable. We show

$$\exists x : \forall y : \phi(x, y) = 1 \leftrightarrow \exists x : \phi'(x, y, z) \text{ has a unique solution.}$$

Let $\phi'(x, y, z) = (\overline{z} \wedge \bigwedge_i \overline{y_i}) \vee \overline{\phi(x, y)}$.

Note that for each assignment of x, $\phi'(x, 0, 0) = 1$. If there exists some x for which $\phi(x, y) = 1$ always, then for this x, $\overline{\phi(x, y)} = 0$ always. For this x, the only way to satisfy ϕ' is to set z and each y_i to 0. It remains to rewrite ϕ' as a CNF formula.

The given $\phi(x, y)$ is in CNF form, so we write $\overline{\phi(x, y)}$ in DNF form using De Morgan's laws. For the ith clause, we introduce the variable c_i to represent whether that clause is satisfied. If there are n clauses, $\overline{\phi(x, y)}$ becomes $(c_1 \vee c_2 \vee \cdots \vee c_n)$. We write the jth literal in the ith clause as $l_{i,j}$, and introduce A as one additional variable. Then the following formula is logically equivalent to ϕ':

$$\phi''(x, y, z, \{c_i\}, A) = (\overline{A} \vee \overline{z}) \wedge \bigwedge_i (\overline{A} \vee \overline{y_i}) \wedge \bigwedge_i \bigwedge_j (\overline{c_i} \vee l_{i,j}) \wedge (A \vee \bigvee_i c_i).$$

Then the following two claims complete the proof.

Claim. Suppose for some assignment x, $\phi(x, y) = 1$ for all y. Then $\phi''(x, y, z, \{c_i\}, A)$ has a unique satisfying assignment.

Since $\phi(x, y) = 1$, $\overline{\phi(x, y)} = 0$. This implies that each $l_{i,j} = 0$ and therefore each $c_i = 0$. This forces $A = 1$, $z = 0$, and $y_i = 0$ for all i.

Claim. If for each x there exists some y where $\phi(x, y) = 0$, then after fixing x, $\phi''(x, y, z, \{c_i\}, A)$ has more than one satisfying assignment.

Setting each $y_i = 0$, $z = 0$, $c_i = 0$ for each i and $A = 1$ gives one satisfying assignment. If there is a setting of the y_i's which makes some of the clauses true, then for the true clauses set $c_i = 1$. Then set $A = 0$ and z to either 0 or 1. This gives additional satisfying assignments for ϕ''. ◀

▶ **Lemma 6.** UQ3SAT *is* Σ_2^P*-hard.*

Proof. The reduction is from UQSAT. Given $\phi(x, y)$, an instance of UQSAT, it remains to transform $\phi(x, y)$ into 3-CNF form. For clauses in $\phi(x, y)$ of length $m > 3$, we imagine a binary tree of height $\log m$ where leaves correspond to clause literals. For all interior nodes of the tree, we introduce a new variable to act as the OR of its two children. If an interior node has children i_1 and i_2, we introduce the variable o along with the following clauses:

$$(\overline{i_1} \vee \overline{i_2} \vee o), \quad (\overline{i_1} \vee i_2 \vee o), \quad (i_1 \vee \overline{i_2} \vee o), \quad (i_1 \vee i_2 \vee \overline{o}).$$

These clauses ensure that o is the OR of i_1 and i_2. Additionally, we add the variable at the root of the tree as a new clause of size 1 to capture the original clause exactly.

To handle clauses with fewer than three literals, we pad the clause with the additional variable a along with the clause $(\overline{a} \vee \overline{a} \vee \overline{a})$ to ensure that a is set to false. ◀

▶ **Lemma 7.** FCP 1-IN-3 SAT *is* Σ_2^P*-complete.*

Proof. The reduction is from UQ3SAT. We use the reduction from Proposition 3.3 in [6] which is a parsimonious reduction from PLANAR 3SAT to POSITIVE PLANAR 1-IN-3 SAT. For this problem, we disregard the planarity and note that the reduction works to map non-planar instances of 3SAT to 1-IN-3 SAT.

We modify the reduction slightly by constructing the assign gadget from Figure 2 for each variable in x. This assign gadget is designed so that any clue to the problem must resolve an ambiguity within the gadget, forcing every clue to assign a value to each variable in x. Therefore, even though FCP 1-IN-3 SAT can give clues for any variable, because of the assign gadgets, clues will only correspond to variables with existential quantifiers in UQ3SAT. ◀

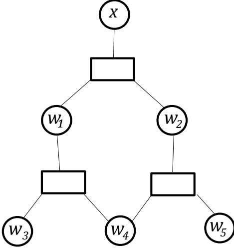


Figure 2 Clue gadget for 1-IN-3 SAT. Each box represents a 1-IN-3 SAT clause with variables corresponding to the three lines connected to the box. x is set to true by setting w_4 to true. x is set to false by setting w_1 or w_2 to true (but not both). If none of the w_i in the clue gadget is assigned, there exists more than one satisfying assignment.

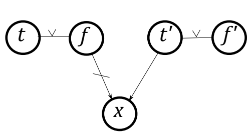


Figure 3 FCP PLANAR 3SAT assign gadget for variable x. The above diagram is written with 4 clauses: $(t \vee f) \wedge (\overline{f} \vee \overline{x}) \wedge (\overline{t'} \vee x) \wedge (t' \vee f')$. The arrows indicate "causal" relationships (if t' is set to true, then x must be assigned to true; if f is set to true, then x must be assigned to false). In order to assign x to true, set f' to false. In order to assign x to false, set t to false. If neither of the t, t', f, or f' variables are set, then the solution is not unique.

▶ **Lemma 8.** *FCP 3SAT is Σ_2^P-complete.*

Proof. We reduce from FCP 1-IN-3 SAT. For each clause of the form $(l_1 \vee l_2 \vee l_3)$ in the FCP 1-IN-3 SAT instance, we construct the following clauses:

$$(l_1 \vee l_2 \vee l_3), \quad (l_1 \vee \overline{l_2} \vee \overline{l_3}), \quad (l_2 \vee \overline{l_1} \vee \overline{l_3}), \quad (l_3 \vee \overline{l_1} \vee \overline{l_2}), \quad (\overline{l_1} \vee \overline{l_2} \vee \overline{l_3}).$$

Note that if $(l_1 \vee l_2 \vee l_3)$ is true in the 1-IN-3 SAT configuration, all of the above clauses are satisfied. Conversely, if all of the above clauses are satisfied, then the clause $(l_1 \vee l_2 \vee l_3)$ has exactly one variable set to true. ◀

▶ **Lemma 9.** *FCP PLANAR 3SAT is Σ_2^P-complete.*

Proof. The reduction is from UQ3SAT. We use the reduction from [4] showing PLANAR 3SAT is NP-complete. The only change is that we introduce clauses to form an assign gadget as shown in Figure 3. The assign gadget guarantees variables with existential quantifiers in UQ3SAT are given clues. ◀

4 FCP Versions of NP-hard Puzzles

In this subsection we show that the FCP versions of popular puzzles are Σ_2^P-complete. It is worth noting that all problems considered in this section are in NP, so their FCP versions are in Σ_2^P due to Theorem 4. Therefore, we focus on obtaining Σ_2^P-hardness reductions.

4.1 Triangle Partition

The Triangle Partition problem is the following:

Given an undirected graph $G = (V, E)$, can we partition E into disjoint triangles?

Holyer [3] showed this problem is NP-complete. We use elements of Holyer's reduction to show FCP Triangle Partition is Σ_2^P-complete. We use the notation as well as elements from the reduction in [3].

▶ **Theorem 10.** *FCP Triangle Partition is Σ_2^P-complete.*

Proof. In order to show FCP Triangle Partition is Σ_2^P-hard, we reduce from FCP 1-IN-3 SAT. We use Holyer's reduction, which reduces 3SAT to Triangle Partition [3]. For the FCP setting, we switch from 3 SAT to 1-IN-3 SAT so that it is not advantageous to provide clues within clause gadgets. The other modification we make is that instead of a single two-way join between literals and variables, we make two two-way joins in other to enforce the variable matches the literal exactly. Since the reduction is from 3SAT in [3], and the three-way join enforces a 1-IN-3 SAT constraint, [3] cannot map the literals and variables directly. Since we are reducing from 1-IN-3 SAT constraints, we can make the literals and variable gadgets be the exact same partition.

This completes the proof, as now a clue on variables in the 1-IN-3 SAT instance corresponds exactly to a clue of the same size on the variable gadgets in the Triangle Partition instance. ◀

4.2 Latin Squares

The Latin Squares problem is the following:

> Given an $n \times n$ grid, with some entries filled in with the numbers 1 to n, can we fill in the remainder of the grid with numbers from 1 to n so that no row or column repeats a number?

Colbourn proved this problem was NP-complete by showing it is equivalent to the Triangle Partition problem restricted to tripartite graphs [1]. We show FCP Latin Squares is Σ_2^P-complete.

▶ **Theorem 11.** *FCP Latin Squares is Σ_2^P-complete.*

Proof. We show hardness via a reduction from FCP Triangle Partition of Tripartite Graphs which is Σ_2^P-complete following Theorem 10 and [1]. The reduction is identical to the one in [1]. The mapping sends Latin Squares entries of the form $L(i,j) = m$ to triangles with vertices (i,j,m) on different parts of the partition, where $L(i,j)$ is the entry on the i-th row and j-th column of the Latin Square L. The mapping between entries and triangles is bijective, implying that a clue of size k to the Latin Squares problem will produce a clue of size k in the Triangle Partition problem. ◀

4.3 Sudoku

The Sudoku problem is as follows:

> Given an $n^2 \times n^2$ grid, divided into n^2 subgrids of size $n \times n$, can we fill in the grid with numbers from $[n^2]$ so that no row, column, or any of the n^2 subgrids contain any repeated numbers?

We will show the FCP problem is Σ_2^P-complete by reduction from FCP Latin Squares. The Latin Square is embedded into the Sudoku puzzle. The columns of the Latin Square are mapped to the parts of columns of the Sudoku. In the mapping, the values are scaled by n.

▶ **Lemma 12.** *FCP Sudoku is Σ_2^P-complete.*

Proof. We note the same reduction as [8] will serve as a reduction from FCP Latin Square to FCP Sudoku. Each entry in the Sudoku puzzle is mapped to an entry in the Latin Squares problem, after diving by n. Therefore, a clue of size k in the constructed Sudoku puzzles can be mapped to a clue of size k in the Latin Squares problem. Figure 4 shows the case when $n = 3$. ◀

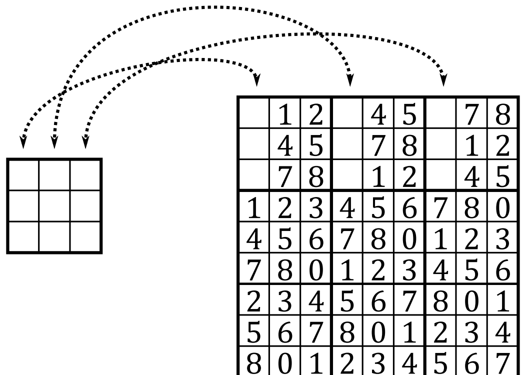


Figure 4 Reduction from Latin Square to Sudoku for $n = 3$ from [8].

4.4 Shakashaka

Shakashaka is a pencil-and-paper puzzle published by Nikoli. A Shakashaka puzzle consists of a rectangular grid of black and white squares. Black squares either contain an integer from $\{0, 1, 2, 3, 4\}$ or can be left blank. Each white square is filled in with a black triangle or is left blank. The filled-in squares become half-black with right isosceles triangles in any of four possible directions (referred to as b/w squares in [2]). The resulting configuration must satisfy the following constraints: each black square with a number c must have c adjacent black isosceles triangles, and the remaining white area must be partitioned into (possibly rotated) rectangles.

The problem is shown to be NP-complete in [2]. The same reduction will show that FCP Shakashaka is Σ_2^P-complete.

▶ **Lemma 13.** *FCP Shakashaka is Σ_2^P-complete.*

Proof. We use the reduction from [2] to reduce from FCP Planar 3SAT. It remains to show that all clue structure of the original problem is preserved in the reduction. Shakashaka entries are comprised of assignments (either to one of the four possible triangle orientations or to be left blank) to the white squares. The key observation is that any assignment to a white square in the reduction can be implied by an assignment in the variable gadget of [2]. Therefore, we can assume without loss of generality that each assignment in an entry is in a unique variable gadget. Thus, the entries correspond exactly to assignments on variables in FCP Planar 3SAT, completing the reduction. ◀

4.5 Akari

Akari, or Light Up, is a pencil-and-paper puzzle consisting of a grid with light and dark squares. Dark squares may be labelled with integers from $\{0, 1, 2, 3, 4\}$. The player places lightbulbs on light squares which illuminate all light squares in the four directions around the lightbulb (although light stops upon a dark square). Equivalently, a square is lit if there exists at least one light bulb connected to it by a straight line of light squares. The goal is to place lightbulbs so that every light square is lit, no two light bulbs illuminate each other, and every numbered dark square has exactly that many light bulbs adjacent to it.

This problem is known to be NP-complete. We show that its FCP version is Σ_2-complete.

▶ **Lemma 14.** *FCP Akari is Σ_2^P-complete.*

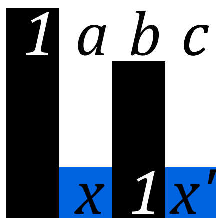


Figure 5 Akari entry gadget. If x or x' is lit, then there is ambiguity regarding placing the light in b or c, in the case of x, and a or b in the case of x'. Lighting up a means x' must be lit, and lighting up c means x must be lit.

Proof. In order to show FCP Akari is Σ_2^P-complete, we reduce from Σ_2 Circuit SAT, using the reduction from [5]. We add the following assign gadgets to a wire to assign the necessary variables. The gadget is shown in Figure 5. Variables with existential quantifiers will need to receive clues to resolve ambiguity. Therefore, clues in Akari will only be given to variables corresponding to existential quantifiers in Σ_2 Circuit SAT. ◀

5 FCP Versions of Easy Problems

In this section, we consider the complexity of the FCP versions of various problems in P.

5.1 Problems Closed under Cluing

▶ **Definition 15.** We call a problem *closed under cluing* if plugging any partial solution into a problem instance results in another instance of the same problem.

For example, 2SAT is closed under cluing since plugging in values for some of the variables will result in another instance of 2SAT.

▶ **Proposition 16.** *If A is a problem closed under cluing in P, then* FCP $A \in$ NP.

Proof. We give the following nondeterministic polynomial time algorithm. FCP A asks us to find a clue set of size at most k. First, nondeterministically pick a clue set of size k. Then for each string character in the solution that has not been assigned a value, try all possible settings of that character and accept if only one setting of that character produces a solvable problem. This step relies on closedness, as we use the fact that the problems that result from setting certain characters of the solution are still solvable in polynomial time.

If for any string character we find that multiple settings give a solvable problem, then our clue set does not give a unique solution. If no setting gives a solvable problem, then our clue set is not valid since no solution is possible. ◀

5.2 FCP 2SAT

Proposition 16 states that FCP 2SAT $\in$ NP. Here, we show that this problem is in fact NP-complete.

▶ **Theorem 17.** *FCP 2SAT is NP-complete.*

We reduce from the NP-hard problem, Minimum Independent Dominating Set (MIDS), which asks whether a graph has an independent dominating set of size at most k.

Figure 6 Reduction from Minimum Independent Dominating Set to FCP 2SAT.

Given a graph $G = (V, E)$, does there exist $S \subset V$, with $|S| \leq k$ such that, for all $v \in V$, either $v \in S$ or $(u, v) \in E$, $u \in S$, and $u, v \in S$ implies that there is no edge between u and v.

We transform G into a formula ϕ. For each edge (u, v), we add the constraint $(\neg u \vee \neg v)$ to the 2SAT formula. The following two lemmas prove that this reduction is correct.

▶ **Lemma 18.** *Suppose G contains an independent dominating set of size at most k, then there exists a clue of the* 2SAT *formula containing at most k variables.*

Proof. Suppose $S \subset V$ is an independent dominating set of size k. Then for each $s \in S$, assign s to true. We claim the formula is uniquely satisfiable.

If x is some variable in ϕ, then x corresponds to some node in G. Since S is a dominating set, let $s \in S$ be a neighbor of x. Then $(\neg x \vee \neg s)$ is a clause in ϕ. For ϕ to be satisfied, x must be false.

Since S is independent, we have no false clauses. ◀

▶ **Lemma 19.** *Suppose ϕ contains a clue of size k. Then G contains an independent dominating set of size at most k.*

Proof. Let C_T and C_F be the set of variables set to true and false, respectively, in the clue for ϕ. Without loss of generality, we may force C_F to be empty. For any variable x in C_F, consider a clause $(\neg x \vee \neg y)$ in ϕ. We can give a clue of identical size by removing x from C_F and including y in C_T.

Also, we note that all variables assigned to true in the satisfying assignment must be in C_T. If a variable x set to true is not in C_T, uniqueness is violated since setting x to false gives another satisfying assignment.

Let $S = C_T$. S must be an independent set because ϕ is satisfiable. In addition, if $v \in G - S$, then since the satisfying assignment to ϕ is unique, v contains a neighbor whose variable is set to true. So S is dominating. ◀

5.3 FCP versions of other easy problems

Consider the following language:

$$L = \{\phi' = (\phi \wedge \overline{z}) \vee z | \phi \text{ is a Boolean formula and } z \text{ does not appear in } \phi\}.$$

Note that if we ask whether ϕ' is satisfiable, the answer is trivially yes. So $L \in \mathsf{P}$. FCP L asks whether there exists a setting of k variables to ϕ' which makes it uniquely satisfiable.

▶ **Proposition 20.** *FCP L is Σ_2^P-complete.*

Proof. We reduce from FCP SAT. If ϕ has a clue of size at most k with a unique solution, then $\phi' = (\phi \wedge \overline{z}) \vee z$ has a partial assignment of size at most $k+1$ with a unique solution. We simply add the assignment for variable z.

If ϕ has no satisfying assignment, then ϕ' requires at least $k+1$ assigned variables for a partial assignment with a unique solution. Let x be a particular satisfying assignment to ϕ' that requires more than k assigned variables. If ϕ has a partial satisfying assignment $\mathbf{x}$ and $z \in \mathbf{x}$, then $\mathbf{x}$ needs at least n assigned variables to be specified. If $\overline{z} \in \mathbf{x}$, then we must specify $\overline{z}$ in the clue, and by assumption we must specify more than k variables. Therefore, if ϕ needs a clue of size greater than k, ϕ' will need a clue of size greater than $k+1$. ◀

Acknowledgements. This work began as a final project for MIT's Algorithmic Lower Bounds class (6.890) taught by Erik Demaine in Fall 2014. Thanks to Sarah Eisenstat and Jayson Lynch for valuable discussion and insight.

References

1. Charles J Colbourn. The complexity of completing partial Latin squares. *Discrete Applied Mathematics*, 8(1):25–30, 1984.
2. Erik D Demaine, Yoshio Okamoto, Ryuhei Uehara, and Uno Yushi. Computational complexity and an integer programming model of shakashaka. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 97(6):1213–1219, 2014.
3. Ian Holyer. The NP-completeness of some edge-partition problems. *SIAM Journal on Computing*, 10(4):713–717, 1981.
4. David Lichtenstein. Planar formulae and their uses. *SIAM Journal on Computing*, 11(2):329–343, 1982. doi:10.1137/0211025.
5. Brandon McPhail. Light up is NP-complete. *Unpublished manuscript*, 2005.
6. Wolfgang Mulzer and Günter Rote. Minimum-weight triangulation is NP-hard. *CoRR*, abs/cs/0601002, 2006. URL: http://arxiv.org/abs/cs/0601002.
7. Takahiro Seta. The complexities of puzzles, cross sum and their another solution problems (ASP). Senior thesis, Univ. of Tokyo, Dept. of Information Science, Tokyo, Japan, Feb 2001. URL: http://www-imai.is.s.u-tokyo.ac.jp/~seta/paper/senior_thesis/seniorthesis.ps.
8. Yato Takayuki and Seta Takahiro. Complexity and completeness of finding another solution and its application to puzzles. *IEICE transactions on fundamentals of electronics, communications and computer sciences*, 86(5):1052–1060, 2003.
9. Seinosuke Toda. PP is as hard as the polynomial-time hierarchy. *SIAM J. Comput.*, 20(5):865–877, October 1991. doi:10.1137/0220053.
10. Nobuhisa Ueda and Tadaaki Nagao. NP-completeness results for nonogram via parsimonious reductions. *preprint*, 1996.
11. Leslie G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8(2):189–201, 1979. doi:10.1016/0304-3975(79)90044-6.
12. Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979. doi:10.1137/0208032.
13. Leslie G. Valiant and Vijay V. Vazirani. NP is as easy as detecting unique solutions. *Theoretical Computer Science*, 47(C):85–93, 1986. doi:10.1016/0304-3975(86)90135-0.

Super Mario Bros. is Harder/Easier Than We Thought

Erik D. Demaine[1], Giovanni Viglietta[2], and Aaron Williams[3]

1 **Computer Science and Artificial Intelligence Laboratory, MIT, 32 Vassar St., Cambridge, MA 02139, USA**
edemaine@mit.edu
2 **School of Electrical Engineering and Computer Science, University of Ottawa, Canada**
viglietta@gmail.com
3 **Division of Science, Mathematics, and Computing, Bard College at Simon's Rock, 84 Alford Rd, Great Barrington, MA 01230, USA**
haron@uvic.ca

Abstract

Mario is back! In this sequel, we prove that solving a generalized level of Super Mario Bros. is PSPACE-complete, strengthening the previous NP-hardness result (FUN 2014). Both our PSPACE-hardness and the previous NP-hardness use levels of arbitrary dimensions and require either arbitrarily large screens or a game engine that remembers the state of off-screen sprites. We also analyze the complexity of the less general case where the screen size is constant, the number of on-screen sprites is constant, and the game engine forgets the state of everything substantially off-screen, as in most, if not all, Super Mario Bros. video games. In this case we prove that the game is solvable in polynomial time, assuming levels are explicitly encoded; on the other hand, if levels can be represented using run-length encoding, then the problem is weakly NP-hard (even if levels have only constant height, as in the video games). All of our hardness proofs are also resilient to known glitches in Super Mario Bros., unlike the previous NP-hardness proof.

1998 ACM Subject Classification F.1.3 Complexity Measures and Classes

Keywords and phrases video games, computational complexity, PSPACE

Digital Object Identifier 10.4230/LIPIcs.FUN.2016.13

1 Introduction

Super Mario Bros.[1] is one of the most famous and iconic video games, launching the 1985 revolution in home console gaming known as the Nintendo Entertainment System (NES/Famicon), and pioneering the genre of side-scrolling platform video games. Super Mario Bros. also launched a series of over 20 games, in total selling over 260 million copies, though the original game remains the best selling at over 40 million copies [11] (and for many years, was even the best-selling video game of all time).

Super Mario Bros. is also well known for being a challenging game, starting the adjective "Nintendo Hard" [6]. To formalize this concept mathematically, we study the computational complexity of a generalized form of Super Mario Bros. Previous work presented at FUN

[1] Super Mario Bros. is a trademark of Nintendo. Sprites are used and stripped and modified ROMs are presented here under Fair Use for the educational purpose of illustrating mathematical theorems.

© Erik D. Demaine, Giovanni Viglietta, and Aaron Williams;
licensed under Creative Commons License CC-BY
8th International Conference on Fun with Algorithms (FUN 2016).
Editors: Erik D. Demaine and Fabrizio Grandoni; Article No. 13; pp. 13:1–13:14
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2014 [1] proved that Super Mario Bros. 1–3, The Lost Levels, and Super Mario World are all NP-hard, but left open whether they were in NP. Here we show that membership in NP is unlikely: Super Mario Bros. is PSPACE-complete (Section 4). This proof uses a monster's location to store state and implement the doors in the PSPACE-completeness metatheorem of [1, 9, 10].

Both our PSPACE-hardness proof and the original NP-hardness proof [1] generalize Super Mario Bros. to have levels of arbitrary size, but also to have the entire level fit "on screen", or equivalently, to have a game engine that remembers the state of off-screen sprites (i.e., *persistent* monsters and items). While the generalization in level size is necessary to study computational complexity, in the real games the screen is much smaller than the level, and the game engine forgets the state of everything substantially off-screen.[2] To model this more "realistic" setup, we consider Super Mario Bros. generalized to have arbitrary level size but only a constant screen size as well as allowing only a constant number of on-screen sprites.[3] (Our "realistic" model also excludes fantastical gameplay elements like horses, boats, and extrasolar objects that were present in Vargomax's (humorous) investigation of Super Mario Bros. and graph coloring [8].)

Within this bounded-screen-size model, we consider two possibilities for how levels can be specified. First, if every tile is encoded explicitly, then we show that Super Mario Bros. becomes solvable in polynomial time (Section 2). Second, if tiles can be encoded implicitly using run-length encoding (where a row of k identical tiles gets encoded as one copy of the tile with a binary encoding of k), then we show that Super Mario Bros. becomes weakly NP-hard (Section 3).[4] This hardness proof also works for levels of constant height, which is a property shared by most Super Mario Bros. games; for example, even the very flexible Super Mario Maker allows levels of height only double that of the screen, or 27 blocks, the same as Super Mario Bros. 3, and about double that of Super Mario Bros. It also relies on many mechanics of Super Mario Bros. previously not exploited: pipes, time limits, multiple lives, 100 coins grant a free life, and levels have checkpoints. Furthermore, the underlying decision problem is not whether Mario can complete a single level (referred to as "reaching the flagpole"), but it is whether Mario can complete a sequence of levels (referred to as "rescuing the princess").

Both of our hardness proofs are resilient to known *glitches* [2, 5], where the implementation of Super Mario Bros. is counter to the intuitive Mario physics with which most players are familiar. Figure 1 shows examples of such glitches; see Section 2 for details. By contrast, the previous NP-hardness proof [1] breaks when such glitch behaviors are permitted.

1.1 Playable gadgets

Our reductions have been implemented and tested in the original NES Super Mario Bros. game. Our modified ROMs can be downloaded from `http://giovanniviglietta.com/files/SMB/gadgets.zip`. We also implemented the two gadgets of Section 4 in Super Mario Maker, although with some minor modifications to cope with the slightly different physics.

[2] The source code [3] defines two screen sizes – the *visible screen* and the somewhat larger *relevant screen* – and all sprites outside of the relevant screen are forgotten.

[3] The Nintendo Entertainment System could draw only eight sprites per scanline (row of the screen) without flickering.

[4] The source code [3] implements such a run-length encoding for runs of blocks or coins of length up to 16, so we consider the natural generalization to runs of blocks or coins of arbitrary length with an efficient encoding.

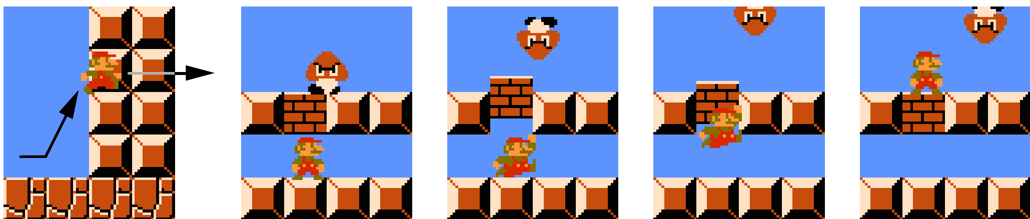


Figure 1 Two glitches: going through a wall and jumping through a brick with a monster on top.

The level IDs are 92F8-0000-005D-F452 (crossover gadget) and A8B5-0000-005D-E79A (open-close door gadget).

2 Standard SMB is polynomial-time solvable

In this section we consider a generalization of the standard Super Mario Bros. (from now on, *SMB*) game that is still quite close to the original, and we prove that the solvability of its levels, and even sequences of levels, can be decided in polynomial time.

2.1 Basic gameplay of SMB-standard

In each level, Mario starts from a specific location, and his goal is to reach a *flagpole* (or an *axe*, in boss levels), which marks the end of the level and leads to the next one, or to the endgame sequence. If Mario dies in a level, he loses one life (of the initial three) and has to play the same level again from the beginning. Nonetheless, each level has a *checkpoint* which, when reached, becomes the new starting location. Occasionally there are hidden *warp pipes* that let Mario skip some levels.

Next we describe *SMB-standard*, which is our first generalization of SMB. We allow the designer to construct arbitrarily large 2-dimensional levels with arbitrarily many objects in them. Accordingly, we allow the initial timer and the amount of lives of Mario to be exponentially large integers (with respect to the size of the level). However, we keep the physics and local effects unaltered, and we keep the screen's size constant, allowing only a constant number of objects on screen, as in the original SMB (where the screen's size is 16×16 tiles, and there can be at most six sprites on screen, including Mario). The screen follows Mario's movements, and all the objects and enemies that exit the screen are forgotten by the game. This means that there is rectangular region in which the action takes place and objects are animated. When this region moves and new elements enter the active area, they are loaded from a read-only level file. On the other hand, the elements that exit the screen are no longer active. As a consequence, if Mario kills an enemy, goes to some other area and then comes back, the enemy is reloaded from the level file in its original position and state, as if it was never killed. In SMB the screen can only scroll to the right, but nothing changes if we allow it to move in all directions in SMB-standard.

There is some pseudo-randomness in SMB: a 7-byte memory area is used as a pseudo-random register; it is deterministically updated at every frame (starting from a fixed seed), and looked up whenever a random number is needed. The illusion of randomness comes from the fact that the frame-by-frame sequence of button presses of a human player is hardly ever the same. In other words, the player's input is the only real source of randomness in the game.

The types of enemies and game elements of SMB are too many to be listed here. A complete definition of all the game mechanics can be found in the game's commented source

Figure 2 World 7-4: if Mario does not guess the correct path, he is sent back a few screens.

code [3]. However, for the purpose of this paper, it is sufficient to highlight just a few aspects of the game. Some of them will be described in this section; others will be introduced when they are needed.

An interesting game element is the *Loop Command* object, which is an invisible tile that is used in some castle levels to make Mario go back a few screens if he does not follow the right path, as Figure 2 exemplifies. When a Loop Command object enters the screen, Mario's position in the screen is checked: if he is in the wrong place, the screen's coordinates change (typically, the x coordinate is decreased by a constant amount). Sometimes, Loop Command objects are *cumulative*: in World 7-4, Mario can be sent back only every third Loop Command object that he encounters, and only if he failed the test on at least one of the previous three Loop Command objects. In SMB-standard, we can generalize this "three" to any (exponentially large) number.

2.2 Glitches

There are also several *glitches* in SMB, some of which are documented in [2, 5]. None of the known glitches affects the results of this section, so we may as well include them in SMB-standard. However, a couple of them will be relevant in the next sections, as they make the game behave counter-intuitively in certain situations (see Figure 1). Namely, when a vertical wall is at least two tiles high, Mario can slightly penetrate between two tiles by jumping towards them at the right speed. Then, if he suddenly steers in the opposite direction, the wall pulls him inside instead of pushing him out. This makes Mario potentially able to walk through every wall consisting of more than a single tile. Also, if Mario is small (i.e., he has not eaten a Super Mushroom) and he hits a brick block with his head twice fast enough, he can jump through the block, provided that there is a monster on top of it.

2.3 Reaching the flagpole in SMB-standard

We start by proving that solving a single level of SMB-standard is a polynomial-time task. For this, we only need a very small set of assumptions, which the interested reader can verify by inspecting [3].

▶ **Theorem 1.** *Reaching the flagpole in a level of SMB-standard without losing lives is a polynomial-time task.*

Proof. Mario's position in the screen is encoded as a pair of integer coordinates (with a precision of 1/16 of a pixel). Hence the possible positions of Mario within the screen are

finitely many. Mario's velocity vector is encoded similarly, and his speed never exceeds a constant value. The same holds for all monsters and moving objects. All the state transitions of each object, as well as all possible interactions between objects, can be computed in constant time. Hence, the transition from one frame to the next can be computed in constant time, because the screen has constant size and there can be only a constant number of interacting objects in it.

It follows that there is only a polynomial number of possible *configurations*, where a configuration represents the state of the game at a given frame, and is characterized by a screen position within the level, at most a constant number of objects on screen (each with a constant-size state), a constant-size pseudo-randomly-generated number, a polynomial-size Loop Command object counter, a Loop Command "failure flag", and the player's input for that frame (which is a combination of button states). Moreover, all the possible one-frame transitions between configurations can be determined in polynomial time. This means we can efficiently construct a polynomial-size *configuration graph*, with several starting vertices $s_1, \dots, s_k$, each corresponding to an initial pseudo-random register, and several finish vertices $t_1, \dots, t_{k'}$, each corresponding to a configuration in which Mario touches the flagpole.

Now, because each directed edge in this graph corresponds to a feasible move (assuming the player hits the right buttons), and such a move makes the timer decrease by a constant fraction $1/f$ of a time unit, solving the level amounts to finding a shortest path from each s_i to any t_j, which is done via a breadth-first traversal of the configuration graph. If any of the shortest paths is longer than f times the available number of time units, then the level is unsolvable. ◀

2.4 Rescuing the princess in SMB-standard

In order to extend the previous proof to sequences of levels, we have to recall how lives work in SMB. Mario gets an extra life whenever he collects a 1-Up Mushroom, or repeatedly jumps on enemies without touching the ground, or collects (a multiple of) 100 coins. He loses a life whenever he fails to reach the flagpole, either because he is killed or because he runs out of time. When he loses the last life, the game is over. Note that Mario does not lose his coins when he dies or completes a level.

The classic SMB game has a fixed limit on the maximum number of lives that Mario can have. In SMB-standard we allow Mario's number of lives to grow unboundedly, which makes our next theorem a little stronger.

▶ **Theorem 2.** *Rescuing the princess in SMB-standard is a polynomial-time task.*

Proof. We will reason as in Theorem 1, with the additional difficulty that now we have to keep track of the number of lives gained and lost.

Let the game have n levels. Each level has a checkpoint, which splits it into two *chunks*. When Mario reaches the same horizontal coordinate of a checkpoint (regardless of his vertical coordinate), the checkpoint is considered reached. Then, the next time he dies in the level, he will restart from the checkpoint, at the smallest possible vertical coordinate. So, the first chunk of each level can only be played from one initial location. The second chunk of a level can be played from at most v possible initial locations, where v is the (polynomial) number of possible vertical coordinates. Indeed, the location with smallest vertical coordinate is the one that occurs after Mario dies in the second chunk of the level, and the other initial locations may occur the first time that Mario enters the second chunk from the first chunk.

Recall that the pseudo-random register is deterministically updated at every frame, and let k be the constant number of possible values that the pseudo-random register can assume.

The crucial observation is that, if the game is beatable with an infinite initial supply of lives, then it is beatable with an initial supply of $2kvn$ lives. Indeed, there are $2n$ level chunks in total, each of which can be replayed several times before moving on to the next one. In turn, there are at most v possible initial positions for Mario in each chunk, and at most k possible initial values of the pseudo-random register. A chunk may be impossible to beat from some initial vertical positions, or for some initial values of the register. Moreover, even if a chunk is always beatable, it may be necessary to finish it when the pseudo-random register has a specific value, because that initial value may be required to beat the next chunk, etc. However, if a specific final value is attainable at all, it must be attainable at the cost of at most k lives.

Because 100 coins grant an extra life and it is impossible to lose coins, we can equivalently think of a coin as $1/100$ of a life. For each of the $2n$ chunks, we build a configuration graph G as in Theorem 1, also adding to the state the initial vertical coordinate of Mario, and the (fractional) number of lives ℓ that Mario has. By the above argument, if at any point $\ell \geq 2kvn$, we can safely assume ℓ to be infinite. Hence we only need encode $200 \cdot kvn + 1$ possible values for ℓ, which can be done using only $O(\log n)$ bits.

Now we connect together the graphs corresponding to different chunks, in such a way that each final vertex of a chunk leads to the initial vertex of the next chunk having a matching pseudo-random register in its state. We also add similar edges corresponding to warp pipes. Furthermore, we add edges corresponding to Mario's deaths, which either lead to the beginning of the current chunk, with one less life, or to a special Game Over vertex having no outgoing edges. We add an extra "zeroth" chunk corresponding to the title screen, with k vertices, one for each possible initial value of the pseudo-random register. Finally, we put a last vertex t, which is reached after beating the last chunk.

The game starts from the vertex s in the zeroth chunk having the appropriate pseudo-random seed, and then proceeds by following the edges in this "macro-graph" corresponding to the player's inputs (or lack thereof) at each frame. For instance, waiting in the zeroth chunk without pressing buttons only makes the pseudo-random register change, while pressing the Start button leads to the appropriate initial vertex of the first chunk. Because the macro-graph has polynomial size, verifying if the vertex t can be reached from s is a polynomial-time task. ◀

3 SMB with run-length encoding is weakly NP-hard

Levels in the original SMB game are encoded using a limited form of compression. For example, a run of up to 16 consecutive coins can be encoded using the same space as a single coin. In this section we consider the hardness of a SMB variation with levels that allow arbitrary *run-length encoding* (RLE), meaning that m consecutive blocks of a fixed type along an individual row can be encoded using $O(\log m)$ bits. We extend this concept to entire levels, as well: we can encode m consecutive copies of the same level by attaching an $O(\log m)$-bit number to the encoding of the level. We refer to this variation as *SMB-RLE*, and we prove that rescuing the princess in this game is weakly NP-hard.

Our reduction is from the Knapsack problem. The main idea is to model item weights by time, and item values by coins. We create a level in which Mario is faced with n independent choices of either collecting v_i coins at a cost of w_i time, or skipping past the coins by using a pipe. Each of these choices corresponds to either adding item i to the knapsack or not, respectively. We set a time limit of W to the level to restrict Mario's choices, and we force Mario to collect at least V coins – and thus $\lfloor V/100 \rfloor$ extra lives – by separating Mario from the princess by a sequence of levels, each of which will cause him to lose one life.

Besides the addition of RLE, our new variation of SMB is similar to the SMB-standard variation introduced in Section 2. In particular, we do not require the persistence or non-persistence of any game elements. However, to make our result stronger, we use levels of constant height that cannot scroll to the left (such as the ones in the original SMB).

The remainder of this section formalizes our Knapsack problem, clarifies our assumptions about the various game elements, and then describes individual levels and overall reduction. Ultimately, we briefly discuss the variation of SMB in which run-length encoding can be applied to blocks but not to whole levels. This version is also weakly NP-hard, provided that the number of coins that grant an extra life (which normally is 100) can be configured arbitrarily.

3.1 Knapsack problem

The following version of the Knapsack problem is also known as the *0-1 Knapsack problem* because each item is indivisible and is either inserted into the knapsack or not.

> *Knapsack problem*
> **Input:** Item weights $w_1, w_2, \ldots, w_n$ and values $v_1, v_2, \ldots, v_n$, a total weight capacity of the knapsack W and a target value V.
> **Output:** Is there a subset of items with total weight $\sum w_i \leq W$ and value $\sum v_i \geq V$?

The problem is NP-complete even if each item's weight is proportional to its value [4], and there is also a well-known dynamic-programming algorithm for solving it. The algorithm runs in *pseudo-polynomial time*, meaning that its running time is polynomial with respect to the magnitude of the numbers involved (as opposed to the number of bits used to represent them). Thus, the problem is weakly NP-complete. We will find it convenient to reduce from a constrained version of the problem in which each item weight is at least 100 times as large as its value.

▶ **Lemma 3** (Knapsack with Large Weights). *The Knapsack problem is weakly NP-complete when restricted to inputs with $w_i \geq 100 \cdot v_i$ for all $1 \leq i \leq n$.*

Proof. The decision problem is unchanged by multiplying the item weights and total capacity by a fixed positive integer. If the integer is an item value, then the maximum magnitude of the numbers in the resulting problem is at most squared. Therefore, the result follows from multiplying every item weight by $100 \cdot v'$, where v' is the maximum value in $\{v_1, \ldots, v_n\}$. ◀

3.2 Game elements and controls

The only game elements we use are solid blocks, pipes, coins, checkpoints, and the timer. We clarify our assumptions about these elements and other aspects about the levels below.

- A level can have arbitrary width, but we only use levels that have a constant height. This differs from the NP-hardness result in [1], which relied upon having levels of arbitrary height and width. In fact, we only use levels that are two blocks high.
- A level consists of a single room. In other words, our hardness result does not require the use of any side rooms or bonus areas.
- A level can have an arbitrary number of pipes. For a Knapsack problem with n items we create a level with n pairs of pipes. The pipes do not nest, in the sense that the x coordinates of the entrances and exits satisfy $in_1 < out_1 < in_2 < out_2 < \cdots < in_n < out_n$.

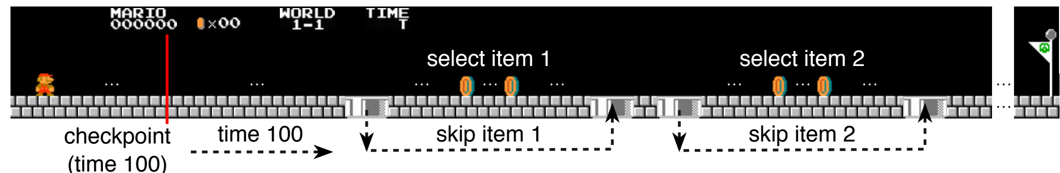


Figure 3 Sketch of a choice level including the first two decision sections.

- Going through a pipe takes constant time, regardless of the distance traveled.
- The screen never scrolls left. This fact, combined with our use of pipes, implies that no game element can be revisited. Thus, persistence is a non-issue, and our reduction will work regardless of whether or not it is assumed.
- The timer for each level can be arbitrarily large. However, we assume that it is always a multiple of 100 to keep it as close to the original game as possible.
- There is one checkpoint per level. If Mario dies after the checkpoint, but before the flagpole, then he is respawned at the checkpoint instead of the beginning of the level. As in the original game, the timer after respawning can differ from the original timer.

Mario will never have reason to jump, move left, move up, or stop running in the levels we create. Furthermore, our designs have natural "walking analogues" that are functionally equivalent if Mario is forbidden from running. Thus, our problem will remain weakly NP-hard if Mario's controls are limited to any subset that includes and .

3.3 Choice level

A *choice level* is parameterized by $2n+1$ integers: $w_1, \ldots, w_n$, $v_1, \ldots, v_n$, and W. The level is comprised of $n+3$ sections (see Figure 3) in the following order.

- The *checkpoint section* begins with a blank screen followed by a checkpoint that will reset the timer to 100 upon Mario's death.
- The *empty section* requires 100 time units to traverse and is otherwise empty.
- There are n *decision sections*, each of which begins with one pipe and ends with one pipe. Taking the first pipe will transport Mario to the second pipe, thereby skipping over the entire section. Otherwise, Mario travels from the first pipe to the second pipe on foot and can collect a series of coins. The number of coins along the ith decision section is v_i and these coins are surrounded by at least one empty screen to the left and right. The path from the first pipe to the second pipe requires w_i units of time to traverse.
- The final section is a *flagpole section* and is the end of the level.

Notice that the first two sections force Mario to complete the level in one life. More specifically, if he respawns at the checkpoint, then he cannot pass the empty section due to the timer. Also notice that the ith decision section is only well defined if t_i is large enough to accommodate c_i coins and the empty screens to the left and right. This will not be an issue with our reduction because we use the constraint discussed in Lemma 3. The level's timer is $T = W + \varepsilon$, where ε is the time required to complete the level using all n pipe pairs.

3.4 Coin level

A *c-coin level* contains c coins for some $0 \leq c \leq 99$ (see Figure 4 (a)). Its time limit of 200 is sufficient for completing the level either by running or walking for all c.

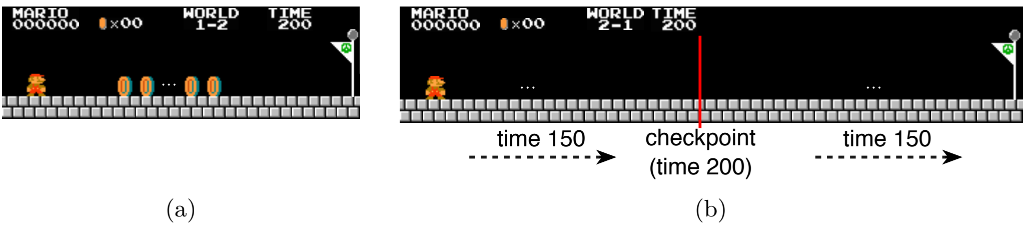


Figure 4 (a) A coin level, and (b) a kill level.

3.5 Kill level

A *kill level* takes 300 time units to complete but only provides a timer of 200 (see Figure 4 (b)). It also has a checkpoint halfway through the level that provides 200 time units after respawning. Mario can complete the level by expending one life to reach the checkpoint, and then by continuing to the flagpole after respawning.

3.6 Weak NP-hardness of SMB-RLE

Now we prove the main result of this section.

▶ **Theorem 4.** *Rescuing the princess in SMB-RLE is weakly NP-hard, even when using levels of height 2, any subset of buttons including and , and the elements in Section 3.2.*

Proof. Consider an instance of the Knapsack problem satisfying Lemma 3. Let $\lceil V/100 \rceil = \ell$. We create an SMB-RLE game consisting of the following levels:

- World 1-1 is a choice level using the integers $w_1, \ldots, w_n$, $v_1, \ldots, v_n$, and W from the Knapsack problem instance.
- World 1-2 is a c-coin level, where $c = 100 \cdot \ell - V$.
- World 2-x is a kill level for all $1 \leq x \leq \ell + 2$.

These levels are well defined by Lemma 3 and the discussion in Section 3.1. For the $\ell + 2$ copies of the kill level in World 2 we make use of run-length encoding, as they could be exponentially many. Notice that Mario can rescue the princess if and only if he has at least $\ell + 3$ lives after completing World 1-2. Considering that Mario starts the game with three lives, this is equivalent to collecting $V + c$ coins on Worlds 1-1 and 1-2, which in turn is equivalent to collecting V coins on World 1-1. Mario can collect V coins on World 1-1 if and only if there is a solution to the Knapsack problem. ◀

3.7 Weak NP-hardness without using run-length encoding for levels

In this section, we show that weak NP-hardness holds even when run-length encoding is allowed only for the blocks within a single level, and not for encoding an entire sequence of levels. (The proof above uses run-length encoding on the level sequence to (just) create the kill levels that force Mario to lose a certain number of lives.) To achieve this goal, we add a different assumption: the amount of coins that Mario has to collect to get an extra life is no longer 100, but it can be any number (decided by the game designer). Then we can modify our previous NP-hardness reduction as follows:

- V coins grant an extra life.
- World 1-1 is a choice level using the integers $w_1, \ldots, w_n$, $v_1, \ldots, v_n$, and W from the Knapsack problem instance.
- Worlds 2-1, 2-2, and 2-3 are kill levels.

Because there are only three kill levels, they can be represented explicitly without using run-length encoding. In order to pass them, Mario needs to have at least four lives by the time he reaches World 2-1. In turn, this is possible if and only if he can collect at least V coins in World 1-1, considering that he starts the game with three lives, and V coins grant an extra life. Hence he can finish all the levels if and only if the Knapsack problem is solvable.

As a consequence, Theorem 4 also extends to this more natural variation of the game.

4 SMB with no reset of off-screen objects is PSPACE-complete

In this section, we assume that the screen does not move in a fixed direction, and that objects off-screen are not reset. In particular, we assume that the engine remembers the positions of all monsters. Reasoning as in Section 2, it is not hard to prove that this variation of SMB, which we call *SMB-general*, is solvable in PSPACE. Indeed, each game configuration can be stored in polynomial space, and the configuration graph can be efficiently navigated. This shows that SMB-general is in NPSPACE, and thus in PSPACE by Savitch's Theorem. In the rest of this section, we prove that SMB-general is PSPACE-complete.

4.1 PSPACE-hardness framework

To prove that SMB-general is PSPACE-hard, we use the "open-close door" framework from [1], similar to metatheorems of [9, 10]. The framework is based on a reduction from Quantified Boolean Formula involving the following elements:

- a player-controlled avatar,
- a starting location and an exit location,
- arbitrarily oriented paths arranged in a plane,
- crossover gadgets, and
- open-close door gadgets.

A *crossover gadget* is a region in which two paths A and B cross each other without leakage. That is, if the avatar is walking along A and traverses the crossover gadget, it cannot end up on B, and vice versa.

An *open-close door gadget* is an object with two states – open and closed – and is intersected by three disjoint paths:

- a *traverse* path, which can be traversed by the avatar if and only if the door is in the open state;
- a *close* path which, when traversed by the avatar, causes the door to close; and
- an *open* path which, when traversed by the avatar, allows the player to open the door, but may not force them to. (The open path may also be a dead end.)

If all the above elements can be implemented in a game, then it is PSPACE-hard [9, 10, 1].

4.2 Starting/exit locations and paths

Of course SMB already has the first three elements, while arbitrarily oriented paths can be implemented by using solid blocks. We will put no Super Mushrooms in our levels, so we may assume that Mario will always be small, and therefore it is sufficient to implement 1-tile-high paths going horizontally and diagonally. It is not hard to construct them without creating walls that are higher than one tile, thus avoiding the walk-through-walls glitch.

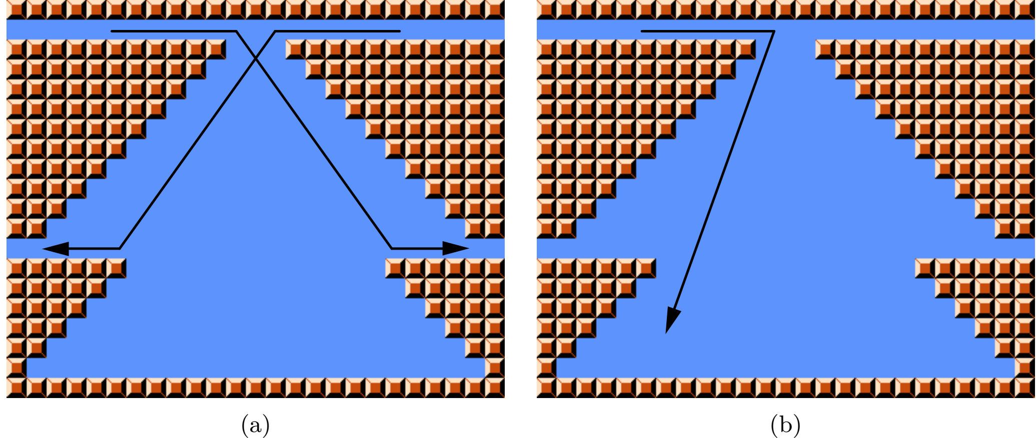


Figure 5 PSPACE-hardness reduction: crossover gadget. (a) Mario can avoid the pit by simply running forward at full speed. (b) If Mario tries to change path, the lack of momentum makes him fall into the pit.

4.3 Crossover gadget

Crossover gadgets are easy to implement with pipes. However, if we insist on not using pipes, we can achieve the same result by using only Mario's physics, thanks to the gadget in Figure 5. A bonus feature of this gadget is that it can be easily adapted to work in almost any platform game. As the first figure shows, Mario can go from the top-left path to the bottom-right path by simply running at full speed. He cannot jump across the 3-tile-wide gap, due to the low ceiling. Moreover, if he tries to go from the top-left path to the bottom-left path, he has to switch direction in mid-air, which makes him lose all his momentum, and makes him fall into the pit, as shown in the second figure. Once in the pit, he can never get out, because it is too deep. Similarly, if he tries to jump from the bottom-right to the bottom-left path, he ends up falling into the pit, because the 13-tile gap between the two paths is too wide. Of course, the gadget works symmetrically in the right-to-left direction.

4.4 Open-close door gadget

Our implementation of the open-close door gadget is illustrated in Figure 6. The three flames represent tiles where rotating firebars are attached (for clarity, only the first flame of each firebar is drawn). Note that firebars do not necessarily have to be attached to solid blocks in SMB (an example is the underwater part of World 8-4).

The monster on the right is a Spiny, which keeps walking back and forth in the 4-tile-wide tunnel. When the Spiny is in this area, the door is considered open. Indeed, in this case Mario can freely walk through the traverse path. However, Mario cannot walk through the close path, because the Spiny is in the way. This is a tough monster, which can be killed only by the fireballs that Mario shoots after picking up a Fire Flower, which is not available in our level. In particular, Small Mario dies if he jumps on a Spiny. If, however, Mario takes the tunnel that runs below the Spiny, he can hit the brick block from below, as in Figure 7. If this is done with the correct timing, it makes the Spiny bounce through the central firebar, and to the other side of the gadget. Note that the brick block does not break, because Mario is small. The purpose of the firebar attached to the brick block is to kill Mario if he triggers the jump-through-brick glitch. Similarly, the central firebar prevents Mario from taking the

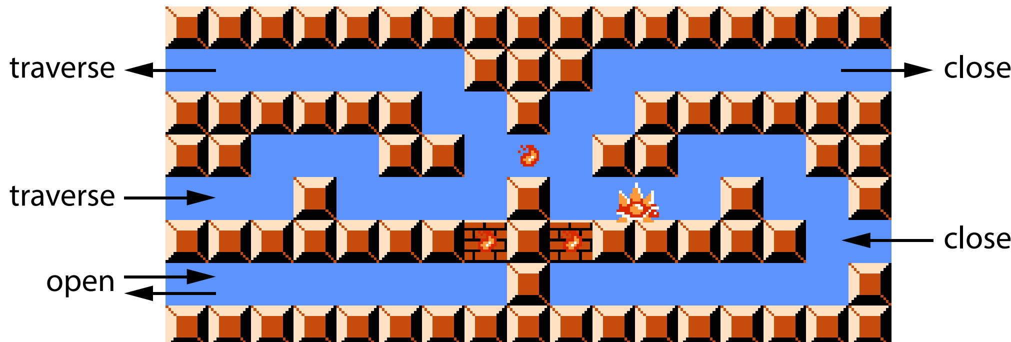


Figure 6 PSPACE-hardness reduction: open-close door gadget (in the open state).

shortcut between the traverse and the close paths, but it does not harm or obstruct the Spiny.

When the Spiny is on the left-hand side of the gadget, Mario can safely go through the close path (see Figure 7). Then the Spiny starts walking back and forth in the 4-tile-wide tunnel on the left, and the door is considered closed. Indeed, by a symmetric argument, Mario is now unable to go through the traverse path because the Spiny is in the way, and he has to reach the open path underneath if he wants to send the Spiny back to the right-hand side, thus opening the door again, and resetting the gadget to its original state.

4.5 PSPACE-hardness of SMB-general

To finalize our construction, we make sure to give Mario enough time to complete the level (provided that it is feasible), by setting the timer to some linear function of the configuration space's size.

Note that the walk-through-walls glitch cannot be exploited in our crossover and open-close door gadgets, because none of them contains 2-tile-high vertical walls. Therefore, as explained in Section 4.1, we have our main result.

▶ **Theorem 5.** *Reaching the flagpole in SMB-general is PSPACE-complete.*

4.6 Proper generation of Spinies

A possible point of criticism on our open-close door gadget is that, although the SMB engine allows the designer to place Spinies anywhere in the levels, the official SMB levels never make use of such a feature. Spinies are never found at specific locations, but they are dynamically thrown at Mario by floating *Lakitus*, as Figure 8 exemplifies.

It is not too hard to incorporate this feature into our levels, if we align all the door gadgets in the upper part of the construction. As the level starts, we can force Mario to walk above each door gadget, where a Lakitu throws a Spiny at him, which eventually falls into a pit that leads to the door gadget below. We can time everything in such a way that the Lakitu can throw only one Spiny per door gadget, if Mario runs at full speed. We can also prevent Mario from entering door gadgets through these pits, by placing firebars in them.

5 Open Problems

There are several natural open problems that arise naturally from our results. We name a few.

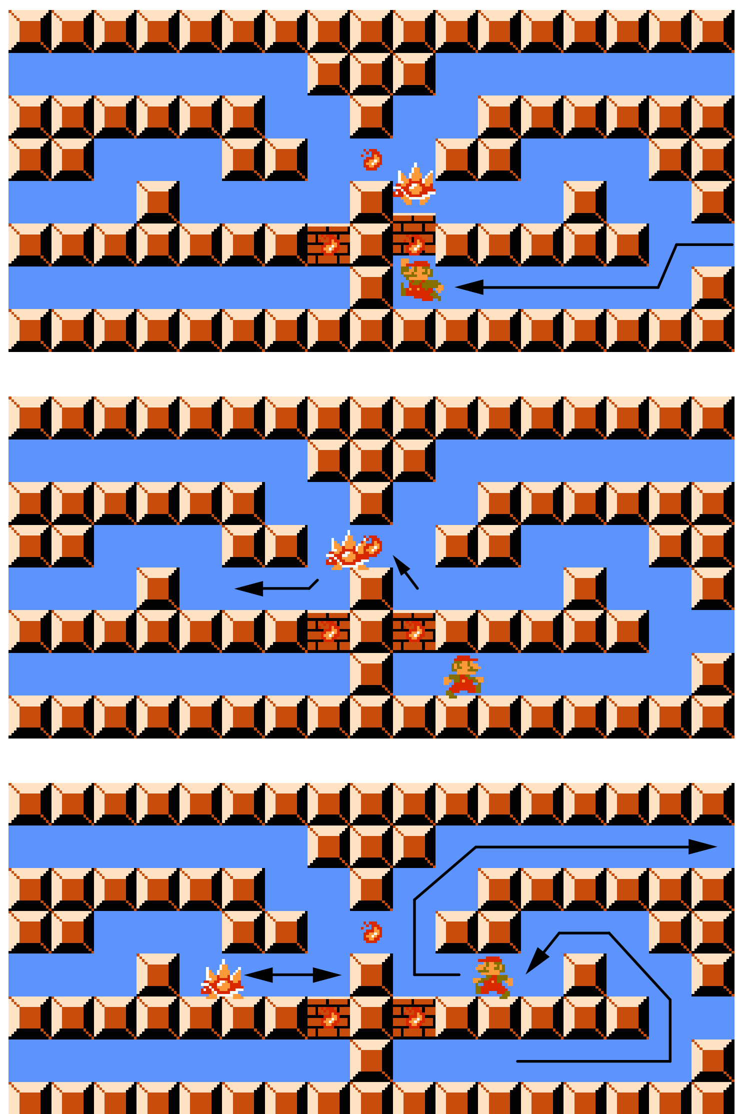


Figure 7 If the Spiny is in Mario's way, it has to be moved to the other side of the gadget. This can be done by hitting the brick block from below.

Figure 8 Lakitu throwing Spiny Eggs, which hatch as they hit the ground, becoming Spinies.

Is reaching the flagpole of run-length-encoded constant-height Super Mario Bros. weakly NP-hard? Is it also NP-complete? Our reduction in Section 3 critically relies on having multiple levels, so we wonder about the complexity of a single level. We note that the strategy of Section 3 could be altered to use *multiple checkpoints* in a single level instead of multiple levels, but we view this as "cheating" because the original Super Mario Bros. uses at most one checkpoint per level (while Super Mario Maker allows two per level).

To prove Theorem 4, we assumed that run-length encoding could be used to represent sequences of levels, as well as sequences of tiles. Then, in Section 3.7 were able to remove this assumption, by introducing the ability to configure the number of coins that have to be collected to gain an extra life. Does Theorem 4 hold if run-length encoding for sequences of levels is not allowed (but it is for blocks), and exactly 100 coins grant an extra life?

Our PSPACE-hardness reduction of Section 4 produces levels of unbounded size in both dimensions. Note that the same result can be obtained for levels of constant height, provided that pipes are used. Is Super Mario Bros. PSPACE-complete also for levels of constant height that do not contain pipes? There is hope for proving this given that constant-width Nondeterministic Constraint Logic is PSPACE-complete [7].

Finally, we suspect that our proofs can be adapted to the many Super Mario Bros. sequels, but this remains to be explored.

References

1. Greg Aloupis, Erik D. Demaine, Alan Guo, and Giovanni Viglietta. Classic Nintendo games are (computationally) hard. *Theoretical Computer Science*, 586:135–160, 2015.
2. Speed Demos Archive. Super Mario Bros. URL: `http://kb.speeddemosarchive.com/Super_Mario_Bros`.
3. doppleganger. A comprehensive Super Mario Bros. disassembly. URL: `http://giovanniviglietta.com/files/SMB/source.asm`.
4. Michael Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.
5. TASVideos. Super Mario Bros. URL: `http://tasvideos.org/GameResources/NES/SuperMarioBros.html`.
6. TV Tropes. Nintendo Hard. URL: `http://tvtropes.org/pmwiki/pmwiki.php/Main/NintendoHard`.
7. Tom C. van der Zanden. Parameterized complexity of graph constraint logic. In *Proceedings of the 10th International Symposium on Parameterized and Exact Computation (IPEC 2015)*, Leibniz International Proceedings in Informatics (LIPIcs), pages 282–293. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015. `doi:10.4230/LIPIcs.IPEC.2015.282`.
8. Vargomax V. Vargomax. Generalized Super Mario Bros. is NP-complete. In *Proceedings of the 6th Binarennial Workshop about Symposium on Robot Dance Party of Conference in Celebration of Harry Q. Bovik's 0x40th Birthday (SIGBOVIK 2007)*, pages 87–88, 2007.
9. Giovanni Viglietta. Gaming is a hard job, but someone has to do it! *Theory of Computing Systems*, 54(4):595–621, 2014.
10. Giovanni Viglietta. Lemmings is PSPACE-complete. *Theoretical Computer Science*, 586:120–134, 2015.
11. Video Game Sales Wiki. Mario. URL: `http://vgsales.wikia.com/wiki/Mario`.

A Rupestrian Algorithm*

Giuseppe A. Di Luna[1], Paola Flocchini[2], Giuseppe Prencipe[3], Nicola Santoro[4], and Giovanni Viglietta[5]

1 **School of Electrical Engineering and Computer Science, University of Ottawa, Ottawa, Canada**
gdiluna@uottawa.ca
2 **School of Electrical Engineering and Computer Science, University of Ottawa, Ottawa, Canada**
flocchin@site.uottawa.ca
3 **Dipartimento di Informatica, Unversità di Pisa, Pisa, Italy**
giuseppe.prencipe@unipi.it
4 **School of Computer Science, Carleton University, Ottawa, Canada**
santoro@scs.carleton.ca
5 **School of Electrical Engineering and Computer Science, University of Ottawa, Ottawa, Canada**
viglietta@gmail.com

Abstract

Deciphering recently discovered cave paintings by the Astracinca, an egalitarian leaderless society flourishing in the 3rd millennium BCE, we present and analyze their shamanic ritual for forming new colonies. This ritual can actually be used by systems of anonymous mobile finite-state computational entities located and operating in a grid to solve the *line recovery problem*, a task that has both self-assembly and flocking requirements. The protocol is totally *decentralized*, fully *concurrent*, provably *correct*, and time *optimal*.

1998 ACM Subject Classification C.2.4 Distributed Systems, F.1.2 Modes of Computation, F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases mobile finite-state machines, self-healing distributed algorithms

Digital Object Identifier 10.4230/LIPIcs.FUN.2016.14

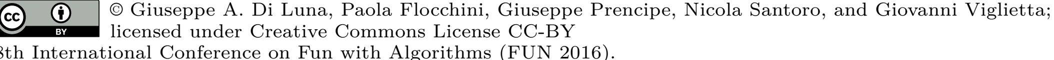
1 Introduction

1.1 Anthropological Discovery

Recently, an archaeological expedition in Budelli has discovered cave paintings and pottery from the 3rd millennium BCE. The artifacts have been attributed to the Astracinca semi-nomadic tribes, whose villages were located in a wide area of northern Sardinia. The Astracinca civilization, undeservedly little known, stood out for the highly sophisticated social organization and the advances in ethno-botanical techniques.

The abundance of resources, due to efficient agricultural techniques, and the high fertility, due to a joyously promiscuous social organization, cyclically brought each village to address the problem of overpopulation. As in many other civilizations, that problem was solved by

* This research has been supported in part by: the Natural Sciences and Engineering Research Council (Canada) under the Discovery Grant program, Prof. Flocchini's University Research Chair, and project PRA_2016_64 "Through the fog" funded by the University of Pisa.


© Giuseppe A. Di Luna, Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Giovanni Viglietta; licensed under Creative Commons License CC-BY

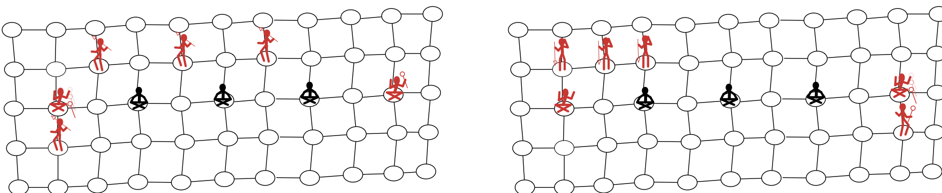


Figure 1 Reproduction of rupestrian paintings discovered in Budelli Island.

selecting a group of settlers, or two of equal size, with the task of creating new settlements, thus solving the overpopulation problem.

Unlike other civilizations, the Astracinca were an egalitarian leaderless society: upon reaching puberty, all members of the tribe were socially equal and decisions were reached by consensus. All agreed-upon social procedures guiding major events were carried out without any member being pre-assigned a special role. Such social procedures have been shrouded in the deepest mystery, and their complexity had been only inferred by cultural anthropologists, because no archaeological finding describing the procedures was available so far.

The found cave paintings (two of which are shown in Figure 1) are remarkable because they contain the description of the procedure used by the Astracinca to create a new settlement, an event of major importance in the life of the tribe; hence the great excitement in deciphering the procedure, and the interest in its intricacies, unparalleled in history.

After a long study of the paintings and pottery found in the cave, the procedure has been finally deciphered; it is indeed composed of a set of intricate rules, defining a shamanistic ritual WonnaGo to be performed by the adult population of the village. Before providing the details, let us give an overview of the discovered process.

The ritual WonnaGo takes place in a large clearing, away from the village, on which a regular grid is engraved, each node of which is large enough to host a person. Before the start of the ritual, a large number of bowls filled with liquid are placed just outside the grid. Each villager picks up a bowl and then sits in an empty node, so to form a line.

Each bowl is filled with liquid from one of two preparations, indistinguishable from the outside; the proportion between the bowls filled by each preparation is unspecified. By analyzing the residues found in the pottery, we know the composition of the two potions. The first potion, based on poppy milk, induces in drinkers a deep comatose state, making movements and communication impossible. Those who consume this substance are catatonically locked in their starting position for the duration of the rite. Upon exiting their comatose state, they continue their life in the old village. The second potion, made from Amanita Muscaria and Cannabis Sativa, selects the *settlers*. The shamanic use of Amanita Muscaria to induce hallucinatory and dissociative states is well known [18].

Dissociative substances may cause a drastic decrease in vision and perceptions; this effect is known as Kalnienk vision [3]. Incredibly, the set of rules predict this eventuality: they only assume the ability to observe the nodes adjacent to the one occupied by a settler, and communication to be limited to the settlers in those locations. The knowledge of this phenomenon by Astracinca also explains the use of a grid with carefully studied proportions. The medical literature has a well-established knowledge of acute intoxication by phytocannabinoids, in particular the short-term deterioration of memory [14]. The rules include this possibility, since each settler needs to remember a limited amount of information that does not depend on the number of participants.

During WonnaGo, the inebriated settlers leave their comrades in a cathatonic state, and eventually form either a new line or two lines of equal length. All the settlers in the

same line eventually become in agreement on the same direction. Once this happens, the ritual ends and the migration starts.

1.2 Technological Interpretations and Contributions

The social structure of the Astracinca civilization and the rules they use in their rituals are surprisingly modern. Indeed, not only do they have impressive similarities with alternative visions of how human systems could function (e.g., [16]), but also directly reflect a variety of current artificial systems ranging from robotic swarms to mobile sensor networks, from biologically inspired systems to mobile software agents. Indeed, systems of *mobile computational entities* that operate in a spatial universe, obeying the same set of rules in a totally decentralized way, with very limited (or no) local memory and communication capabilities, are almost ubiquitous. From an algorithmic point of view, such systems are being extensively and intensively studied, in particular within distributed computing. They include

the *metamorphic* robots (e.g., [5, 6, 20]), the amoeba-inspired *amebots* [8, 9], the finite-state *mobile robotic sensors* (e.g., [4, 13]), and the extensively studied *oblivious mobile robots* (e.g., [1, 7, 11, 12, 19]). In some of these systems the entities can move in a continuous spaces (e.g., $\mathbb{R}^2$); in others the movements occur in a discrete space (e.g., $\mathbb{Z}^2$). In particular, extensive investigations have been carried out when the computational entities are identical finite-state machines operating and moving in a (possibly incomplete) grid (e.g., [4, 8, 9, 10, 13, 15, 17]).

The model implied by the Astracinca ritual is precisely that of anonymous mobile finite-state machines located and operating in a grid in a fully synchronous system. Each entity is a finite-state machine, can move in the grid from node to node, and is able to communicate with the entities located at neighboring nodes; since the entities are finite-state, the amount of information exchanged in a communication is bounded by a constant. The entities are anonymous and behaviorally identical; that is they have no distinguished identifiers and they all execute the same algorithm. The entities do not rely on a common coordinate system: each entity fixes a local orientation of the grid, but different entities may have different orientations. How the communication between two neighboring entities is performed (e.g., accessing a shared variable, reading the other entity's state, wireless transmission, etc.) is irrelevant for our investigation. Analogously irrelevant is how an entity performs its movement (e.g., extending and contracting its body, using wheels, transported by a service robot, etc).

The problem addressed by the WonnaGo ritual has two aspects. It is first of all a problem of *self-assembly* and *self-repair* of the system: initially located on a line, upon the (possible) failure of some entities, the non-faulty ones must reform the line excluding any faulty element. Solving this problem requires formulating a set of rules (the algorithm) that will allow the entities to form the line within finite time, regardless of the initial distribution and number of faults and of the local orientations of the non-faulty entities. It is also a problem of *coordinated moving* or *flocking*: the non-faulty entities must move away (possibly forever) while maintaining the line formation.

Unfortunately both these tasks, as formulated, are actually *unsolvable*, even in fully synchronous systems. In fact, there are initial configurations where unbreakable symmetries make it impossible to form a single line. Similarly, even if the non-faulty entities are all on a single line (e.g., if there are no faulty entities), if their number is even, there are assignments of the local orientations that render the flocking of the line impossible. Both impossibilities are circumvented by requiring that either one or two lines of equal size be formed, and that each formed line migrates maintaining its line formation; we shall call this problem *line recovering*. The Astracinca must have been aware of those impossibilities; in fact, the WonnaGo ritual meets precisely this requirement.

In this paper we present the deciphered ritual and analyze its effectiveness. We prove that the set of deciphered rules always and correctly leads the non-comatose participants to form either a single line or two lines of equal size, and each line moves to find new locations. We also prove that the short ritual allows the process to be performed in optimal time.

Additionally, we provide a ritual simulator. The C source code, a pre-compiled binary for Windows systems, and the relative instructions can be found at `http://giovanniviglietta.com/files/rupestrian/Simulator.zip`.

Summarizing, we show that the Astracinca have developed a synchronous protocol for the *line recovery problem*; the protocol is totally *decentralized*, fully *concurrent*, provably *correct*, and *time optimal*.

2 System Model and Rite Purpose

We consider the space to be an infinite unoriented anonymous mesh $G(V, E)$, i.e., the nodes in V are all equal, edges are bidirectional and unlabeled, G is constituted by an infinite number of rows and columns. The tribe is a set of n persons in distinct positions in G. Each person p is modeled as a tuple (x, s, dir, pre, b), where $x \in V$ represents the person's *position*, $s \in S$ is a *state* (where S is a fixed finite set of possible states, with $S \supseteq \{sleeper, settler\}$), $dir, pre \in D = \{up, down, left, right, none\}$ represent two *directions* to which the person is pointing (the current direction dir and the previous location pre), and $b \in \{0, 1\}$ is a *bump flag*.

The purpose of the bump flag is to let a person know that they bumped into someone while trying to move to a location, as will be explained shortly.

Given a node $x \in V$, $N(x)$ is the set of its four neighbors. For explanation purposes, we use a global reference system, which allows us to give consistent labels in $D \setminus \{none\}$ to the edges from x to $N(x)$. This reference system is unknown to the persons. Each person p has their own reference system: when on node x, person p associates to each node in $N(x)$ a direction in $D \setminus \{none\}$. The *neighborhood* of p in x is an ordered list of elements $\{up, down, left, right\}$. An element is *empty* if on the corresponding node, with respect to the reference system of p, there is no person. Otherwise, if there is an agent $p' = (y, s', dir', pre', b)$ in that direction, then the element is $(s', T(p, p', dir'), T(p, p', pre'))$, where T is a function that translates the direction d' of p' in the reference system of p. Essentially, a person can see the states of persons on the neighboring nodes and their directions, but not their coordinates and their bump flags.

Time is divided in fixed size intervals (*rounds*), $r \in \mathbb{N}^+$. At each round r, every person with state $s \neq sleeper$ is activated. Upon activation, a person performs some operations based on their *view* (i.e., their state, directions and neighborhood at the beginning of r). The operations to be executed are determined according to a set of rules, called *ritual*, which is the same for all persons. Given a person's view at the beginning of a round, the ritual specifies whether the person must move and where, and indicates the new state and the directions at the end of the round. A person in node x at round r may move to any $y \in N(x)$. If at round r several persons move towards the same empty node y, only one of them succeeds and will be at node y at the beginning of round $r + 1$. All other persons remain in their nodes and at the beginning of round $r + 1$, and will have the bump flags set.

Given a set of persons we say that they are on a *straight line* if they are all on the same row or column of G. We say that they are on a *compact straight line* if they are on a *straight line* and the subgraph induced by their positions is connected. We say that a set of persons is oriented in direction d if their directions dir once translated in the global reference system

are all equal to d. Initially, at round $r = 1$, all persons are positioned on a compact straight line (the "initial line"), $f \geq 0$ of them have state *sleeper*, all the other $n - f \geq 5$ have state *settler* and directions set to *none*.

The *line recovery* problem is solved at round r^* if, for any round $r \geq r^*$, the initial *settlers* form a straight line and they are oriented in a certain direction, or they form two straight lines of equal size but with opposite orientations.

3 The WonnaGo Ritual

3.1 Overall Description

In the following, we will assume that persons can exchange fixed-size messages: this can be easily simulated in our model. Also, if not otherwise specified, the variable *dir* of a person p stores the movement's direction of p, that is, it stores the location where p intends to move; when no ambiguity arises, we will use the expression "direction of p" to indicate the content of *dir*. Similarly, the content of variable *pre* stores the location of p in the previous round; again, when no ambiguity arises, we will use the expression "previous location of p" to denote the content of this variable. We will say that p is *pointing at* a person p' if p and p' are neighbors, and the direction *dir* of p is toward the location occupied by p'. Finally, when a person p changes state from s, we will say that p *becomes* s.

The WonnaGo ritual is divided in several sub-rites: Exodus, Explorer Divination, Marker Creation, Chief Identification, The Chosen One, Opposite Sides, and Same Side. Let L_0 be the row where the initial line is placed, and L_1, L_{-1} the two rows adjacent to L_0.

The rite starts by checking whether there are no *sleepers*: in this case, all *settlers* are already in a compact line. This scenario is detected during the Exodus sub-rite, started (at the first round) by the settlers who occupy the extreme positions of the starting configuration, i.e., by the two persons having only one neighbor. These two extreme *settlers* send a special message inside the line: if the two messages meet, there are no *sleepers*; otherwise, the Exodus gets interrupted.

Should there be *sleepers*, the second sub-rite (the Explorer Divination) is performed, started by all the *settlers* who have a *sleeper* neighbor. In this sub-rite some *settlers* become *explorers* and move out of the line. The selection of the *explorers* is made in such a way that their movement does not create "gaps" of more than two consecutive empty positions anywhere in the original line (this property is crucial to detect the end of the line in subsequent sub-rites). Notice that it is possible that both the Exodus and the Explorer Divination sub-rite are started concurrently, in which case the Exodus process will die.

After stepping out of the initial line, the *explorers* start moving in a direction of their choice, and the Marker Creation sub-rite begins. The goal of this sub-rite is to place an *explorer* at each end of the original line so to mark it for subsequent rites. To achieve this, the *explorers* move along the chosen direction.

An *explorer* that sees the end of the line (i.e., three consecutive empty positions), moves immediately after it and becomes a *marker* with *scepter flag* set. After the *marker(s)* are created, the other explorers continue to move towards the end of the line: this is handled in the Chief Identification sub-rite.

The main goal of the Chief Identification sub-rite is to select at most two *explorers* as *chiefs*. In particular, when an *explorer* reaches a *marker* with scepter flag set, they become a *chief*, and the *marker* loses the scepter. Then, the *chief* inverts direction and tries to reach the other end of the line, i.e., the other *marker*. Due to concurrency, several scenarios can occur which make the *chief* understand whether it is unique or not. In case there are two

chiefs, each of them will understand whether they are located in different lines (L_1, L_{-1}) or on the same line (L_1 or L_{-1}) with opposite directions. Depending on the situation, a new sub-rite begins (The Chosen One, Opposite Sides, or Same Side).

The Chosen One sub-rite handles the easiest of the three possible outcomes of the Chief Identification sub-rite: in this case one of the *chiefs* is the first to reach also the other *marker* with scepter flag set (or reaches an extreme of the line with no *marker* at all), and becomes the (unique) *chosen one*. During this sub-rite, the *chosen* moves through the line, collecting anyone who is not *sleeper*,

forming a procession that will complete the assigned task.

In the Opposite Sides sub-rite, the two *chiefs* realize (when they reach the *marker* at the opposite end of their respective line) to be located on two different lines (i.e., one on L_1 and the other on L_{-1}): in this case, each *chief* becomes a *collector*, and a two-phase process is started. In the first one, a *collector* goes to the other *marker*, moving every two rounds, and collecting all people encountered on the way as well as the *settlers* still on L_0. The second phase is a counting process, which is an attempt to establish which of the processions formed by the two *collectors* is the longest. If one of the two processions is longer, its *collector* is elected, performs a final loop of the line, collecting everybody, and eventually forms a unique straight line, thus completing the task. Otherwise, in the case the two processions have the same length, the two *chiefs* move along opposite directions, until two distinct straight lines are formed, thus completing the task.

The Same Side sub-rite occurs when a *chief* meets another *chief*: in this case, the two *chiefs* realize to be on the same line, say L_1, and that they are moving in opposite directions. As soon as the two *chiefs* meet, they become *opposers*, switch directions, and move along the new direction, collecting everybody they encounter along the way. When an *opposer* reaches a *marker*, they start the final *collecting* phase: they keep moving in the same direction (on L_{-1}) and collect the *settlers* still on L_0. Eventually, the two *opposers* meet on L_{-1}: at this point, messages are exchanged among the people within the processions led by the two *opposers*, in an attempt of electing one of the *opposers*. If this is possible, the *opposer* that gets elected starts moving until the procession forms a straight line, thus completing the task. Otherwise (i.e., it is not possible to elect a unique *opposer*), the two *opposers* move along opposite directions, until two straight lines of equal size are formed, thus completing the task.

In the following the sub-rites are detailed; the complete set of rules and the reproductions can be found at the end of the paper.

3.2 Sub-rites

Exodus. In the Exodus sub-rite the *settlers* detect if there are no *sleepers* in the system, and in that case they elect one or two *exodus.leaders*, who will lead the migration. This is done as follows: in the first round, a *settler* detects if they are at the extreme of the line, i.e., they have only one neighbor. If this is the case, the *settler* becomes a *marker* with scepter flag set. At the end of the first round the *marker* sends an "Exodus?" message to an active neighbor, if any exists. A *settler* receiving such a message propagates it to the next *settler*. If there are no *sleepers*, then the two "Exodus?" messages meet; at that point, depending on the parity of the number of *settlers*, either one or two lines of equal size are formed. Otherwise (i.e., there are *sleepers*) the Explorer Divination sub-rite is eventually performed.

Explorer Divination. The sub-rite Explorer Divination is used to bootstrap the other sub-rites, and it is executed if at least one *sleeper* is present. The purpose of the rite is to select

at least three *explorers* among the *settlers*, who will move out of the line without creating empty "gaps" of more than two consecutive positions. This is done as follows. If a *settler* has a *sleeper* neighbor, then they become an *explorer* and they notify this decision to the neighboring *settler*, if any exists. Upon receiving such a notification, a *settler* becomes *settler.notified*. Any *settler.notified* who is not neighbor of another *settler.notified* becomes an *explorer* as well.

At this point (the fourth round) all *explorers* step out of the initial line.

Marker Creation. In this sub-rite the *explorers* move along the line until they find the end, which is detected by seeing three consecutive empty locations. The first *explorer* reaching the end of the line becomes a *marker* with scepter flag set and stays there. If two *explorers* try to become *marker* on the same extreme at the same time, only one is allowed to do so. Note that, when two *explorers* meet, they cannot pass through each other, and therefore they simply switch directions.

Depending on the initial configuration, either one or two *markers* are created by the end.

Chief Identification. The purpose of the Chief Identification sub-rite is to let at most two *explorers* become *chiefs*, and for a *chief* to determine if it is unique. An *explorer* who reaches a *marker* with scepter flag set receives the scepter flag from the *marker* and becomes a *chief*; should two *explorers* reach the same *marker* with scepter flag set at the same time, the *marker* will give the scepter to only one of them.

A newly elected *chief* now has to determine if they are the only one; to do so, they switch direction trying to reach the other extremity of the line. If the *chief* meets an *explorer* coming from opposite direction, it "virtually" continues its walk by switching roles with the explorer: the *explorer* becomes *chief* and switches direction, and the old *chief* becomes a *disciple* and stops. Similarly, if a *chief* and a *disciple* meet, they switch roles. If an *explorer* meets a *marker* without scepter flag or a *disciple*, it becomes *disciple* and stops.

There are three possible scenarios: (1) the *chief* reaches the other extremity, finding a *marker* with scepter flag or three empty locations; (2) the *chief* reaches the other extremity, finding a *marker* without scepter; (3) the *chief* meets another *chief*. Scenario (1) implies that the *chief* is unique; in this case the sub-rite The Chosen One starts. Scenario (2) implies that there are two *chiefs* who are moving on two different lines, adjacent to the initial one; in this case the sub-rite Opposite Sides is started. Scenario (3) means that there are two *chiefs* who are moving on the same line, adjacent to the initial one; in this case the sub-rite Same Side starts.

The Chosen One. The rules of the sub-rite The Chosen One are executed when a *chief* reaches a *marker* with scepter flag or detects that there is no *marker* on that side (three empty spots). When this happens, the *chief* becomes the *chosen*. The goal is for the *chosen* to collect everybody and eventually form a single line. To do so, the *chosen* reverses direction and moves, having everyone they meet follow them according to the Recruitment Procession rules described later. This movement is performed as follows: when the *chosen* meets a *disciple*, the *chosen* becomes a *follower* and the *disciple* becomes the *chosen*. When a *chosen* meets an *explorer*, a similar thing happens. In this process, a *settler* who sees a procession of *followers* will try to join the procession. Eventually, the *chosen* will reach the *marker*. When this happens, the *marker* becomes the *chosen*, and the *chosen* a *follower*. If there are no *disciples* around the *marker*, the *chosen* moves outside of the line and takes as direction the other endpoint of the line.

Opposite Sides. This sub-rite starts when a *chief* on L_1 (respectively, L_{-1}) reaches a *marker* m without the scepter flag: the chief understands that there is another *chief* moving in the same direction (clockwise or counter-clockwise) on L_{-1} (respectively, L_1). The *chief* becomes *collector*, switches direction, and moves toward the other *marker* m'; the *collector* moves every two rounds. During this swipe of L_1, the *collector* recruits all the encountered people, including the *settlers* still on L_0, thus forming a procession.

After at most $2n$ rounds, they reach the other *marker* m'. Let us assume for now that, during the swipe, they recruited at least one person. The *collector*, say x, and the closest recruited *follower*, say z, start now a phase to determine whether or not it is possible to form a unique procession. If not, two distinct processions of the same length will be formed. Specifically, at round r, z becomes *mover* and moves on L_1 towards *marker* m', eventually reaching it. Meanwhile, at round $r + 2$, x becomes *collector.counting* and does not move (i.e., x remains close to *marker* m). After a finite number of rounds, a *marker* will have as neighbors both a *collector.counting* and a *mover*. When this occurs, the *marker* signals this event to both of them, say at round $r' > r + 2$. At this time, the *collector.counting* and the *mover* simultaneously change state, becoming *probes*.

The *marker* also memorizes the line where the *collector.counting* lies; this information will be used to break possible symmetric scenarios.

Let us now focus on one of the two *markers*, say m: the two *probes* generated by m start moving towards m', one on L_1 and the other on L_{-1}, using the *probe move* protocol described below. In Section 4, we will prove that both *probes* reach m' at the same time if and only if the two processions formed by the two *collectors* have the same number of people; in fact, should one procession be smaller than the other, the *probe* traversing it will reach a *marker* before the other. In any case, each *marker* will know whether the two processions have the same length and, if not, which one is smaller.

The *probes* move according to the following protocol: each *probe* has a modulo-5 counter, initially set to 1. If the counter is greater than 0, the *probe* decrements it at each round. When the counter reaches 0, the *probe* moves to the next location. If the next location is empty, the *probe* moves and the counter remains 0. If, instead, the next location is occupied by a person p, then the *probe* "virtually" moves having p take the state of the *probe*, while the *probe* takes the state of p; in this case, p adds 1 to the counter (it adds another 1 to the counter if there is a *prefollower* pointing at p; refer to Section 3.3). If a *probe* reaches a *marker*, they first decrement the counter until 0, they then become a *follower* pointing towards the *marker*. There are two exceptions to this general rule: (E1) if at the same time two *probes* with counter 0 are neighbors of the same person p, they both take the state of p, p waits an appropriate number of rounds and signals to both neighbors (i.e., the old *probes*) to become *probe* pointing away from them with an appropriate set counter; (E2) two *probes* may move to the same empty location, with one of them bumping back. When this happens, both *probes* change direction but the one that did not bump sets the counter in such a way that they wait one round less; this is done to account for the fact that the bumping person did not move.

Eventually, a *probe* reaches a *marker*. If a *marker* detects that on the memorized side there is a new *probe* with counter 0, and on the other side there is no one or there is a *probe* with counter greater than 0, then the *marker* becomes a *consul* and points to the side opposite to the one they memorized. In this case a *consul* behaves as a *chosen* (see The Chosen One sub-rite), and does a loop around L_0 collecting all people (both the ones encountered on their way and the *settlers* still on L_0). Note that in this case the other *marker* does nothing, waiting to be collected. If at a given round a *marker* detects that on

the memorized side there is a new *probe* with counter 0, and on the other side there is also a *probe* b with counter 0, then that the two processions have the same length (see Section 4). In this case, the *marker*, say m, becomes a *follower* and signals to b to become a *consul*; the *consul* will lead the procession away from L_0. Symmetrically, the same will happen on the side of the other *marker* m'.

At the beginning of this sub-rite, we assumed that the *collector* recruited at least one person. If this is not the case, when the *collector* reaches a *marker*, they become a *collector.counted* and move towards the other *marker*. If a *marker* sees a *collector.counted* and a *collector.counting*, they elect the *collector.counted*, appointing them a *consul*. Also in this case, a *consul* behaves as a *chosen*, and does a loop around L_0 collecting everyone encountered on the way (including the *settlers* still on L_0).

Same Side. In the Same Side case a *chief* meets another *chief*; let us assume, without loss of generality, they are both on L_1. When this happens both *chiefs* become *opposers*, they switch directions and move towards a *marker*. If an *opposer* moving towards a *marker*, say m, meets an *explorer* on its way, and their directions are opposite, the *opposer* continues moving collecting the *explorer* (i.e., the *opposer* becomes a *follower*, and the *explorer* becomes an *opposer* that still moves towards m). Eventually, the *opposer* reaches a *marker*: when this happens, the *opposer* moves to the spot occupied by the *marker* collecting the *marker* (i.e., the *opposer* becomes a *follower*, the *marker* becomes *opposer* and sets the follow-me flag, continuing to move in the same direction of the old *opposer*). From the next round on, the *opposer* (who is now on L_{-1}) starts collecting all encountered people (the ones met on L_{-1}, as well as the *settlers* still on L_0), thus forming their own procession.

Eventually, one of the two following scenarios can occur: (1) the two *opposers* are at distance 1 and between them there is a *disciple* (see Chief Identification sub-rite). In this case, both *opposers* become *followers* and the *disciple* becomes *opposer.winner*. (2) The two *opposers* meet on L_{-1}. When this occurs, they first wait two rounds (giving enough time to their immediate *followers* to reach them). Then, they both change state to *opposer*.waiting. At this point, each *opposer*.waiting starts a *straight line check* phase, by sending a *"straight line query"* to their procession (see Section 3.3) to figure out if it is possible to select a unique winner to lead the procession. Three cases can occur. (a) The two processions have different length: one of the two *opposer.waiting* is elected, becoming an *opposer.winner*, while the other *opposer* becomes a *follower*. (b) The two processions have equal length, and one procession is not already forming a straight line: the *opposer.waiting* who leads this procession becomes a *follower* and the other becomes *opposer.winner*. (c) The processions have equal length and they are both forming a straight line: in this case two *opposer.winners* are elected.

In cases (1), (2).a and (2).b, the only *opposer.winner* will lead the final migration; in case (2).c, the two *opposer.winners* set the flag tail, reverse direction, and each of them will lead the migration of their own procession.

3.3 Sub-Phases

The following two processes, *Recruitment Procession*, and *Straight Line Query*, are required in some of the rites described in the previous section to create and maintain a procession, and to compare two created processions, respectively.

Recruitment Procession. A key procedure of the rite is the construction of a procession, a group of people following a designated leader during the migration. The leader is called the

head of the procession; all other people in the processions are called the *followers*, and the last *follower* is the tail (i.e., they have the tail flag set).

Let us show the details of the procession creation and maintenance.

A procession is *created* by a head; initially the head is also the tail of the procession. If the head wants to recruit *settlers* (i.e., people who are still on L_0) then they set the additional follow-me flag. When a *settler* sees a head p with the follow-me flag, they change their state to *prefollower*; furthermore, they memorize the direction, the tail flag, and the previous location of p. At the next round this *prefollower* will try to move in the location where p was seen. If such a move succeeds, they change state to *follower* and update their direction, tail flag, and previous location to the memorized values. If instead the location is occupied by another person p', who must also be part of a procession, the *prefollower* memorizes the direction and tail flag of p' and replicates the aforementioned steps; eventually, the *settler* will be able to leave L_0 and join the procession.

During the whole ritual, it might happen that new *followers* join the procession, hence the tail changes: if a *follower* has the tail flag set, and sees in their previous location a *follower* or a *prefollower* whose direction is pointing at then, then they lose the tail flag and skip the current round.

The procession *moves* according to the following general rule: a *follower* whose direction is not occupied by anyone and whose previous location is occupied moves towards the location pointed by the direction.

There are three exceptions to this rule: (E1) a tail moves regardless of the presence of someone in its previous location; (E2) if a *follower* moved in the previous round and they find that there is no person in their direction, than they change the direction to point to the only *follower* (or head) who is present in a location perpendicular to the old direction, towards L_0. This rule is used by *followers* to move *around* the extremes of L_0, thus allowing the procession to *loop* around the starting line; (E3) the last exception is given by a head that has both follow-me and tail flag set: in this case, when the head sees that there is a *settler* who may join the procession, they move to the next location and they wait for one round (giving time to the *settler* to join the procession).

A special procedure is executed when the head meets a *marker* or someone on their path: in this case, the head becomes a *follower* and a new head is elected (as an example, see Rules 4.F, 4.G of The Chosen One case).

Straight-Line Query. A stationary head can check if their own procession is aligned or not, by sending a *"query message"* to the closest follower in the procession (technically, they send the message to themselves, and then they process this message as a *follower*). Upon reception of the *"query message"*, a *follower* checks if there is a *settler* (or a *prefollower*) in a "non-previous" location who is pointing at them; if so, they wait two rounds. After waiting, they send the *"query message"* to the *follower* in the previous location. The *"query message"* is then propagated in this direction along the procession until it reaches the tail. When the message reaches a tail, they wait using the same rules of the *followers*, and then they send a *"straight message"* back towards the head (again, technically, they start the propagation of this message by sending it to themselves). This message is thus propagated through the *followers* in the procession, with the additional rule that it is changed from *"straight message"* to *"not straight message"* by a *follower* who has both direction and previous location set to vertical. When the head receives the *"straight/not straight message"*, the query terminates.

4 Analysis

We assume that at least $n - f \geq 5$ *settlers* participate in the rite. A settler steps out of the initial line if and only if they become an *explorer* or if they see a procession. Thus we can observe that, if $f = 0$, the *settlers* do not move outside the initial row.

▶ **Lemma 1.** *The* Exodus *sub-rite terminates and it elects an* exodus.leader *if and only if* $f = 0$*. If n is even, then two* exodus.leaders *are elected; if n is odd, one* exodus.leader *is elected. When the* Exodus *sub-rite terminates the rite specifications are satisfied.*

▶ **Lemma 2.** *If $f > 0$, then at the end of the third round we have that either there is at least one* marker *with flag scepter and two* explorers*, or there are at least three* explorers*.*

From the rules of the Explorer Divination sub-rite we have the following corollary.

▶ **Corollary 3.** *At the end of the third round, between the endpoints of the initial line, there cannot be three consecutive empty locations.*

In the following we assume $f > 0$.

▶ **Lemma 4.** *There exists a round $r \in [3, 4n + 3]$ in which the rite is in one of the following configurations:*

(C1) *There is a* chosen (The Chosen One).

(C2) *There are two* markers *without sceptre and two* chiefs *on opposite sides of the initial line* (Opposite Sides).

(C3) *There are two* markers *without sceptre and two* chiefs *on the same side of the initial line* (Same Side).

Let a procession be called *pious* if it does not contain two consecutive empty spots and no follower without tail has empty spots both ahead and behind in the procession.

▶ **Lemma 5.** *All the processions generated by* WonnaGo *are pious.*

▶ **Corollary 6.** *Let us consider a* follower *who is not a tail. If at round r there is no one in the* follower*'s previous location, then there will be a* follower *at round $r + 1$.*

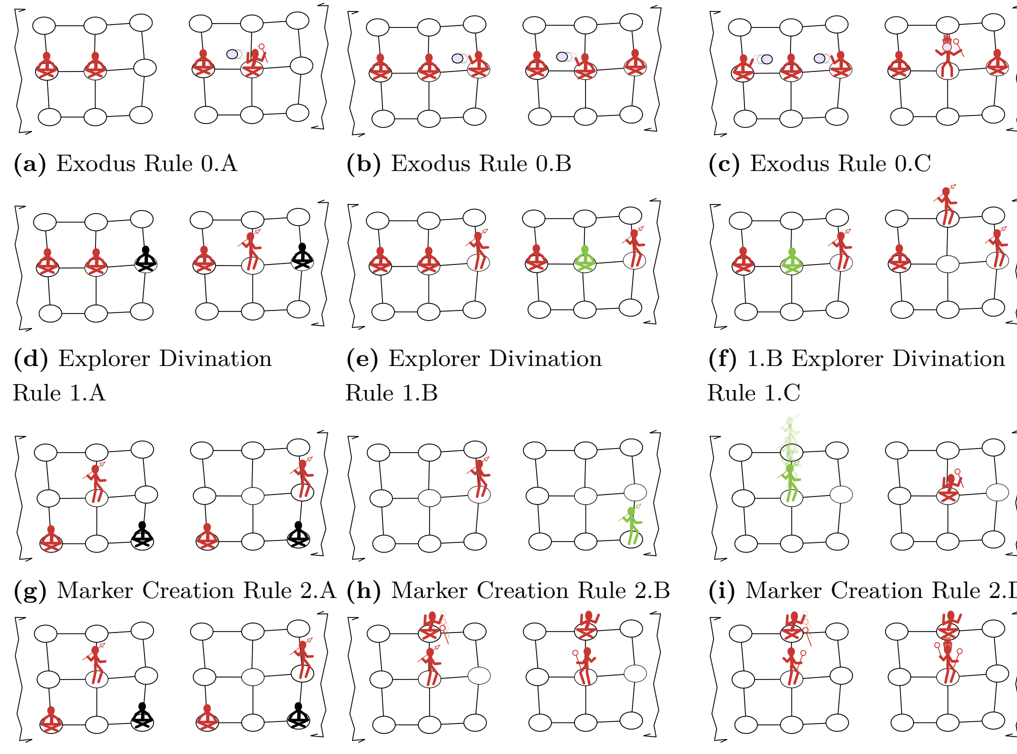
▶ **Corollary 7.** *If there is a unique procession, the head occupies a new location within a constant number of rounds.*

▶ **Theorem 8.** *If a* chosen *is created, then it is unique, and line recovery is achieved within $\mathcal{O}(n)$ rounds.*

▶ **Theorem 9.** *Let the tribe reach, at round r, a configuration in which there are two* markers *without scepter and two* chiefs *on opposite sides of the initial line. Then, line recovery is achieved within $\mathcal{O}(n)$ rounds.*

▶ **Theorem 10.** *Let the tribe reach, at round r, a configuration in which there are two* markers *without scepter and two* chiefs *on the same sides of the initial line. Then, line recovery is reached in $\mathcal{O}(n)$ rounds.*

Due to Lemma 4 and Theorems 8, 9, 10, line recovery is achieved in $\mathcal{O}(n)$ rounds.



(a) Exodus Rule 0.A **(b)** Exodus Rule 0.B **(c)** Exodus Rule 0.C

(d) Explorer Divination Rule 1.A **(e)** Explorer Divination Rule 1.B **(f)** 1.B Explorer Divination Rule 1.C

(g) Marker Creation Rule 2.A **(h)** Marker Creation Rule 2.B **(i)** Marker Creation Rule 2.D

(j) Marker Creation Rule 2.E **(k)** Chief Identification Rule 3.A **(l)** The Chosen one Rule 4.A

Figure 2 Reproduction of paintings depicting specific rules of the WONNAGO.

5 Forming a compact straight line

The WONNAGO ritual solves a convergent task. However, it is also possible to modify the rite to obtain a solution to the stronger *compact line recovery* problem, where an explicit termination is required and the final configuration has to be a compact line (or two compact lines of equal length). To terminate explicitly, the head of a procession has to know when all *settlers* (or half if there exists another procession with opposite direction) have joined the procession and the procession is on a straight line. Knowing this, the head stops moving and, within n rounds, all *followers* will be in a compact position.

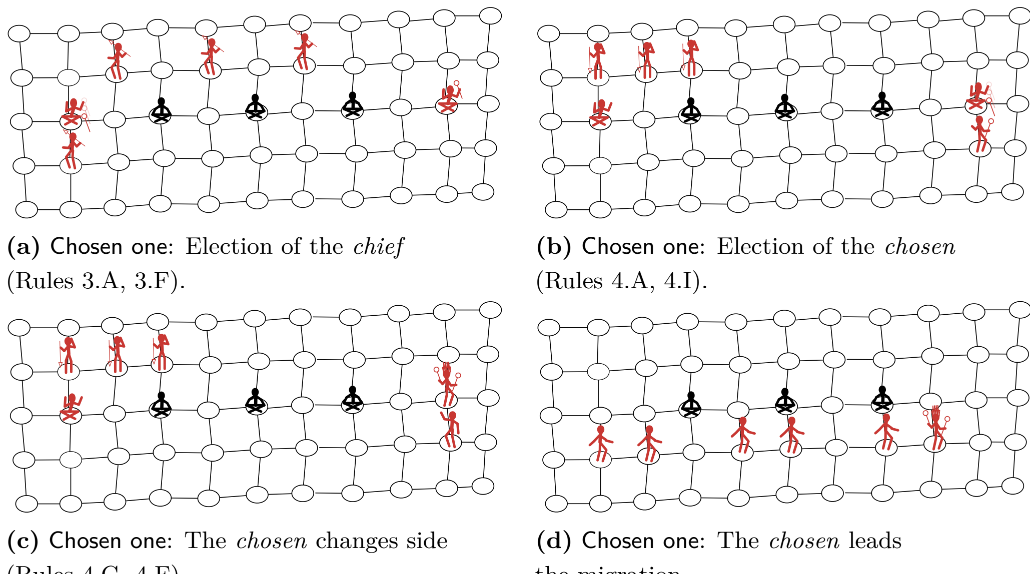
6 The Rules

- **Exodus 0:**
  - **Rule 0.A: If** I am a *settler* and I have only one neighbour and it is the first round, **then** I become a *marker* with flag sceptre. Moreover, if my neighbour is non-*sleeper* I ask they “Exodus?”.
  - **Rule 0.B:If** I am a *settler* and I receive for first time an “Exodus?” from only one neighbour and my other neighbour is a *settler*, **then** I ask they “Exodus?”.
  - **Rule 0.C: If** I am a *settler* and I receive “Exodus?” from both neighbours, **then** I become an *exodus.leader* and I set *dir* to point one of my neighbours.
  - **Rule 0.D: If** I am a *settler* and I receive an “Exodus?” message in two consecutive rounds, **then** I become an *exodus.leader* and I set as *dir* the direction of the first received message and as *pre* the opposite.

- **Rule 0.E: If** I am an *exodus.leader*, **then** I send a "setdir message" to my neighbour pointed by *dir* and a "setbackdir message" to my neighbour pointed by *pre*.
- **Rule 0.F: If** I am a *settler* or a *marker* and I receive a "setdir message" from neighbour p in location x, **then** I set my *dir* to point at the opposite location of x and I send the message to neighbour in location pointed by *dir*.
- **Rule 0.G: If** I am a *settler* or a *marker* and I receive a "setbackdir message" from neighbour p in location x, **then** I set my *dir* to point at location of x and I send the message to neighbour in location opposite to x.

- **Explorer Divination (1):**
  - **Rule 1.A: If** I am a *settler* and I have two neighbours and I am near a *sleeper*, **then** I become an *explorer*. My direction *dir* will be pointing at the *sleeper*. If they are both sleepers the tie is broken arbitrary.
  - **Rule 1.B:If** it is the second round and I am a *settler* and I have only one *explorer* as neighbour, **then** I become a *settler*.notified. My direction *dir* will be pointing at my *explorer* neighbour.
  - **Rule 1.C: If** it is the third round and (I am an *explorer* or (I am a *settler*.notified and I do not have as neighbour a *settler*.notified)), **then** I become *explorer* and I move perpendicularly to my neighbours (up or down).
- **Marker Creation (2):**
  - **Rule 2.A:If** I am an *explorer* and there is no one in the *dir* direction and there is no (*marker*, *premarker*, *chosen*, *opposer* or *collector*) in my neighbourhood and on the initial line I have not seen three consecutive empty locations, **then** I move to that location.
  - **Rule 2.B:If** I am an *explorer* and on the initial line I have seen three consecutive empty locations, **then** I set my state to *premarker* and I move to occupy a free location on L_0.
  - **Rule 2.C:If** I am a *premarker* and I did bumped, **then** I become an *explorer*.
  - **Rule 2.D: If** I am a *premarker* and I did not bumped, **then** I become a *marker* with flag sceptre.
  - **Rule 2.E:If** I am an *explorer* and I see an *explorer* in my direction, **then** I switch the direction *dir* and I skip the current round.
- **Chief Identification (3):**
  - **Rule 3.A:If** I am an *explorer* and I see a *marker* with flag sceptre and I receive the sceptre from they, **then** I become a *chief* and I switch the direction *dir*.
  - **Rule 3.B:If** I am an *explorer* and I see a *marker* with flag sceptre and I do not receive the sceptre from they, **then** I become a *disciple*.
  - **Rule 3.C:If** I am an *explorer* and I see a *marker* without sceptre **then** I become a *disciple*.
  - **Rule 3.D:If** I am an *explorer* and there is a *chief* pointed by my *dir* and the *dir* of the *chief* is pointing at me, **then** I become a *chief* and I switch direction *dir*.
  - **Rule 3.E:If** I am an *explorer* and there is a *disciple* pointed by *dir*, **then** I become a *disciple*.
  - **Rule 3.F:If** I am a *marker* with flag sceptre and I see some *explorers* and I do not see a *chief*, **then** I unset the flag sceptre and I give the sceptre to only one *explorer*.
  - **Rule 3.G:If** I am a *chief* and there is an *explorer* pointed by my *dir* and the *dir* of the *explorer* is pointing at me, **then** I become a *disciple*.
  - **Rule 3.H:If** I am a *chief* and there is a *disciple* pointed by my *dir*, **then** I become a *disciple*.

- **Rule 3.I:If** I am a *disciple* and there is a neighbour *chief* whose direction *dir* is pointing a me, **then** I become a *chief* and my direction *dir* will be the one of the old *chief*.



(a) Chosen one: Election of the *chief* (Rules 3.A, 3.F).

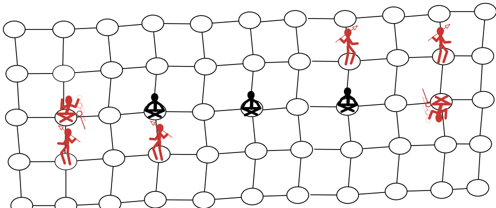
(b) Chosen one: Election of the *chosen* (Rules 4.A, 4.I).

(c) Chosen one: The *chosen* changes side (Rules 4.G, 4.F).

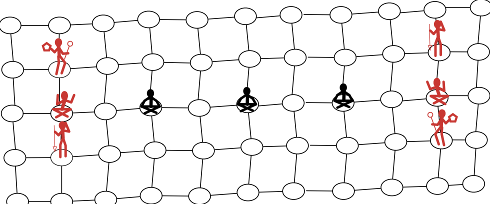
(d) Chosen one: The *chosen* leads the migration.

Figure 3 Reproduction of paintings depicting a rite in which a *chosen* is elected.

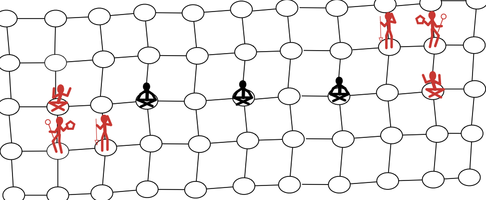
- **The Chosen one (4):**
  - **Rule 4.A:If** I am a *chief* and I see a *marker* with flag sceptre, **then** I become the *chosen*, I set the follow-me flag and the tail flag and I skip the round.
  - **Rule 4.B:If** I am *chosen* and there is a person pointed by *dir*, **then** I become a *follower* and I unset the follow-me flag.
  - **Rule 4.C:If** direction *dir* of a *chosen* is pointing at me, **then** I copy the state of the *chosen* and I set direction *prev* to point the location the old *chosen*.
  - **Rule 4.D:If** I am *chosen* and I do not see a *marker* and in my direction *dir* there is no one and my direction *dir* is not pointing to a location outside L_1, L_0, L_{-1}, **then** I move to that location.
  - **Rule 4.E:If** I am *chosen* and I do not see a *marker* and in my direction *dir* there is no one and my direction *dir* pointing to a location outside L_1, L_0, L_{-1}, **then** I switch *dir* to point towards the farther endpoint of the line.
  - **Rule 4.F:If** I am *chosen* and I am a neighbour of a *marker* without sceptre, **then**: I become *follower* and I set *dir* to point at the *marker*.
  - **Rule 4.G:If** I am a *marker* and I am a neighbour of a *chosen*, **then** I copy the state of the *chosen* and I set direction *prev* to point at location the old *chosen* and as direction *dir* the opposite location.
  - **Rule 4.H:If** I am a *chief* and on the line I see three consecutive empty locations, **then**: I set my state to *chosen*, I set the follow-me flag and the tail flag and I set as direction *dir* the only location in L_0.
  - **Rule 4.I:If** I am a *marker* with flag sceptre and I see a *chief*, **then** I unset the flag sceptre.
- **Opposite Sides (5):**
  - **Rule 5.A: If** I am a *chief* and I reach a *marker* M without sceptre, **then** I become a *collector*, I set the follow-me and tail flags, and I switch direction *dir*.



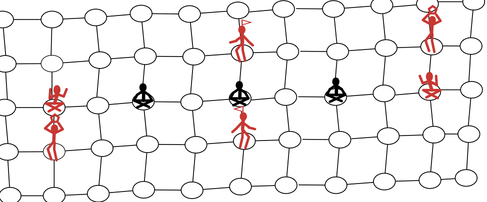
(a) Opposite Sides: Two *chiefs* are elected on the opposite side (Rules 3.A, 3.F).



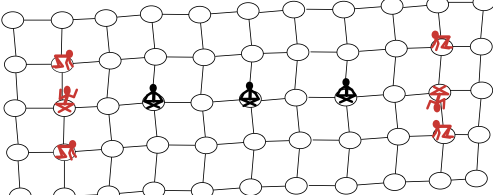
(b) Opposite Sides: The two *chiefs* become *collector*.



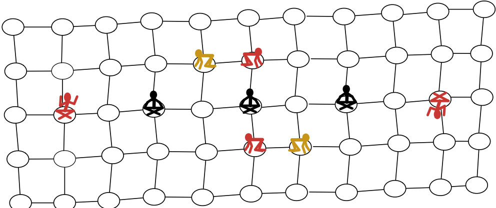
(c) Opposite Sides: Two *collector*s order the *followers* to become mover (Rule 5.E).



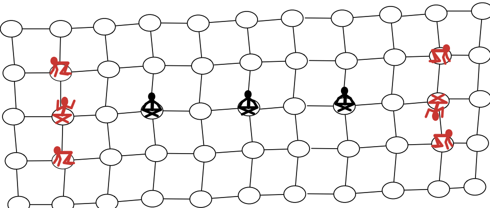
(d) Opposite Sides: The *movers* are going towards the other *marker* (Rule 5.H).



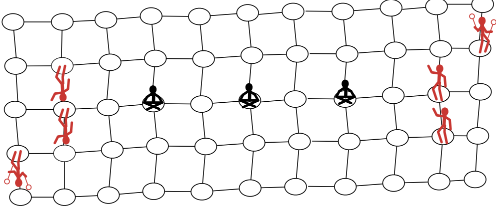
(e) Opposite Sides: The *probes* start moving on both sides of the line (Rule 5.N). Each *marker* memorised the side of the *collector.counting* (Rule 5.K).



(f) Opposite Sides: Bumping *probes*. Two *probes* have not bumped (colored in red), they immediately move (5.P). The *probes* in yellow have bumped, they have $probecounter = 1$ and they will decrease their counters in this round (Rules 5.O, 5.M).



(g) Opposite Sides: The *probes* reach the *marker* at same time on both sides. This triggers (Rule 5.U).



(h) Opposite Sides: Two processions with equal size are migrating in opposed directions.

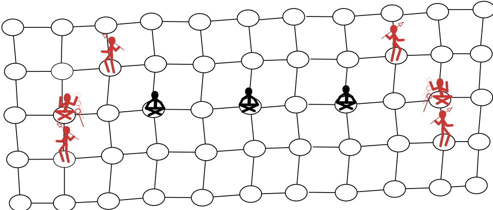
Figure 4 Reproduction of paintings depicting a rite in which the Opposite Sides case arise.

- **Rule 5.B: If** I am a *collector* and there is no one in the direction of *dir* and I do not see a *marker* and the round is even, **then** I move.
- **Rule 5.C: If** I am a *disciple* and the *dir* of a *collector* is pointing at me and the round is even, **then** I become a *collector* and I copy the directions $dir, prev$ and follow-me flag of old *collector* but not the tail flag.
- **Rule 5.D: If** I am a *collector* and my *dir* is pointing at *disciple* and the round is even,

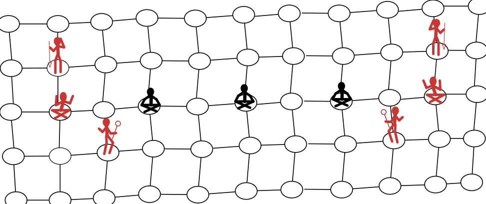
then I become a *follower*, if the tail flag is set I retain it.

- **Rule 5.E: If** I am a *collector* and I have reached the *marker* M', **then** I wait for one round then I signal to the *follower* pointed by *prev* to become a *mover* and after waiting two rounds I become a *collector.counting*. If there is no such *follower*, then I become a *collector .counted* and I switch the direction in *dir*.
- **Rule 5.F: If** I am a *mover* and my *dir* is pointing at a *follower*, **then** I become a *follower* and I copy the state of the pointed *follower*.
- **Rule 5.G: If** I am a *follower* and the *dir* of a *mover* is pointing at me, **then** I become a *mover* and I copy the *dir* of the old *mover*.
- **Rule 5.H: If** I am a *mover* or a *collector .counted* and there is no one pointed by *dir* and I do not see a *marker*, **then** I move in the direction.
- **Rule 5.I: If** I am a *marker* and I see a *collector .counted* and a *collector .counting*, **then** I signal the victory to *collector .counted*.
- **Rule 5.J: If** I am *collector .counted* and I receive the victory signal, **then** I become a *consul* and I point the *marker*. A *consul* obeys to the same rules of the *chosen*. If somebody see a *consul* they do as if they see a *chosen*.
- **Rule 5.K: If** I am a *marker* and I see a *mover* and a *collector .counting*, **then** I send a probing signal to both and I memorise the side of the *collector .counting* in *dir* for future reference.
- **Rule 5.L: If** I am a *mover* or a *collector .counting* and I receive a probing signal from the marker, **then** I become a *probe* and I set *probecounter* $= 1$ and I switch direction.
- **Rule 5.M: If** I am *probe* with *probecounter* ≥ 1, and there is no *probe* whose *dir* is pointing at me with *probecounter* $= 0$, **then** I decrease my *probecounter*.
- **Rule 5.N: If** I am a *probe* and my *probecounter* is 0, and in there is no one in the direction pointed by *dir*, **then** I move in my direction.
- **Rule 5.O: If** I am a *probe* and I see a *probe* whose *dir* is pointing at me, and one of us has *probecounter* $= 0$, **then** I take the state of the other one, I add 1 to *probecounter* and I further add one to *probecounter* if I see a *prefollower* whose *dir* is pointing at me. Moreover, If I bumped then I set a visible bumped flag in my state.
- **Rule 5.P: If** I am a *probe* and behind me there is a *probe* with visible bumped flag set, **then** I decrease my *probecounter*, If the *probecounter* is 0 then I move in the direction of *dir*.
- **Rule 5.Q: If** I am a *probe*, my *probecounter* $= 0$ in my direction *dir* there is a person that is not a *probe*, **then** I copy the state of the other.
- **Rule 5.R: If** I am not a *probe*, and there is only one *probe* whose *dir* is pointing at me with *probecounter* $= 0$, **then** I become a *probe* I copy the direction *dir* from the old probe and I set *probecounter* $= 1$ and I further add one to *probecounter* if I see a *prefollower* pointing at me.
- **Rule 5.S: If** I am not a *probe*, and there are two *probes* whose *dir* are pointing at me with *probecounter* $= 0$, **then** I wait one round and a further round if I see a *prefollower* whose *dir* is pointing at me. Then, I order each of my neighbours to become a *probe* with *dir* pointing away from me and set their *probecounter* to 2 or 1, respectively if they see a *prefollower* pointing at them or not. At the next round I take my original non-*probe* state.
- **Rule 5.T: If** I am *probe* and I have reached a *marker* and my *probecounter* $= 0$, **then** I become a *follower* and I set *dir* to point the *marker*.
- **Rule 5.U: If** I am a *marker*, the first *probe* that reaches me and has *probecounter* $= 0$ is on the side I memorised in Rule 5.K **then**

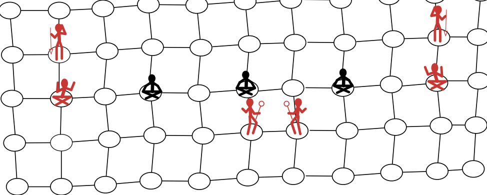
  * if on the other side there is a *probe* p that has *probecounter* > 0 or there is no one, **then** I become a *consul* and I point in the direction of p. A *consul* obeys to the same rules of the *chosen*. If somebody see a *consul* they do as if they see a *chosen*.
  * if on the other side there is a *probe* p that has *probecounter* $= 0$, **then** I order p to become *consul* pointing in the direction opposite to the other *marker*, and I become *follower* pointing at p and setting as *pre* the opposite.



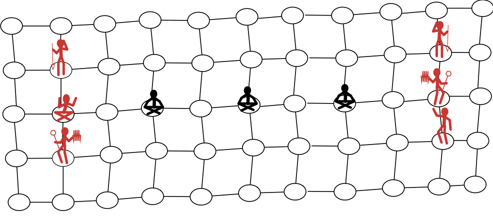
(a) Same Side: Two *chiefs* are elected on the same side (Rules 3.A, 3.F).



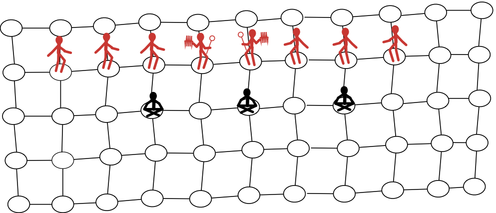
(b) Same Side: Two *chiefs* are moving towards each other.



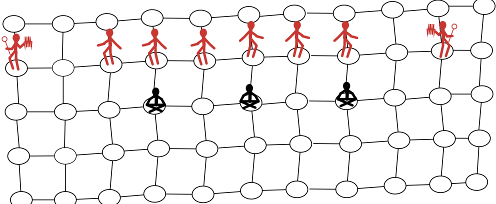
(c) Same Side: Two *chiefs* met (Rule 6.A).



(d) Same Side: Two *opposers* are changing side (Rule 6.F, 6.G).



(e) Same Side: Two *opposer* meet. They are executing the Straight Line Query.



(f) Same Side: Migration of two processions of equal size.

Figure 5 Reproduction of paintings depicting a rite in which the Same Side case arise.

- **Same Side (6):**
  - **Rule 6.A: If** I am a *chief* and there is another *chief* pointed by me, **then**: I become *opposer*, I set the tail flag and I switch direction.
  - **Rule 6.B:If** I am an *explorer* and there is a *opposer* pointed by me and the *opposer* is pointing at me, **then** I become an *opposer* and I switch direction for *dir* and I set *prev* to point at the location the old *opposer*.
  - **Rule 6.C:If** I am a *opposer* and there is an *explorer* pointed by me and the *explorer* is pointing at me, **then** I become a *follower*.
  - **Rule 6.D:If** I am *opposer* and I see a *disciple*, **then** I become a *follower*.
  - **Rule 6.E:If** I am a *disciple* and I see only one *opposer*, **then** I become the *opposer* and I set *prev* to point at the old *opposer* and *dir* to point at the location of the neighbour *disciple* if they exist otherwise the free location closer to the initial line.

- **Rule 6.F:If** I am a *marker* and I see an *opposer*, **then** I become an *opposer*, I set the follow-me flag and I set *prev* to point at the old *opposer* and *dir* as the opposite.
- **Rule 6.G:If** I am *opposer* and I see a *marker*, **then** I become a *follower* and I point at the *marker*.
- **Rule 6.H:If** I am *opposer* and I see another *opposer* pointed by me for two consecutive rounds, **then** I do a query in my procession and I set my state to $opposer.counting$.
- **Rule 6.I:If** I am an $opposer.counting$ and I see a $opposer.winner$, **then** I become a *follower*.
- **Rule 6.J:If** I am an $opposer.counting$ and my query terminates and the result is "straight" and I do not see an $opposer.winner$, **then** I become an $opposer.winner$.
- **Rule 6.K:If** I am an $opposer.counting$ and my query terminates and the result is "not-straight" and I do not see an $opposer.winner$, **then** I become an $opposer.winner$.
- **Rule 6.L:If** I am an $opposer.winner$ and the previous round I was an $opposer.counting$ and I have as neighbour an $opposer.winner$, **then** if we are both "straight" I set the tail flag and I switch direction *dir*, if I'm the only one "straight" I switch direction *dir*, if I'm not "straight" I become a *follower*.
- **Rule 6.M:If** I am a *follower* and there is an $opposer.winner$ that is pointing at me, **then** I become a $opposer.winner$ and I switch direction *dir*, I clear my tail flag, and I set *prev* to point to the old $opposer.winner$.
- **Rule 6.N:If** I am $opposer.winner$ and there is a *follower* pointed by me, **then** I become a *follower* and I point the *follower*.
- **Rule 6.O:If** I am a *disciple* and I see two *opposers*, **then** I become an $opposer.winner$.
- **Rule 6.P:If** I am a *opposer* there is no one in the *dir* direction and I do not see a *marker*, **then** I move.

- **Procession (7):**
  - **Rule 7.A: If** I am a *follower* without tail flag and I see a *follower* in my previous position and I do not see anyone in *dir* direction, **then** I move towards my direction.
  - **Rule 7.B: If** I am a *settler* or a *settler*.notified and I see an head with follow-me flag in location l, **then** I set my state to *prefollower*; from the person in l I copy their tail flag and their *dir* direction in my *pre*, and I point towards l.
  - **Rule 7.C: If** I am a *prefollower* and I bumped and there is no one in direction *dir*, **then** I move towards *dir*.
  - **Rule 7.D: If** I am a *prefollower* and I bumped and there is someone in direction *dir* , **then** I copy the tail flag from the person in *dir*.
  - **Rule 7.E: If** I am a *prefollower* and I moved, **then** I set my state to *follower* and I set my *dir* to be the *pre* and *pre* as the opposite of *dir*, the queue flag is set to the memorised value.
  - **Rule 7.F: If** I have a tail flag set and in the location pointed by *pre* there is a *prefollower* or *follower* neighbour pointing at me, **then** I unset my tail flag.
  - **Rule 7.G: If** I am a *follower* and I moved in the previous round and I do not see a *follower* or an head in direction *dir*, **then** I set *dir* to point towards the only head or *follower* perpendicular to my old *dir*.
  - **Rule 7.H:If** I am the head of a *followers* procession and I have the tail flag set and one of my neighbours is a *settler*, **then** I set the waiting flag and I move in my direction.
  - **Rule 7.I:If** I am the head of a *followers* procession, and at the previous round I had set the waiting flag, **then** I reset the waiting flag and I do not move in this round.
  - **Rule 7.J: If** I am the head of a *followers* procession, I do not have the tail flag, and I do not see a *follower* in my previous position, **then** I do not move.

  - **Rule 7.K: If** I am a *prefollower* and I bumped and in direction *dir* there is a *probe*, **then**, I skip this round.
- **Straight Line Query (8):**
  - **Rule 8.A:** If I receive a "query message" and there is no *settler* (or a *prefollower*) pointing at me, **then** if I am a tail, I send the "straight message" to myself. Otherwise I forward the "query message" at the person pointed by *pre*.
  - **Rule 8.B:** If I receive a "query message" and there is a *settler* (or a *prefollower*) pointing at me, **then** I skip two rounds. Then if I am a tail I send the "straight message" to myself. Otherwise I forward the "query message" at the person pointed by *pre*.
  - **Rule 8.C:** If I receive a "straight message", **then** If I my *dir* and *pre* are vertical I forward a "not-straight message". Otherwise, I forward "straight message".
  - **Rule 8.D:** If I receive a "not-straight message", **then** I forward a "not-straight message".
  - **Rule 8.E:** If I am an *opposer* and I receive a "not-straight message", **then** my query is terminated and the result is "not-straight".
  - **Rule 8.F:** If I am an *opposer* and I receive a "straight message", **then** my query is terminated and the result is "straight".

7 Conclusion

The ritual that we have analyzed in this paper is the *short* WonnaGo. The cave contains other paintings depicting a more complex version of the rite known as the *long* WonnaGo. The long rite was used in the autumnal months, when the new flowers of Cannabis have the greatest quantity of THC. It is very likely that the Astracinca had knowledge of the temporal distortion caused by phytocannabinoids, an effect studied only recently by modern medicine [2]. Essentially, the long ritual takes into account that different settlers may have a different perception of the passage of time, and that furthermore they can unpredictably slow down both in their movements and communication. We are studying the *long* WonnaGo, and we are confident that the Astracinca devised an algorithm for *line recovery* in the *asynchronous* settings. However, we have yet to fully understand the many subtleties of the *long* WonnaGo, which remains a challenging open problem.

References

1 N. Agmon and D. Peleg. Fault-tolerant gathering algorithms for autonomous mobile robots. *SIAM Journal on Computing*, 36(1):56–82, 2006.
2 Z. Atakan, P. Morrison, M.G. Bossong, R. Martin-Santos, and J.A. Crippa. The effect of cannabis on perception of time: a critical review. *Current Pharmaceutical Design*, 18(32):4915–22, 2012.
3 D.G. Barceloux. *Medical Toxicology of Drug Abuse*. Wiley, 2012.
4 L. Barriere, P. Flocchini, E. Mesa-Barrameda, and N. Santoro. Uniform scattering of autonomous mobile robots in a grid. *International Journal of Foundations of Computer Science*, 22(3):679–697, 2011.
5 Z.J. Butler, K. Kotay, D. Rus, and K. Tomita. Generic decentralized control for lattice-based self-reconfigurable robots. *International Journal of Robotics Research*, 23(9):919–937, 2004.
6 G. Chirikjian. Kinematics of a metamorphic robotic system. In *Proceedings of the 1994 International Conference on Robotics and Automation (ICRA)*, pages 449–455, 1994.

7 M. Cieliebak, P. Flocchini, G. Prencipe, and N. Santoro. Distributed computing by mobile robots: gathering. *SIAM Journal on Computing*, 41(4):829–879, 2012.

8 Z. Derakhshandeh, S. Dolev, R. Gmyr, A. Richa, C. Scheideler, and T. Strothmann. An algorithmic framework for shape formation problems in self-organizing particle systems. In *Proceedings of the 2nd International Conference on Nanoscale Computing and Communication*, pages 21:1–21:2, 2015.

9 Z. Derakhshandeh, R. Gmyr, T. Strothmann, R.A. Bazzi, A. Richa, and C. Scheideler. Leader election and shape formation with self-organizing programmable matter. In *Proceedings of the 21st International Conference on DNA Computing and Molecular Programming (DNA)*, pages 117–132, 2015.

10 Y. Emek, T. Langner, J. Uitto, and R. Wattenhofer. Solving the ANTS problem with asynchronous finite state machines. In *Proceedings of the 41st International Colloquium on Automata, Languages, and Programming (ICALP)*, 471–482, 2014.

11 P. Flocchini, D. Ilcinkas, A. Pelc, and N. Santoro. Ring exploration by asynchronous oblivious robots. *Algorithmica*, 65(3):562–583, 2013.

12 P. Flocchini, G. Prencipe, and N. Santoro. *Distributed Computing by Oblivious Mobile Robots*. Morgan & Claypool, 2012.

13 T.R. Hsiang, E. Arkin, M. Bender, S. Fekete, and J. Mitchell. Algorithms for rapidly dispersing robot swarms in unknown environments. In *Proceedings of the 5th Workshop on Algorithmic Foundations of Robotics (WAFR)*, pages 77–94, 2002.

14 J.E. Joy, S.J. Watson, Jr., and J.A. Benson. *Marijuana and medicine: assessing the science base*. National Academies Press, 1999.

15 B. Keller, T. Langner, J. Uitto and R. Wattenhofer. Overcoming obstacles with ants. In *Proceedings of the 19th International Conference on Priniciples of Distributed Systems (OPODIS)*, 2015.

16 R. Nozick. *Anarchy, State, and Utopia*. Basic Books, 1974.

17 A.L. Rosenberg. Finite-state robots in the land of Rationalia. In *Proceedings of the 27th International Symposium on Computer and Information Sciences*, 3–11, 2013.

18 G. Samorini. *Animals and Psychedelics: The Natural World and the Instinct to Alter Consciousness*. Park Street Press, 2002.

19 I. Suzuki and M. Yamashita. Distributed anonymous mobile robots: formation of geometric patterns. *SIAM Journal on Computing*, 28(4):1347–1363, 1999.

20 J.E. Walter, J.L. Welch, and N.M. Amato. Distributed reconfiguration of meta-morphic robot chains. *Distributed Computing*, 17(2):171–189, 2004.

Building a Better Mouse Maze

Jessica Enright[1] and John D. Faben[2]

1 **University of Stirling, Stirling, UK**
jae@cs.stir.ac.uk
2 **Glasgow, United Kingdom**
jdfaben@gmail.com

Abstract

Mouse Maze is a Flash game about Squeaky, a mouse who has to navigate a subset of the grid using a simple deterministic rule, which naturally generalises to a game on arbitrary graphs with some interesting chaotic dynamics. We present the results of some evolutionary algorithms which generate graphs which effectively trap Squeaky in the maze for long periods of time, and some theoretical results on how long he can be trapped. We then discuss what would happen to Squeaky if he couldn't count, and present some open problems in the area.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, G.2.2 Graph Theory, G.2.1 Combinatorics

Keywords and phrases graph evolutionary genetic algorithm traversal

Digital Object Identifier 10.4230/LIPIcs.FUN.2016.15

1 Introduction

There is a mouse (named Squeaky) who is trapped in a maze. He follows a very simple deterministic rule to decide which square to visit next – always choosing the least visited available neighbour of his current square, and breaking ties by preferring to go down and left over right and up (in that order). This seemingly simple rule leads Squeaky to behave in unpredictable ways, and results in an interesting game designing mazes to trap Squeaky for as long as possible.

In this paper, we first define the natural generalisation of the Mouse Maze game to a problem on general graphs, and then we explore upper and lower bounds on the question of how long Squeaky can be trapped in the maze. We get upper bounds from theoretical considerations, and establish lower bounds using genetic algorithms to search possible maze configurations. We then consider the question of whether Squeaky could still escape if he were not so good at counting, and finally end with some open questions.

1.1 Some history

The implementation of Mouse Maze that sparked our interest is a Flash game hosted on Kongregate[1]. It was written by Tom Fraser, who is Kongregate user CuriousGaming, and is based on the game Micro Mouse, which was written by Jeremy Hammett for PCW Games for the Commodore 64 in 1984 [3].

The game was introduced to the mathematics department at Queen Mary University of London in the summer of 2009 by Michael Brough. He pioneered the approach of using

[1] http://www.kongregate.com/games/CuriousGaming/mouse-maze

© Jessica Enright and John Faben;
licensed under Creative Commons License CC-BY
8th International Conference on Fun with Algorithms (FUN 2016).
Editors: Erik D. Demaine and Fabrizio Grandoni; Article No. 15; pp. 15:1–15:12
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

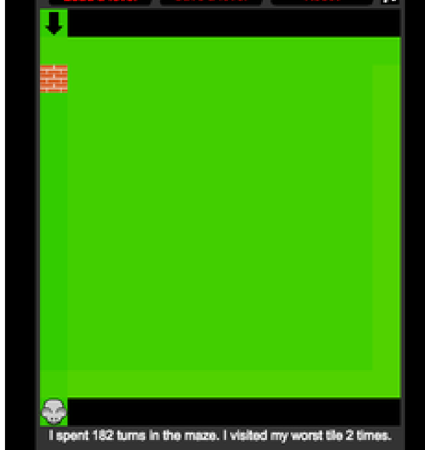


Figure 1 Squeaky has finished navigating a maze with a single block. The brick walls show squares that Squeaky is blocked from visiting, warmer colours are squares he has visited more often.

genetic algorithms to generate mazes which take a long time for the mouse to navigate. This was taken up by a small group of researchers, including Andrew Drizen, who has held the high score on Kongregate since then. An informal proof of the folklore result in Theorem 7 was known in 2009, but to our knowledge no-one has previously demonstrated an explicit finite upper bound for how long it can take Squeaky to escape a maze.

2 What is mouse maze?

2.1 The original game

To quote the instructions for the original game:

> Draw a maze. Release the mouse. Confound it for as long as you can. Squeaky follows a primitive algorithm; he always chooses the square he has visited least. Squeaky prefers down and right. Squeaky doesn't like ups and lefts.

Drawing a maze is accomplished by removing cells from the 13x13 grid, preventing Squeaky from visiting those cells. This can lead to some surprisingly erratic behaviour from the mouse – for example, on the blank grid from the original game, he will take just 14 moves to exit. However, if just one square is blocked off from his route, it will take him 180 turns to escape, as shown in Figure 1 (we use a slightly different counting convention to the Flash game, which gets 182 turns for this maze).

2.2 Generalised mouse maze

We define a mouse maze problem as follows. All graphs $G = (V, E)$ will be labelled, undirected, simple and loopless.

▶ **Definition 1.** A **maze** $M = (G, s, f, r)$ is tuple consisting of a graph G along with two vertices $\{s, f\} \in v(G)$ where s is the start vertex and f is the exit vertex, and a function for each vertex $v \in V(G), r_v : N(v) \to \mathbb{N}$, which gives Squeaky's preference order on the neighbours of v. We say a maze is **feasible** if s is in the same connected component of G as f.

▶ **Definition 2.** We say that Squeaky **runs** a maze M by executing the following algorithm.

```
v ← s                                   ▷ v is Squeaky's current position, he starts at s
for x ∈ V(G) do
    visits_0(x) ← 0                     ▷ Initially, he has visited each vertex 0 times
end for
t ← 1
while v ≠ f do                          ▷ keep going until he escapes
    for x ∈ V(G) \ {v} do
        visits_t(x) ← visits_{t−1}(x)
    end for
    visits_t(v) ← visits_{t−1}(v) + 1
    next ← ∞
    x ← None                            ▷ x will be the next vertex Squeaky visits
    for w ∈ N(v) do                     ▷ consider these from most preferred to least preferred
        if visits_t(w) < next then
            x ← w
            next ← visits(x)
        end if
    end for
    v ← x                               ▷ update Squeaky's position
    t ← t + 1
end while
```

▶ **Definition 3.** For any maze $M = (G, s, f, r)$, for any vertex $v \in V(G)$, we define $visits(v)$ to be the number of times Squeaky visits v if he starts at s and runs through the maze until he gets to f. While Squeaky is running through the maze, we define $visits_t(v)$ to be the number of times Squeaky has visited v after t steps.

▶ **Definition 4.** For each maze M, $visits(M)$ is defined to be $\sum_{v \in V(G)} visits(v)$

▶ **Example 5.** The original game is played on a subgraph of the 13x13 grid graph, with r_v being such that Squeaky prefers to go down, right, left then up in that order, where possible. Getting a high score is the problem of choosing a subgraph H of the grid such that $visits(H)$ is maximal.

3 Squeaky can always escape

3.1 An upper bound exists

The astute reader may have noticed that we haven't actually established that the function *visits* is well-defined. Here we show that it is. In particular, if M is feasible, then Squeaky will eventually visit f.

The intuition for this is simple – if Squeaky doesn't escape, then he must enter an infinite loop, but if he enters an infinite loop, then there is a square which is adjacent to that infinite loop which is only visited finitely many (perhaps 0) times, but Squeaky would eventually prefer to go there than stay inside the loop, which is a contradiction. We formalise this intuition below, and give a finite upper bound on $visits(M)$ based on the degree structure of G. We will use the following concepts:

- The **distance** between any two vertices $v, w \in G$, denoted $d_G(v, w)$, is the length of the shortest path in G which goes from v to w.

- The **diameter** of a graph G, denoted $diam(G)$, is the longest shortest path in G, that is, the maximum distance between any two vertices in G.
- The **neighbourhood** of a vertex in $V(G)$, denoted $N_G(v)$, or more commonly $N(v)$, if G is implicit from the context, is the set of vertices $\{u \in V(G) | (u, v) \in E(G)\}$.
- The **degree** of a vertex $v \in V(G)$, denoted $deg(v)$, is the number of neighbours of v in G, or $|N_G(v)|$
- The **maximum degree** of a graph G, denoted $\Delta(G)$, is the maximum degree of any vertex in $V(G)$.

We are now ready to prove that Squeaky will always escape any feasible maze (thus justifying our choice of the word 'feasible'). We use the following technical lemma.

▶ **Lemma 6.** *If Squeaky leaves a vertex v for the kth time at iteration t, the following holds,*

$$\forall w \in N(v),\ visits_t(w) \geq \lfloor \frac{k}{deg(v)} \rfloor .$$

Proof. If $k < deg(v)$, this is clearly true, as $\frac{k}{deg(v)} < 1$, so the RHS of the above expression is equal to 0. If $k = deg(v)$, the claim is that every neighbour of v has been visited at least once. But this must be true, as Squeaky has left v $deg(v)$ times, so if there is a neighbour of v which has been visited 0 times, say w, then on one of those occasions Squeaky chose to visit a neighbour of v which had already been visited once in preference to w, but this is a contradiction.

The expression on the right only increases when $deg(v)$ divides k, so if it is true for all multiples of $deg(v)$ then it must also be true for all other k. Assume that it is true for all $j < k$, and that k is a multiple of $deg(v)$, *i.e.* $\exists q \in \mathbb{N}$ such that $k = q * deg(v)$. Now, by induction, every neighbour of v had been visited at least $(q - 1)$ times when Squeaky had left v $(q-1)deg(v)$ times. But Squeaky has made $deg(v)$ more visits to v since then, and has chosen neighbours to visit exactly $deg(v)$ more times. Assume that there is still some vertex $w \in N(v)$ which only been visited $q - 1$ times, then as in the previous paragraph, Squeaky has at some point chosen to visit a neighbour of v which had already been visited q times in preference to w, which is a contradiction. ◀

This lemma will form the basis for both of the next two proofs. First, we prove that Squeaky will eventually escape the maze, then we will give a bound on how long it takes him. What the lemma essentially says is that Squeaky can never visit a vertex much more often than he visits all of its neighbours. Using this idea, we show that if Squeaky visits any vertex more than a certain number of times, then he must also eventually visit f, since there is a chain of neighbours which connects f to this vertex.

▶ **Theorem 7.** *For any maze $M = (G, s, f, r)$, if M is feasible, then when Squeaky runs through M, he will eventually reach f.*

Proof. In the below, we assume that $\Delta(G) > 1$ if not, G consists of a collection of isolated edges and vertices, and the result is trivial.

If Squeaky never visits f, then eventually there is some vertex v which is visited $\Delta(G)^{2diam(G)}$ times. We will show that if this is the case, then Squeaky has in fact visited f, a contradiction. Let t be the time step immediately after Squeaky leaves v this many times.

First, consider neighbours of v. By Lemma 6,

$$\forall w \in N(v) visits_t(w) \geq \lfloor \frac{\Delta(G)^{2diam(G)}}{deg(v)} \rfloor$$

and since $deg(v) \leq \Delta(G)$,

$$visits_t(w) \geq \Delta(G)^{2diam(G)-1}.$$

But then it is certainly the case that Squeaky has left w at least $\Delta(G)^{2(diam(G)-1)}$ times, since $\Delta(G) > 1$, so this is smaller than the RHS of the above expression. By a second application of the Lemma, if we let z be a vertex at distance 2 from v, then z is adjacent to some vertex at distance 1 from v, and we have,

$$visits_t(z) \geq \Delta(G)^{2(diam(G)-1)-1}$$

and in particular, Squeaky has left z at least $\Delta(G)^{2(diam(G)-2)}$ times, since $\Delta(G) > 1$, so this is less than $\Delta(G)^{2(diam(G)-1)-1)}$.

By repeated application of the lemma, we get that for any vertex at distance k from v,

$$visits_t(x) \geq \Delta(G)^{2(diam(G)-k)-1}.$$

But by definition $d_G(v, f) < diam(G)$, so in particular,

$$visits_t(f) \geq \Delta(G)^{2(diam(G)-diam(G))-1} \geq 1$$

and Squeaky has visited every vertex in G. ◀

3.2 A finite upper bound

Note that in above proof it seems that $\Delta(G)^{2diam(G)}$ is in some sense bigger than the number of visits we needed to ensure that Squeaky visited every vertex. Something closer to $\Delta(G)^{diam(G)}$ should suffice. Using the fact that we know Squeaky will eventually escape, we prove an upper bound on how long it takes him which is indeed of the order $\Delta(G)^{diam(G)}$. Note that we use the result of Theorem 7 in the very first step of this proof, where we assume that Squeaky eventually visits f.

▶ **Theorem 8.** *The total number of turns Squeaky spends in any maze M on a graph G before he visits every vertex is bounded above by,*

$$|V(G)| \sum_{i=1}^{diam(G)} \Delta(G)^i.$$

Proof. Since Squeaky stops when he visits f for the first time, we know that $visits(f) = 1$. We will show that no vertex in $V(G)$ was visited more than $\sum_{i=1}^{diam(G)} \Delta(G)^i$ times.

First consider vertices in $v \in N(f)$. We claim that $visits(v) \leq \Delta(G)$. The logic here is similar to that in the proof of Lemma 6. Assume that Squeaky did visit v more than $deg(v)$ times, and consider the situation just after he left v for the $deg(v)^{th}$ time, at time t. At this time, $visits_t(f)$ was 0, by assumption (since Squeaky got back to v again, in order for $visits(v) > deg(v)$), but then at some time t_1 in those $deg(v)$ occasions he must have chosen to visit a vertex $w \in N(v)$ with $visits_{t_1}(w) \geq 1$ in preference to visiting f, but this is a contradiction, so $visits(v) \leq deg(v) \leq \Delta(G)$.

Now we consider the vertices at distance 2 from f. Let w be such a vertex. By Lemma 6,

applying the result in the previous paragraph, we have,

$$\Delta(G) \geq \lfloor \frac{visits(w)}{deg(w)} \rfloor \quad (1)$$

$$\geq \lfloor \frac{visits(w)}{\Delta(G)} \rfloor \quad (2)$$

$$\Delta(G) + 1 > \frac{visits(w)}{\Delta(G)} \quad (3)$$

$$\Delta(G)(\Delta(G) + 1) > visits(w) \quad (4)$$

Now assume for all $j < k$, and all vertices $x \in V(G)$ such that $d(x, f) \leq j$ we have $\sum_{i=1}^{j} \Delta(G)^i > visits(x)$. We will show that for all vertices y at distance k from f, we have $\sum_{i=1}^{k} \Delta(G) > visits(y)$. Any vertex at distance k from f is adjacent to some vertex at distance $k-1$ from f, let x be such a vertex, then by our induction hypothesis, we have, $\sum_{i=1}^{k-1} \Delta(G)^i > visits(x)$. Let t be the last time that Squeaky left y, then $visits(y) = visits_t(y)$, but by Lemma 6, we have $visits_t(x) \geq \lfloor \frac{visits_t(y)}{deg(y)} \rfloor$, and since $visits(x) \geq visits_t(x)$, combining these inequalities gives:

$$\sum_{i=1}^{k-1} \Delta(G)^i > \lfloor \frac{visits(y)}{deg(y)} \rfloor \quad (5)$$

$$\geq \lfloor \frac{visits(y)}{\Delta(G)} \rfloor \quad (6)$$

$$\sum_{i=1}^{k-1} \Delta(G)^i + 1 > \frac{visits(y)}{\Delta(G)} \quad (7)$$

$$\Delta(G)(\sum_{i=1}^{k-1} \Delta(G)^i + 1) > visits(y) \quad (8)$$

$$\sum_{i=1}^{k} \Delta(G)^i > visits(y) \quad (9)$$

But the farthest any vertex can be from f in G is $diam(G)$, so for every vertex $v \in V(G)$, we have

$$\sum_{i=1}^{diam(G)} \Delta(G)^i > visits(v) \quad (10)$$

And since $visits(M)$ is the sum over all vertices in $V(G)$ of $visits(v)$, and every vertex satisfies the above inequality,

$$|V(G)| \sum_{i=1}^{diam(G)} \Delta(G)^i > visits(M) \quad (11)$$

◀

4 Lower bounds: the search for a high score

The ultimate question in mouse maze is: what is the highest achievable score? That is, given certain constraints on (G, s, f, r), what is the maximum value of $visits(M)$ where M is a maze on (G, s, f, r). The original game constrains G to be a subgraph of the 13x13 grid, with

s being in the top row, f being in the bottom row, and r being a preference for down, right, left, up in that order. In the following we restrict our attention to games on the square grid with this preference function. We present a brief summary of some types of algorithm that have been used in the search for better mouse mazes, with a discussion of the effectiveness of these algorithms on smaller grids, where the exact solution can be computed. Finally we present the best known maze for the original problem, which was found using a genetic algorithm and some local optimisation.

4.1 Flipping squares in a grid

Since we will only be considering mazes where G is a subset of some 2-dimensional grid graph, in the following it will often be easier to refer to 'square of the grid' rather than 'vertices of G'. We will also talk about 'flipping' squares.

▶ **Definition 9.** A maze M is obtained from another maze M_0 by **flipping** square (i, j) if M_0 and M_1 are identical except for the fact that the square of the grid graph corresponding to the $(i, j)^{th}$ coordinate is either in $V(M_0)$ and not in $V(M_1)$ or vice versa. We say that M_0 and M_1 are **neighbours**. We will also define the Hamming distance between two mazes which are subgraphs of a given grid. This is the number of vertices that need to be flipped in one to reach the other. This corresponds exactly to the Hamming distance between the natural matrix representation of subgraphs of the grid.

4.2 Some algorithms

We present below five types of algorithm which we have used to generate mazes to trap Squeaky. For a summary of these techniques, the reader is referred to [4].

In **random search**, we simply generate feasible mazes at random and evaluate them. In practice, we limit random search to generating mazes by considering each grid square independently and deciding whether to include it with a fixed probability p. This can be used as a baseline for other methods.

In **hill-climbing**, we pick a maze at random (using the same method for picking random mazes we used in random search), evaluate that maze, then evaluate all of its neighbours. If any neighbour has a higher value for the *visits* function, we then evaluate all of that maze's neighbours, and so on until there are no further improvements in the evaluation function to be found.

In **simulated annealing** we again start with a random maze, M. We pick one of its neighbours, N uniformly at random, and evaluate $visits(N)$ (we keep picking until N is feasible). If $visits(N) > visits(M)$, then we move to N and recurse. If $visits(N) < visits(M)$, then we still move to N with some probability dependent on the difference, which reduces as the algorithm runs (a process known as cooling). This allows simulated annealing to escape local maxima, in contrast to hill-climbing.

In an **evolutionary search**, we pick a random subset of the search space – that is a random group of mazes, and we evaluate all of them. We then take the top n of these to be 'parents' for the next generation, and mutate these parents slightly to produce a new generation. In particular, to generate a member of the new generation, we pick a member of the list of parents at random, we then decide for each vertex independently whether to flip this with a fixed probability, and the resultant maze is placed into the next generation.

In a **genetic search**, more than one parent is combined to produce each child. There are a variety of ways in which this combination can be achieved. We discuss these a little more in Section 4.4, but the simplest, and the on which has in practice proved most effective

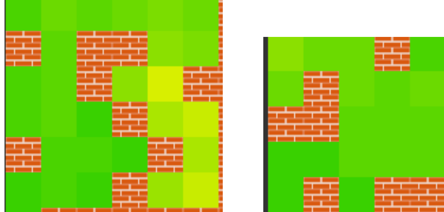


Figure 2 The best possible mazes on the 6x6 and 5x5 grids, which take 115 turns and 54 turns to navigate respectively. Squeaky starts in the top left and exits in the bottom left. The brick walls show squares that Squeaky is blocked from visiting, lighter colours are squares he has visited more often.

is **uniform crossover**. We evaluate each member of the current population, and take the top n to be parents. We then select two parents uniformly at random (so both parents can be the same). For each vertex in the grid, we pick a parent at random and include that vertex in the child if it is included in that parent. We may then apply some mutation to the children, in the same way that we did for the evolutionary algorithm.

4.3 Some smaller mazes

For small grids, it is possible to completely enumerate all mazes (possibly with some restrictions on s and f). We present the optimal mazes for a 5 by 5 and 6 by 6 grids in Figure 2. These were obtained by a simple exhaustive search on all possible feasible mazes. The last of these took approximately 60 days of parallel computation, so extending this analysis using the same method to a 7 by 7 grid is prohibitive, as there are approximately 10,000 times as many feasible 7 by 7 grids as feasible 6 by 6 grids. 7 by 7 may be achievable with more aggressive pruning of the search space to identify feasible mazes, and faster maze evaluation implementations. Finding optimal mazes for 4 by 4 and 3 by 3 grids can be done by hand, and is left as an exercise for the interested reader.

We ran each of the algorithms described in Section 4.2 to search for mazes on the 6 by 6 grid until they had evaluated 1 million mazes each. Monte Carlo simulation shows that of the 2^{34} possible subgraphs of the 6x6 grid including the start and the finish vertex, approximately 4% are feasible, so each of these algorithms has searched around 0.01% of the total search space – much more than we could reasonably hope to achieve for a 13 by 13 grid. The table below presents the results of these searches. Note that the best possible score, represented by the grid in Figure 2 is 115.

Search method	Highest *visits*	Comments
Random Search	67	We used p=0.25, the fraction of blocked squares in the optimal 6x6 grid. We searched until 1 million feasible mazes were evaluated.
Hill Climbing	87	We started with a random maze with $p = 0.25$. The search was restarted whenever a local optimum was reached, and continued until 1 million mazes had been evaluated.
Simulated Annealing	109	We evaluated 1 million mazes, using a simple linear cooling function.
Evolutionary Algorithm	109	Using a mutation rate of 0.06, and picking 10 parents from each generation of 100 mazes, we ran 10,000 iterations, evaluating 1 million mazes.
Genetic Algorithm	111	Using a mutation rate of 0.06 and uniform crossover, picking 10 parents from each of 100 mazes, we ran 10,000 iterations, evaluating 1 million mazes.

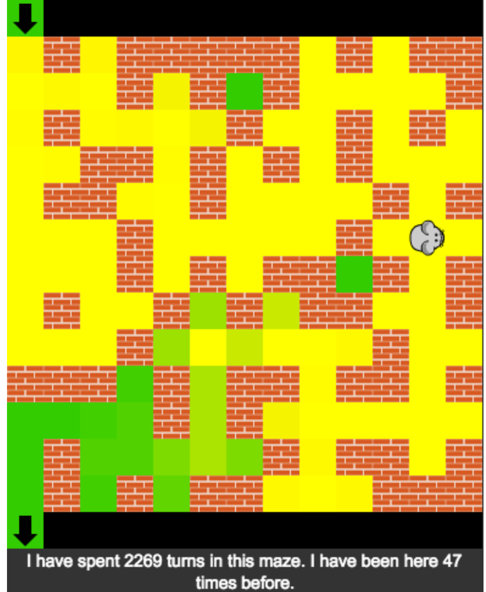


Figure 3 The best known maze, which takes Squeaky 36314 turns to navigate, shown as Squeaky is running through it. Squeaky starts in the top left and exits in the bottom left. The brick walls show squares that Squeaky is blocked from visiting, lighter colours are squares he has visited more often.

4.4 The Genetic Algorithm

The results in Section 4.3 are only suggestive, but they do indicate that an evolutionary/-genetic approach may be suitable for finding good mazes to trap Squeaky. This is backed up by previous experience. Indeed, M_{34188}, which has held the high score on Kongregate since 2009 was generated by Andrew Drizen using a simple genetic algorithm with uniform crossover [2]. Despite having some success on the 6 by 6 grid, simulated annealing has proved ineffective on 13 by 13 grids. After searching 10 million mazes, we failed to find one with a visits number exceeding 2000.

It may seem that the uniform crossover operator doesn't do a good job of preserving the structure in the parents, which is usually a feature of good crossover operators for evolutionary algorithms [4]. However, various crossover operators which do seem more likely to have this property were tried, including point crossover (choosing everything above and to the left of a random point from one parent, and everything else from the other); multi-point crossover (essentially point crossover, but with multiple points); line crossover (choosing each row or column of the offspring to come entirely from one or other parent) and an operator in which we chose 2x2 grids from the parents at random. All of these were abandoned as they didn't produce any promising solutions (defined as >10,000 visits) after evaluation of over 10 million mazes in each case. In contrast, we were able to produce a large number of mazes with > 20,000 visits using uniform crossover.

4.5 The Best Known Maze

In our search for better solutions, we were able to find several mazes which exceeded 25,000 visits, but we have failed to identify a more promising area of the search space than that identified by Andrew Drizen in 2009. However, we have managed by locally searching the

Figure 4 A violin plot, on a log scale, showing the distribution of the visit numbers of the neighbours of M_{36314} (top) compared to a similar plot for a random maze feasible with the same number of blocked squares (bottom).

area around that maze to produce a small improvement on the lower bound – we present M_{36314}, depicted in Figure 3, with $visits(M_{36314}) = 36314$.

M_{36314} is very close to the previous high scoring maze (Hamming distance = 10). It was found by the authors using that maze as a starting point. We used a simple evolutionary algorithm starting from a population consisting entirely of that maze with a very low mutation rate and no crossover, an approach that had proved successful in exploiting the results of genetic algorithms in our own searches. This produced a maze which took Squeaky 36118 turns to escape. We then did an exhaustive search of the local neighbourhood of this maze – searching all mazes within Hamming distance 3, and successfully located M_{36314}. There is no maze within Hamming distance 3 of M_{36314} with a higher number of visits than 36314.

4.6 Local Search is limited

The chart in Figure 4 summarises $visits(M)$ for all feasible mazes which are within a Hamming distance of 2 of the best maze found so far, M_{36314}, compared with the mazes which are within Hamming distance 2 of a random feasible subgraph of the grid with the same number of vertices.

Note that while several mazes close to M_{36314} do have high visit numbers, there are also mazes which are directly adjacent to our optimal maze which have a tiny number of visits – in particular, there is one maze N such that $visits(N) = 14$, which is the minimum possible for a maze on the 13 by 13 grid, but N is adjacent to M_{36314} (N is the maze obtained by removing the brick which blocks Squeaky from heading straight to the exit). We can contrast this with the neighbours of a randomly chosen feasible maze with the same number of vertices removed from the grid as M_{36314}. This maze only had a visit number of 56. Some of its neighbours have visit numbers higher than this, but none exceed 200.

5 Mice who cannot count properly

We've shown that Squeaky can escape any maze, even on an arbitrary graph, but to do so, he might need to count to a very large number. Typical mice probably cannot count that high, though there is some evidence that at least some mice can count to 40 [1]. We therefore consider k-modular mice which count mod k. Apart from their counting, these mice have

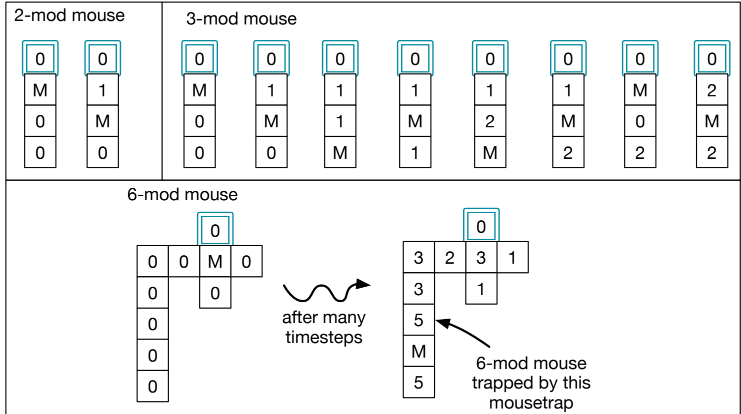


Figure 5 Initial maze states and eventual resulting mousetraps for a 2-mod mouse (top left), a 3-mod mouse (top right), and a 6-mod mouse (bottom). All intermediate timesteps are shown for the 2- and 3-mode mice, for both of which the mousetrap vertex set consists only of second vertex from the top. In all three the exit is the double-outlined box, the M is the mouse position, and the numbers indicate number of visits at a node.

the same directional preferences as Squeaky. We show that, unlike Squeaky, these types of mice can be trapped in a maze – counting is an important skill! We give examples of some small mazes that can trap k-mod mice for small values of k.

In our constructions, we use the idea of a *mousetrap*. A set of vertices V_c is a mousetrap of a k-mod mouse if at time t:

- for every vertex $v \in V_c$, $visit_t(v) = k - 1$
- in $G[V \backslash V_c]$), the mouse is not in the same component as the exit
- at every vertex $u \in N(V_c)$ that the mouse can reach without passing through a member of V_c, the mouse directionally prefers some neighbour of u that is not in V_c over its neighbours that are in V_c.

▶ **Lemma 10.** *If at some time t there is a mousetrap for a k-mod mouse, the mouse will never escape.*

Proof. It suffices to show that the mouse will never visit a vertex of the mousetrap, as all paths from the mouse to the exit go through the mousetrap. We proceed by contradiction. Consider the first time t' after t at which the mouse visits a member v of V_c. Then at $t' - 1$, the mouse was at a neighbour u of v that was reachable from the mouse's position at t without passing through V_c. By the definition of a mousetrap, we know that the mouse directionally prefers some neighbour $w \in N(v) \backslash V_c$ to v. Because $visits_{t'}(v)$ and $visits_{t'-1}(v)$ are the maximum possible value, and v is not directionally preferred, the mouse should not visit v at t', a contradiction. ◀

We can trap 2-mod mice and 3-mod mice the same single path, as in Figure 5. Note that the construction for a 3-mod mousetrap depends on the fact that $2 + 1 < 2 \pmod 3$, so that the visit number of a vertex can go down). We have investigated similar maze constructions for small values of k, up to $k = 6$ (Figure 5) We can more easily trap a general k-mod mouse in a graph that is not a subgraph of the grid.

▶ **Lemma 11.** *We can trap a k-mod mouse in a maze for which the graph consists of a tree with a single vertex of degree greater than two, and that vertex has degree $k + 1$.*

Proof. We define maze (G, s, f, r) as follows: Let G be a tree with a single vertex s of degree greater than two. Vertex s has k neighbours $v_1...v_k$, and one of those neighbours, v_{k-1}, has a neighbour w. All other neighbours $v_1...v_{k-2}$, and v_k are leaves. We need only define our directional preference function r for non-leaves (because at leaves Squeaky has no choice of where to go, so his preferences don't matter). The only non-leaves in this graph are s and v_{k-1}. First, we give the preference function at s: r_s is a function from $v_1...v_k$ to $\mathbb{N}$, where $r_s(v_i) = i, 1 \leq i \leq k$. The directional preference function $r_{v_{k-1}}$ maps from $\{s, w\}$, with $r_{v_{k-1}}(s) = 2$ and $r_{v_{k-1}}(w) = 1$ – that is, Squeaky prefers w to s at v_{k-1}. Finally, let f be v_k.

So, informally, Squeaky is in a star with a short path in place of one of the leaves, where he starts at the central vertex. The exit is in his least-preferred direction from the central vertex, and the short path is in his second least-favourite direction. Once he is on the short path, he prefers going away from the central vertex.

It remains to show that he will become trapped. First, observe that Squeaky will not visit the exit until he has visited all other neighbours of s at least once. In fact, Squeaky, starting at s, will visit v_1, then return to s, then v_2, then return to s, and so forth. So, when Squeaky is visiting v_i, where $i \leq k-1$ for the first time, he has already visited each of v_j, where $j < i$ exactly once: and has visited (and left) s exactly i times. Then when Squeaky visits v_{k-1}, $visits(s) = k-1$ (the maximum possible value), all paths from v_{k-1} to f pass through s, and Squeaky directionally prefers w to s. Therefore $\{s\}$ is a mousetrap at this timestep. ◀

6 Open questions

There is one obvious question which we leave open: what is the high score? We provide the maze with the highest known number of visits, but there is still a lot of room between this and the upper bound in Theorem 8. One could also consider the crossed paths version of the game – rather than maximising the amount of time Squeaky spends in the maze, maximise the number of times he visits a single vertex.

Other interesting questions include the generation of new mazes for Mouse Maze 2. This is a variant of the original game in which edges, rather than vertices are deleted from the 13x13 grid. In general the related question would be: how do we trap Squeaky for as long as possible in a maze given certain constraints on the graph, preference function and start and finish vertices?

Acknowledgements. We want to thank Andrew Drizen for generously sharing the maze which had led to the previous high score with us, Michael Brough for bringing the game to our attention all those years ago, and Tom Fraser for his implementation, which made it available for all of us to play.

References

1 Bilgehan Çavdaroğlu and Fuat Balcı. Mice can count and optimize count-based decisions. *Psychonomic Bulletin & Review*, pages 1–6, 2015. doi:10.3758/s13423-015-0957-6.

2 Andrew Drizen. private communication, 2016.

3 Jane Green, editor. *Personal Computer World: Best of PCW Software for the Commodore 64*. Century Communications, 1984.

4 Zbigniew Michalewicz and David B. Fogel. *How to Solve It: Modern Heuristics*. Springer, 2000.

Recognizing a DOG is Hard, But Not When It is Thin and Unit*

William Evans†[1], **Mereke van Garderen**‡[2], **Maarten Löffler**§[3], and **Valentin Polishchuk**¶[4]

1 **University of British Columbia, Canada**
will@cs.ubc.ca
2 **University of Konstanz, Germany**
mereke.van.garderen@uni-konstanz.de
3 **Utrecht University, The Netherlands**
m.loffler@uu.nl
4 **Linköping University, Sweden**
valentin.polishchuk@liu.se

Abstract

We define the notion of *disk-obedience* for a set of disks in the plane and give results for *disk-obedient graphs* (DOGs), which are disk intersection graphs (DIGs) that admit a planar embedding with vertices inside the corresponding disks. We show that in general it is hard to recognize a DOG, but when the DIG is thin and unit (i.e., when the disks are unit disks), it can be done in linear time.

1998 ACM Subject Classification F.2.2 [Nonnumerical Algorithms and Problems] Geometrical problems and computations – Routing and layout

Keywords and phrases graph drawing, planar graphs, disk intersection graphs

Digital Object Identifier 10.4230/LIPIcs.FUN.2016.16

1 A Concise Introduction to Disk-Obedience

Consider a set of disks in the plane. The *disk intersection graph* (DIG) induced by the set has a vertex for every disk and an edge between two vertices whose disks intersect. We will often identify the DIG vertices with the disks that induce the graph, and speak, e.g., about "disks of the DIG". An *embedding* of the DIG assigns a location in the plane for each vertex of the DIG; the graph edges are straight-line segments connecting their endpoints. An embedding is *planar* if the edges do not intersect except at common endpoints.

---

* Research on the topic of this paper was initiated at the 2nd International Workshop on Drawing Algorithms for Networks in Changing Environments (DANCE 2015) in Langbroek, The Netherlands, supported by the Netherlands Organisation for Scientific Research (NWO) under project no. 639.023.208.
† W. Evans is supported by NSERC.
‡ M. van Garderen is supported by the European Union's 7th Framework Programme for research, technological development and demonstration under grant agreement n° 1133 (Project CARIB) and ERC grant agreement n° 319209 (Project NEXUS1492). The project CARIB is financially supported by the HERA JRP (http://www.heranet.info) which is co-funded by AHRC, AKA, BMBF via PT-DLR, DASTI, ETAG, FCT, FNR, FNRS, FWF, FWO, HAZU, IRC, LMT, MHEST, NWO, NCN, RANNÍS, RCN, VR and The European Community FP7 2007-2013, under the Socio-economic Sciences and Humanities programme.
§ M. Löffler is supported by the Netherlands Organisation for Scientific Research (NWO) under grant 639.021.123.
¶ V. Polishchuk is supported by grant 2014-03476 from the Sweden's innovation agency VINNOVA.

© William Evans, Mereke van Garderen, Maarten Löffler, and Valentin Polishchuk;
licensed under Creative Commons License CC-BY

The question we consider is whether a DIG can be embedded so that its vertices obey the location constraints imposed by the disks, and the embedding is planar. The main definitions in this paper are those of *obedience* and a *DOG*:

▶ **Definition 1.** An embedding of a DIG is called *disk-obedient* if each vertex is contained in its corresponding disk.

▶ **Definition 2.** A DIG is a *disk-obedient graph (DOG)* if it admits a planar disk-obedient embedding.

We study computational complexity of the following problem:

Is the DIG a DOG?

That is, for a given DIG, we want to find out whether or not it has a planar, disk-obedient embedding.

1.1 Motivation and Related Work

DIGs have been extensively studied in the literature due to their wide applications in a variety of domains. In particular, in wireless sensor networks (WSNs), the *unit-disk graph* (UDG) has been the prevailing basic model for communication and sensing, and many generalizations of the UDGs (civilized graphs [19], quasi-UDGs [21, 5, 14], Vietoris-Rips complex [30, 13, 15], bisectored UDGs [27], etc.) have been introduced. The most natural generalization is, of course, to disks of two radii (modeling, e.g., two types of communication devices in the WSN or the asymmetry in sending/receiving range [12, 11]) or of arbitrary sizes. Embedding DIGs in the plane (and/or finding a *realization* of a DIG) was viewed as an important practical problem in WSN and ad hoc networks, because it translates the network connectivity information into virtual coordinates [20] and can serve as the first step in aiding topology extraction [18, 31], which may be subsequently used in geometric routing algorithms (e.g., GOAFR [22] and GFG/GPSR [6]); a planar embedding may be preferred since it provides the cleanest picture of the network.

In graph drawing and network visualization, minimizing crossings in the embedding is the central investigated question. The problem in this paper is related to the so called *Anchored* Planar Graph Drawing (AGD) (shown to be NP-hard in [1]): Given a planar graph G and an initial placement for its vertices, produce a planar straight-line drawing of G such that each vertex is at distance at most 1 from its original position. AGD was motivated by the scenario in which the disks represent cities (disk sizes represent city sizes) and the graph shows some relation between cities; in the drawing, it makes sense to keep a city vertex inside the city. Our DIG/DOG problem is different from AGD since for us, the DIG G whose planar disk-obedient drawing is sought, is fully defined by the disks (G is the DIG), while in AGD, G can be an arbitrary planar graph given in the input *in addition* to the disks. Moreover, [1] studied AGD only for *non-overlapping* disks (as observed in [1], if the disks are allowed to overlap, AGD becomes as difficult as the long standing open problem of strip planarity testing [2]; however, as also noted in [1], if the disks can have different radii and are allowed to overlap, then AGD is trivially hard by reduction from planar drawing extension [28]); for us, the disks must overlap, for otherwise the DOG recognition problem is void (G is empty). Placing vertices inside the disks of a DIG disks to produce C-oriented planar drawings of bounded-degree graphs was studied in [17]; however, the disks in [17] only touch (no overlap), making crossings impossible.

Our particular motivation for introducing disk-obedience comes from dealing with data uncertainty [7, 26, 25, 8, 24] which arises, e.g., due to objects moving (possibly with different

and/or unknown speeds) from some last-known locations (starting possibly at different and/or unknown times). It is often known which objects could currently be close to each other (potential incidents / dog fights / airplanes in conflict / criminal activities), which defines the DIG. Now, one wants to show this graph as clearly as possible, which means (by GD tradition) to draw it without crossings.

Just for FUN, we experimented with turning our problem into a game of trying to embed a given DIG in a planar disk-obedient way. The game can be programmed in interactive geometry software like GeoGebra. Initial experience indicates no concrete reason why any of the variants would not make for an engaging game that should totally become the next hit on the iStore. It has been a FUN tradition to analyze computational hardness of various games, and for once, our paper gives the computational complexity treatment to a game *before* the game storms the market. We invite the reader to play with our simple sample instance at `http://tube.geogebra.org/m/BzVYK21A`; it is not hard, but requires a tiny bit of thinking outside the circle (at least a few 9-year-old testers have politely found it non-trivial).

1.2 Paper Outline

Section 2 gives initial observations on disk-obedience, proving that a *shallow* DIG (a DIG with depth 2) is always a DOG. It follows that triangle-free DIGs are DOGs, i.e., that triangles are the only problematic places around which a DIG may be disk-disobedient. This motivates us to look more closely at triangles in DOGs. In Section 3 we show that in general, for DIGs which may have triangles close to each other, testing disk-obedience is NP-hard. In Section 4 we prove that for unit DIGs (i.e., DIGs for unit disks) having close-by triangles is essential for the hardness: we define thin unit DIGs whose triangles are well separated, and give a linear-time algorithm to recognize a thin unit DOG. The biggest question we leave unsolved is the hardness of recognizing general unit DOGs, i.e., deciding whether a given unit DIG is a DOG; we discuss it (and possible related positive results) in Section 5.

2 DIGs with Depth 2 are DOGs

We assume that our DIGs are connected. Of course, only planar graphs can have planar disk-obedient embeddings. However, not all of them do, as Figure 1 shows. Define the *depth of a point* in the plane as the maximum number of disks containing the point, and the *depth of a DIG* as the maximum number of disks having a point in common. If a DIG has depth 5 or larger, it cannot be a DOG since K_5 is not planar. Depth-3 DIGs may be DOGs (e.g., 3 disks that coincide) or not (as Fig. 1 shows); similarly, depth-4 DIGs may be DOGs (e.g., 4 disks that coincide) or not (just add a disk to Fig. 1 to create a depth-4 point – adding more disks to a DIG that is not a DOG will not make a planar disk-obedient embedding possible).

▶ **Observation 3.** Depth-2 DIGs are DOGs.

Proof. Let $D_1, D_2, \ldots, D_n$ be an arrangement of disks with depth at most two, where D_i has center c_i. Embed vertices of the DIG at the disk centers. Suppose edges c_1c_3 and c_2c_4 intersect at point x (Fig. 2), and suppose wlog that x lies in D_1 and D_4. Let $c_1a = D_1 \cap c_1c_3$ and $c_4b = D_4 \cap c_2c_4$. Since D_1 and D_3 intersect, $a \in D_3$ and since D_2 and D_4 intersect $b \in D_2$. Since c_1a and c_4b intersect at x, by the triangle inequality, $|c_1b| + |c_4a| < |c_1a| + |c_4b|$, which implies either $b \in D_1$ or $a \in D_4$ and thus b or a is contained in three disks; a contradiction. ◀

▶ **Corollary 4.** *DIGs without triangles are DOGs.*

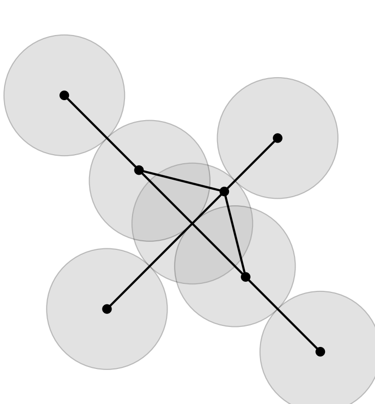


Figure 1 An example of a planar DIG that has no planar disk-obedient embedding.

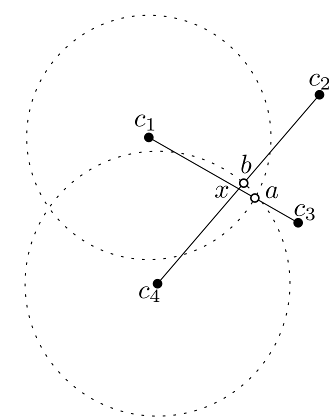


Figure 2 Either a or b is in three disks.

3 Recognizing a DOG is Hard

In this section, we prove that deciding whether a DIG is a DOG is NP-hard. We reduce from planar 3-SAT [23].

Let S be a planar 3-SAT formula. We can lay out its variable-clause graph in a planar way. Globally speaking, we will emulate every edge leading from a variable to a clause by a *variable chain*: a set of disks such that their DIG has exactly two possible planar disk-obedient embeddings. One of these embeddings will encode the value `true`; the other will encode the value `false`. Then, at each variable, we add a *variable gadget*: a configuration of disks that ensures all chains belonging to the same variable have to be in the same state. At each clause, we add a *clause gadget*: a configuration of disks that ensures at least one of the three incident chains must be in the correct state (either `true` or `false`, depending on the literal in the clause). Then, the entire configuration of disks will have a planar disk-obedient embedding if and only if S can be satisfied.

Variable chains

A variable chain is a sequence of kissing unit disks (from now on called *big disks*), together with a number of triples of disks (*small disks*) overlapping each kissing point, as shown in Figure 3a. The embedding of the vertices corresponding to the big disks, together with the edges connecting the vertices, will be called the *variable path*. The embedding of the vertices corresponding to the small disks in a triple, together with the edges connecting the vertices, will be called a *small path*. In a planar disk-obedient embedding a small path should not intersect the variable path. This means that the whole small path will need to be embedded either fully below or fully above the variable path. To be able to embed the small path below (resp., above) the variable path, the variable path must have points of all three small disks below (resp., above) the path. That is, the variable path must intersect either the top or the bottom small disk of a triplet. Both are possible: see Figures 3b and 3c. Moreover, by making the radii of the small disks sufficiently close to the radius of the big disks, we can ensure that two neighbouring triples of small paths cannot be in opposite configurations simultaneously. This way, the DIG of a variable chain will have exactly two distinct planar disk-obedient embeddings.

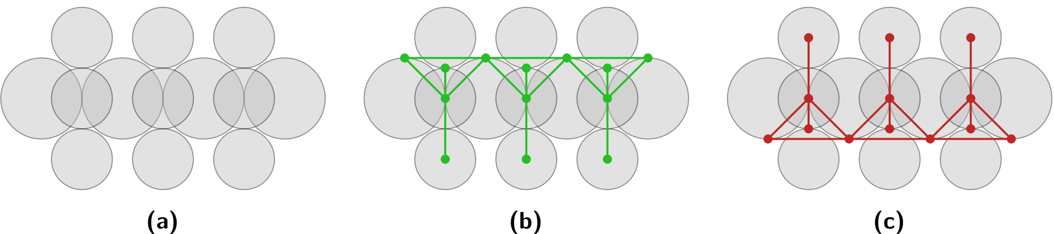


Figure 3 A variable chain and its two valid states.

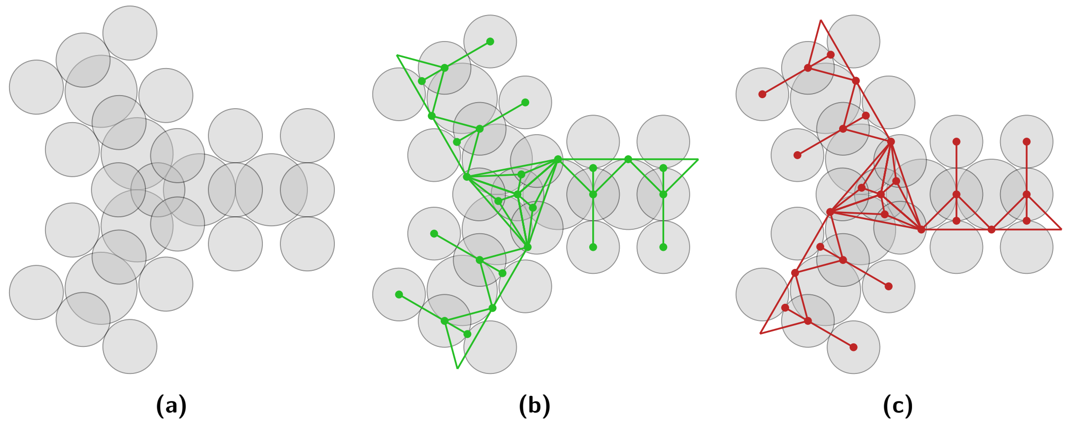


Figure 4 An alignment gadget and its two valid states.

Alignment gadgets

We use the *variable alignment gadget* to force all chains emanating from the same variable to have the same value. In principle, the number of times a variable appears in a 3-SAT instance is unlimited (though of course bounded by the number of clauses). We will not attempt to design a generic gadget working for an arbitrary-degree variable, but instead replace each variable vertex by a tree of degree-3 nodes and install the alignment gadget at each node. This way, using multiple gadgets each aligning only three chains, we can align as many chains as required.

To align three chains, we let the three big disks at the ends of the chains kiss. Centered on this location, we add a claw consisting of four small disks. Figure 4a shows the configuration. If all three chains are in the same state, then there exists a planar disk-obedient embedding of the entire alignment gadget, as can be seen in Figures 4b and 4c. It can be seen by inspection that in any other combination of states, there is no planar disk-obedient embedding.

The alignment gadget can also be used as an inverter to make the chain arrive at the clause in the correct orientation (in the inverter, one of the three variable chains will end without connecting to a clause).

Clause gadgets

Finally, we design *clause gadgets* for the clause vertices of S. In a clause, three variable chains end, and we place them so that they all kiss a single *clause disk* of the same radius as the big disk in the variable chains. The chains make an angle of 150° before they kiss. On top of this, we add a number of much smaller disks (from now on, *tiny disks*), as depicted in Figure 5a.

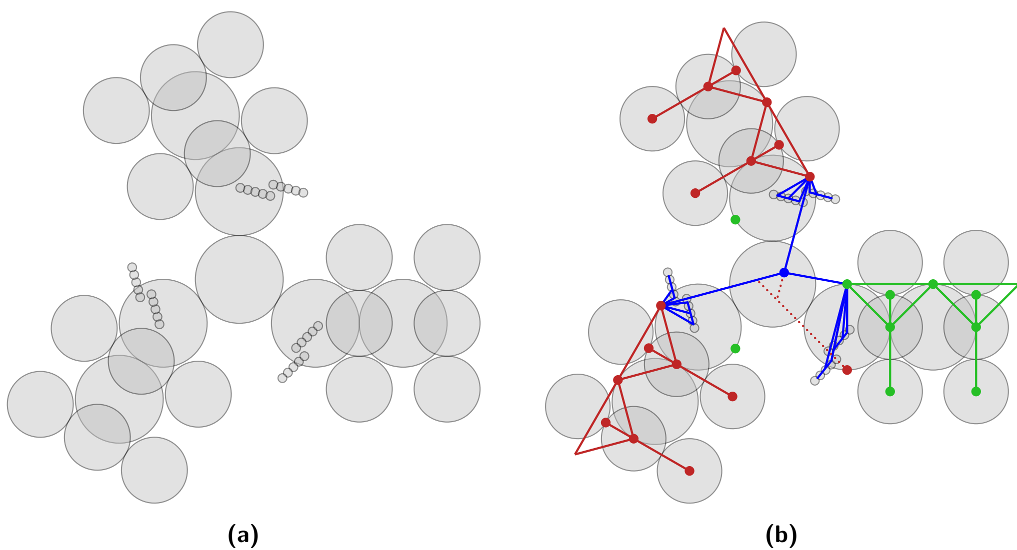


Figure 5 A clause gadget and a possible satisfied state.

The tiny disks essentially form walls that cannot be intersected by any edge in the solution embedding. We place these walls in such a way that if a variable chain is in the wrong state, the only way for the final variable disk to connect to the clause disk is if the vertex of the clause disk is placed on a specific line (or, in fact, narrow cone) determined by the location of the final vertex of the variable path and a gap in the wall. We generate three such lines—one for each variable—and make sure the three lines do not pass through a common point. Now, if at most two of the three variables are in the wrong state, we can still place the clause vertex, but if all three are in the wrong state, this is no longer possible and there is no planar disk-obedient embedding.

Thus, S can be satisfied if and only if there is a planar disk-obedient embedding, i.e., if the DIG built from S is a DOG.

4 Recognizing Thin Unit DOGs is Easy

In this section, we study disk-obedience for unit DIGs (i.e. unit disk graphs). We prove that (in accordance with common sense) if potential disk-disobedience spots are distant, each of them can be handled separately.

We start by defining some terms used later in the section:

▶ **Definition 5.** The *hippodrome* $H(A, B)$ of two intersecting disks A, B is the union of all possible segments that could realize the edge AB (i.e., the convex hull of $A \cup B$). Each of the two connected components of the difference $H(A, B) \setminus (A \cup B)$ is called a *delta* and denoted by $\Delta(A, B)$.

Suppose that vertices of a DOG are embedded at their disks centers, and (alas!) the embedding is not planar. Recall from Section 2 that each crossing must involve a triangle (by the proof of Observation 3, edges with endpoints in centers of depth-2 disks do not intersect). Let abc be a triangle in the DOG; let A, B, C be the corresponding disks. If an edge pq of the DOG intersects the edge ab, then the disks P and Q of p and q resp. are not too far from A and B; in fact, they must be within some constant distance < 8 from the triangle (because the hippodromes $H(A, B)$ and $H(P, Q)$ intersect, and a hippodrome has diameter < 4). But

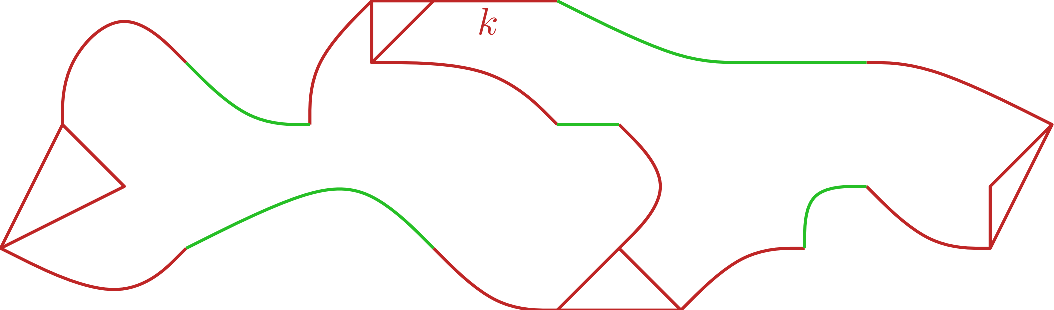


Figure 6 A schematic picture of a thin DOG. Our algorithm places green vertices at the disk centers, and works on each red component independently.

in any DOG, the number of disks within a constant distance from any point cannot be too large, or else a depth-5 point appears. Thus, there are only a constant number of disks within distance k of the triangle for any constant k.

Before embedding the (potential) DOG, we color its disks red and green: the disks that are closer than k to any triangle (including all the triangle disks themselves) are *red*, and the others are *green* (the same color applies to both a disk and its vertex). Let monochromatic edges inherit the color from their endpoints. Inspired by work on thin grids, suggesting that problems on them tend to be easier than on general grids [3, 4], we define *thin* DIGs as those composed from triangles connected by (longish) *isolated* paths such that far from the triangles the different paths do not come close to each other. A path in a DIG is *isolated* if the hippodrome of any adjacent disks A and B on the path may only intersect disks of vertices adjacent to A or B.

▶ **Definition 6.** A DIG is *thin* if

- the distance between any two triangles is larger than $3k$ ($k = 16$ suffices)
- removal of all disks within distance $k/2$ of a triangle decomposes the graph into a set of isolated paths.

▶ **Definition 7.** A *thin DOG* is a thin DIG admitting a planar disk-obedient embedding (Fig. 6).

In what follows, we consider only *connected* thin DIGS, since a DIG is a DOG if and only if its components are DOGs. The crucial properties of thin DIGs, following from the definition, are that all green vertices have degree ≤ 2 and are adjacent to vertices (red or green) of degree ≤ 2 (since they are far from any triangle); any pair of adjacent vertices that are close (≤ 8) to a green vertex are part of an isolated path (thus their hippodrome does not intersect the green disk or its adjacent disks unless they are part of the same path); and no green vertex can be on a short cycle (length $< k/2$) in the DIG (otherwise the cycle would be disconnected from the rest of the graph, since every vertex close to a green vertex has degree ≤ 2).

Given a thin DIG, we check whether each red triangle and its red component is a DOG and obtain a disk-obedient embedding for it. Since there is only a constant number of vertices per red component (otherwise we know the red component is not a DOG), this can be done in constant time (in an appropriate model of computation, like realRAM) – e.g., by formulating DOG recognition as a mathematical program whose variables are the coordinates of the vertices, and checking the program for feasibility. If no "local" red DOG can be found, this certifies that the whole DIG is not a DOG either. Otherwise, if all red components are DOGs,

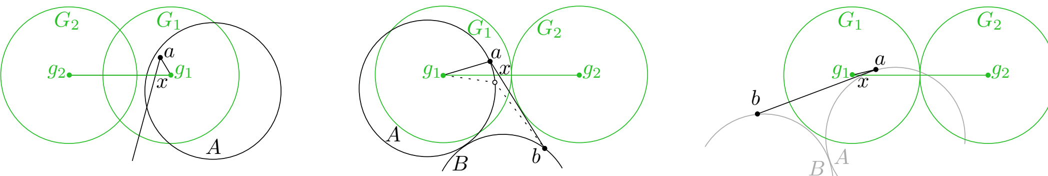


Figure 7 Left: If $x \in A \cup B$, move a to x. Middle: The new edges are dotted and the new location for a is the hollow circle. Right: $x \in \Delta(A, B), \{g_1, g_2\} \cap (A \cup B) = \emptyset$: move g_1 to x.

we claim that the whole DIG is also a DOG. To that end, we embed the green vertices at their disk centers and resolve any edge intersections this might create. In what follows, we consider the different cases depending on the colors of the endpoints of intersecting edges. Note that all four of these endpoints cannot be red.

First, suppose there is a green edge g_1g_2 that intersects an edge ab (a and b can be of arbitrary colors); let G_1, G_2, A, B be the corresponding disks, and let $x = g_1g_2 \cap ab$ be the intersection point. Since g_1, g_2 are at the disks centers, $x \in G_1 \cup G_2$. Consider the two cases depending on where x is w.r.t. $A \cup B$:

$x \in \boldsymbol{A \cup B}$**:** Then $(G_1 \cup G_2) \cap (A \cup B) \neq \emptyset$, and hence there exists an edge (say, g_1a) between a green vertex and one of a, b. We move a to x (Fig. 7, left), removing the intersection and, since the new edges are sub-segments of the old, creating no new intersection.

$x \in \boldsymbol{\Delta(A, B)}$**:** There are 2 subcases:

- If one of the green centers is inside $A \cup B$ (say, $g_1 \in A$), we rotate g_1a around g_1 until a has moved over g_1g_2 (we initially keep $|g_1a|$ fixed, i.e., moving a on the circular arc, but if a gets onto the boundary of A, we move it along the boundary during the rotation); ba rotates around b at the same time. There is no other edge that the rotated g_1a and ba could intersect, for otherwise there is a triangle or a degree-3 disk (Fig. 7, middle).
- Otherwise, either there is a degree-3 disk among G_1, G_2, A, B (a contradiction), or x is in one of G_1, G_2, meaning that there is an edge between a green vertex and one of a, b (say the edge is g_1a). In this case, we move g_1 along g_1g_2 to x to resolve the intersection (Fig. 7, right).

Finally, suppose there is an intersection of a bichromatic edge gr with an edge ab. We may assume that ab is a red edge r_1r_2, since the bichromatic case is more restrictive, meaning that the location of a green vertex is at the center of its disk, while a red vertex may be anywhere within its disk. Let G, R, R_1, R_2 be the corresponding disks, and let $x = gr \cap r_1r_2$ be the intersection point.

We claim that at most one of gr_1, r_1r, rr_2, and gr_2 exist. If two exist that share a vertex (e.g., gr_1 and r_1r), then they form a triangle with one of gr or r_1r_2 (contradiction). If two exist that do not share a vertex (e.g., gr_1 and rr_2), then they form a cycle with gr and r_1r_2, contradicting the no-short-cycle property.

Suppose G intersects R_1 or R_2 (say wlog R_1) so gr_1 is an edge in the DIG. If $x \in G$, we can move g to x and eliminate the crossing (Fig. 8, left). This doesn't introduce any new crossings since the new edge segments are sub-segments of the old ones. (Note: Vertex g has one edge to r_1 and one to r; the first is replaced by xr_1 and the second by xr, both of which are sub-segments of the original drawing. Vertex g has no further adjacent edges.) Similarly, if $x \in R_1$, we can move r_1 to x to eliminate the crossing. If neither of these is true, we move r_1 to the intersection of gr with the boundary of R_1 (Fig. 8, right).

Otherwise, suppose R intersects R_1 or R_2 (say wlog R_1) so rr_1 is an edge. As before, we can eliminate the intersection if $x \in R$ or $x \in R_1$ (Fig. 9, left). If neither of these is true, we

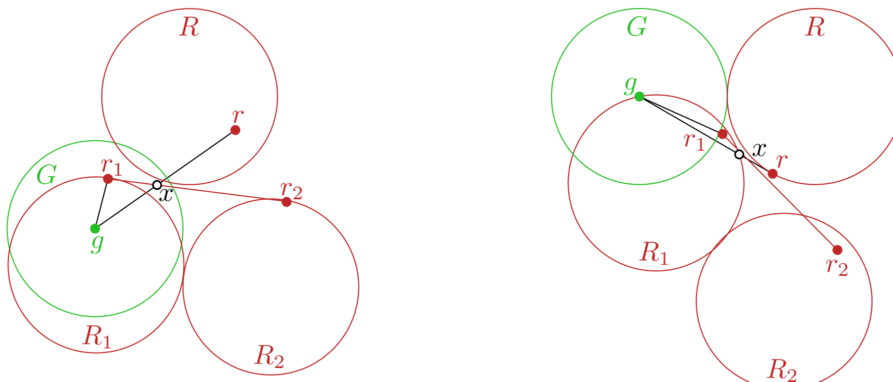


Figure 8 Edge gr_1 is in DIG. Left: $x \in G$. The new location of g is the hollow circle. Right: $x \notin G \cup R_1$. The new location for r_1 is the hollow circle.

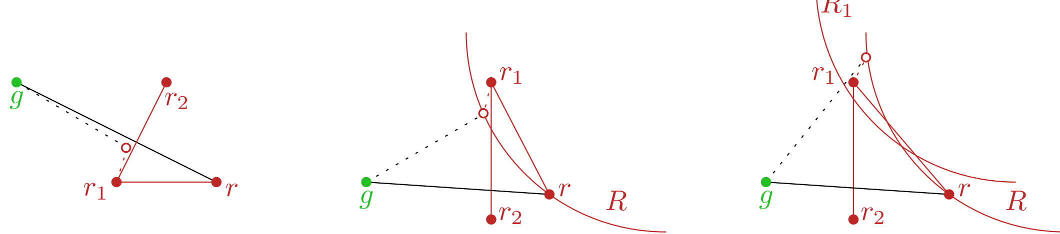


Figure 9 The fixes are dotted, the new locations for r are the hollow circles. Left: $x \in R$. Middle and right: $x \notin R$ (r is shown already pulled onto ∂R).

move r back towards x along gr as much as possible, i.e., onto the boundary of R. We then rotate gr around g, moving r along the boundary of R (again, no other edges can be in the vicinity of the moved edges since that would imply a vertex with degree at least 3): if the boundary of R intersects r_1r_2, then r is moved until it goes just over the intersection (Fig. 9, middle); otherwise, r is moved until gr jumps over r_1 (Fig. 9, right).

Finally, suppose none of the edges gr_1, r_1r, rr_2, and gr_2 exist. The intersection x must happen in a delta of R_1, R_2, with the segment r_1r_2 "cutting off" a cap of R in which x lies (Fig. 10). This implies that disk R intersects the hippodrome $H(R_1, R_2)$, which violates the path-isolation property.

5 Discussion

We showed that in general recognizing DOGs is hard, but recognizing thin unit DOGs is easy. Our hardness proof in Section 3 uses disks of different radii, and a natural question is how hard it is to recognize a unit DOG (i.e., the complexity of deciding planar disk-obedience for unit DIGs). We were not able to settle the complexity of the problem, and comment on it here.

Some parts of our hardness construction can be done with unit disks. For instance, the variable chains can be made to work with unit disks by carefully placing them (see our game instance at `http://tube.geogebra.org/m/BzVYK21A`). Similarly, it is possible to create a splitter gadget with unit disks. However, the clause gadget crucially relies on the abilities to build "walls" of small disks, and while it is possible to grow these small disks significantly, it

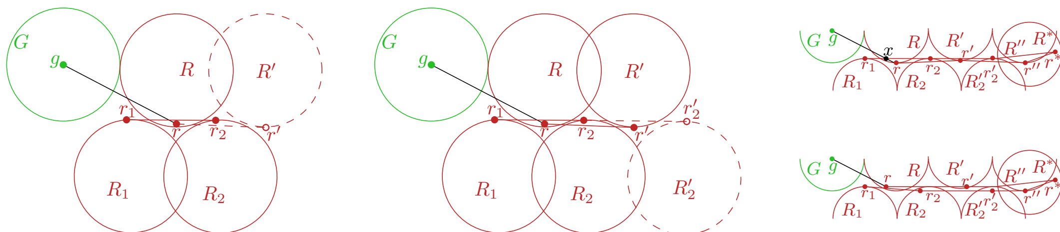


Figure 10 $r \in R$ is inside the hippodrome $H(R_1, R_2)$, which violates path isolation.

appears not to be possible to make them the same size as the other disks. In principle, we could potentially try to reduce from a different NP-hard version of SAT (not the standard 3SAT) to deciding disk-obedience for UDGs – Schaefer's dichotomy theorem [29, 9] tells which versions are hard; however, were not able to find a hard version whose `true` (and only `true`) clauses could be encoded by DOGs.

As far as our positive result goes, we believe that our problem is fixed-parameter tractable with the running time of the FPT algorithm depending on the ratio ρ of largest to smallest radius of the disks. Indeed, the fact that it is enough to look only at a certain number k of red vertices around each triangle is based on the packing argument (the disks that can intersect a triangle edge must be close to the triangle, and if there are too many such disks, they will define a point of high depth), which works for DIGs with arbitrary-radius disks (k will be a function of ρ, of course). Similarly, the arguments about non-intersection of the colorful edges (or about fixing the intersections) can be potentially adapted to apply to different-size disks, thus possibly extending the whole algorithm to general DOGs.

Our problem can be naturally extended to shapes other than disks, defining the intersection graph; for any class of shapes it may be asked whether a given -IG is a -OGs. For instance, rectangle intersection graphs (RIGs) have also been studied earlier [10, 16] but the question "Is the RIG a ROG?" has yet to be answered (even in its simplest form "Is the SIG a SOG?" when the rectangles are squares). It could be interesting to see for which shapes the -IG/-OG problem is polynomially solvable and also for which shapes the "shape-obedience game" is fun to play.

Acknowledgements. We thank the anonymous reviewers for their helpful comments. We also thank Boris Klemz for noticing that our thin DIGs were not quite thin enough in an earlier draft.

References

1 Patrizio Angelini, Giordano Da Lozzo, Marco Di Bartolomeo, Giuseppe Di Battista, Seok-Hee Hong, Maurizio Patrignani, and Vincenzo Roselli. Anchored drawings of planar graphs. In *Graph Drawing*, volume 8871 of *Lecture Notes in Computer Science*, pages 404–415. Springer, 2014.

2 Patrizio Angelini, Giordano Da Lozzo, Giuseppe Di Battista, and Fabrizio Frati. Strip planarity testing. In Stephen K. Wismath and Alexander Wolff, editors, *Graph Drawing*, volume 8242 of *Lecture Notes in Computer Science*, pages 37–48. Springer, 2013.

3 Esther M Arkin, Michael A Bender, Erik D Demaine, Sándor P Fekete, Joseph SB Mitchell, and Saurabh Sethia. Optimal covering tours with turn costs. *SIAM Journal on Computing*, 35(3):531–566, 2005.

4 Esther M Arkin, Sándor P Fekete, Kamrul Islam, Henk Meijer, Joseph SB Mitchell, Yurai Núñez-Rodríguez, Valentin Polishchuk, David Rappaport, and Henry Xiao. Not being (super) thin or solid is hard: A study of grid Hamiltonicity. *Computational Geometry*, 42(6):582–605, 2009.

5 Lali Barrière, Pierre Fraigniaud, Lata Narayanan, and Jaroslav Opatrny. Robust position-based routing in wireless ad hoc networks with irregular transmission ranges. *WCMC*, 3(2):141–153, 2003. `doi:10.1002/wcm.108`.

6 Prosenjit Bose, Pat Morin, Ivan Stojmenović, and Jorge Urrutia. Routing with guaranteed delivery in ad hoc wireless networks. *Wireless networks*, 7(6):609–616, 2001.

7 Kevin Buchin, Irina Kostitsyna, Maarten Löffler, and Rodrigo I Silveira. Region-based approximation algorithms for visibility between imprecise locations. In *ALENEX*, pages 94–103. SIAM, 2015.

8 Kevin Buchin, Maarten Löffler, Pat Morin, and Wolfgang Mulzer. Preprocessing imprecise points for Delaunay triangulation: Simplified and extended. *Algorithmica*, 61(3):674–693, 2011. `doi:10.1007/s00453-010-9430-0`.

9 Hubie Chen. A rendezvous of logic, complexity, and algebra. *ACM Comput. Surv.*, 42(1):2:1–2:32, December 2009. `doi:10.1145/1592451.1592453`.

10 M.B. Cozzens. *Higher and Multi-dimensional Analogues of Interval Graphs*. PhD thesis, Rutgers University, 1981.

11 Hongwei Du, Xiaohua Jia, Deying Li, and Weili Wu. Coloring of double disk graphs. *J. Global Opt*, 28(1):115–119, 2004. `doi:10.1023/B:JOGO.0000006750.85332.0f`.

12 Alon Efrat, Sándor P Fekete, Joseph SB Mitchell, Valentin Polishchuk, and Jukka Suomela. Improved approximation algorithms for relay placement. *ACM Transactions on Algorithms*, 12(2):20, 2015.

13 M. Gromov. Hyperbolic groups. In S. M. Gersten, editor, *Essays in Group Theory*, pages 75–263. Springer New York, 1987.

14 Marja Hassinen, Joel Kaasinen, Evangelos Kranakis, Valentin Polishchuk, Jukka Suomela, and Andreas Wiese. Analysing local algorithms in location-aware quasi-unit-disk graphs. *Discr Appl Math*, 159(15):1566–1580, 2011. `doi:10.1016/j.dam.2011.05.004`.

15 Jean-Claude Hausmann. On the Vietoris-Rips complexes and a cohomology theory for metric spaces. In F. Quinn, editor, *Annals of Mathematics Studies*, volume 138, pages 175–188, Princeton, N.J., 1995. Princeton University Press. Prospects in topology : proceedings of a conference in honor of William Browder.

16 Hiroshi Imai and Takao Asano. Finding the connected components and a maximum clique of an intersection graph of rectangles in the plane. *Journal of Algorithms*, 4(4):310–323, 1983. `doi:http://dx.doi.org/10.1016/0196-6774(83)90012-3`.

17 Balázs Keszegh, János Pach, and Domotor Palvolgyi. Drawing planar graphs of bounded degree with few slopes. *SIAM Journal on Discrete Mathematics*, 27(2):1171–1183, 2013.

18 Alexander Kröller, Sándor P Fekete, Dennis Pfisterer, and Stefan Fischer. Deterministic boundary recognition and topology extraction for large sensor networks. In *SoDA*, pages 1000–1009, 2006.

19 Sven O Krumke, Madhav V Marathe, and SS Ravi. Models and approximation algorithms for channel assignment in radio networks. *Wireless networks*, 7(6):575–584, 2001.

20 Fabian Kuhn, Thomas Moscibroda, and Rogert Wattenhofer. Unit disk graph approximation. In *Proceedings of the 2004 joint workshop on Foundations of mobile computing*, pages 17–23. ACM, 2004.

21 Fabian Kuhn, Roger Wattenhofer, and Aaron Zollinger. Ad hoc networks beyond unit disk graphs. *Wireless Networks*, 14(5):715–729, 2007. `doi:10.1007/s11276-007-0045-6`.

22 Fabian Kuhn, Rogert Wattenhofer, Yan Zhang, and Aaron Zollinger. Geometric ad-hoc routing: of theory and practice. In *PoDC*, pages 63–72, 2003.

23 David Lichtenstein. Planar formulae and their uses. *SIAM J. Comput.*, 11(2):329–343, 1982. doi:10.1137/0211025.

24 Maarten Löffler. Existence and computation of tours through imprecise points. *IJCGA*, 21(1):1–24, 2011. doi:10.1142/S0218195911003524.

25 Maarten Löffler and Wolfgang Mulzer. Unions of onions: Preprocessing imprecise points for fast onion decomposition. *JoCG*, 5(1):1–13, 2014. URL: http://jocg.org/index.php/jocg/article/view/140.

26 Maarten Löffler and Marc van Kreveld. Largest and smallest convex hulls for imprecise points. *Algorithmica*, 56(2):235–269, 2010.

27 John Nolan. Bisectored unit disk graphs. *Networks*, 43(3):141–152, 2004. doi:10.1002/net.10111.

28 Maurizio Patrignani. On extending a partial straight-line drawing. *Int. J. Found. CS*, 17(5):1061–1070, 2006. doi:10.1142/S0129054106004261.

29 Thomas J. Schaefer. The complexity of satisfiability problems. In *SToC'78*, STOC'78, pages 216–226. ACM, 1978. doi:10.1145/800133.804350.

30 L. Vietoris. Über den höheren zusammenhang kompakter räume und eine klasse von zusammenhangstreuen abbildungen. *Mathematische Annalen*, 97(1):454–472, 1927. doi:10.1007/BF01447877.

31 Yue Wang, Jie Gao, and Joseph S B Mitchell. Boundary recognition in sensor networks by topological methods. In *12th annual conference on Mobile computing and networking*, pages 122–133, 2006.

Counting Circles Without Computing Them

Rudolf Fleischer

GUtech, Muscat, Oman; and
Fudan University, Shanghai, China
`rudolf.fleischer@gutech.edu.om`

Abstract

In this paper we engineer a fast algorithm to count the number of triangles defined by three lines out of a set of n lines whose circumcircle contains the origin. The trick is *not* to compute any triangles or circles.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Lines arrangement, triangle, circumcircle, inscribed angle theorem

Digital Object Identifier 10.4230/LIPIcs.FUN.2016.17

1 Introduction

I am a lousy programmer. I think I have a solid theoretical knowledge of basic and clever data structures and algorithms, but solving problems quickly and implementing them error-free on the first attempt (versus finding logical and programming errors incrementally in repeated rounds of trial-and-error) is a completely different story. I am in good company here, even experienced algorithmicists can, even without the pressure of a relentlessly ticking clock, blunder and design (and sometimes publish) wrong algorithms [5].

Since we recently started to train student teams for the ACM Collegiate Programming Contest [1], We found it appropriate to do a little bit of programming training ourselves to better understand why our students sometimes do not manage in a contest to solve problems that are seemingly straightforward to solve. Simple answer: even problems that appear simple from the elevated viewpoint of a theoretician can be tricky to solve in a contest if the contestant lacks programming experience.

The best way to gain experience is to solve many problems from previous contests, and one platform that provides an excellent training environment is the Codeforces website [10] that not only gives access to many old contest problems but also regularly hosts contests with newly designed problem sets. Usually, no model solutions are provided for the contest problems, and reverse engineering participants solutions to understand the underlying algorithms can be frustratingly difficult because submissions are often highly optimized and nearly always without any comments.

In a recent Codeforces contest [10] the following problem, titled *Ruminations on Ruminants*, was given in category D, the second highest difficulty level [2].

> Kevin Sun is ruminating on the origin of cows while standing at the origin of the Cartesian plane. He notices n lines $\ell_1, , \ell_2, \ldots, \ell_n$ on the plane, each representable by an equation of the form $ax + by = c$. He also observes that no two lines are parallel and that no three lines pass through the same point.
>
> For each triple (i, j, k) such that $1 \leq i < j < k \leq n$, Kevin considers the triangle formed by the three lines ℓ_i, ℓ_j, ℓ_k. He calls a triangle *original* if the circumcircle of that triangle passes through the origin. Since Kevin believes that the circles of

© Rudolf Fleischer;
licensed under Creative Commons License CC-BY
8th International Conference on Fun with Algorithms (FUN 2016).
Editors: Erik D. Demaine and Fabrizio Grandoni; Article No. 17; pp. 17:1–17:7
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

bovine life are tied directly to such triangles, he wants to know the number of original triangles formed by unordered triples of distinct lines.

Recall that the circumcircle of a triangle is the circle which passes through all the vertices of that triangle.

Of course, it is straightforward to solve this problem by first computing all triangles and then counting the number of circumcircles that pass through the origin. Unfortunately, this $O(n^3)$ time algorithm is too slow to solve problems with $n = 2,000$ lines with integer coefficients $|a_i|, |b_i|, |c_i| \leq 10,000$ representing line ℓ_i within the required time bound of four seconds on the Codeforces server which has a similar performance as a regular desktop PC. A MacBook needs already 2.5 seconds just to enumerate all triples of lines, adding the complex calculations to determine the circumcircle and testing whether it passes through the origin would increase this time by more than a factor of ten.

With these constraints on the problem size and running time it was clear that we need to find at least a quadratic algorithm (it takes only 3 ms on a MacBook to enumerate $\binom{2000}{2}$ pairs of lines). While many Codeforces problems have straightforward solutions based on elementary data structures like, for example, balanced search trees and priority queues, this problem gave me a hard time. None of the standard tricks from the bag of computational geometry algorithms like plane sweep, duality, divide-and-conquer, Voronoi diagram, etc., seemed to work to bring down the running time from cubic to quadratic complexity. There is no apparent locality in the problem, the lines forming original triangles can be close together or far apart, and no clever preprocessing or ordering of the lines seems to give an advantage that would allow us to not compute many of the $\binom{n}{3}$ triangles and circumcircles to obtain a better running time.

In this paper, we will show how to solve this problem in time $O(n^2 \log n)$ (i.e., nearly quadratic) by *not* computing any triangles or circles and only using the basic arithmetic operations $+, -, \star, /$, i.e., without computing roots and trigonometric functions which usually make life miserable for geometrical algorithm designers because of the inherent rounding errors. It is actually a common trick in combinatorics to count X instead of Y if that is what we are really interested in. A classical example from computational geometry is the problem of counting the number of cells in a simple arrangement of n lines in general position [4] by counting the number of vertices, instead. There are $1 + n + \binom{n}{2}$ cells because there are n infinite-downward rays separating the $n + 1$ cells of the arrangement that are unbounded in the downward direction; the remaining cells all have a unique lowest vertex, each vertex is the lowest one of a unique cell, and there are $\binom{n}{2}$ vertices.

This paper is organized as follows. In Section 2, we will introduce the notations and formulas from geometry we need to formulate and solve the problem. In Section 3, we will explain the algorithm and analyze its correctness and run-time. In Section 4, we will shortly discuss how the algorithm was implemented and fine-tuned to run faster by a factor of eight.

2 Geometry Preliminaries

In this section we define the problem and introduce some notations. The input is a set of n lines $\ell_1, \ldots, \ell_n$. Each line ℓ_i is represented by the equation

$$a_i x + b_i y = c_i \ , \tag{1}$$

where $a_i^2 + b_i^2 > 0$. We assume the lines are in general position, i.e., no two lines are parallel and no three lines pass through the same point. The first assumption implies

$$a_i b_j \neq a_j b_i \tag{2}$$

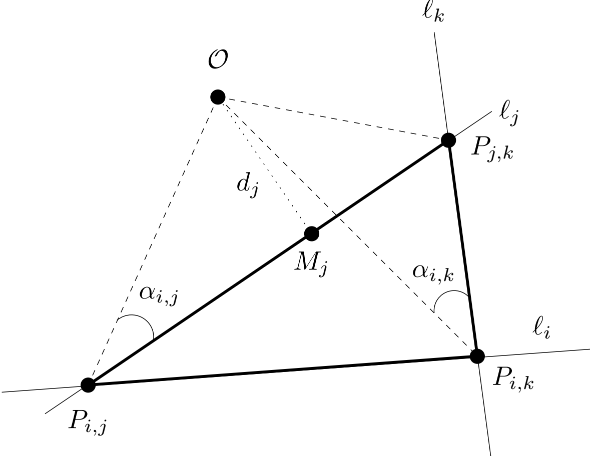


Figure 1 The triangle $T_{i,j,k}$ (in bold) formed by the lines ℓ_i, ℓ_k, ℓ_k.

for all $i \neq j$. Any three distinct lines ℓ_i, ℓ_j, ℓ_k, $1 \leq i < j < k \leq n$, therefore define a triangle $T_{i,j,k}$ with a unique circumcircle $C_{i,j,k}$. We call $T_{i,j,k}$ *original* if $C_{i,j,k}$ passes through the origin $\mathcal{O}$. The task in the *Original Triangle Counting Problem (OTC)* is to count the number of original triangles defined by the given set of n lines.

Before we solve this problem efficiently, let us introduce a few more notations and formulas from computational geometry. This also gives the reader the chance to digress and try to find an efficient solution herself before continuing to read this paper. The intersection point $P_{i,j} = (x_{i,j}, y_{i,j})$ of lines ℓ_i and ℓ_j, $1 \leq i < j \leq n$, can be computed as

$$x_{i,j} = \frac{b_i c_j - b_j c_i}{b_i a_j - b_j a_i}, \qquad y_{i,j} = \frac{a_i c_j - a_j c_i}{a_i b_j - a_j b_i}. \tag{3}$$

The point $M_j = (s_j, t_j)$ on line ℓ_j closest to the origin (see Figure 1) can be computed as

$$s_j = \frac{a_j c_j}{a_j^2 + b_j^2}, \qquad t_j = \frac{b_j c_j}{a_j^2 + b_j^2} \tag{4}$$

and the square of its distance d_j from the origin as

$$d_j^2 = \frac{c_j^2}{a_j^2 + b_j^2}. \tag{5}$$

A central element of our algorithm will be the angle under which we can see $\mathcal{O}$ from line ℓ_j in point $P_{i,j}$. We denote this angle by $\alpha_{i,j}$. We can compute the square of the sine of $\alpha_{i,j}$ as

$$\sin^2(\alpha_{i,j}) = \frac{d_j^2}{x_j^2 + y_j^2}. \tag{6}$$

We would like to use the right-hand side of Equation 6 to represent the angle $\alpha_{i,j}$ because we can compute it with basic arithmetic operations. However, since $\sin(\frac{\pi}{2} - \beta) = \sin(\frac{\pi}{2} + \beta)$ for any β, we cannot distinguish between angles smaller and larger than $\frac{\pi}{2}$. We will therefore use $\frac{d_j^2}{x_j^2 + y_j^2}$ to represent $\alpha_{i,j}$ if the three points $\mathcal{O}$, $P_{i,j}$, and M_j are counterclockwise oriented

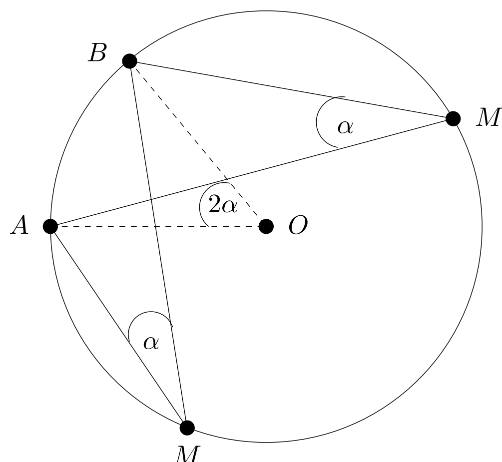


Figure 2 An illustration of the Inscribed Angle Theorem [3].

(i.e., when walking from $\mathcal{O}$ to M_j via $P_{i,j}$ we turn left at $P_{i,j}$, as in Figure 1), and we will use $-\frac{d_j^2}{x_j^2+y_j^2}$ if $\mathcal{O}$, $P_{i,j}$, and M_j are clockwise oriented.

The following theorem is a generalization of Thales's Theorem.

▶ **Theorem 1** (Inscribed Angle Theorem, [3]). *An angle α inscribed in a circle is half of the central angle 2α that subtends the same arc on the circle. Therefore, the angle does not change as its vertex is moved to different positions on the circle.*

We actually need to rephrase the theorem for our purposes, see Figure 2.

▶ **Theorem 2** ([3]). *Given two points A and B, the set of points M in the plane for which the angle AMB is equal to α is an arc of a circle through A and B. The measure of the angle AOB, where O is the center of the circle, is 2α.*

3 The Algorithm

We will now describe an efficient algorithm for solving OTC. It is based on Theorem 2 which we eventually remembered from another paper long ago [9]. It implies the following Corollary, see Figure 1. Note that the validity of the corollary does not depend on a particular way to draw the lines (i.e., the order in which the intersection points appear on the circle), see for example a different configuration in Figure 3. Please remember that we represent the angles $a_{i,j}$ as described after Equation 6.

▶ **Corollary 3.** *The circumcircle $C_{i,j,k}$ of triangle $T_{i,j,k}$ passes through the origin $\mathcal{O}$ if and only if $P_{i,j}$ and $P_{i,k}$ both see the line segment $\overline{\mathcal{O}P_{j,k}}$ under the same angle, i.e., if and only if $\alpha_{i,j} = \alpha_{i,k}$.*

Proof. We need to distinguish two cases, whether $P_{i,j}$, $P_{i,k}$, and $P_{j,k}$ appear in this order clockwise or counterclockwise on the circumcircle $C_{i,j,k}$ of triangle $T_{i,j,k}$. In the former case, $P_{i,j}$ and $P_{i,k}$ lie in different half-spaces defined by the line through $\mathcal{O}$ and $P_{j,k}$ (see Figure 3), while in the latter case they are in the same half-space (see Figure 1). Below we only give the proof of the Corollary in the latter case, from Figure 3 it should be clear that the proof of the other case is similar.

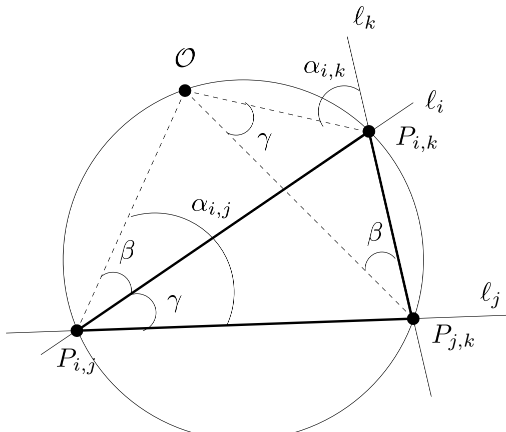


Figure 3 A different configuration for triangle $T_{i,j,k}$ (in bold) formed by the lines ℓ_i, ℓ_k, ℓ_k. Note that $\alpha_{i,j} = \beta + \gamma = \alpha_{i,k}$.

We use the Inscribed Angle Theorem with $A = \mathcal{O}$, $B = P_{j,k}$, and $P_{i,j}$ and $P_{i,k}$ in the role of M. If $\alpha_{i,j} = \alpha_{i,k}$, then $P_{i,j}$ and $P_{i,k}$ see the line segment $\overline{\mathcal{O}P_{j,k}}$ under the same angle. Now Theorem 2 implies that the four points $\mathcal{O}$, $P_{j,k}$, $P_{i,k}$, and $P_{i,j}$ must lie on a circle, which is the circumcircle $C_{i,j,k}$ of triangle $T_{i,j,k}$. That means, $C_{i,j,k}$ contains the origin. ◀

We can now formulate an efficient algorithm for OTC. First note that we can w.l.o.g. assume that no line goes through the origin. If there is only one such line, it cannot contribute to any original triangle and we can delete it. If there are two such lines, any triangle they form with a third line is an original triangle, so we can increase the triangle count by $n-2$ and delete the two lines through the origin.

1. Compute all intersection points and store point $P_{i,j}$, $1 \leq i < j \leq n$, on line ℓ_i together with the angle $\alpha_{i,j}$ (using Equation 6).
2. Sort for each line ℓ_i, $1 \leq i \leq n$, the intersections points $P_{i,j}$, $j \neq i$, on the line by angle $\alpha_{i,j}$;
3. If the same angle appears k times on line ℓ_i for some $k \geq 2$, then any pair of the corresponding intersection points induce an original triangle, i.e., we can increase the triangle counter by $\binom{k}{2}$.

Although we do not use trigonometric functions, we may experience rounding error problem when computing the squares of the sine values of the angles. However, if the lines have integer coefficients of absolute value at most 10^5, then we can define two angles to be equal if they are less than 10^{-14} apart, for example. As one reviewer pointed out, we may also avoid the rounding problems by using exact rationals instead of error-prone doubles or floats.

The run-time of step 1 is $O(n^2)$, step 2 needs time $O(n^2 \log n)$, and step 3 needs time $O(n^2)$. Thus, the run-time of the algorithm is $O(n^2 \log n)$.

▶ **Theorem 4.** *We can solve OCT in time $O(n^2 \log n)$.*

4 Engineering Speed-Ups

At the moment, there are 127 correct solutions for OTC listed on Codeforces, the fastest one running in 109 ms [11] on the most difficult test case (apparently using a similar algorithm

as the one described in this paper, but submitted three weeks later than our solution). Our first correct submission [6] needed 1,684 ms, well within the time bounds of 4 seconds, but nevertheless a bit disappointing considering the brilliant elegance and simplicity of our algorithm.

So we set out to identify and remove the bottlenecks in our program code. Our first bad design decision had been to actually compute all original triangles three times, namely once for each line bounding the triangle. Computing the original triangles only once reduced the run-time by a factor of two to 826 ms [7]. Note that we do not achieve a speed-up factor of three. The reason is that originally each $P_{i,j}$ was stored twice, once on line ℓ_i and once on line ℓ_j; in the new implementation it was only stored on line ℓ_i if $i < j$, thus saving half of the work.

The second bad design decision had been to sort all intersection point angles in one single batch which may look slightly more elegant on paper but is less efficient in practice. So the next speed-up came from sorting the angles separately for each line. Asymptotically there is no difference in the run-time, it is $O(n^2 \log n)$ in both cases, but the constant factors differ by a factor of four. In the first variant, we sort $\binom{n}{2}$ angles which needs approximately $\binom{n}{2} \log \binom{n}{2} \approx n^2 \log n$ comparisons. In the improved variant, we successively sort $n-1, n-2, \ldots, 1$ values which requires $(n-1)\log(n-1) + (n-2)\log(n-2) + \cdots + 1 \log 1 < \frac{1}{2} n^2 \log n$ comparisons. We gain another factor of two because we compare pairs of (line,angle) in the first variant, i.e., each comparison in the sorting step actually requires two comparisons, while the sorting in the second variant only requires one comparison to compare two angles. Consequently, the run-time dropped to 187 ms [8], which is currently the 11th best run-time on the Codeforces server. Actually not too bad for a lousy programmer.

Acknowledgements. We would like to thank the reviewers for their helpful comments, and also for generously ignoring some shortcomings of the original draft that was written in a great hurry shortly before the submission deadline.

References

1. ACM-ICPC International Collegiate Programming Contest. `https://icpc.baylor.edu`.
2. Problem 603D–36: Ruminations on Ruminants. *Codeforces*, Contest 342, Division 1, 2015, Dec 1. `http://codeforces.com/contest/603/problem/D`.
3. Wikipedia contributors. Inscribed angle, Date retrieved: 20 February 2016 15:40 UTC. Permanent link: `https://en.wikipedia.org/w/index.php?title=Inscribed_angle&oldid=699778165`.
4. Wikipedia contributors. Arrangement of lines, Date retrieved: 29 February 2016 15:24 UTC. Permanent link: `https://en.wikipedia.org/w/index.php?title=Arrangement_of_lines&oldid=702141041`.
5. R. Fleischer. FUN with implementing algorithms. In E. Lodi, L. Pagli, and N. Santoro, editors, *Proceedings of the 1998 International Conference FUN with Algorithms (FUN'98)*, pages 88–98. Carleton Scientific, Proceedings in Informatics 4, 1999.
6. R. Fleischer. Problem 603D–36: Submission 14741117. *Codeforces*, Contest 342, Division 1, 2015, Dec 10. `http://codeforces.com/contest/603/submission/14741117`.
7. R. Fleischer. Problem 603D–36: Submission 16228941. *Codeforces*, Contest 342, Division 1, 2016, Jan 20. `http://codeforces.com/contest/603/submission/16228941`.
8. R. Fleischer. Problem 603D–36: Submission 16231449. *Codeforces*, Contest 342, Division 1, 2016, Jan 20. `http://codeforces.com/contest/603/submission/16231449`.

9 R. Fleischer and Y. Wang. On the camera placement problem. In *Proceedings of the 20th International Symposium on Algorithms and Computation (ISAAC'09).* Springer Lecture Notes in Computer Science 5878, pages 255–264, 2009.

10 M. Mirzayanov. Codeforces: The only programming contests web 2.0 platform, 2010–2016. `http://codeforces.com`.

11 Z. Shi. Problem 603D–36: Submission 15094164. *Codeforces*, Contest 342, Division 1, 2015, Dec 30. `http://codeforces.com/contest/603/submission/15094164`.

Large Peg-Army Maneuvers

Luciano Gualà[1], Stefano Leucci[2], Emanuele Natale[2], and Roberto Tauraso[4]

1 **Università di Roma Tor Vergata, Italy**
guala@mat.uniroma2.it
2 **Sapienza Università di Roma, Italy**
leucci@di.uniroma1.it
3 **Sapienza Università di Roma, Italy**
natale@di.uniroma1.it
4 **Università di Roma Tor Vergata, Italy**
guala@mat.uniroma2.it

Abstract

Despite its long history, the classical game of peg solitaire continues to attract the attention of the scientific community. In this paper, we consider two problems with an algorithmic flavour which are related with this game, namely Solitaire-Reachability and Solitaire-Army. In the first one, we show that deciding whether there is a sequence of jumps which allows a given initial configuration of pegs to reach a target position is NP-complete. Regarding Solitaire-Army, the aim is to successfully deploy an army of pegs in a given region of the board in order to reach a target position. By solving an auxiliary problem with relaxed constraints, we are able to answer some open questions raised by Csakany and Juhasz (Mathematics Magazine, 2000).

1998 ACM Subject Classification G.2.0 General

Keywords and phrases Complexity of Games, Solitaire Army

Digital Object Identifier 10.4230/LIPIcs.FUN.2016.18

1 Introduction

Not so very long ago there became widespread an excellent kind of game, called Solitaire, where I play on my own, but as if with a friend as witness and referee to see that I play correctly. A board is filled with stones set in holes, which are to be removed in turn, but none (except the first, which may be chosen for removal at will) can be removed unless you are able to jump another stone across it into an adjacent empty place, when it is captured as in Draughts. He who removes all the stones right to the end according to this rule, wins; but he who is compelled to leave more than one stone still on the board, yields the palm. This game can more elegantly be played backwards, after one stone has been put at will on an empty board, by placing the rest with it, but the same rule being observed for the addition of stones as was stated just above for their removal. Thus we can either fill the board, or, what would be more clever, shape a predetermined figure from the stones; perhaps a triangle, a quadrilateral, an octagon, or some other, if this be possible; but such a task is by no means always possible: and this itself would be a valuable art, to foresee what can be achieved; and to have some way, particularly geometrical, of determining this. – Gottfried Wilhelm Leibniz[1]

[1] Original Latin text in *Miscellanea Berolinensia* **1** (1710) 24, the given translation is from [3].

© Luciano Gualà and Stefano Leucci and Emanuele Natale, and Roberto Tauraso;
licensed under Creative Commons License CC-BY
8th International Conference on Fun with Algorithms (FUN 2016).
Editors: Erik D. Demaine and Fabrizio Grandoni; Article No. 18; pp. 18:1–18:15
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

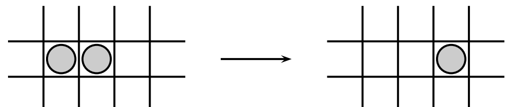


Figure 1 A peg solitaire move.

In this work we investigate some computational and mathematical aspects of the classical game known as *peg solitaire*. In peg solitaire, we have a grid graph (the *board*) on each of whose nodes (the *holes*) there may be at most one *peg*. The initial configuration of pegs evolves by performing one of the following four moves (the *jumps*): for each triple of horizontally or vertically adjacent nodes, if the first and the second nodes are occupied by pegs and there is no peg on the third one, then we can remove the two pegs and place a new one on the third node (see Figure 1). A *puzzle* of peg solitaire is defined by an initial and a final configuration, and consists of finding a sequence of moves that transforms the initial configuration into the final one.

Because of the complexity generated by such simple rules [12, 14, 9, 6], the game has attracted the attention of many mathematically-inclined minds over its long history (for which we refer the reader to Beasley's writings on the topic [7, 4, 5]). In fact, we started with a quotation from Leibniz, and the origin of the game may well precede his time. Despite such a respectable age, the problem of deciding which peg solitaire puzzles can be solved is far from settled [4].

The present paper contributes to such investigation by considering two specific problems:

- Solitaire-Reachability: given a target position and an initial configuration of pegs on a finite board, we wish to determine whether there exists a sequence of moves that allows some peg to be placed in the given target position.
- Solitaire-Army: given a target position and a region of an infinite board, find an initial configuration of pegs inside that region, and a finite sequence of moves that allows some peg to be placed in the given target position.

We study the former problem from a computational point of view and we prove that Solitaire-Reachability is NP-complete. This result, which we discuss in Section 2, is a significant step in understanding why peg solitaire puzzles are intrinsically difficult. They are actually still a good testbed for artificial intelligence techniques [19, 18]. Indeed, the first computational hardness result for this game was proved in 1990 in [22], but was limited to those peg solitaire puzzles in which the final configuration is required to have only one peg (and hence the goal was that of cleaning the entire board).

The Solitaire-Army problem has received a lot of attention from the mathematical and game-connoisseur community [7, 1, 8]. In this body of works, the part of the board where no pegs are allowed in the initial configuration is called *desert*, and the typical goal is reaching the farthest distance inside the desert.

In the classical example, introduced by J. H. Conway, the desert is a half-plane. Conway devised an elegant potential function argument which shows that, for any initial configuration, and no matter what sequence of moves one may attempt, no peg can reach a distance larger than four in the desert [17].

Other shapes for the desert have been considered. For instance, in [11], the authors focus on square-shaped and rhombus-shaped deserts. Here the natural target position is the center of the square/rhombus, and the goal is that of finding the largest size of the desert for which the puzzle is solvable. Among other results, in [11] it is shown that the 9×9 square-shaped

and 13×13 rhombus-shaped deserts are solvable, while the 13×13 square-shaped and 17×17 rhombus-shaped are not. Therefore the problems whether the 11×11 square-shaped and 15×15 rhombus-shaped deserts are solvable were left open[2].

In Section 3 we discuss our contribution to the SOLITAIRE-ARMY problem. We develop a general approach that can be used to attack such kind of puzzles, and as a byproduct, we are able to show that both the 11×11 square-shaped and 15×15 rhombus-shaped deserts are actually solvable. The main idea underlying our technique is considering an auxiliary problem where the constraint that any node can have at most one peg is *relaxed* to allowing pegs to be stacked. We actually allow each node to have any integer number of pegs, including negative. This setting has the advantage that the order of the moves is immaterial. The auxiliary problem admits a natural compact Integer Linear Programming (ILP) formulation which, when the puzzle is simple enough, can be safely solved by means of an ILP solver. Once the relaxed problem is solved we provide also an efficient algorithm which *converts* the relaxed solution into a solution for the original problem. To appreciate the combinatorial beauty of our solutions, we recommend to visit the gallery of animations provided at `http://solitairearmy.isnphard.com`.

Interestingly enough, an analogous relaxation was considered in [10] for a similar problem where a different set of moves is used (the so-called *pebbling* game). The authors claim the equivalence between the relaxed and the original problem (see Lemma 3 in [10]) but the proof is omitted as an 'easy induction argument'. Apparently, E. W. Dijkstra disagrees with such statement: "*it would have been nice if 'the easy induction argument' had been shown: a few colleagues and I spent* $1\frac{1}{2}$ *hours on not finding it*" [13]. Since our equivalence result holds for a large class of moves which includes both pegs and pebbles, we also obtain their result as a corollary.

Other related results. Our hardness reduction contributes to the line of research investigating the computational complexity of combinatorial games [16, 2, 15]. As for positive results regarding peg solitaire, it has been shown that, on a rectangular board of fixed height, the set of initial configurations that can be reduced to a single peg form a regular language [21, 20]. Finally, the idea of formulating peg solitaire puzzles as integer linear programs was already suggested in [3] and effectively used in [19].

2 Our Hardness Reduction

2.1 Overview of the Reduction

Here we consider the problem SOLITAIRE-REACHABILITY: given an initial configuration of pegs on a finite board, it asks to decide whether there exists a sequence of moves that cause a peg to be placed in a given target position. We prove that SOLITAIRE-REACHABILITY is NP-complete.

Our reduction is from the *planar circuit satisfiability problem* (PCSAT for short): in PCSAT we are given a *boolean network* represented as a planar directed acyclic graph G having a single *sink* vertex $t \in V(G)$ (i.e., a vertex having out-degree 0). Each other vertex in $V(G)$ is either an *input* vertex or a *NAND* vertex. The vertex t is required to have in-degree 1, input vertices must be sources in G (i.e. their in-degree must be 0), while NAND vertices must have in-degree exactly 2 and out-degree at least 1. The problem consists in determining

[2] Note that the square/rhombus should have the side of odd length in order for the center of the desert to be well-defined.

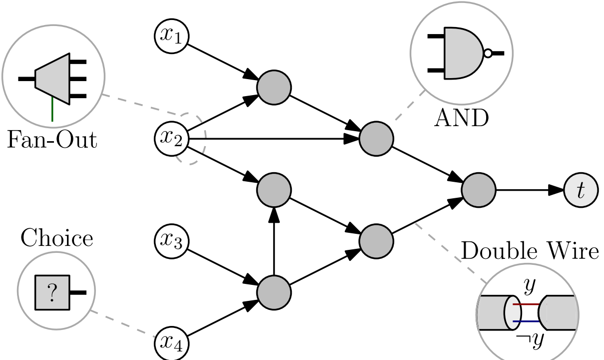


Figure 2 An instance of PCSat corresponding to the formula $(\neg x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3 \vee \neg x4)$. Gadgets that we need to implement are highlighted.

whether it is possible to assign a truth value $\pi(u) \in \{\text{True}, \text{False}\}$ to each vertex $u \in G$ in order to satisfy the following properties:

- the truth value assigned to a NAND vertex u is the NAND of the truth values of its two in-neighbors, i.e., if v_1 and v_2 are the in-neighbors of u, we have $\pi(u) = \neg(\pi(v_1) \wedge \pi(v_2))$;
- the value assigned to the sink vertex t coincides with the truth value of its only in-neighbor;
- $\pi(t) = \text{True}$.

Notice that the assignment $\pi(\cdot)$ is completely determined by the truth values of the input vertices. It is well known that this problem is NP-hard.[3] Clearly, the graph can be thought as a *boolean circuit* consisting of *links* and *gates* which computes a boolean output as a function of the circuit's inputs $x_1, x_2, \dots$.

Given any instance of PCSat, we will build an instance of Solitaire-Reachability which simulates the behavior of such a circuit. We encode the circuit's mechanics using *dual-rail logic*: each edge of the network will be transformed into a *dual-rail wire* consisting of two *single wires* which will always "carry" opposite boolean values. In order to perform such a transformation we will make use of some gadgets: the *choice gadget* will be used to encode input vertices, the *NAND gadget* will represent a NAND vertex, and the *fan-out gadget* will allow to split a single double-wire into multiple double-wires to be fed as input into other gadgets. For technical reasons, the fan-out gadget will output an additional boolean *control signal* on a dedicated single wire, which we will call *control wire*. Intuitively, if we require this control signal to be True, then the correct operation of the fan-out gadget is guaranteed. All these control wires can be safely brought outside of the area of the board that contains the gadgets and all the other wires. Once this has been done, we can finally AND together the boolean signals carried by all these control lines along with the signal carried by the double wire corresponding to the unique edge entering the output vertex t. Since G is planar we will have no intersections between double-wires. However, the same does not hold for control wires. Whenever a control wire intersects a double wire we will use a suitable additional gadget which we call *control-crossover* that ensure that, whenever the control signal is True, the signal carried by the double wire will not be affected.

Figure 2 shows a possible instance of PCSat and highlights the gadgets we need to implement, while Figure 3 is a high level picture of the associated instance of Solitaire-

[3] See, for example, the notes of lecture 6 of the course "Algorithmic Lower Bounds: Fun with Hardness Proofs" by Prof. Erik Demaine (`http://courses.csail.mit.edu/6.890/fall14/lectures/`).

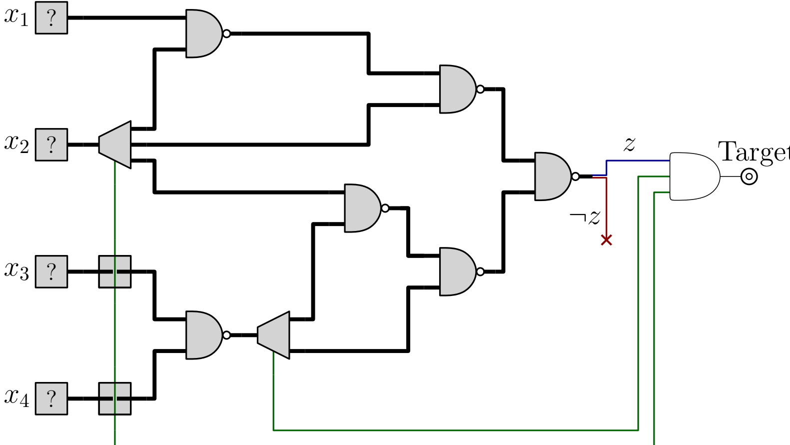


Figure 3 A high-level picture of the instance of SOLITAIRE-REACHABILITY corresponding to the instance of PCSAT shown in Figure 2. Double wires are bold while single wires are thin. Control wires (in green) are ANDed together with the single wire carrying the positive signal z of the circuit output. The negated signal $\neg z$ is ignored.

REACHABILITY. Clearly, the circuit shown in Figure 3 will be implemented as a certain configuration of pegs on a board. The output of the last AND gate will correspond to the target position of SOLITAIRE-REACHABILITY: a peg can be brought to the target position iff there is a truth assignment of the inputs that causes the circuit to output TRUE.

2.2 Description of the Gadgets

In this section we describe the gadgets used by our reduction. We will need to design gates for both binary and dual-rail logic. We will use the following color convention: the former will be colored white, while the latter will be in gray.

2.2.1 Binary Logic Gadgets

We start by describing the gadgets of our construction that deal with binary logic. We have single wires and binary logic gates. Intuitively a single wire "carries" a boolean signal from its first endpoint to the other. The truth value of the signal is encoded as follows: if there is a peg on the first endpoint then the signal is TRUE, and the wire will allow the peg to be moved to the other endpoint via a sequence of moves. On the converse, if there is no peg on the first endpoint then no move can be made and therefore no peg can be placed on the second endpoint, which encodes a FALSE signal. The implementation of a single wire is straightforward and it is shown in Figure 4. In order to deal with parity issues we can use the Shift gadget (also shown in Figure 4) that allows to shift a wire by one row/column of the board. This gadget can also be used to force the signal to flow in one direction.

As far as gates are concerned, each gate takes one or more boolean inputs (which are again encoded by the presence or absence of pegs in certain input positions), and has one or more boolean outputs which are a function of the inputs. If an output is TRUE, this means that there exists a sequence of moves which places a peg on the output position of the gate. Otherwise, if an output is FALSE, no sequence of moves can place a peg on the output position of the gate.

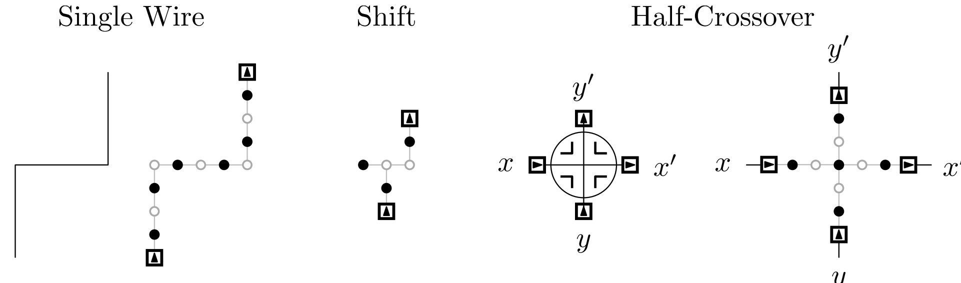


Figure 4 Symbols and implementations of the wire (left), shift (center), and half-crossover (right). In the half-crossover, if the inputs x and y are not both true, then $x' = x$ and $y' = y$. Otherwise either x' or y' will be true (but not both).

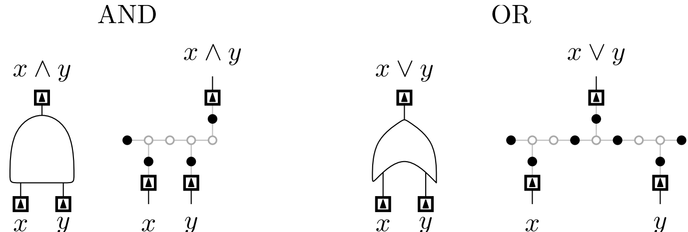


Figure 5 Implementation of the AND and OR binary logic gates.

Binary Logic Half-Crossover

Intuitively, this gadget allows for two single wires to safely cross each other if at least one of them is False. When both signals are True they cannot both cross, and one of them will become False.

More formally, the half-crossover has two inputs x and y and two outputs x' and y'. If x and y are not both true, then x' is true iff x is true, and y' is true iff y is true. Otherwise, if both x and y are true, then only one of x' and y' can be set to true. The gadget is shown in Figure 4.

Binary Logic AND and Binary Logic OR

The binary logic AND and OR gadgets have two inputs and one output and their implementations are shown in Figure 5. In the AND gadget it is possible to place a peg on the output position only when both pegs are present on the input positions while, in the OR gadget, it is possible to output a peg iff at least one input peg is present. Clearly, multiple copies of both the AND and the OR gadgets can be chained together in order to simulate AND and OR gates with multiple inputs.

Binary Logic Fan-Out

This gadget allows to duplicate an input signal x. For technical reasons it also has an additional output signal c that we call control signal. Whenever c is True, this gate acts as a classical fan-out: the values of the two outputs x' and x'' (see Figure 6) will coincide to the value of x. We will always require all the control lines to be True. In order to implement this gate we need an additional gadget that we call AXB gate.

The AXB gate takes three inputs a, x, b (in order) and computes a single output whose value is True iff x is True or both a and b are True. The implementation is given in

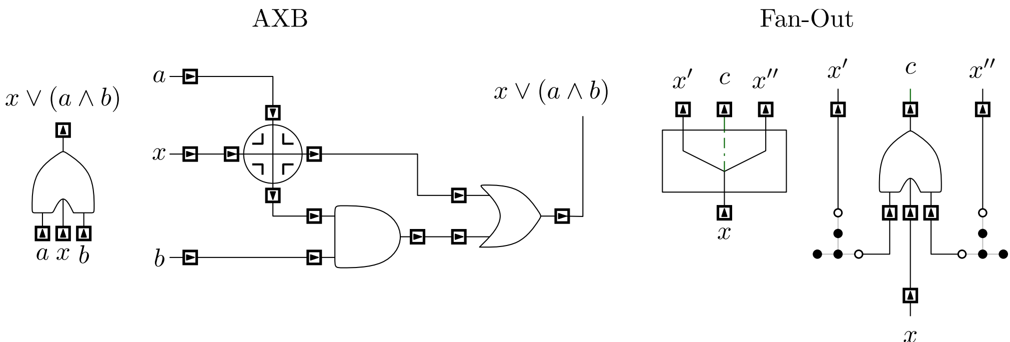


Figure 6 Implementation of the AXB and fan-out binary logic gates.

Figure 6. Notice that if x is TRUE, then a TRUE signal can traverse the half-crossover and trigger the OR-gate, which outputs TRUE. If x is FALSE, then the only way for the gate to output TRUE is to have both a and b set to TRUE. Indeed, in this case, a can traverse the half-crossover and reach the AND gate together with b. Both this signals are need to cause the AND and the OR gates to output TRUE.

Now we argue on the correctness of the fan-out gadget. Assume that the output control signal c is TRUE. This means that either x is TRUE and hence two pegs can be placed on the output positions of x' and x'', or x is FALSE which means that the other two inputs of the AXB gate must both be true, which implies that no peg can reach x' or x''.

Binary Logic Control-Crossover

This gadget allows a control signal c to safely cross a the signal x carried by single wire. The gadget has two outputs c' and x' and its implementation is shown in Figure 7. If c' is required to be TRUE, then c must be also true and x' will be TRUE iff x is TRUE. Notice that if $c' =$ FALSE then x' can be freely set to TRUE or FALSE. However, we will always require c' to be set to TRUE.

We now argue on the correctness of the gadget. We only need to consider the case $c' =$ TRUE. It is easy to see that, in order for c' to be TRUE, c must be TRUE as well. Indeed if $c =$ FALSE, then at least one of the two inputs of the AND gate must be false. Let us assume $c =$ TRUE. Either $x =$ TRUE or $x =$ FALSE. In the first case, the gate can output $x' =$ TRUE and $c' =$ TRUE: we can use the peg from c to allow the peg from x to reach the fan-out gate which can now output three TRUE signals. Notice that this requires the peg from x to jump over the peg from c. In the other case, i.e. $x =$ FALSE, we have that the gate can output $x' =$ FALSE and $c' =$ TRUE by using the peg from c to activate the OR gate, and the control signal from the fan-out gate to activate the end gate. Now, we only need to show that the gate cannot output $x' =$ TRUE and $c' =$ TRUE. Indeed, as $c' =$ TRUE, the control signal from the fan-out gate must be true as well. The claim follows since, in this case, there is no way for a TRUE signal to reach the input of the fan-out gate.

2.2.2 Dual-Rail Logic Gadgets

Double Wire and Dual-Rail Logic Choice

These two gadgets are straightforward and their implementation are shown in Figure 8. Double wires will be drawn in bold to distinguish them from single wires. Moreover we will refer to the pair of signals x and $\neg x$ as $\hat{x}$. In the choice gadget, the central peg must

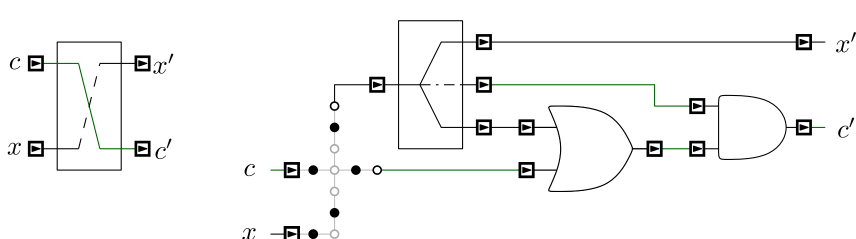


Figure 7 Implementation of the control-crossover gadget. If $c' = \text{TRUE}$, then $c = \text{TRUE}$ and $x' = x$. If $c' = \text{TRUE}$ then x' is unrelated to x and can be either TRUE or FALSE.

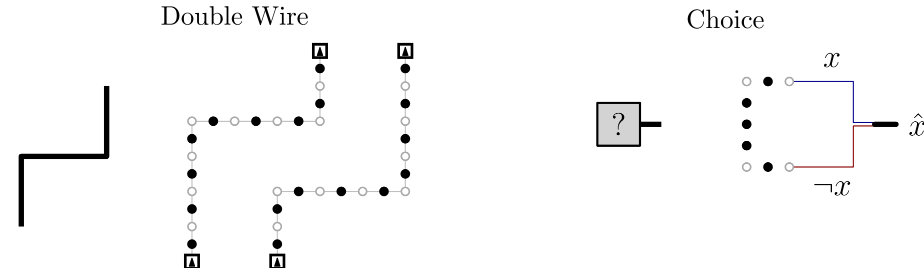


Figure 8 Implementation of a double wire and of the dual-rail logic choice gadget.

jump over either the peg on its top (that corresponds to x) or the peg on its bottom (that corresponds to $\neg x$).

Dual-Rail Logic NAND

We implement this gate by separately building a dual-rail NOT gate and a a dual-rail AND gate, see Figure 9. To compute $\neg\hat{x}$ from $\hat{x}$, we just need to exchange the roles of the single wires corresponding to x and $\neg x$, hence the NOT gate is actually just a half-crossover. The AND gate computes the logic AND of $\hat{x}$ and $\hat{y}$ by separately computing $x \wedge y$ and $\neg(x \wedge y) = \neg x \vee \neg y$. Notice that, every time we use a half-crossover, at least one of its inputs is FALSE.

Dual-Rail Logic Fan-Out

This gate is shown in Figure 10 and it is similar to the binary logic fan-out except that it works on double wires. The gadget makes use of two binary logic fan-outs whose control lines are crossed with the single wires using binary logic control crossovers. The additional control output c will be TRUE iff all the inner control lines are TRUE as well, hence ensuring the correct operation of the gadget. Notice that we also need to cross a single wire carrying the signal $\neg x$ with another single wire carrying $\neg x$ but this can be safely done by using an half-crossover.

Dual-Rail Logic Control-Crossover

This gadget allows for a control wire to safely cross a double wire: if the control output c' is TRUE, then the control input c is true as well and the output x' is equal to the input x

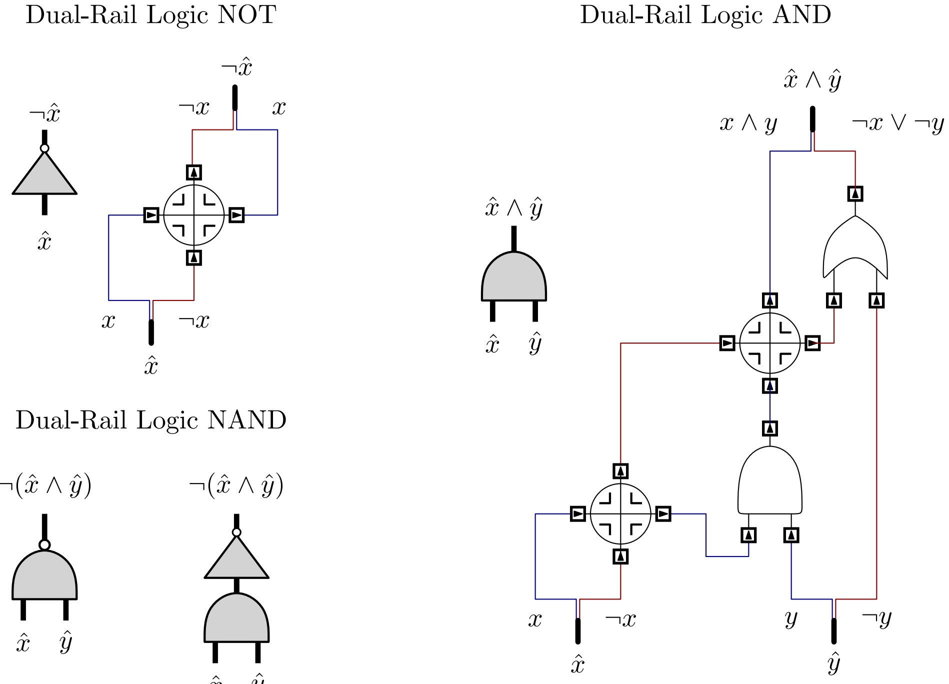


Figure 9 Implementation of the dual-rail logic NOT, AND, and NAND gates.

(see Figure 11). Its implementation is straightforward as it suffices to use two binary logic control-crossovers to cross c with the single wires corresponding to x and $\neg x$, respectively.

3 The Solitaire-Army problem

3.1 Overview

In this section we focus on the Solitaire-Army problem which, given a target position g and a region R of an infinite board, requires finding an initial configuration of pegs inside R, and a finite sequence of moves such that a peg is finally placed in g. We will call *desert* the part of the board where no pegs are allowed in the initial configuration. Moreover, when we consider a configuration of the board, for any position p, we denote by $n(p)$ the number of pegs placed at p in the configuration[4].

When trying to solve an instance of Solitaire-Army, it is convenient to consider the *reversed* version of the game, as if we were rewinding a video of someone trying to solve the puzzle. More formally, in the reverse Solitaire-Army, a move is defined as follows. Let $\mathbf{p} = (p_1, p_2, p_3)$ be a generic triple of vertically or horizontally adjacent positions p_1, p_2 and p_3. A move consists of decreasing by one $n(p_3)$ and increasing by one both $n(p_1)$ and $n(p_2)$, provided that $n(p_1) = n(p_2) = 0$ and $n(p_3) = 1$. Clearly, an instance of Solitaire-Army is solvable if and only if, in the reversed version of the game, it is possible to reach a configuration with no peg in the desert, starting from an initial configuration with a single peg placed in g. In the rest of this section, we will consider always the reversed version of Solitaire-Army and we will refer to it as simply Solitaire-Army.

[4] We remark that in Solitaire-Army it always holds $n(p) \in \{0, 1\}$.

Dual-Rail Logic Fan-out

Figure 10 Implementation of the dual-rail logic fan-out. Whenever $c' =$ TRUE, the gadget duplicates $\hat{x}$ into $\hat{x}'$ and $\hat{x}''$.

We are now ready to describe a general approach that can be conveniently used to attack an instance of SOLITAIRE-ARMY. One of the most annoying issue when trying to solve the problem is the fact that the order of the moves does matter. To circumvent this difficulty, we simplify the game by defining a *relaxed* version of SOLITAIRE-ARMY that we call R-SOLITAIRE-ARMY. In this variant, given a triple $\mathbf{p} = (p_1, p_2, p_3)$ on the board, a move consists of decreasing $n(p_3)$ by one and increasing both $n(p_1)$ and $n(p_2)$ by one, with no constraints on the values of $n(p_1)$, $n(p_2)$ and $n(p_3)$. Therefore we allow each position to have any integer number of pegs, including negative. Here the goal is to reach a configuration in which there are zero pegs in the desert and every position in the region R is occupied by one or zero pegs.

Our main contribution of this section is the following theorem, which shows the equivalence among the two versions of the game. The proof is deferred to the next Section.

▶ **Theorem 1.** R-SOLITAIRE-ARMY *is solvable if and only if* SOLITAIRE-ARMY *is solvable. Moreover, any solution for* R-SOLITAIRE-ARMY *can be transformed in polynomial time into a solution for* SOLITAIRE-ARMY *(with at most the same number of moves).*

We emphasize that Theorem 1 actually holds for a wider class of games (including the *pebbling* game [10]). More precisely, we can define a generalized (reversed) SOLITAIRE-ARMY problem in which the set of moves is specified by a collection of tuples $\mathbf{p}^{(i)} = (p_1^{(i)}, \dots, p_{\ell_i}^{(i)})$ of vertices of the board. In this game a move consists of removing a peg from the vertex $p_{\ell_i}^{(i)}$ and of adding a peg in the remaining vertices of the tuple, provided that $n(p_{\ell_i}^{(i)}) = 1$ and $n(p_1^{(i)}) = \dots = n(p_{\ell_i - 1}^{(i)}) = 0$.

Since it turns out that R-SOLITAIRE-ARMY admits a compact integer linear programming formulation (ILP), then several SOLITAIRE-ARMY puzzles can be solved by using a good

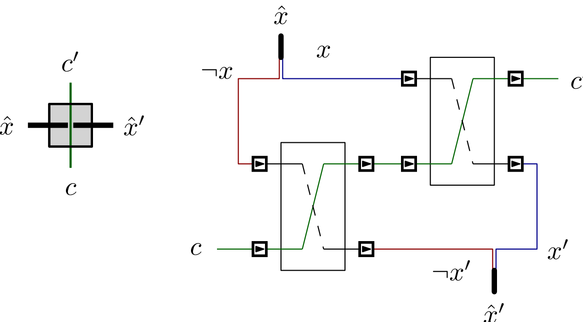


Figure 11 Implementation of the dual-rail logic control-crossover.

Table 1 Solutions to SOLITAIRE-ARMY of particular interest.

Desert	Goal vertex	Number of moves
Square 7×7	Center of the desert	15
Square 9×9	Center of the desert	39
Square 11×11	Center of the desert	212
Square 12×12	One of the four centers of the desert	301
Square 11×11 at the border of the board, when the board is a half-plane	Center of the desert	246
Square 11×11, when the board is the union of three half-planes tangent to the desert	Center of the desert	241
Rhombus 15×15 (i.e. with axes of length 15)	Center of the desert	176
Rhombus 15×15 at the border of the board, when the board is a diagonal half-plane	Center of the desert	202
Rhombus 15×15, when the board is the union of three half-planes tangent to the desert	Center of the desert	183

ILP solver[5]. Table 1 summarizes some of the results that we obtained by using our general approach. We also provide a gallery of *animated* solutions at `http://solitairearmy.isnphard.com`. As an example, in Figure 12, we illustrate a final configuration for the SOLITAIRE-ARMY in which the initial target position is the center of a 11×11 square-shaped desert.

3.2 Proof of Theorem 1

Recall that we are considering the reversed version of SOLITAIRE-ARMY. For any finite board, let n be the number of positions. Let $\mathbf{0}$ be the $n \times 1$ vector whose entries are all 0s, and $\mathbf{1}$ be the $n \times 1$ vector whose entries are all 1s. Let $\mathbf{b}$ be a $n \times 1$ integer vector with $\mathbf{0} \leq \mathbf{b} \leq \mathbf{1}$ which represents the initial configuration of pegs. Let $\mathbf{c}$ be a $n \times 1$ integer vector with $\mathbf{0} \leq \mathbf{c} \leq \mathbf{1}$ which represents the constraints on the final configuration. In particular the zero entries of $\mathbf{c}$ specify the region that at the end should be cleared of pegs (the desert).

[5] We used Gurobi Optimizer, that Gurobi Optimization Inc. has gently made freely available to academic users for research purposes.

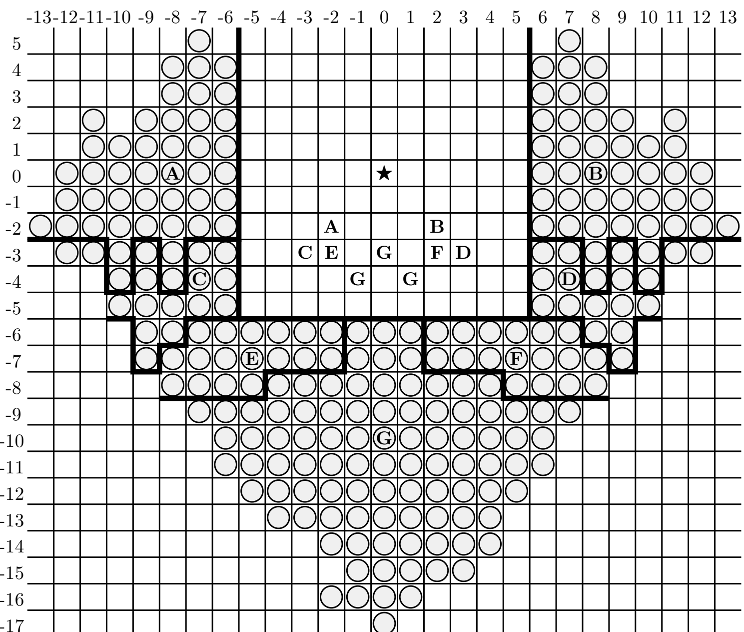


Figure 12 The initial deployment of the SOLITAIRE-ARMY which sends a peg to the center of a square 11×11 desert (this problem was left open in [11]). The army is divided into seven platoons each denoted by a letter. The letters inside the square represent the positions reached by the corresponding platoons. When those positions are taken, it is easy to conquer the center denoted by a star. The complete sequence of moves is given in the Section 3.3.

An $n \times m$ matrix $A = (a_{ij})$ defines the set of moves on the board: for $j = 1, \ldots, m$, the jth move adds $a_{ij} \in \{0, 1, -1\}$ pegs to the ith position. The *fundamental assumption* on this matrix is that each column of A, a move, has at most a -1 entry. No further assumption on the structure of matrix A is made.

According of this notation, we state the relaxed problem (R-SOLITAIRE-ARMY): find a $m \times 1$ integer vector $\mathbf{x} \geq \mathbf{0}$ such that

$$\mathbf{0} \leq A\mathbf{x} + \mathbf{b} \leq \mathbf{c}.$$

Note that the component $(A\mathbf{x} + \mathbf{b})_i$ gives the number of pegs at the ith position after we applied $(\mathbf{x})_j$ times the jth move for $j = 1, \ldots, m$ to the initial configuration $\mathbf{b}$.

On the other hand, the original problem SOLITAIRE-ARMY can be rephrased as follows. Find $m \times 1$ unit vectors $\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_N$ (a *unit vector* is a vector which has an entry 1 and 0s elsewhere) such that

$$\mathbf{0} \leq A\mathbf{x}_k + \mathbf{b} \leq \mathbf{1} \quad \text{for } 1 \leq k \leq N-1 \quad \text{and} \quad \mathbf{0} \leq A\mathbf{x}_N + \mathbf{b} \leq \mathbf{c}.$$

where $\mathbf{x}_k := \sum_{t=1}^{k} \mathbf{u}_t$. Hence if a solution $\mathbf{x}$ of R-SOLITAIRE-ARMY is available, we need to find a decomposition of $\mathbf{x}$ into an ordered sum of unit vectors which represents a specific sequence of feasible moves for SOLITAIRE-ARMY.

For a generic vector $\mathbf{x}$, we denote by $|\mathbf{x}|$ the sum of the entries of $\mathbf{x}$. The following preliminary lemma describes a crucial property we will need later.

▶ **Lemma 2.** *Let* $\mathbf{x}$ *be a solution of* R-SOLITAIRE-ARMY *and let* $\mathbf{y}$ *be a* $m \times 1$ *integer vector such that*

$$\mathbf{0} \leq \mathbf{y} \leq \mathbf{x} \quad \textit{and} \quad A\mathbf{y} \geq \mathbf{0}.$$

If $\mathbf{y}$ *is maximal, i. e. there is no unit vector* $\mathbf{u} \leq \mathbf{x} - \mathbf{y}$ *such that* $\mathbf{y}' := \mathbf{y} + \mathbf{u}$ *satisfies the above inequalities, then* $\mathbf{x}' := \mathbf{x} - \mathbf{y}$ *is another solution of* R-Solitaire-Army*.*

Proof. We have to show that $\mathbf{0} \leq A\mathbf{x}' + \mathbf{b} \leq \mathbf{c}$. The inequality on the right is trivial:

$$A\mathbf{x}' + \mathbf{b} \leq (A\mathbf{x}' + \mathbf{b}) + A\mathbf{y} = A\mathbf{x} + \mathbf{b} \leq \mathbf{c}.$$

If the remaining inequality $A\mathbf{x}' + \mathbf{b} \geq \mathbf{0}$ does not hold then there is some $i \in \{1, \dots, n\}$ such that $(A\mathbf{x}' + \mathbf{b})_i \leq -1$ which implies

$$(A\mathbf{x}')_i \leq -1 - (\mathbf{b})_i \leq -1.$$

Hence there exists a unit vector $\mathbf{u} \leq \mathbf{x}'$ such that $(A\mathbf{u})_i = -1$. We claim that $A(\mathbf{y} + \mathbf{u}) \geq \mathbf{0}$, which contradicts the maximality of $\mathbf{y}$. As regards the ith entry,

$$(A(\mathbf{y} + \mathbf{u}))_i = (A\mathbf{x} + \mathbf{b})_i - (A\mathbf{x}' + \mathbf{b})_i + (A\mathbf{u})_i \geq 0 + 1 - 1 = 0.$$

For $l \neq i$, due to the fundamental assumption on the matrix A, we have $(A\mathbf{u})_l \geq 0$ and the proof is complete. ◀

In order to prove Theorem 1, it suffices to show that if R-Solitaire-Army is solvable then Solitaire-Army is solvable as well. The proof is by induction on the the number of moves $|\mathbf{x}|$ of the solution of R-Solitaire-Army $\mathbf{x}$. If $|\mathbf{x}| = 1$ then we set $\mathbf{u}_1 := \mathbf{x}$ and we are done. Assume that $|\mathbf{x}| > 1$ and any $\mathbf{x}'$ solution of R-Solitaire-Army with $|\mathbf{x}'| < |\mathbf{x}|$ can be transformed into a solution of Solitaire-Army.

Let $\mathbf{x}_0 = \mathbf{0}$. We need to show that if we are at the step k with $0 \leq k < |\mathbf{x}|$ and $\mathbf{0} \leq A\mathbf{x}_k + \mathbf{b} \leq \mathbf{1}$ then we are able to move forward to step $k+1$: there exists a unit vector $\mathbf{u} \leq \mathbf{x} - \mathbf{x}_k$ such that $\mathbf{0} \leq A\mathbf{x}_{k+1} + \mathbf{b} \leq \mathbf{1}$ where $\mathbf{x}_{k+1} := \mathbf{x}_k + \mathbf{u}$.

We divide the proof of this fact into two claims.

▶ **Claim 3.** *There is a unit vector* $\mathbf{u}$ *such that* $\mathbf{u} \leq \mathbf{x} - \mathbf{x}_k$ *and* $A(\mathbf{x}_k + \mathbf{u}) + \mathbf{b} \geq \mathbf{0}$*.*

Proof. Let $\mathbf{z} := \mathbf{x} - \mathbf{x}_k$, then $\mathbf{z} \geq 0$ and $\mathbf{z} \neq \mathbf{0}$.

If $A\mathbf{z} \geq \mathbf{0}$ then we extend $\mathbf{z}$ to a maximal vector $\mathbf{y}$ such that $\mathbf{z} \leq \mathbf{y} \leq \mathbf{x}$ and $A\mathbf{y} \geq \mathbf{0}$. By Lemma 2, $\mathbf{x}' := \mathbf{x} - \mathbf{y}$ is another solution for R-Solitaire-Army with $|\mathbf{x}'| < |\mathbf{x}|$ and, by the induction hypothesis, it can be transformed into a solution of Solitaire-Army.

On the other hand, if $A\mathbf{z} \geq \mathbf{0}$ does not hold, then there is some $i \in \{1, \dots, n\}$ such that $(A\mathbf{z})_i \leq -1$ and there is a unit vector $\mathbf{u} \leq \mathbf{z}$ such that $(A\mathbf{u})_i = -1$. If $l \neq i$ it follows that

$$(A(\mathbf{x}_k + \mathbf{u}) + \mathbf{b})_l = (A\mathbf{x}_k + \mathbf{b})_l + (A\mathbf{u})_l \geq (A\mathbf{x}_k + \mathbf{b})_l \geq 0.$$

As regards the ith entry, $A\mathbf{x} + \mathbf{b} \geq \mathbf{0}$ implies

$$(A\mathbf{x}_k + \mathbf{b})_i = (A\mathbf{x} + \mathbf{b})_i - (A\mathbf{z})_i \geq 0 + 1 = 1$$

and therefore

$$(A(\mathbf{x}_k + \mathbf{u}) + \mathbf{b})_i = (A\mathbf{x}_k + \mathbf{b})_i + (A\mathbf{u})_i = (A\mathbf{x}_k + \mathbf{b})_i - 1 \geq 1 - 1 = 0.$$

Hence $A(\mathbf{x}_k + \mathbf{u}) + \mathbf{b} \geq \mathbf{0}$. ◀

▶ **Claim 4.** *There is a unit vector* $\mathbf{u}$ *such that* $\mathbf{u} \leq \mathbf{x} - \mathbf{x}_k$ *and* $\mathbf{0} \leq A(\mathbf{x}_k + \mathbf{u}) + \mathbf{b} \leq \mathbf{1}$*.*

Proof. Let $\mathcal{U}$ be the set of unit vectors $\mathbf{u} \leq \mathbf{x} - \mathbf{x}_k$ such that $A(\mathbf{x}_k + \mathbf{u}) + \mathbf{b} \geq \mathbf{0}$. The set $\mathcal{U}$ is not empty by Claim 3. If Claim 4 holds then the proof is complete. Otherwise given any $\mathbf{u} \in \mathcal{U}$ there is some $i \in \{1, \ldots, n\}$ such that

$$(A\mathbf{x}_k + \mathbf{b})_i = 1 \quad \text{and} \quad (A\mathbf{u})_i = 1.$$

On the other hand, $\mathbf{0} \leq A\mathbf{x} + \mathbf{b} \leq \mathbf{c}$ implies

$$(A(\mathbf{x} - \mathbf{x}_k - \mathbf{u}))_i = (A\mathbf{x} + \mathbf{b})_i - (A\mathbf{x}_k + \mathbf{b})_i - (A\mathbf{u})_i \leq 1 - 1 - 1 = -1,$$

and there is a unit vector $\mathbf{v} \leq \mathbf{x} - \mathbf{x}_k - \mathbf{u}$ such that $(A\mathbf{v})_i = -1$. Hence

$$(A(\mathbf{x}_k + \mathbf{v}) + \mathbf{b})_i = (A\mathbf{x}_k + \mathbf{b})_i + (A\mathbf{v})_i = 1 - 1 = 0$$

which means that $\mathbf{v} \in \mathcal{U}$.

By iterating this process, we generate a sequence of elements of $\mathcal{U}$. Since $\mathcal{U}$ is finite, this sequence has a minimal cycle, $\mathbf{u}_1, \ldots, \mathbf{u}_{r-1}, \mathbf{u}_r = \mathbf{u}_1$ and, by construction, for $2 \leq t \leq r$, $(A\mathbf{u}_t)_i = -1$ implies that $(A\mathbf{u}_{t-1})_i = 1$. Setting $\mathbf{z} := \sum_{t=2}^{r} \mathbf{u}_t$, we have $\mathbf{0} \leq \mathbf{z} \leq \mathbf{x}$, $A\mathbf{z} \geq \mathbf{0}$ and $\mathbf{z} \neq \mathbf{0}$. Now, as in Claim 3, we extend $\mathbf{z}$ to a maximal vector $\mathbf{y}$. Then, by Lemma 2, $\mathbf{x}' := \mathbf{x} - \mathbf{y}$ is another solution for R-Solitaire-Army with $|\mathbf{x}'| < |\mathbf{x}|$ and finally we apply the induction hypothesis. ◀

Finally, observe that the proof is constructive and can be easily turned into a polynomial-time algorithm that performs the transformation.

3.3 Explicit Solution to the Solitaire-Army Problem of Figure 12

Given a configuration of pegs on a board, we denote a jump with (x,y,d), where $d \in \{\uparrow, \downarrow, \leftarrow, \rightarrow\}$ indicates the direction of the jump that has to be made by the peg in position (x, y) of the board. The moves that allows the peg army in Figure 12 to reach the center are listed below.

Platoon A: (-7,-2,→), (-9,-2,→), (-11,-2,→), (-13,-2,→), (-10,-4,↑), (-10,-2,→), (-10,0,↓), (-12,0,→), (-11,2,↓), (-12,-1,→), (-11,-2,→), (-8,-2,→), (-8,-4,↑), (-9,-2,→), (-6,-2,→), (-6,0,↓), (-6,2,↓), (-6,4,↓), (-8,1,→), (-10,1,→), (-7,3,↓), (-7,5,↓), (-9,2,→), (-8,4,↓), (-8,-1,→), (-10,-1,→), (-7,1,↓), (-7,3,↓), (-9,0,→), (-11,0,→), (-8,2,↓), (-7,-2,→), (-5,-2,→), (-7,-1,→), (-7,1,↓), (-9,0,→), (-8,-1,→), (-6,-1,→), (-7,0,→), (-6,2,↓), (-6,0,→), (-4,0,↓), (-4,-2,→).
Platoon B: symmetric moves of **Platoon A**.
Platoon C: (-8,-6,↑), (-10,-5,→), (-12,-3,→), (-10,-3,→), (-9,-7,↑), (-9,-5,↑), (-7,-3,→), (-6,-5,↑), (-7,-5,↑), (-6,-3,→), (-8,-3,→), (-8,-5,↑), (-9,-3,→), (-7,-3,→), (-5,-3,→).
Platoon D: symmetric moves of **Platoon C**.
Platoon E: (-2,-7,↑), (-3,-7,↑), (-4,-7,↑), (-6,-6,→), (-5,-8,↑), (-5,-6,→), (-6,-8,↑), (-7,-6,→), (-7,-8,↑), (-8,-8,↑), (-8,-6,→), (-6,-6,→), (-3,-6,↑), (-4,-6,↑), (-4,-4,→), (-2,-5,↑).
Platoon F: symmetric moves of **Platoon E**.
Platoon G: (-3,-9,↑), (-5,-9,→), (-7,-9,→), (-6,-11,↑), (-6,-9,→), (-3,-10,↑), (-5,-10,→), (-5,-12,↑), (-4,-12,↑), (-3,-11,↑), (-5,-10,→), (-4,-9,↑), (-4,-7,→), (-3,-9,↑), (1,-7,↑), (1,-9,↑), (3,-9,←), (5,-9,←), (7,-9,←), (6,-11,↑), (6,-9,←), (4,-9,←), (3,-11,↑), (5,-10,←), (5,-12,↑), (4,-12,↑), (4,-14,↑), (3,-8,←), (3,-10,↑), (5,-10,←), (4,-8,←), (1,-8,↑), (1,-10,↑), (3,-10,←), (2,-12,↑), (4,-12,←), (3,-14,↑), (1,-11,↑), (1,-13,↑), (1,-15,↑), (3,-15,←), (1,-16,↑), (3,-12,←), (1,-12,↑), (0,-7,↑), (0,-9,↑), (-2,-9,→), (-1,-11,↑), (-1,-13,↑), (-1,-15,↑), (-3,-12,→), (-2,-14,↑), (-4,-13,→), (-3,-10,→), (2,-8,←), (2,-10,↑), (1,-10,↑), (0,-8,↑), (2,-8,←), (0,-9,↑), (0,-11,↑),

(-2,-11,→), (0,-12,↑), (-2,-12,→), (0,-13,↑), (2,-13,←), (0,-14,↑), (2,-14,←), (-2,-13,→), (-2,-8,→), (-1,-10,↑), (-1,-7,↑), (-3,-7,→), (-1,-8,↑), (1,-6,↑), (0,-6,↑), (0,-8,↑), (0,-10,↑), (0,-12,↑), (0,-14,↑), (0,-16,↑), (-2,-16,→), (0,-17,↑), (0,-15,↑), (0,-13,↑), (0,-11,↑), (0,-9,↑), (0,-7,↑), (-1,-6,↑), (0,-5,↑).

Finale: (-3,-3,→), (3,-3,←), (-1,-4,↑), (1,-4,↑), (-2,-2,→), (0,-3,↑), (2,-2,←), (0,-2,↑).

References

1. M. Aigner. Moving into the Desert with Fibonacci. *Math. Magazine*, 70(1):11–21, 1997.
2. G. Aloupis, E. D. Demaine, A. Guo, and G. Viglietta. Classic Nintendo games are (computationally) hard. *Theoretical Computer Science*, 586:135–160, 2015.
3. J. D. Beasley. *The Ins and Outs of Peg Solitaire*. Oxford University Press, 1985.
4. J. D. Beasley. Solitaire: Recent Developments. *arXiv:0811.0851*, 2008.
5. J. D. Beasley. John and Sue Beasley's Webpage on Peg Solitaire, 2015. URL: `http://jsbeasley.co.uk/pegsol.htm`.
6. G. I. Bell. A Fresh Look at Peg Solitaire. *Mathe.Magazine*, 80(1):16–28, 2007.
7. G. I. Bell, D. S. Hirschberg, and P. Guerrero-Garcia. The minimum size required of a solitaire army. *arXiv/0612612*, 2006.
8. E. R. Berlekamp, J. H. Conway, and R. K. Guy. *Winning Ways for Your Mathematical Plays, Volume 2*. AK Peters, 2002.
9. A. Bialostocki. An application of elementary group theory to central solitaire. *The College Mathematics Journal*, 29(3):208, 1998.
10. F. Chung, R. Graham, J. Morrison, and A. Odlyzko. Pebbling a Chessboard. *The American Mathematical Monthly*, 102(2):113–123, 1995.
11. B. Csakany and R. Juhasz. The Solitaire Army Reinspected. *Math. Magazine*, 73(5):354–362, 2000.
12. E. W. Dijkstra. The checkers problem told to me by M.O. Rabin, 1992. URL: `http://www.cs.utexas.edu/users/EWD/ewd11xx/EWD1134.PDF`.
13. E. W. Dijkstra. Only a matter of style?, 1995. URL: `http://www.cs.utexas.edu/users/EWD/ewd12xx/EWD1200.PDF`.
14. M. Gardner. *The Unexpected Hanging and Other Mathematical Diversions*. University of Chicago Press, 1991.
15. L. Guala, S. Leucci, and E. Natale. Bejeweled, Candy Crush and other match-three games are (NP-)hard. In *CIG 2014*, pages 1–8, 2014.
16. R. A. Hearn and E. D. Demaine. *Games, puzzles, and computation*. AK Peters, 2009.
17. R. Honsberger. A problem in checker jumping. *Mathematical Gems II*, pages 23–28, 1976.
18. C. Jefferson, A. Miguel, I. Miguel, and S. A. Tarim. Modelling and solving English Peg Solitaire. *Computers & Operations Research*, 33(10):2935–2959, 2006.
19. M. Kiyomi and T. Matsui. Integer Programming Based Algorithms for Peg Solitaire Problems. In *Computers and Games*, number 2063 in LNCS, pages 229–240. Springer, 2000.
20. C. Moore and D. Eppstein. One-Dimensional Peg Solitaire, and Duotaire. *arXiv/0008172*, 2000.
21. B. Ravikvmar. Peg-solitaire, string rewriting systems and finite automata. In *Algorithms and Computation*, volume 1350, pages 233–242. Springer, 1997.
22. R. Uehara and S. Iwata. Generalized Hi-Q is NP-Complete. *IEICE TRANSACTIONS*, E73-E(2):270–273, 1990.

Loopless Gray Code Enumeration and the Tower of Bucharest

Felix Herter[1] and Günter Rote[2]

1 **Institut für Informatik, Freie Universität Berlin, Takustr. 9, 14195 Berlin, Germany**
avealx@zedat.fu-berlin.de

2 **Institut für Informatik, Freie Universität Berlin, Takustr. 9, 14195 Berlin, Germany**
rote@inf.fu-berlin.de

Abstract

We give new algorithms for generating all n-tuples over an alphabet of m letters, changing only one letter at a time (Gray codes). These algorithms are based on the connection with variations of the Tower of Hanoi game. Our algorithms are loopless, in the sense that the next change can be determined in a constant number of steps, and they can be implemented in hardware. We also give another family of loopless algorithms that is based on the idea of working ahead and saving the work in a buffer.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Tower of Hanoi, Gray code, enumeration, loopless generation

Digital Object Identifier 10.4230/LIPIcs.FUN.2016.19

1 Introduction: Binary reflected Gray code and the Tower of Hanoi

1.1 The Gray code

The Gray code, or more precisely, the reflected binary Gray code G_n, orders the 2^n binary strings of length n in such a way that successive strings differ in a single bit. It is defined inductively as follows, see Figure 1 for an example. The Gray code $G_1 = 0, 1$, and if $G_n = C_1, C_2, \dots, C_{2^n}$ is the Gray code for the bit strings of length n, then

$$G_{n+1} = 0C_1, 0C_2, \dots, 0C_{2^n},\ 1C_{2^n}, 1C_{2^n-1}, \dots, 1C_2, 1C_1 \tag{1}$$

In other words, we prefix each word of G_n with 0, and this is followed by the reverse of G_n with 1 prefixed to each word.

1.2 Loopless algorithms

The Gray code has an advantage over alternative algorithms for enumerating the binary strings, for example in lexicographic order: one can change a binary string $a_n a_{n-1} \dots a_1$ to the successor in the sequence by a single update of the form $a_i := 1 - a_i$ in constant time. However, we also have to *compute* the position i of the bit which has to be updated. A straightforward implementation of the recursive definition (1) leads to an algorithm with an optimal overall runtime of $O(2^n)$, i.e., constant average time per enumerated bit string.

A stricter requirement is to compute each successor string in constant *worst-case* time. Such an algorithm is called a *loopless* generation algorithm. Loopless enumeration algorithms

© Felix Herter and Günter Rote;
licensed under Creative Commons License CC-BY
8th International Conference on Fun with Algorithms (FUN 2016).
Editors: Erik D. Demaine and Fabrizio Grandoni; Article No. 19; pp. 19:1–19:19
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

000000	001011	010111	110100	101110
000001	001001	010110	111100	101111
000011	001000	010010	111101	101101
000010	011000	010011	111111	101100
000110	011001	010001	111110	100100
000111	011011	010000	111010	100101
000101	011010	110000	111011	100111
000100	011110	110001	111001	100110
001100	011111	110011	111000	100010
001101	011101	110010	101000	100011
001111	011100	110110	101001	100001
001110	010100	110111	101011	100000
001010	010101	110101	101010	

0000	0111	0222	1112	1001	2120	2220
0001	0112	1222	1111	1000	2110	2221
0002	0102	1221	1110	2000	2111	2222
0012	0101	1220	1120	2001	2112	
0011	0100	1210	1121	2002	2102	
0010	0200	1211	1122	2012	2101	
0020	0201	1212	1022	2011	2100	
0021	0202	1202	1021	2010	2200	
0022	0212	1201	1020	2020	2201	
0122	0211	1200	1010	2021	2202	
0121	0210	1100	1011	2022	2212	
0120	0220	1101	1012	2122	2211	
0110	0221	1102	1002	2121	2210	

Figure 1 The binary Gray code G_6 for 6-tuples and the ternary Gray code for 4-tuples.

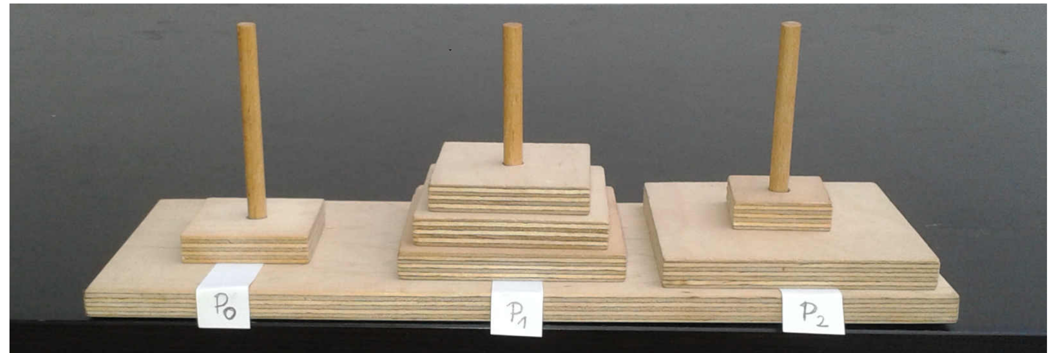


Figure 2 The Tower of Hanoi with $n = 6$ (square) disks. When running the algorithm HANOI from Section 1.5, the configuration in this picture occurs together with the bit string 110011. (The relation between the positions of the disks and this bit string is not straightforward, cf. [13, Section 3].) The next disk to move is D_1; it moves clockwise to peg P_0, and the last bit is complemented. The successor in the Gray code is the string 110010. After that, D_1 pauses for one step, while disk D_3 moves, again clockwise, from P_1 to P_2, and the third bit from the right is complemented, leading to the string 110110.

for various combinatorial structures were pioneered by Ehrlich [3], and different loopless algorithms for Gray codes are known, see Bitner, Ehrlich, and Reingold [1] and Knuth [10, Algorithms 7.2.1.1.L and 7.2.1.1.H]. These algorithms achieve constant running time by maintaining additional pointers.

1.3 The Tower of Hanoi

The Tower of Hanoi is the standard textbook example for illustrating the principle of recursive algorithms. It has n disks $D_1, D_2, \ldots, D_n$ of increasing radii and three pegs P_0, P_1, P_2, see Fig. 2. The goal is to move all disks from the peg P_0, where they initially rest, to another peg, subject to the following rules:

1. Only one disk may be moved at a time: the topmost disk from one peg can be moved on top of the disks of another peg
2. A disk can never lie on top of a smaller disk.

For moving a tower of height n, one has to move disk D_n at some point. But before moving disk D_n from peg A to B, one has to move the disks $D_1, \ldots, D_{n-1}$, which lie on top

of D_n, out of the way, onto the third peg. After moving D_n to B, these disks have to be moved from the third peg to B. This reduces the problem for a tower of height n to two towers of height $n-1$, leading to the following recursive procedure.

> *move_tower*(k, A, B): (Move the k smallest disks $D_1 \dots D_k$ from peg A to peg B)
> **if** $k \leq 0$: **return**
> *auxiliary* $:= 3 - A - B$; (*auxiliary* is the third peg, different from A and B.)
> *move_tower*$(k-1, A, \textit{auxiliary})$
> move disk D_k from A to B
> *move_tower*$(k-1, \textit{auxiliary}, B)$

1.4 Connections between the Tower of Hanoi and Gray codes

The *delta sequence* of the Gray code is the sequence $1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, \dots$ of bit positions that are updated. (In contrast to the usual convention, we number the bits starting from 1.) This sequence has an obvious recursive structure which results from (1). It also describes the number of changed bits when incrementing from i to $i+1$ in binary counting. Moreover, it is easy to observe that the same sequence also describes the disks that are moved by the recursive algorithm *move_tower* above. It has thus been noted that the Gray code G_n can be used to solve the Tower of Hanoi puzzle, cf. Gardner [4]. In the other direction, the Tower of Hanoi puzzle can be used to generate the Gray code G_n, see Buneman and Levy [2].

Several loopless ways to compute the next move for the Tower of Hanoi are known, and they lead directly to loopless algorithms for the Gray code. We describe one such algorithm.

1.5 Loopless Tower of Hanoi and binary Gray code

From the recursive algorithm *move_tower*, it is not hard to derive the following fact.

▶ **Proposition 1.** *If the tower should be moved from P_0 to P_1 and n is odd, or if the tower should be moved from P_0 to P_2 and n is even, the moves of the odd-numbered disks always proceed in forward ("clockwise") circular direction: $P_0 \to P_1 \to P_2 \to P_0$, and the even-numbered disks always proceed in the opposite circular direction: $P_0 \to P_2 \to P_1 \to P_0$.*

In the other case, when the assumption does not hold, the directions are simply swapped. Since we are interested not in moving the tower to a specific target peg, but in generating the Gray code, we stick with the proposition as stated.

> *Algorithm* **HANOI**. Loopless algorithm for the binary Gray code.
> Initialize: Put all disks on P_0.
> **loop**:
> Move D_1 clockwise.
> Let D_k be the smaller of the topmost disks on the two pegs that don't carry D_1.
> If there is no such disk, terminate.
> Move D_k clockwise if k is odd; otherwise, move it counterclockwise.

To obtain the Gray code, we simply set $a_k := 1 - a_k$ whenever we move the disk D_k [2]. See Fig. 2 for a snapshot of the procedure.

We would not need the clockwise/counterclockwise rule for D_k: Since we must not put D_k on top of D_1, there is anyway no choice of where to move it [2]. We have chosen the above formulation since it is better suited for generalization.

1.6 Overview

In the remainder of this paper, we will generalize these connections to Gray codes for larger radixes (alphabet sizes). Section 2 is devoted to ternary Gray codes and their connections to the so-called *Towers of Bucharest*. After defining Gray codes with general radixes in Section 3, we extend the ternary algorithm from Section 2 to arbitrary odd radixes m in Section 4, and even to mixed (odd) radixes (Section 6). In Section 5, we generalize the binary Gray code algorithm of Section 1.5 to arbitrary even m. Finally, in Section 8, we develop loopless algorithms bases on an entirely different idea of "working ahead" that is related to converting amortized running-time bounds to worst-case bounds. In the appendix, we give prototype code for simulating our main algorithms (Sections 4, 5, and 8) in PYTHON. The preprint [8] contains simulations of all our algorithms.

2 Ternary Gray codes and the Towers of Bucharest

A ternary Gray code enumerates the 3^n n-tuples $(a_n, \dots, a_1)$ with $a_i \in \{0,1,2\}$. Successive tuples differ in one entry, and in this entry they differ by ± 1.

The following simple variation of the Towers of Hanoi will yield a ternary Gray code $(m = 3)$: *We disallow the direct movement of a disk between pegs P_0 and P_2*: a disk can only be moved to an adjacent peg. We call this the Towers of Bucharest.[1] This version of the game was already considered in 1944 (not under this name) by Scorer, Grundy, and Smith [13, Section 4(iii)] and has been thoroughly investigated, see Chapter 8 in the extensive monograph about the Tower of Hanoi by Hinz, Klavžar, Milutinović, and Petr [9].

Figure 3 shows the state space of the Towers of Bucharest in comparison with the Towers of Hanoi. In accordance with this figure, we can make the following easy observations:

▶ **Proposition 2.**

1. *In the Towers of Hanoi, there are three possible moves from any position, except when all disks are on one peg: In these cases, there are only two possible moves.*
2. *In the Towers of Bucharest, there are two possible moves from any position, except when all disks are on peg P_0 or P_2: In those cases, there is only one possible move.*

---

[1] The custom of naming variations of the Tower of Hanoi game after different cities, instead of using ordinary names such as "three-in-a-row" [12], has a long tradition. The name "Towers of Bucharest" has been suggested by Günter M. Ziegler. Several legends rank themselves around these towers.
A little count from Transylvania had conquered the whole country and had become a powerful Lord. In order to celebrate his glory, he built a magnificent palace in the capital city Bucharest, and he suppressed his people as best he could. He also had a dog named Heisenberg. In a nearby monastery, the monks had golden disks of different sizes on three pegs, and they had played the Towers of Hanoi game for centuries. It was already forseeable that the game was drawing towards its conclusion. According to an ancient prophecy, the palace of the ruler of the country would crumble and his rule would come to an end when the game would be finished. When the count, who called himself king by this time, heard this story, he did not like it. *First* of all, he had the monks beheaded and told them to do some useful job instead. *Secondly*, he removed the pegs with the discs and took them to his palace. He made sure that they were placed very far away from each other: The first peg was put in the South wing, the second peg in the North wing, and the third peg again in the South wing. One may wonder why he did not place them in some more logical arrangement like South-South-North or South-North-North, or perhaps North-middle-South or South-middle-North. The reader will soon understand that this placement was a clever decision when she or he learns what else he did. The North wing could only be reached from the South wing through the middle wing, or by going out on the street and reentering on the other side, but I don't think it is very wise to go into the street carrying a heavy golden disk.
Anyway, his *third* action was his most wicked and smartest move: It occurred to him that *he was powerful and he was the ruler, and he therefore had the power to change the rules.* He decreed that the discs can only be moved between the first and the second peg or between the second and the third peg. Direct moves between the first and third peg were henceforth forbidden. This would delay the moves of

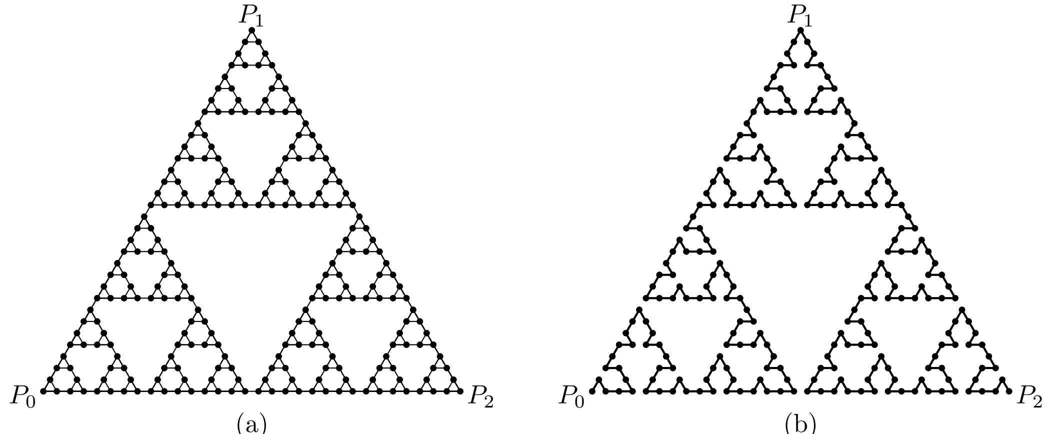


Figure 3 The state graphs of (a) the Tower of Hanoi and (b) the Tower of Bucharest with $n = 5$ disks.

Proof.

1. The disk D_1 can be moved to any of the other pegs (two possible moves). In addition, the smaller of the topmost disks on the other pegs (if those pegs aren't both empty) can be moved to the other peg which is not occupied by D_1.
2. If the disk D_1 is in the middle, it can be moved to any of the other pegs, but no other move is possible. If the disk D_1 is on P_0 or P_2, it has only one possible move, and the smaller of the topmost disks on the other pegs (if those pegs aren't empty) also has one possible move, similarly as above. ◀

Both games have the same set of 3^n states, corresponding to the possible ways of assigning each disk to one of the pegs P_0, P_1, P_2. The nodes in the corners marked P_0, P_1, P_2 represent the states where all disks are on one peg. The graph of the Towers of Hanoi in Figure 3a approaches the Sierpiński gasket. The optimal path of length $2^n - 1$ is the straight path from P_0 to P_2. (The directions of the edges in this drawing of the state graph are not directly related to the pegs that are involved in the exchange, and the relation between a state and

the disks, since they always had to be carried all the way from the South wing to the North wing and back. As shown in this article, the consequences of the new rule in delaying the game are even more spectacular. These measures were definitely overcautious, in particular since nobody was there to move the discs any more, and moreover, the pegs with the golden discs were carefully guarded. Nevertheless, he was worried that his wife and children would wander around in the palace and play with the disks, thereby setting the prophecy into motion again, like in that movie, "Jumanji" with Robin Williams. He was not sure how the guarding officers would behave in a conflict between the loyalty to their orders and the authority of his family members. You may draw your own conclusions, but in my opinion, this count, or king if you wish, was pretty paranoid. In the end, it served him nothing. He was swept away by the revolution. What became of the golden disks? Nobody knows. It is sometimes claimed that they were hidden in a subterranean cave, and hobby archaeologists are still looking for them occasionally. But probably they have found their way to the black market. Today, tourists that visit the palace are led to a stump on the floor in the North wing, which is supposedly the remainder of one of the pegs. The South wing is closed for restoration.
Another story, even more unbelievable but no less bloody than the first one, puts the Towers of Bucharest in the context of the legendary caliph Harun-al-Rashid. One night, the caliph was again wandering through the streets of Baghdad, as usual dressed like an ordinary businessman, in order to assure himself that the people were still loving his reign, admiring his wisdom, and praising his justice. He noticed a crowd of lookers-on who were gathered around a man and a woman sitting on the ground side by side, silently and solemnly executing the moves of the Towers of Bucharest. One of them would pick up a disk and set it on an adjacent peg. By the rules of the game, there was always one of them for whom the two involved pegs were easy to reach, and this was the one who carried out the move. Only on the infrequent occasions when one of the larger and heavier disks had to be moved, they helped each other. The man wore a

its position on the drawing is not straightforward.) By contrast, we see that the graph of the Towers of Bucharest in Figure 3b is a single path through all nodes.

Let us see why this is true. By Proposition 2, this graph has maximum degree 2, and it follows that it must consist of a path between P_0 and P_2 (the only degree-1 nodes), plus a number of disjoint cycles. However, it is known that the path has length $3^n - 1$ and does therefore indeed go through all nodes. Since we will prove a more general statement later (Theorem 3), we only sketch the argument here: Solving the problem recursively in an analogous way to the procedure *move_tower*, we reduce the problem of moving a tower of n disks from P_0 to P_2 (or vice versa) to three problem instances with $n - 1$ disks, plus two movements of disk D_n, and the resulting recursion establishes that $3^n - 1$ moves are required.

The states of the Towers of Bucharest correspond in a natural way to the ternary n-tuples: The digit $a_i \in \{0, 1, 2\}$ gives the position of disk D_i. It follows now easily that the solution of the Towers of Bucharest yields a ternary Gray code: Since we can move only one disk at a time, it means that we change only one digit at a time, and by the special rules of the Towers of Bucharest, we change it by ± 1. This connection has already been noted earlier; it is explicitly mentioned in Graham, Knuth, and Patashnik [5, Exercises 1.2–1.3, p. 17, with answers on p. 483], or Guan [6, Theorem 4]. In fact, the algorithm produces *the* ternary reflected Gray code, which we are about to define below in Section 3; see also Theorem 3. Moreover, since there are only two possible moves, one just has to always choose the move which does not undo the previous move, and this leads to a very easy loopless Gray code enumeration algorithm.

It is remarkable that ternary Gray codes can be generated on the same hardware as binary Gray codes (Fig. 2). In the context of generating the ternary Gray code, the Gray code string can be directly read off the disks. For example, the configuration in Fig. 2 represents the string 211102. It is D_1's turn to move, and the disk D_1 will make two steps to the left, generating the strings 211101 and 211100, and pauses there for one step, while disk D_3 moves to the right, leading to the string 211200, etc.

---

cowboy hat, and the woman was in her bikini. After all, it might have been the Towers of Hanoi that they played. Some witnesses have later reported that they had seen a disc jumping between the first and the third peg, but this has never been conclusively confirmed.
May that as it be, something unexpected happened. As the khaliph was engrossed in watching the spectacle and drew a bit closer, a small door in the wall beside him opened, which he had not noticed before. It gave onto a small garden. The moon had risen over the rooftops, and her light gave a sort-of surreal atmosphere to the whole scene. In the middle of the garden, at the corner of a fountain, a woman sang, accompanying herself on the lute. She had a beautiful voice, a bit like Mariah Carey or Adele. The calif would have listened longer, but he was quickly escorted into a house, where a maid-servant took charge of him and handed him a black gown. “Hurry up, you are late. We were waiting only for you!” The gown covered his whole stature and hid his face, and he entered a room that was barely lit by an open fire. Seven other men in black gowns were already sitting on small stools in a circle around the fire. One stool was free, and he sat down. Beside the fire, there was a small ivory model of the Towers of Bucharest, with the four largest of the $n = 6$ disks on the final peg. Disc 1 was on the middle peg, and disk 2 was on the first peg. The kaliph, having watched the game just before, understood immediately what that meant. Nowbody said a word. The tension rose. After six minutes, a lady entered and addressed them. She was the singer from the fountain. “Gentlemen. You have sworn to come to my rescue when I would be in need. Now the time has come to fulfill your oath. You see seven discs of different sizes. He who will draw the smallest disk will bring me the head of the detestable caliph Harun-al-Rashid (ca. 763–809). Should he fail to fulfill this task, the other eight will kill *him*, and we will come together and draw again.” With these words, she dropped the discs into a chalice. In silence, each man picked a disk. The kaliph was last to draw. As he opened his hand, sure enough, he found the smallest disc, disc number 1. He rose and said: “Fair lady, I will fulfill your order as I have promised. But pray tell me: by which deeds or words has the kalif enraged you so much that you wish him to

3 Gray codes with general radixes

An m-ary Gray code enumerates the n-tuples $(a_n, \dots, a_1)$ with $0 \le a_i < m$, changing a single digit at a time by ± 1. The m-ary reflected Gray code can be recursively described as follows: Let $C_1, C_2, \dots, C_{m^n}$ be the Gray code for the strings of length n. Then the strings of length $n+1$ are generated in the order

$$\begin{aligned} &C_1 0, C_1 1, C_1 2, \dots, C_1(m-2), C_1(m-1), \quad C_2(m-1), C_2(m-2), \dots, C_2 2, C_2 1, C_2 0, \\ &C_3 0, C_3 1, C_3 2, \dots, C_3(m-2), C_3(m-1), \quad C_4(m-1), C_4(m-2), \dots, C_4 2, C_4 1, C_4 0, \\ &C_5 0, C_5 1, C_5 2, \dots, C_5(m-2), C_5(m-1), \quad \dots \end{aligned} \tag{2}$$

Each digit alternates between an upward sweep from 0 to $m-1$ and a return sweep from $m-1$ to 0.

The PYTHON program in Appendix A.4 implements this recursive definition directly.

4 Generating the m-ary Gray code with odd m

For odd m, the ternary algorithm can be generalized. We need m pegs $P_0, \dots, P_{m-1}$. The leftmost peg P_0 and the rightmost peg P_{m-1} play a special role. The other pegs are called the *intermediate pegs*.

Algorithm **ODD**. Generation of the m-ary Gray code for odd m.
Initialize: Put all disks on P_0.
loop:
Move D_1 for $m-1$ steps, from P_0 to P_{m-1} or vice versa.
Let D_k be the smallest of the topmost disks on the $m-1$ pegs that don't carry D_1.
If there is no such disk, terminate.
Move D_k by one step:
If D_k is on P_0 or P_{m-1}, there is only one possible direction where to go.
Otherwise, the disk D_k continues in the same direction as in its last move.

Figure 4 shows an example with $m = 5$. This game with 5 pegs is called the Tower of Klagenfurt, after the birthplace of the senior author.[2]

In this procedure, the movement of D_1 is "externally given", whereas the movement of the other disks, whenever D_1 is at rest, is somehow "determined by the algorithm". It is not obvious that the algorithm does not violate the rules by putting a larger disk on top of D_1.

▶ **Theorem 3.** *Algorithm ODD generates the m-ary reflected Gray code defined in* (2).

Proof. It is clear from the algorithm that the last digit, which is controlled by the movement of D_1, changes in accordance with (2). We still have to show that when we discard the last digit and observe only the movement of the disks $D_2, \dots, D_n$, the algorithm produces the Gray code for the strings of length $n-1$. This is proved by induction.

By the rules of the algorithm, whenever D_1 rests, the disk that moves is D_2, unless D_2 is covered by D_1. Let us now observe the motion pattern of D_1 and D_2 that results from

[2] When Klagenfurt was founded, it was surrounded by a swamp. The swamp was inhabited by a dinosaur, the so-called *Lindworm*. The Lindworm would regularly come to the city and eat citizens. Occasionally, she would devour one of the towers of the city. The coat of arms of Klagenfurt shows the Lindworm dragon in front of the only remaining tower, see Figure 4. (Initially, there were five towers.) Over the centuries, the swamp has been drained, and the Lindworm is practically extinct.

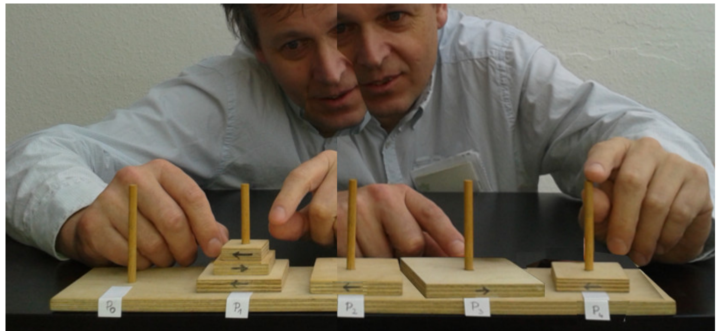


Figure 4 The Towers of Klagenfurt. This configuration represents the string 321411 over the radix $m = 5$. The next step of the Gray code moves the smallest disk D_1 onto peg P_0, changing the string to 321410. After that, disk D_2 moves from P_1 to P_2 and the next string is 321420. In the background, the two-headed Lindworm monster.

this rule. We start with D_1 on top of D_2, say, on peg P_0, with D_1 about to start its sweep. Whenever D_1 pauses for one step, D_2 will make a step towards P_{m-1}. After D_2 reaches P_{m-1}, it turns out that, because m is odd, D_1 will make its next sweep from P_0 to P_{m-1}, resting on top of D_2. Now, since D_2 is covered, it will be one of the *other* disks $D_3, D_4, \ldots$ that will move. Then the same routine repeats in the other direction.

If we now ignore D_1 and look only at the motions of the other disks, the following pattern emerges: D_2 makes $m-1$ steps from one end to the other, and then the smallest disk that is not covered by D_2 makes its move, according to the rules. This is precisely the same procedure as Algorithm ODD, with D_2 taking the role of the externally controlled disk. By induction, this algorithm correctly produces the Gray code for the strings of length $n-1$, and it does not put a larger disk on top of D_2. Since the larger disks are moved only when D_2 lies under D_1, it follows that a larger disk is never moved on top of D_1 either. ◀

One can actually apply one induction step of the proof in the opposite direction, introducing an additional "control disk" D_0 which does not have a digit associated with it. Its only role is to alternately cover P_0 and P_{m-1} and exclude the covered peg for the selection of the disk D_k that should be moved. The algorithm becomes simpler because it does not have to treat D_1 separately from the other disks. The program in [8, Appendix A.3] applies this idea to the algorithm of Section 6 below.

5 Generating the m-ary Gray code with even m

For even m, we generalize Algorithm HANOI, which solves the case $m = 2$. We use $m + 1$ pegs $P_0, \ldots, P_m$, which we arrange in a cyclic clockwise order. We stipulate that disks D_i with odd i move only clockwise, and disks with even i move only counterclockwise.

Algorithm **EVEN**. Generation of the m-ary Gray code for even m.
Initialize: Put all disks on P_0.
loop:
  Move D_1 for $m-1$ steps, in clockwise direction.
  Let D_k be the smallest of the topmost disks on the m pegs that don't carry D_1.
  If there is no such disk, terminate.
  Move D_k by one step, in the direction determined by the parity of k.

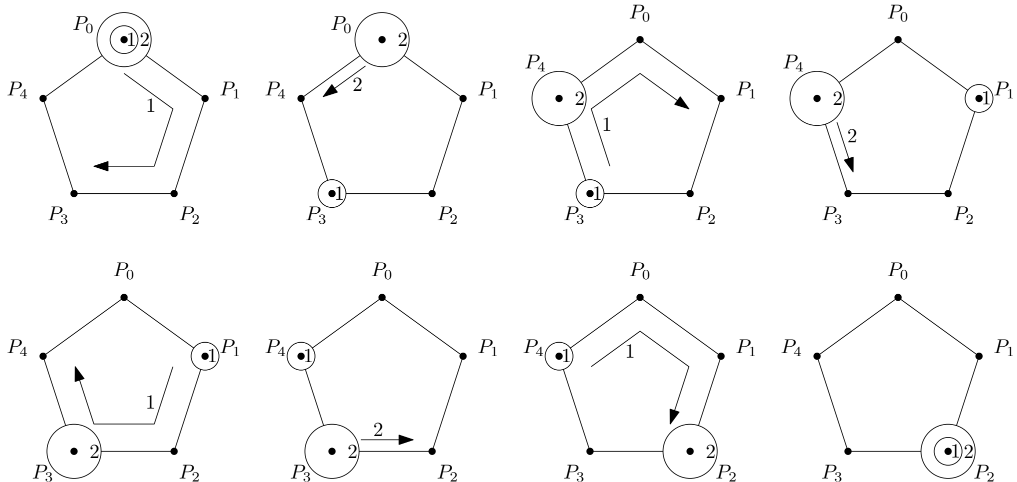


Figure 5 One period of movement of the two smallest disks D_1 and D_2 when Algorithm EVEN generates all tuples over an alphabet of size $m = 4$ using $m + 1 = 5$ pegs.

The Gray code is determined by changing the digit a_i whenever disk D_i is moved. The digit a_i runs through the sequence $0, 1, 2, \ldots, m-2, m-1, m-2, \ldots, 2, 1, 0, 1, 2, \ldots$. Thus it changes always by ± 1, but we have to remember whether it is on the increasing or the decreasing part of the cycle. The position of disk D_i is no longer directly correlated with the digit a_i; thus the digits a_i have to be maintained separately, in addition to the disks on the pegs. It is far from straightforward to relate the disk configuration to the Gray code.

For example, when carrying out the algorithm for $m = 4$, the configuration in Figure 4 appears when the string is 211030. Disk D_1 has just made three steps and is going to rest for one step. The next step moves D_3 clockwise, since 3 is odd, and the string is changed to 211130. After that, D_1 resumes its clockwise motion, and the string changes into 211131.

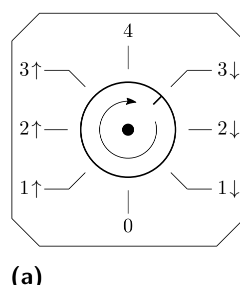
▶ **Theorem 4.** *Algorithm EVEN generates the m-ary reflected Gray code defined in* (2).

Proof. This follows along the same lines as Theorem 3. When we look at the pattern of motion of D_1 and D_2, we observe again that D_2 makes $m - 1$ steps until it is covered by D_1, see Fig. 5: After the first move of D_2, the clockwise cyclic distance from D_1 to D_2 is 1, and with each move of D_2, this distance increases by 1. Thus, after $m - 1$ moves, the distance becomes $m - 1$, and D_1 will land on top of D_2 with its next sweep. ◀

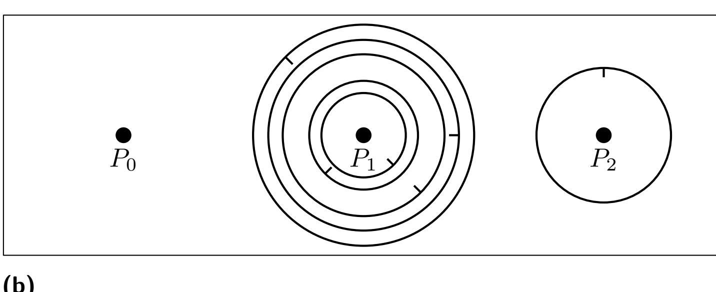
Except for $m = 3$ and $m = 2$, Algorithms ODD and EVEN do not generate a shortest sequence of moves to the target configuration, even if moves are allowed only between adjacent pegs (or cyclically adjacent pegs, in a direction depending on the disk parity). We could not come up with some natural constraints under which our algorithms give a shortest solution.

6 The Towers of Bucharest++

In Algorithm ODD, the intermediate pegs $P_1, \ldots, P_{m-2}$ will always be available for selecting the smallest disk D_k to be moved. Thus, one can coalesce these pegs into one peg, keeping only the two extreme pegs separate. With three pegs, we can use the same hardware as the Tower of Bucharest, but we have to record the value of the digits, since they are no longer expressed by the position. A simple method is to provide the disks with *marks* that indicate



(a)



(b)

Figure 6 (a) The upgraded disk of the Towers of Bucharest++ and the meaning of its positions, for $m = 5$. (b) The situation of Figure 4, compressed to 3 pegs.

the value as well as the direction of movement, which we have to remember anyway. Each disk cycles through $2m - 2$ values, potentially augmented with direction information:

$$0,\ 1\!\uparrow,\ 2\!\uparrow,\ \dots,\ (m-2)\!\uparrow,\ m-1,\ (m-2)\!\downarrow,\ \dots 2\!\downarrow,\ 1\!\downarrow,\ 0,\ 1\!\uparrow,\ \dots \tag{3}$$

We can encode this information like a dial with $2m - 2$ equally spaced directions, as shown in Fig. 6a. A disk whose mark shows 0 is always on the left peg P_0. A disk whose mark shows $m - 1$ is always on the right peg P_2. Otherwise, it is on the middle peg P_1. When we say we *turn a disk*, this means that we turn it clockwise to the next dial position, and if necessary, move it to the appropriate peg.

Algorithm **ODD-COMPRESSED**. Generation of the m-ary Gray code for odd m.
Initialize: Put all disks on P_0, and turn them to show 0.
loop:
Turn disk D_1 $m - 1$ times until it arrives at one of the extreme pegs P_0 or P_2.
Let D_k be the smaller of the topmost disks on the two pegs not covered by D_1.
If there is no such disk, terminate.
Turn D_k once.

The digits a_i can be read off from the dial positions. Correctness follows by comparison with Algorithm ODD, checking that the transition between successive states is preserved when merging the intermediate pegs into one peg. ◀

This algorithm can now even be generalized to mixed-radix Gray codes for the n-tuples $(a_n, \dots, a_1)$ with $0 \le a_i < m_i$, for some sequence of radixes $m_i \ge 2$, provided that all m_i are odd.

7 Simulation

All our algorithms can be easily simulated in software on a digital computer.[3] A stack will do for each peg. If there are k pegs, the algorithm takes $O(k)$ time to compute the next move and accordingly produce the next element of the Gray code sequence. If m is constant, then $k = m$ or $k = m + 1$ in Algorithms ODD and EVEN, and these algorithms can pass as loopless algorithms. If k is large, Algorithm ODD can be replaced by ODD-COMPRESSED, which has only 3 pegs, independent of m.

[3] Nowadays, most households will more readily have access to a computer than to a tower of Hanoi.

To make a truly loopless algorithm out of Algorithm EVEN, at the expense of an increased overhead, we can use the following easy fact, which follows directly from the algorithm statement.

▶ **Lemma 5.** *In the algorithms EVEN, ODD, and ODD-COMPRESSED, when a disk D_k is moved, all smaller disks $D_1, \ldots, D_{k-1}$ are on the same peg.*

To get a loopless implementation, the set of disks on a peg has to be maintained as a sequence of maximal intervals of successive integers, instead of storing them as a stack in the usual way. Then, whenever D_1 is at rest, the disk D_k to be moved can be determined in constant time as the smallest missing disk on the peg containing D_1.

8 Working ahead

While we are at the topic of Gray codes, we might as well mention another approach for loopless generation of Gray codes, which results from a generally applicable technique for converting amortized bounds into worst-case bounds. We start from the observation that was already mentioned in connection with the delta-sequence in Section 1.4:

▶ **Proposition 6.** *Consider the enumeration of the n-tuples $(b_n, \ldots, b_1)$ with $0 \le b_i < m_i$ in* lexicographic order. *If, between two successive tuples of the sequence, the j rightmost digits are changed, then, at the corresponding transition in the Gray code, the j-th digit from the right is changed.* ◀

We can thus find the position j that has to be changed in the Gray code by lexicographically "incrementing" n-tuples $(b_n, \ldots, b_1)$ in a straightforward way:

Algorithm **DELTA**. Generation of the delta-sequence for the Gray code.
  Initialize: $(b_n, \ldots, b_2, b_1) := (0, 0, \ldots, 0, 0)$
  $Q :=$ an empty list
  loop:
    $j := 1$
    while $b_j = m_j - 1$:
      $b_j := 0$
      $j := j + 1$
      if $j = n + 1$: TERMINATE
    $b_j := b_j + 1$
    $Q.append(j)$

The delta sequence is stored in Q. It is known that the *average* number of loop iterations for producing an entry of Q is less than 2. We use this fact to coordinate the *production* of entries Q by Algorithm DELTA with their *consumption* in the Gray code generation, turning Q into a buffer of bounded capacity. This leads to the following loopless algorithm:

Algorithm **WORK-AHEAD**.
Generation of the Gray code.

```
procedure STEP:
  if b_j = m_j − 1:
    b_j := 0
    j := j + 1
  else:
    if Q is not filled to capacity:
      b_j := b_j + 1
      Q.append(j)
      j := 1
```

```
(a_n, ..., a_2, a_1) := (0, ..., 0, 0)
(d_n, ..., d_2, d_1) := (1, ..., 1, 1)
(b_{n+1}, b_n, ..., b_2, b_1) := (0, 0, ..., 0, 0); m_{n+1} := 2
Q := queue of capacity B := ⌈n/2⌉, initially empty
j := 1
loop:
    visit the n-tuple (a_n, ..., a_2, a_1)
    STEP
    STEP
    remove j from Q
    if j = n + 1: TERMINATE
    a_j := a_j + d_j
    if a_j = 0 or a_j = m_j − 1: d_j := −d_j
```

The procedure STEP on the left side encompasses one loop iteration of Algorithm DELTA. By programmer's license, we have moved the initialization $j := 1$ of the loop variable to the end of the previous loop. We have also moved the termination test $j = n + 1$ to the side of the consumer. Accordingly, we had to extend the n-tuple b into an $(n + 1)$-tuple, setting m_{n+1} arbitrarily to 2. When Q is full, nothing is done in the procedure STEP, and the repeated call of STEP will try to insert the same value into Q. Thus, apart from the termination test, a repeated execution of STEP will faithfully carry out Algorithm DELTA.

The Gray code algorithm on the right couples two production STEPs with one consumption step, which takes out an entry j of Q and carries out the update $a_j := a_j \pm 1$. Every digit a_j must cycle up and down through its values in the sequence (3), and thus, we have to remember the direction $d_j = \pm 1$ in which it moves, as in Algorithm ODD.

To show that the algorithm is correct, we have to ensure that the queue Q is never empty when the algorithm retrieves an element from it. This is proved below in Lemma 7.

The clean way to terminate the algorithm would be to stop inserting elements into Q as soon as $j = n + 1$ is *produced* in STEP, as in Algorithm DELTA. Instead, termination is triggered when the value $j = n + 1$ is *removed* from Q. Due to this delayed termination test, a few more iterations of STEP can be carried out, but they cause no harm.

For the *binary* Gray code ($m_i = 2$ for all $i = 1, \dots, n$), the algorithm can be simplified. With a slightly larger buffer Q of size $B' := \max\{\lceil \frac{n+1}{2} \rceil, 2\}$, the test whether Q is filled to capacity can be omitted, see Lemma 9 below. The reason is that the average number of production STEPs per item approaches 2 in the limit, and accordingly, the queue automatically does not grow beyond the minimum necessary size. The directions d_i are of course also superfluous in the binary case.

The idea of "working ahead" is opposite to the approach of delaying work as long as possible that underlies many "lazy" data structures and also lazy evaluation in some functional programming languages. In a similar vein, Guibas, McCreight, Plass, and Janet R. Roberts [7] have obtained worst-case bounds of $O(\log k)$ for updating a sorted linear list at distance k from the beginning. Their algorithm works ahead to hedge against sudden bursts of activity. Our setting is much simpler, because we do not depend on the update requests of a "user" and we can plan everything in advance.

At a different level of complexity, the idea of working ahead occurs in an algorithm of Wettstein [14, Section 6]. This trick, credited to Emo Welzl, is used to achieve *polynomial delay* between successive solutions when enumerating non-crossing perfect matching of a planar point set, despite having to build up a network with exponential space in a preprocessing phase.

8.1 An alternative STEP procedure

As an alternative to the organization of Algorithm WORK-AHEAD, we can incorporate the termination test into the STEP procedure:

procedure STEP′:
  if $j = n + 1$: TERMINATE
  if $b_j = m_j - 1$:
    $b_j := 0$
    $j := j + 1$
  else:
    if Q is not filled to capacity:
      $b_j := b_j + 1$
      $Q.append(j)$
      $j := 1$

With this modified procedure STEP′, the termination test in the main part of Algorithm WORK-AHEAD can of course be omitted. We also need not extend the arrays b and m to $n + 1$ elements.

The algorithm still works correctly because there are no unused entries in the queue when STEP′ signals termination. Let us prove this:

The termination signal is sent instead of producing the value $j = \bar{\rho}(k) = n + 1$ for $k = m_0 m_1 \dots m_{n-1}$. Generating this signal takes $n + 1$ iterations of STEP. In this time, no new values are added to the queue. Let us assume that the production of $\bar{\rho}(k)$ was started during iteration k_0, and the buffer was filled with $B_0 \le B$ entries at that time. The first of these entries is consumed at the end of iteration k_0, and all B_0 entries of the buffer have been used up at the beginning of iteration $k_0 + B_0$. By this time, at most $2B_0 \le 2B \le n + 1$ iterations of STEP were carried out and contributed to the production of the termination signal. It follows that when STEP discovers that $j = n + 1$, no unused entries are in the stack, and it is safe to terminate the program.

It is important not to "speed up" the program by moving the termination test into the **if**-branch after the statement $j := j + 1$. Also, we must use exactly the prescribed buffer size for Q. Therefore, this variation is incompatible with the simplification for the binary case mentioned above.

8.2 Correctness proofs for the work-ahead algorithms

We define the ruler function ρ and the modified ruler function $\bar{\rho}$ with respect to a sequence of radixes $m_1, \dots, m_n$ as follows:

$$\rho(k) := \max\{\, i : 0 \le i \le n,\ m_1 m_2 \dots m_i \text{ divides } k \,\}, \qquad \bar{\rho}(k) := \rho(k) + 1$$

Then the k-th value that is entered into Q is $\bar{\rho}(k)$, and for computing this value, Algorithm DELTA needs $\bar{\rho}(k)$ iterations, and accordingly, Algorithm WORK-AHEAD needs $\bar{\rho}(k)$ STEPs.

▶ **Lemma 7.** *In Algorithm WORK-AHEAD, the buffer Q never becomes empty.*

Proof. We number the iterations of the main loop as $1, 2, \dots, m_1 m_2 \dots m_n$. In the last iteration, the algorithm terminates.

Let us show that the queue Q is not empty in iteration k. We distinguish two cases.

(i) Up to and including iteration k, two repetitions of STEP were always completed.

(ii) Some repetitions of STEP had no effect because the buffer Q was full.
In case (i), production of all values $\rho(i)$ for $i = 1, \dots, k$ requires

$$S(k) := \sum_{i=1}^{k} \bar{\rho}(i)$$

calls to STEP. To show that these calls are completed by the time when $\bar{\rho}(k)$ is needed, we have to show

$$S(k) \le 2k. \tag{4}$$

In case (ii), let k_0 be the last iteration when an execution of STEP was "skipped". This means that the queue Q was filled to capacity B just before removing the value $j = \bar{\rho}(k_0)$, and it contained the values $\bar{\rho}(k_0), \bar{\rho}(k_0 + 1), \dots, \bar{\rho}(k_0 + B - 1)$. Since then, STEP was called $2(k - k_0)$ times, and $\bar{\rho}(k)$ is ready when it is needed, provided that

$$1 + \sum_{i=k_0+B+1}^{k} \bar{\rho}(i) \le 2(k - k_0)$$

whenever $k \ge k_0 + B$. The left-hand side of this inequality is the number of necessary STEPs for computing the values up to $\bar{\rho}(k)$. Computing $\bar{\rho}(k_0 + B)$ takes just one more STEP, since the STEP that would have stored this value in Q was abandoned in iteration k_0. Setting $k' = k_0 + B$, we can express the inequality equivalently as

$$S(k) - S(k') \le 2(k - k' + B) - 1 \text{ for } k' \le k \tag{5}$$

We can write an explicit formula for $S(k)$:

$$S(k) = k + \left\lfloor \frac{k}{m_1} \right\rfloor + \left\lfloor \frac{k}{m_1 m_2} \right\rfloor + \dots + \left\lfloor \frac{k}{m_1 m_2 \dots m_n} \right\rfloor .$$

Since all $m_i \ge 2$, we get $S(k) \le k + k/2 + k/4 + k/8 + \dots + k/2^n < 2k$, proving (4). For the other bound (5), we use the relation $\lfloor x \rfloor - \lfloor x' \rfloor < x - x'$ and get

$$S(k) - S(k') < (k - k') + (k - k') \cdot (\tfrac{1}{2} + \tfrac{1}{4} + \tfrac{1}{8} + \dots + \tfrac{1}{2^n}) + n < 2(k - k') + n.$$

Since the left-hand side is an integer, we obtain $S(k) - S(k') \le 2(k - k') + n - 1$ and this implies (5) since the buffer size $B := \lceil \frac{n}{2} \rceil$ satisfies $2B \ge n$. ◀

In Algorithm WORK-AHEAD, the STEPs should generate entries $\bar{\rho}(1), \bar{\rho}(2), \dots$ of Q up to $\bar{\rho}(N)$, where $N := m_1 m_2 \dots m_n$.

The following lemma shows that production of the STEPs may overrun their target by at most one. Since the algorithm has already made provisions to generate $\bar{\rho}(N) = n + 1$ by extending the arrays b and m to size $n + 1$ instead of n, this one extra entry does not cause any harm.

▶ **Lemma 8.** *In Algorithm WORK-AHEAD, the last entry that is added to Q is $\bar{\rho}(N)$ or $\bar{\rho}(N + 1)$.*

Proof. The production of $\bar{\rho}(N) = n + 1$ takes $n + 1 \ge 2B$ STEPs. It follows that the buffer Q is empty when $\bar{\rho}(N) = n + 1$ is inserted, regardless of whether the production of $\bar{\rho}(N)$ is started in the first or second STEP of an iteration.

If the production of $\bar{\rho}(N) = n + 1$ is completed in the second STEP of an iteration, it is thus immediately consumed, which leads to termination. If $\bar{\rho}(N)$ is completed in the first STEP of an iteration, the second STEP will produce the value $\bar{\rho}(N + 1) = 1$, but then the algorithm will terminate as well. ◀

Finally, we prove the simplification of the algorithm for the binary case.

▶ **Lemma 9.** *In the binary version of Algorithm WORK-AHEAD, i.e., when $m_i = 2$ for all $i = 1, \ldots, n$, the buffer Q automatically never gets more than $B' := \max\{\lceil \frac{n+1}{2} \rceil, 2\}$ entries, even if the test in STEP whether the buffer is full is omitted.*

Proof. Let us assume for contradiction that the buffer becomes overfull in iteration k, $1 \le k \le 2^n$. This means that, before $j = \bar{\rho}(k)$ is removed from Q, the $2k$ STEP operations have produced more than $k - 1 + B'$ values. But this is impossible, since, as we will show, the production of the first $k_1 = k + B'$ values would have taken

$$S(k_1) = k_1 + \left\lfloor \frac{k_1}{2} \right\rfloor + \left\lfloor \frac{k_1}{2^2} \right\rfloor + \cdots + \left\lfloor \frac{k_1}{2^n} \right\rfloor > 2k$$

STEPs. To show the last inequality, we first consider the case $k_1 < 2^n$. We apply the inequality $\lfloor x \rfloor > x - 1$ and obtain $S(k_1) > 2k_1 - k_1/2^n - n$, and since $k_1/2^n < 1$ and $S(k_1)$ is an integer, we get

$$S(k_1) \ge 2k_1 - n = 2k + 2B' - n > 2k.$$

Let us now see at what time $\bar{\rho}(k_1)$ for $k_1 \ge 2^n$ is entered into Q. When $k_1 = 2^n$, no round-off takes place in the formula for $S(k_1)$, and we have $S(2^n) = 2 \cdot 2^n - 1$. This shows that the production of $\bar{\rho}(2^n)$ is completed in the first STEP of iteration 2^n. In the second STEP of this iteration, $\bar{\rho}(2^n + 1) = 1$ is added to Q. Thus, when $\bar{\rho}(2^n)$ is about to be retrieved, the buffer contains $2 \le B'$ elements. Then the algorithm terminates, and no more elements are produced. ◀

9 Concluding Remarks

By our approach of modeling the Gray code in terms of a motion-planning game, we were able get a mixed-radix Gray code only when all radixes m_i are odd. It remains to find a model that would work for different even radixes or even for radixes of mixed parity.

Another motion-planning game which is related to the binary Gray code is the *Chinese Rings* puzzle, see Gardner [4], Knuth [10, pp. 285–286], or Scorer, Grundy, and Smith [13]. (Knuth [10, Solution to Ex. 7.2.1.1–(10), p. 679] gives a brief survey of the early literature, mentioning references that date back as far as the 16th century.) The goal is to detach a series of interlocked rings from a bar. Like the Towers of Bucharest, the Chinese rings allow at most two possible moves in every state. Each move removes or replaces a single ring. By simulating the Chinese rings directly, one can therefore obtain another loopless algorithm for the binary Gray code, see Misra [11], Knuth [10, Solution to Ex. 7.2.1.1–(12b)]. However, this algorithm does not seem to extend to other radixes. (Scorer et al. [13, Section 5] analyzed a generalization of the Chinese Rings. We did not check whether it leads to interesting Gray codes.)

Acknowledgements. We thank Don Knuth for leading us to the reference [7] about the work-ahead approach, and we thank Sandi Klavžar for pointing us to the earlier references [5] and [6] that connect the Towers of Bucharest to ternary Gray codes.

References

1 James R. Bitner, Gideon Ehrlich, and Edward M. Reingold. Efficient generation of the binary reflected Gray code and its applications. *Commun. ACM*, 19(9):517–521, 1976.

2 Peter Buneman and Leon Levy. The towers of Hanoi problem. *Information Processing Letters*, 10(4–5):243–244, 1980.
3 Gideon Ehrlich. Loopless algorithms for generating permutations, combinations, and other combinatorial configurations. *J. Assoc. Comput. Mach.*, 20(3):500–513, July 1973.
4 Martin Gardner. The curious properties of the Gray code and how it can be used to solve puzzles. *Sci. American*, 227:106–109, 1972.
5 Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. Addison-Wesley, 1989.
6 Dah-Jyh Guan. Generalized Gray codes with applications. *Proc. Natl. Sci. Council, Republic of China (A)*, 22(6):841–848, 1998.
7 Leo J. Guibas, Edward M. McCreight, Michael F. Plass, and Janet R. Roberts. A new representation for linear lists. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, STOC'77, pages 49–60, New York, NY, USA, 1977. ACM.
8 Felix Herter and Günter Rote. Loopless Gray code enumeration and the Tower of Bucharest. Preprint arXiv 1604.06707 [cs.DM], April 2016.
9 Andreas M. Hinz, Sandi Klavžar, Uroš Milutinović, and Ciril Petr. *The Tower of Hanoi – Myths and Maths*. Birkhäuser, 2013.
10 Donald E. Knuth. *Combinatorial Algorithms, Part 1*, volume 4A of *The Art of Computer Programming*. Addison-Wesley, 2011.
11 Jayadev Misra. Remark on Algorithm 246. *ACM Trans. Math. Software*, 1(3):285, 1975.
12 Amir Sapir. The towers of Hanoi with forbidden moves. *The Computer Journal*, 47(1):20–24, 2004.
13 R. S. Scorer, P. M. Grundy, and C. A. B. Smith. Some binary games. *The Mathematical Gazette*, 28(280):96–103, 1944.
14 Manuel Wettstein. Counting and enumerating crossing-free geometric graphs. Preprint arXiv 1604.05350 [cs.CG], April 2016.

A Appendix: PYTHON simulations of some algorithms

These programs run equally with Python 2.7 and Python 3. The pegs, the string a, and the array of `directions` are kept as global variables. See [8] for the complete set of programs.

A.1 Algorithm ODD, Section 4

```
def initialize_m_ary_odd(m,n):
    global pegs, a, direction
    pegs = tuple([] for k in range(m))
    for i in range(n,0,-1):
       pegs[0].append(i)
    a = (n+1)*[0] # a[0] and direction[0] is wasted
    direction = (n+1)*[+1]

def Gray_code_m_ary_odd(m):
    "generate n-tuple of entries from the set {0,1,...,m-1}"
    visit()
    while True:
       for _ in range(m-1): # repeat m-1 times:
           move_disk(m, peg=a[1])
           visit()
```

```
        k=find_smallest_disk(exclude=a[1]) # smallest disk not covered by D1
        if k==None: return
        move_disk(m,peg=k)
        visit()

def move_disk(m,peg): # move topmost disk on peg
    disk = pegs[peg].pop()
    peg     += direction[disk]
    a[disk] += direction[disk]
    pegs[peg].append(disk)
    if peg==m-1: direction[disk] = -1
    elif peg==0: direction[disk] = +1

def visit(): # print the current string and the contents of the pegs
    print ("".join(str(x) for x in reversed(a[1:])) + " "+
           " ".join("P{}:".format(k)+",".join(map(str,p))
                                 for k,p in enumerate(pegs)))

def find_smallest_disk(exclude=None):
    list_d_k = [(p[-1],k) for k,p in enumerate(pegs) if k!=exclude and p]
    if list_d_k:
        _,k = min(list_d_k) # smallest disk not covered by D1
        return k
    return None

initialize_m_ary_odd(m=3,n=6) # run the program for a test
Gray_code_m_ary_odd(m=3)
```

A.2 Algorithm EVEN, Section 5

```
def initialize_m_ary_even(m,n):
    initialize_m_ary_odd(m+1,n) # use m+1 pegs

def Gray_code_m_ary_even(m):
    peg_disk1 = 0 # position of disk D1
    visit()
    while True:
        for _ in range(m-1): # repeat m-1 times:
            turn_disk(m,peg=peg_disk1)
            peg_disk1 = (peg_disk1+1) % (m+1)
            visit()
        k = find_smallest_disk(exclude=peg_disk1)
             # smallest disk not covered by D1
        if k==None: return
        turn_disk(m,peg=k)
        visit()

def turn_disk(m,peg): # move the topmost disk on peg
    disk = pegs[peg].pop()
```

```
    if disk%2==1:
        peg = (peg + 1) % (m+1)
    else:
        peg = (peg - 1) % (m+1)
    pegs[peg].append(disk)
    a[disk] += direction[disk]
    if a[disk]==m-1: direction[disk] = -1
    elif a[disk]==0: direction[disk] = +1

initialize_m_ary_even(m=4,n=6) # run the program for a test
Gray_code_m_ary_even(m=4)
```

A.3 Algorithm WORK-AHEAD, Section 8

```
def STEP():
    global j, b,m,B,Q
    if b[j]==m[j]-1:
        b[j]=0
        j += 1
    else:
        if len(Q)<B:
            b[j] += 1
            Q.append(j)
            j = 1

def initialize_work_ahead(n):
    global a,b,direction,B,Q,j
    a = (n+1)*[0] # a[0], b[0], and direction[0] is wasted
    direction = (n+1)*[+1]
    b = (n+2)*[0]
    from collections import deque
    B = (n+1)//2
    Q = deque()
    j = 1

def Gray_code_work_ahead(n):
    while True:
        VISIT()
        STEP()
        STEP()
        j = Q.popleft()
        if j==n+1: break
        a[j] += direction[j]
        if a[j] in (0,m[j]-1): direction[j] *= -1

def VISIT():
    print ("".join(str(x) for x in reversed(a[1:])))

# run the program for a test:
```

```
n=4
m=[0]+[2,4,5,2]+[2] # initial 0 and final 2 are artificial
initialize_work_ahead(n)
Gray_code_work_ahead(n)
```

A.4 General mixed-radix Gray code generation according to the recursive definition of Section 3

In order to have a reference implementation for comparing the results, we give a program straight from the definition (2) of Section 3, extended to arbitrary mixed radices $(m_1, \dots, m_n)$.

```
def mixed_Gray_code(ms):
    "a generator for the mixed-radix Gray code"
    if ms:
        m = ms[0] # radix for the least significant digit
        G1 = mixed_Gray_code(ms[1:])
        while True:
            g = next(G1)
            for lastdigit in range(m):
                yield g+(lastdigit,)
            g = next(G1)
            for lastdigit in reversed(range(m)):
                yield g+(lastdigit,)
    else:
        yield () # produce one element: the empty list

for g in mixed_Gray_code([2,4,5,2]): # run the program for a test
    print ("".join(str(x) for x in g))
```

Convex Configurations on Nana-kin-san Puzzle*

Takashi Horiyama[1], Ryuhei Uehara[2], and Haruo Hosoya[3]

1 **Saitama University, Japan**
2 **Japan Advanced Institute of Science and Technology, Japan**
3 **Ochanomizu University, Japan**

Abstract

We investigate a silhouette puzzle that is recently developed based on the golden ratio. Traditional silhouette puzzles are based on a simple tile. For example, the tangram is based on isosceles right triangles; that is, each of seven pieces is formed by gluing some identical isosceles right triangles. Using the property, we can analyze it by hand, that is, without computer. On the other hand, if each piece has no special property, it is quite hard even using computer since we have to handle real numbers without numerical errors during computation. The new silhouette puzzle is between them; each of seven pieces is not based on integer length and right angles, but based on golden ratio, which admits us to represent these seven pieces in some nontrivial way. Based on the property, we develop an algorithm to handle the puzzle, and our algorithm succeeded to enumerate all convex shapes that can be made by the puzzle pieces. It is known that the tangram and another classic silhouette puzzle known as Sei-shonagon chie no ita can form 13 and 16 convex shapes, respectively. The new puzzle, Nana-kin-san puzzle, admits to form 62 different convex shapes.

1998 ACM Subject Classification G.2.1 Counting problems

Keywords and phrases silhouette puzzles, nana-kin-san puzzle, enumeration algorithm, convex polygon

Digital Object Identifier 10.4230/LIPIcs.FUN.2016.20

1 Introduction

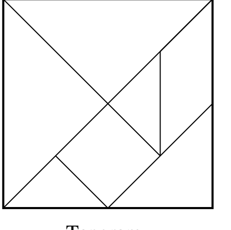
A *silhouette puzzle* is a game where, given a certain set of polygons, one must decide whether all of them can be placed in the plane in such a way that their union is a target figure or letter[1]. Rotation and reflection are allowed but scaling and overlapping are not. Formally, a set of polygons S can *form* a polygon P if there is an isomorphism up to rotation and reflection between a partition of P and the polygons of S (i.e. a bijection $f(\cdot)$ from a partition of P to S such that x and $f(x)$ are congruent for all x).

The *tangram* is a set of polygons consisting of a square of material cut by straight incisions into different-sized pieces. See the left diagram in Figure 1. Of anonymous origin, their first known reference in literature is from 1813 in China [7]. The tangram has grown to be extremely popular throughout the world; now, over 2000 dissection and related puzzles exist for it ([7, 3]). Less famous is a quite similar Japanese puzzle called *Sei-shonagon Chie no Ita*

* This work was partially supported by JSPS KAKENHI Grant Number 26330009, 15K00008 and MEXT Kakenhi Grant Number 24106004, 24106007.

[1] This is also called a *dissection puzzle* in some context. However, dissection puzzle rather asks how to cut a given polygon into a few pieces so that they can be rearranged to the target polygon. The most famous one is called the haberdasher's problem created by Henry Dudeney [1], which asks to transform a square into a regular triangle by cutting into four pieces.

© Takashi Horiyama, Ryuhei Uehara, and Haruo Hosoya;
licensed under Creative Commons License CC-BY
8th International Conference on Fun with Algorithms (FUN 2016).
Editors: Erik D. Demaine and Fabrizio Grandoni; Article No. 20; pp. 20:1–20:14
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



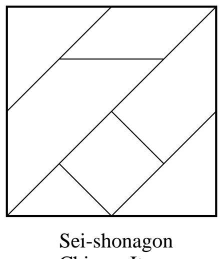


Figure 1 (Left) Tangram and (Right) Sei-shonagon Chie no Ita.

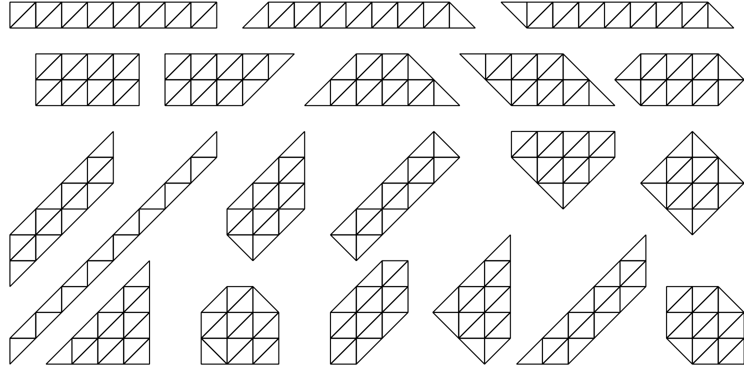


Figure 2 All 20 potential convex polygons by tangram type puzzles.

(the right diagram in Figure 1). Sei-shonagon was a courtier and famous novelist in Japan, but there is no evidence that the puzzle existed a millennium ago when she was living. *Chie no ita* means *wisdom plates*, which refers to this type of physical puzzle. This puzzle is said to be named after Sei-shonagon's wisdom. Historically, it first appeared in literature in 1742, bit older than the tangram [7].

Wang and Hsiung considered the number of possible convex (filled) polygons formed by the tangram [9]. They first noted that, given sixteen identical isosceles right triangles, one can create the tangram pieces by gluing some edges together. (In technical word in puzzle society, each piece is a *polyabolo*.) So, clearly, the set of convex polygons one can create from the tangram is a subset of those that sixteen identical isosceles right triangles can form. Embedded in the proof of their main theorem, Wang and Hsiung [9] demonstrate that sixteen identical isosceles right triangles can form exactly 20 convex polygons. These 20 polygons are illustrated in Figure 2. The tangram can realize thirteen of those 20. Since the same idea works for the Sei-shonagon Chie no Ita, it is quite natural to ask how many of these twenty convex polygons the Sei-shonagon Chie no Ita pieces can form. In [2], Fox-Epstein, Katsumata and Uehara showed that (1) Sei-shonagon Chie no Ita achieves sixteen convex polygons out of twenty, (2) there are four sets of seven convex polygons in this manner that can form nineteen convex polygons out of twenty (Figure 3), and we cannot improve any more in this context.

For the tangram and the Sei-shonagon Chie no Ita, the key idea is that each piece consists of congruent right isosceles triangles. On the other hand, even if the pieces are in quite simple forms, the silhouette puzzle can be intractable when it has many pieces. In [4], they prove that this problem is NP-complete for general n pieces even if each piece is a rectangle of size $1 \times x_i$ for some integer x_i except only one polygon with 6 vertices. We remark that,

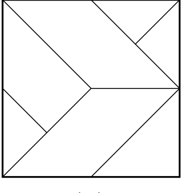
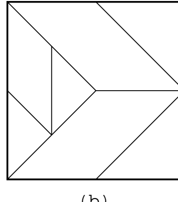
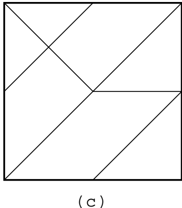
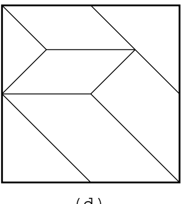

(a)

(b)

(c)

(d)

Figure 3 Four patterns that can form nineteen convex polygons.

the goal in their paper is to form "line symmetric shape," that is, the target shape itself is not explicitly given. However, even if the goal rectangle is explicitly given, the proof in [4] still works and we obtain NP-completeness in our framework with the same set of pieces. In fact, in our framework of silhouette puzzle, we can further improve their result to the set of only rectangles by splitting the last polygon with 6 vertices into three rectangles. In this case, all pieces and the goal shape are just rectangles.

Another interesting problem is ETS polygons discussed in [5] and [6]: Given an integer n, we determine whether a convex n-gon can be obtained by gluing equilateral triangles and squares in an edge-to-edge manner. In [5], this problem is treated as a dissection problem, and it is shown that n can only be an integer from 3 to 12. In [6], all possible sets of exterior angles for n-gons are listed, and for each of the sets, an edge-to-edge glued shape is given as an example.

As seen the fact that there are over 2,000 dissection for the tangram [7, 3], when we consider arbitrary polygons, even a set of seven pieces seems to be intractable since there are essentially infinitely many polygons that can form from the seven pieces of the tangram. (We remark that the tangram itself contains seven convex pieces.) Therefore, it is reasonable to consider as the framework of a silhouette puzzle with the following two assumptions; (1) each piece is convex and (2) target polygon is also convex. Even under this assumption, we still have many variants of problems; in fact, even with only two pieces, the following theorem is mentioned by Uematsu [8]:

▶ **Theorem 1** ([8]). *For any given positive integer $n > 2$, we have a set of two convex polygons that can form $2n$ different convex polygons.*

Proof. (Sketch) We here show for general case $n > 4$ since $n = 3$ and $n = 4$ are special cases. Let P_n be a regular n-gon. Then we first construct the first piece P'_n by bending P_n little a bit to satisfy the following conditions; each edge has length 1, and every angle is distinct. The second piece T is a shallow triangle; its base edge is of length 1, and two other edges are of length $1/2 + \epsilon_1$ and $1/2 - \epsilon_2$ with $0 < \epsilon_1 - \epsilon_2$. By setting ϵ_1 and ϵ_2 sufficiently small, attaching T at the base edge to P'_n, we have n different convex polygons, and by flipping T, we also have the other n different convex polygons. ◀

We note that P and its mirror image P^R are regarded as the same shape. We also generalize this idea and obtain exponential lower bound for the number of pieces:

▶ **Theorem 2.** *For any given positive integer $n > 4$, we have a set of $n + 1$ convex polygons of $4n$ vertices in total that can form $2^n \cdot n!$ different convex polygons.*

Proof. Let P'_n be the n-gon used in the proof of Theorem 1, which is obtained by bending a regular n-gon little a bit without changing the length of each edge. We construct a set S of n shallow triangles constructed in a similar manner in the proof of Theorem 1. We make all

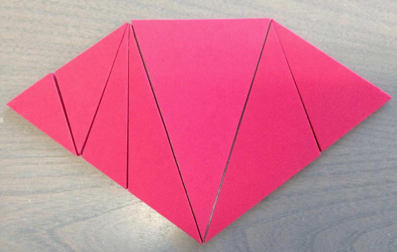


Figure 4 Package (left) and seven pieces (right) of Nana-kin-san puzzle.

triangles in S distinct from each other. In this time, we can attach these n triangles at each of n edges of P', and each different way produces a distinct convex shape. We count them up. Once we fix an edge of P' as a special edge, the ordering of the triangles makes $n!$ ways. In each ordering, each triangle has two ways to attach it to P' by flipping. Therefore, in total, we have $2^n \cdot n!$ different convex polygons from this set of $n+1$ pieces. ◀

From the viewpoint of the design of algorithms, such detailed real numbers in Theorems 1 and 2 are not welcome since we have to take care of numerical errors in computation. On the other hand, the tangram and the Sei-shonagon Chie no Ita are rather too simple to use computer since they are based on the unit tile of identical isosceles right triangles. (In fact, most results in [2] are obtained without using computer.)

Recently, one of the authors produces a new silhouette puzzle named "Nana-kin-san puzzle[2]." This puzzle is designed based on the golden ratio triangles (Figure 4). As the same as the tangram and Sei-shonagon Chie no Ita, it consists of seven pieces. However, each piece is a triangle based on the golden ratio, and hence these edges are not of integer lengths.

From the property of the golden ratio, this puzzle has beautiful features from the viewpoint of both of lengths and angles. That is, as shown in Figure 5, most edge lengths can be described quite simple form using golden ratio $\varphi = \frac{\sqrt{5}+1}{2} = 1.68\ldots$ that satisfies $\varphi^2 - \varphi - 1 = 0$. Moreover, each of all angles of these triangles can be represented by a multiple of 18°. These facts are useful from the viewpoint of design of algorithms. Namely, each angle equal to $18k°$ for some k can be represented by just positive integer k, and each edge can be represented by a simple equation $a_0 + a_1\varphi + a_2\alpha$ for some natural numbers a_0, a_1, a_2 in $\{0, 1, 2\}$. That is, all computations can be done over integer operations, which mean that we do not take care of numerical errors in computation.

In a sense, the Nana-kin-san puzzle is a puzzle between the one based on a unit tile like polyominoes, polyabolos, and so on, and the general puzzle shown in Theorems 1 and 2. At a glance, since all pieces are "similar" triangles, this puzzle might seem to be simpler than the other silhouette puzzles like the tangram and the Sei-shonagon Chie no Ita that consist of more variant shapes. However, this is not the case. In this paper, we propose a simple algorithm that generates all convex shapes that can be formed by the Nana-kin-san puzzle. Comparing to the similar puzzles based on polyabolos (the tangram can form 13, and the Sei-shonagon Chie no Ita can form 16, and its theoretical upper bound is 19 in that framework), it is surprisingly many. The Nana-kin-san puzzle can form 62 different convex shapes.

[2] In Japanese, "nana," "kin," and "san" mean "seven," "golden," and "three," respectively. That is, this name indicates seven golden-ratio triangles.

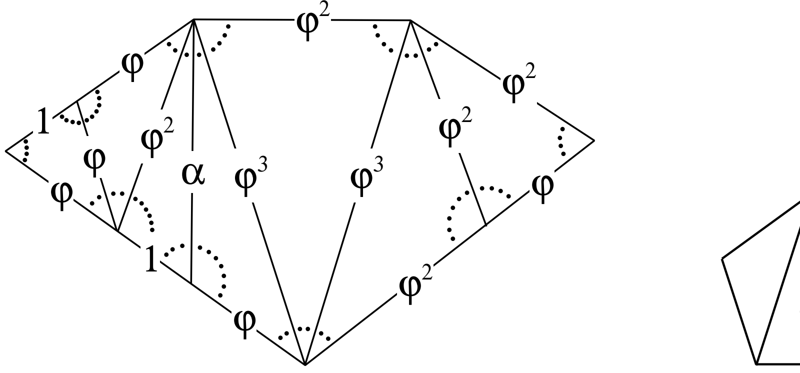


Figure 5 Lengths and angles of Nana-kin-san puzzle.

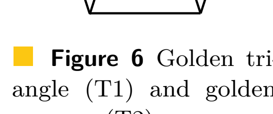


Figure 6 Golden triangle (T1) and golden gnomon (T2).

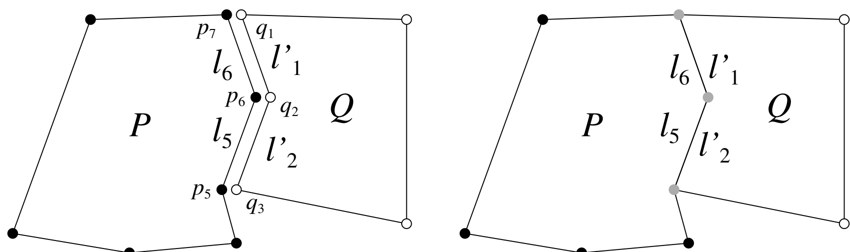


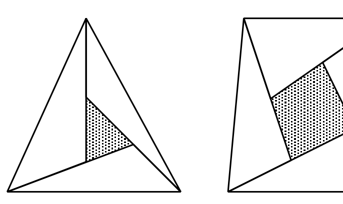
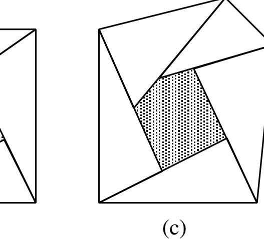
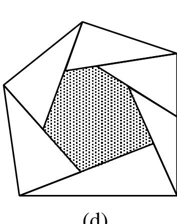
Figure 7 Edge-to-edge gluing of P and Q at (p_5, p_6, p_7) and (q_1, q_2, q_3).

2 Nana-kin-san puzzle and its property

The details of the Nana-kin-san puzzle is described in Figure 5. In the figure, each black dot indicates 18°. That is, each angle can be represented by $18k°$ for some positive integer k. In the figure, the unit length is described by 1, and the other lengths are described by φ, φ^2, φ^3, and α. Here, φ means the golden ratio $1.618\ldots$ that satisfies $1+\varphi = \varphi^2$, and $\alpha = \sqrt{5+2\sqrt{5}}$. We here remark that for the golden ratio, we have $\varphi^2 = 1+\varphi$ and $\varphi^3 = 1+2\varphi$. In general, for any positive integer $k > 1$, φ^k can be represented by a linear expression $a_0 + a_1\varphi$ for some positive integers a_0 and a_1 uniquely. We here remark that the Nana-kin-san puzzle is designed based on two triangles made from a regular pentagon; we can obtain two isosceles triangles from a regular pentagon (Figure 6). We call these triangles *golden triangle* (T1) and *golden gnomon* (T2), respectively.

For two polygons P and Q, we introduce an *edge-to-edge gluing* of P and Q as follows. Let P be a polygon with n vertices $p_0, p_1, \ldots, p_{n-1}$ in counterclockwise order[3], and Q a polygon with m vertices $q_0, q_1, \ldots, q_{m-1}$. Let ℓ_i be the length of the edge (p_i, p_{i+1}), and ℓ'_j the length of the edge (q_j, q_{j+1}). Then we can edge-to-edge glue P and Q at some paths $(p_i, p_{i+1}, \ldots, p_{i+k})$ and $(q_j, q_{j+1}, \ldots, q_{j+k})$ if and only if for each $0 \le h \le k$, (1) $\ell_{i+h} = \ell'_{j+k-h-1}$, (2) at each pair of vertices p_{i+h} and q_{j+k-h}, the summation of two angles at these two vertices makes 360°, (3) at pair of vertices p_i and q_{j+k}, the summation of two angles at these two vertices makes less than 360°, and (4) at pair of vertices p_{i+k} and q_j,

[3] In this paper, all indices are computed $\bmod n$, where n is the number of vertices of the polygon.

(a) (b) (c) (d)

Figure 8 Not pseudo-guillotine cut separable cases.

the summation of two angles at these two vertices makes less than 360°. See Figure 7 for a simple example of an edge-to-edge gluing of P and Q at (p_5, p_6, p_7) and (q_1, q_2, q_3).

Let assume that a polygon P can be formed by two polygons P_1 and P_2 by an edge-to-edge gluing at two paths on P_1 and P_2. Then we call this shared path in P *pseudo-guillotine cut* of P. Then we have the following useful property of the Nana-kin-san puzzle:

▶ **Theorem 3.** *In the Nana-kin-san puzzle, every convex polygon can be obtained by repeating edge-to-edge gluing.*

Proof. Let P be any convex polygon formed by the Nana-kin-san puzzle. To derive a contradiction, we assume that P cannot be split into two pieces by any pseudo-guillotine cut. As similar in the ordinary guillotine cut, if P cannot be split into two pieces by any pseudo-guillotine cut, P should be surrounded by a series of triangles as shown in Figure 8 (some triangles can be composed by two or more pieces which can be split by pseudo-guillotine cut).

In the figure, each gray area is a hole surrounded by three, four, five, and six triangles in (a), (b), (c) and (d), respectively. Here, each hole should be filled by a set of triangles. Hence, the cases (c) and (d) are already impossible since we have only seven triangles in the Nana-kin-san puzzle contains seven pieces. Therefore, only considerable cases are (a) and (b). It is not difficult to check that each case cannot be achieved by using the seven pieces of the Nana-kin-san puzzle. ◀

By Theorem 3, we can say that it is sufficient to check the repetition of edge-to-edge gluing of the Nana-kin-san puzzle to generate all convex shapes.

3 Algorithm

In this section, we describe the details of our algorithm for enumerating all convex polygons using seven pieces of the Nana-kin-san puzzle.

3.1 Data structure

We design a special data structure for this problem, which has applications to the other problems with similar properties. Let P be a polygon with n vertices $p_0, p_1, \ldots, p_{n-1}$ in counterclockwise order. Hereafter, we assume that P is made from some pieces of the Nana-kin-san puzzle. Then P is described by two linked lists $(\ell_0, \ell_1, \ldots, \ell_{n-1})$ and $(d_0, d_1, \ldots, d_{n-1})$, where ℓ_i is the length of the edge $e_i = (p_i, p_{i+1})$ and d_i is the inner angle at the vertex p_i. We here note that each d_i takes a value of $(18 \times k)°$ for some positive integer k. Therefore, each d_i needs to store this integer k. Moreover, each ℓ_i can be represented by a 3-tuple $(\ell_{i,0}, \ell_{i,1}, \ell_{i,2})$ such that $\ell_i = \ell_{i,0} + \ell_{i,1} \times \varphi + \ell_{i,2} \times \alpha$. By the property of the golden ratio,

we can confirm that any length ℓ_i of P can be represented in this form for some positive integers $\ell_{i,0}, \ell_{i,1}$, and $\ell_{i,2}$, and these positive integers are uniquely determined.

For a polygon P, we denote by P^R its mirror image. From the viewpoint of the representation, for a polygon P represented by $p_0, p_1, \ldots, p_{n-1}$, its mirror image P^R can be represented by $p_{n-1}, \ldots, p_1, p_0$. Based on this representation, we define a *canonical form* of P as follows. First, we fix some vertex as p_0. Then we can obtain the corresponding string of integers $(d_0, d_1, \ldots, d_{n-1}, \ell_0, \ell_1, \ldots, \ell_{n-1})$ (precisely, each d_i is represented by an integer k_i with $d_i = (18 \times k_i)°$, and each ℓ_i is represented by a sequence of three integers $\ell_{i,0}, \ell_{i,1}, \ell_{i,2}$). For each vertex of P, we can compute the corresponding string of integers. Among them, we employ the lexicographically first one as the canonical representation of this P. It is not difficult to see that any two polygons P and P', P is congruent with P' if and only if their canonical representations are the same string. We maintain each polygon P by its canonical representation. (Note that P and P^R have the different canonical representation in general.)

3.2 Algorithm description

Now we describe the algorithm we use to check all convex polygons made from the seven pieces of the Nana-kin-san puzzle. Based on Theorem 3, if we have a convex polygon P made from some pieces of the Nana-kin-san puzzle, P always can be cut into two convex polygons by one pseudo-guillotine cut. Therefore, we can apply inductive construction of convex polygons made from these seven pieces. First, we initialize the set S_0 of polygons by the seven pieces $\{P_0, P_1, \ldots, P_6\}$ of the Nana-kin-san puzzle. In general, we keep the set of shapes P such that each of them is made from some pieces of S_0. That is, P consists of the shape described by its canonical representation, and the subset of S that forms P. That is, in the first set S_0, each piece P_i (with $0 \le i \le 6$) has its canonical representation and it is associated with the set $\{P_i\}$. In general step, we grow the set S_0 and add convex polygons that can be formed by some pieces of the Nana-kin-san puzzle. We denote this general set by S which starts from S_0.

In general step, we pick up two polygons P and P' from S such that they do not share any common piece in S_0. Then we glue P to P' in all possible ways, obtain new polygons P'', and add them into S as follows (we also apply the same algorithm for P^R and P'):

Step 1: For each i and j, pick up e_i from P and e_j from P'.

Step 2: If $\ell_i \neq \ell_j$, we do nothing for this pair. If $\ell_i = \ell_j$, we construct a new polygon by gluing e_i to e_j as follows. The new polygon is described by two linked lists $(\ell'_{j+1}, \ell'_{j+2}, \ldots, \ell'_{j-1}, \ell_{i+1}, \ell_{i+2}, \ldots, \ell_{i-1})$ and $(d_i+d'_{j+1}, d'_{j+2}, \ldots, d'_{j-1}, d_{i+1}+d'_j, d_{i+2}, \ldots, d_{i-1})$. Let the resulting polygon P'' be represented by $(\ell''_0, \ell''_1, \ldots)$ and $(d''_0, d''_1, \ldots)$.

Step 3: We here search the vertex p_k of degree 360° and remove it. Let we have $(\ldots, d''_{k-1}, 20, d''_{k+1}, \ldots)$ (we remind that each d_i represents $(18 \times d_i)°$). Then, intuitively, the edges e_{k-1} and e_k are overlapping with sharing the point p_k. If $\ell_{k-1} \neq \ell_k$, we conclude that this gluing is fault since P'' is not convex. Otherwise, the list $(\ldots, d''_{k-1}, 20, d''_{k+1}, \ldots)$ is replaced by $(\ldots, d''_{k-1}+d''_{k+1}, \ldots)$ and $(\ldots, \ell''_{k-2}, \ell''_{k-1}, \ell''_k, \ell''_{k+1}, \ldots)$ is replaced by $(\ldots, \ell''_{k-2}, \ell''_{k+1}, \ldots)$.

Step 4: We here search the vertex p_k of degree 180° and remove it. Let we have $(\ldots, d''_{k-1}, 10, d''_{k+1}, \ldots)$. Then, intuitively, the edges e_{k-1} and e_k are on the same line with sharing the point p_k. Then the list $(\ldots, d''_{k-1}, 10, d''_{k+1}, \ldots)$ is replaced by $(\ldots, d''_{k-1}, d''_{k+1}, \ldots)$ and $(\ldots, \ell''_{k-2}, \ell''_{k-1}, \ell''_k, \ell''_{k+1}, \ldots)$ is replaced by $(\ldots, \ell''_{k-2}, \ell''_{k-1} + \ell''_k, \ell''_{k+1}, \ldots)$.

Step 5: If d''_k is greater than 20 for some k (i.e., the inner angle at vertex p_k is greater than 360), forget P'' and go to Step 1.

Step 6: Add the canonical form of P'' into S, and go to Step 1.

3.3 Correctness of the algorithm

We here show the correctness of the algorithm. If the algorithm outputs a convex polygon P that uses seven pieces of the Nana-kin-san puzzle, it is easy to see that we can construct it. Therefore, it is sufficient to show that all possible convex polygons are enumerated by the algorithm. By Theorem 3, any convex polygon P that uses all pieces can be divided into two polygons by pseudo-guillotine cut. Then, for each (not necessarily convex) polygon, using Theorem 3 repeatedly, we finally obtain the set S_0 of the seven pieces of the Nana-kin-san puzzle. Therefore, the algorithm is correct and we obtain all possible convex polygons made from the Nana-kin-san puzzle.

4 Results

In this section, we show the results of the algorithm. In our experiment, we use the following system: Intel Core i7-3770K (3.50GHz), 32 GB Memory. The computation time is 675 seconds, and the memory consumption is 15 MB. We obtain 563 possible convex polygons that can be formed by a subset of the seven pieces of the Nana-kin-san puzzle. Among them, the number of the convex polygons that can be formed by the all of the seven pieces of the Nana-kin-san puzzle is 62. They consist with 3 tetragon, 24 pentagons, 29 hexagons and 6 heptagons. Their shapes are shown in Figure 9. Since this is a silhouette puzzle, we just only show the all possible convex polygons without cutting lines. Their solutions are given in Appendix. The whole 563 convex polygons can be found at `http://www.al.ics.saitama-u.ac.jp/horiyama/research/puzzle/7kin3_puzzle/`.

We also applied the algorithm to the tangram and the Sei-shonagon Chie no Ita. We obtain 93 and 100 possible convex polygons that can be formed by a subset of the seven pieces of the tangram and the Sei-shonagon Chie no Ita, respectively. Among them, the number of the convex polygons that can be formed by the all of the seven pieces of the tangram and the Sei-shonagon Chie no Ita, respectively, is 13 and 16. The 62 out of 563 possible convex polygons of the Nana-kin-san puzzle suggests the rich potential of that puzzle. On the computation time, interestingly, they have a significant difference : 65 seconds for the tangram, and 40,920 seconds for the Sei-shonagon Chie no Ita.

References

1. Henry Ernest Dudeney. *The Canterbury Puzzles.* Dover, 1958.
2. Eli Fox-Epstein, Kazuho Katsumata, and Ryuhei Uehara. The Convex Configurations of "Sei Shonagon Chie no Ita," Tangram, and Other Silhouette Puzzles with Seven Pieces. *IEICE Trans. on Inf. and Sys.*, accepted, 2016.
3. Martin Gardner. *Origami, Eleusis, and the Soma Cube.* The New Martin Gardner Mathematical Library. Cambridge, 2008.
4. Jason S. Ku, Erik D. Demaine, Matias Korman, Joseph Mitchell, Yota Otachi, Marcel Roeloffzen, Ryuhei Uehara, Yushi Uno, and Andre van Renssen. Symmetric Assembly Puzzles are Hard, Beyond a Few Pieces. In *The 18th Japan Conference on Discrete and Computational Geometry and Graphs (JCDCGG 2015)*, 2015.
5. Joseph Malkevitch. Problem 707, solution by M. Goldberg. *Math. Mag.*, 42:158, 1969.
6. Joseph Malkevitch. Tiling Convex Polygons with Equilateral Triangles and Squares. *Annals of the New York Academy of Science*, 440:299–303, 1985.
7. Jerry Slocum and Jacob Botermans. *The Tangram Book: The Story of the Chinese Puzzle with Over 2000 Puzzles to Solve.* Sterling Publishing, 2004.
8. M. Uematsu. Personal communication. 2015.
9. Fu Traing Wang and Chuan-Chih Hsiung. A Theorem on the Tangram. *The American Mathematical Monthly*, 49(9):596–599, 1942.

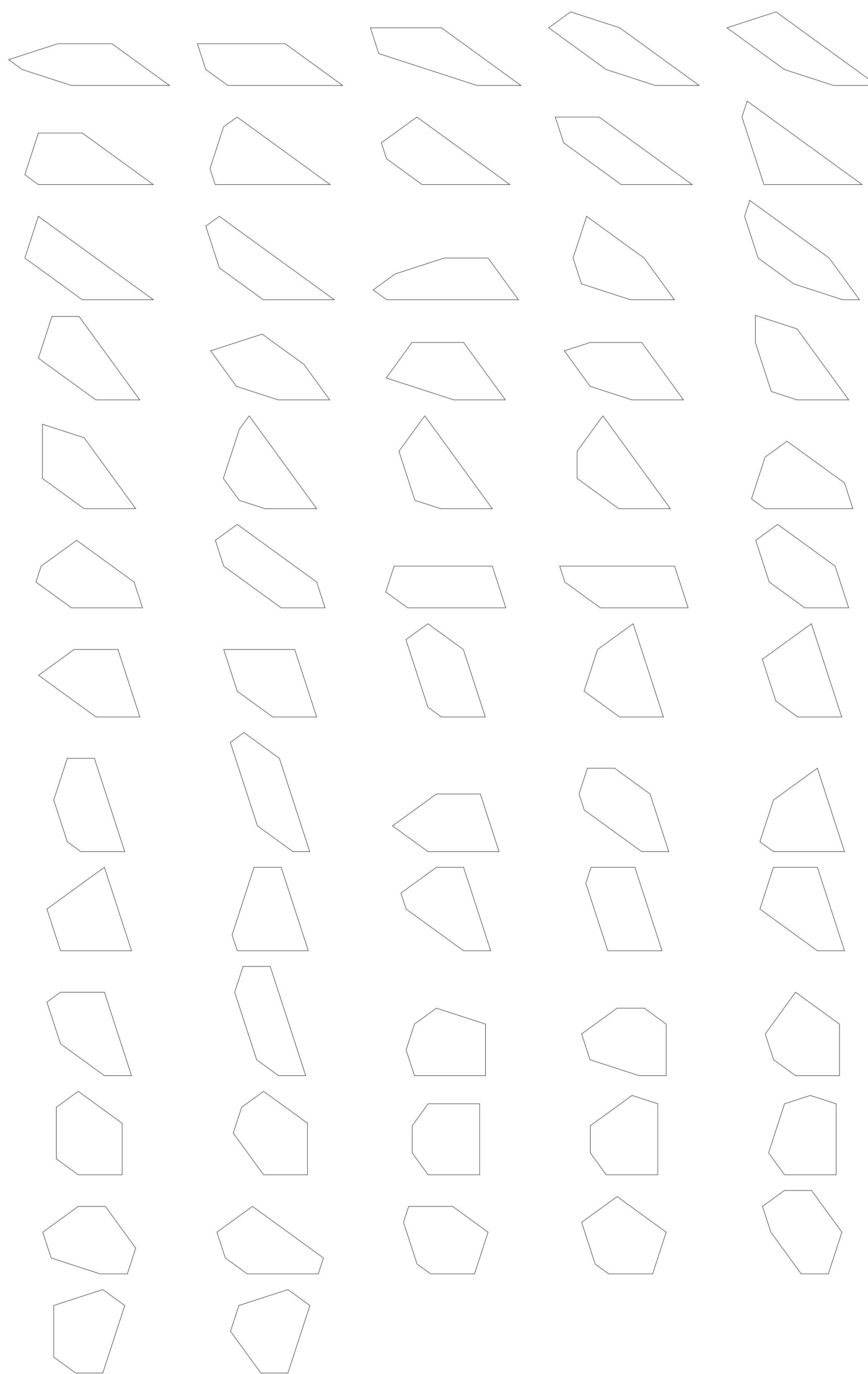
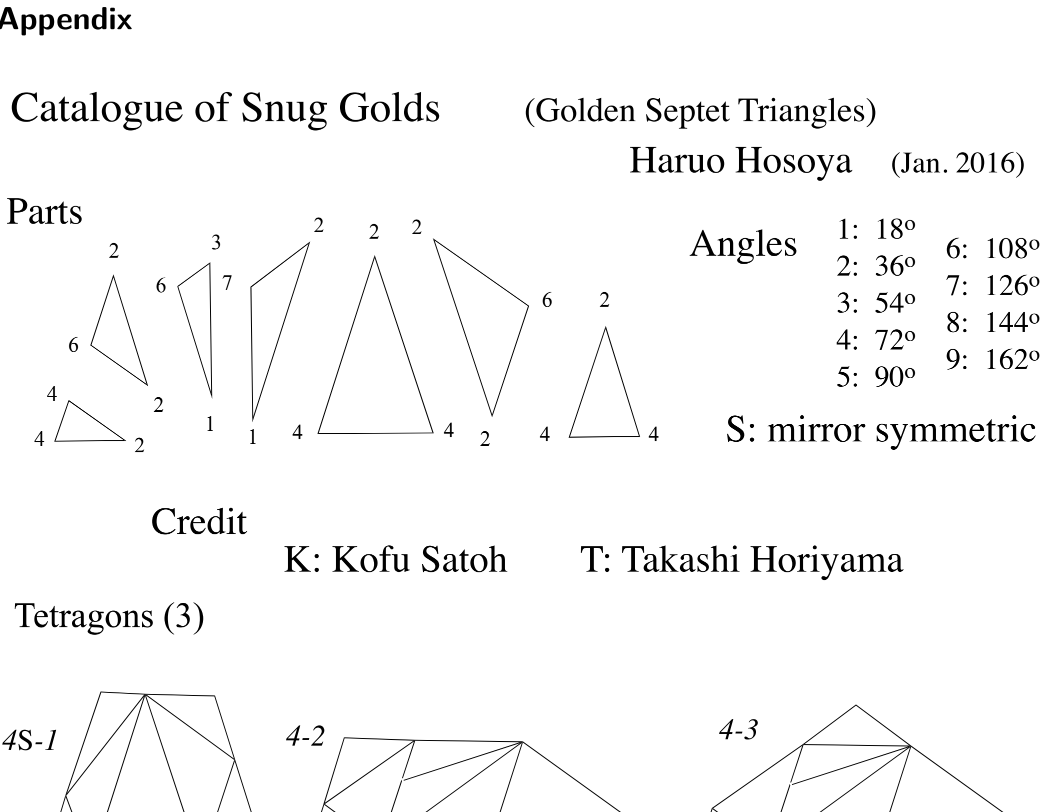
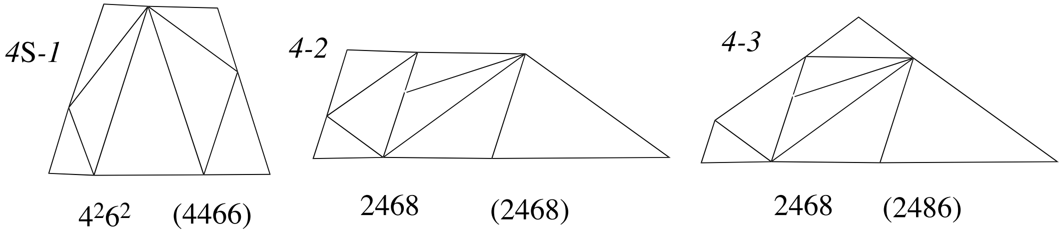


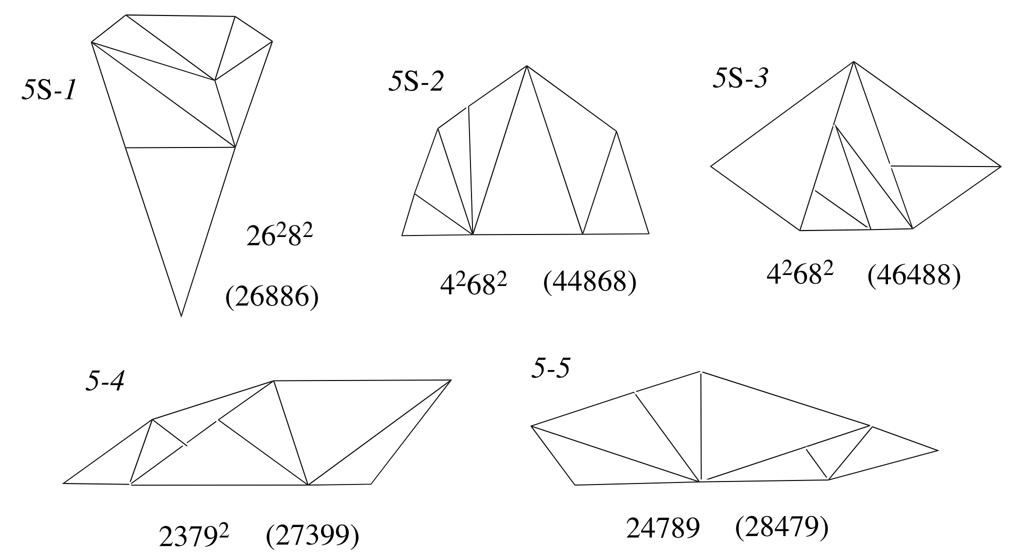
Figure 9 62 convex polygons formed by 7 pieces of the Nana-kin-san puzzle.

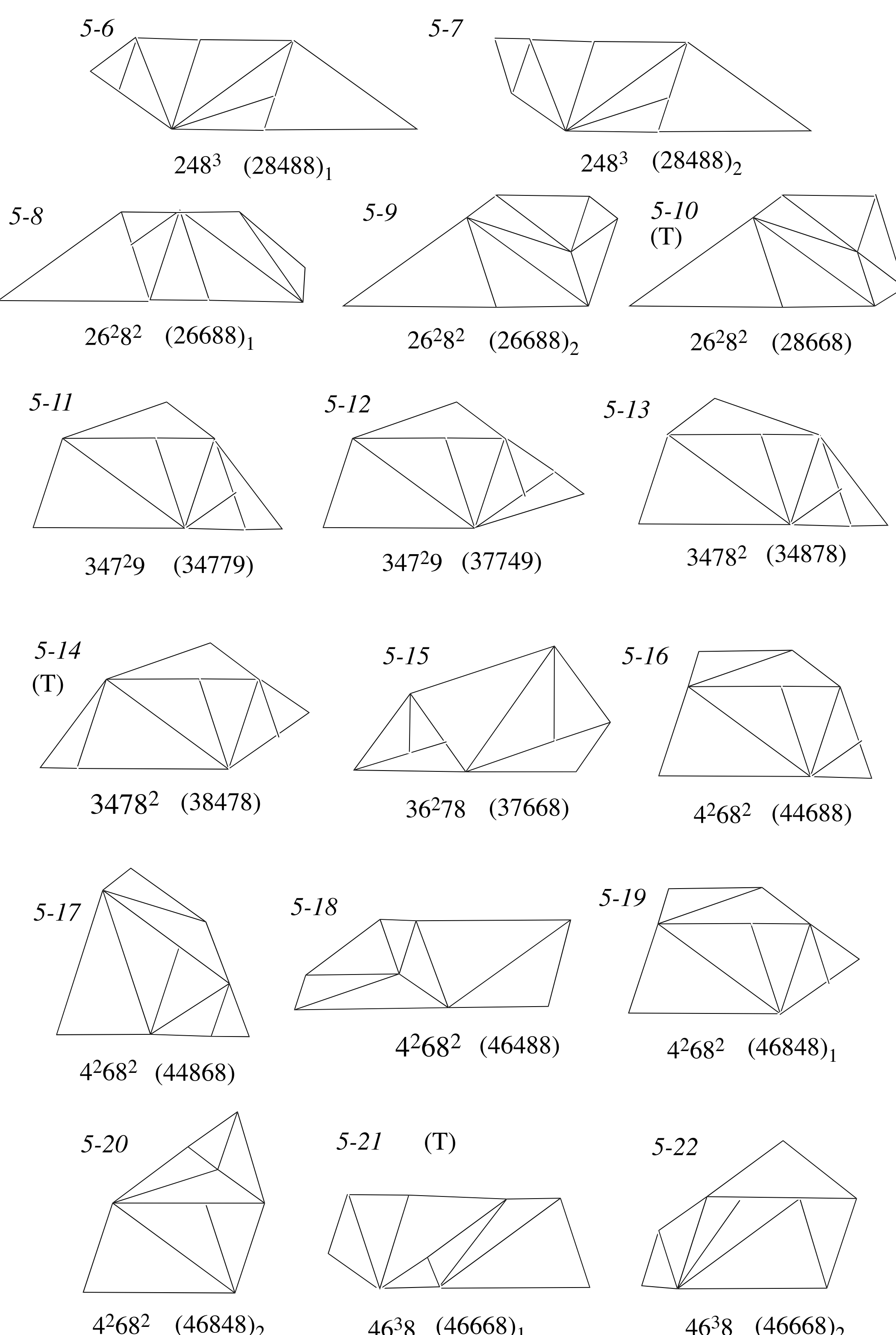
Appendix

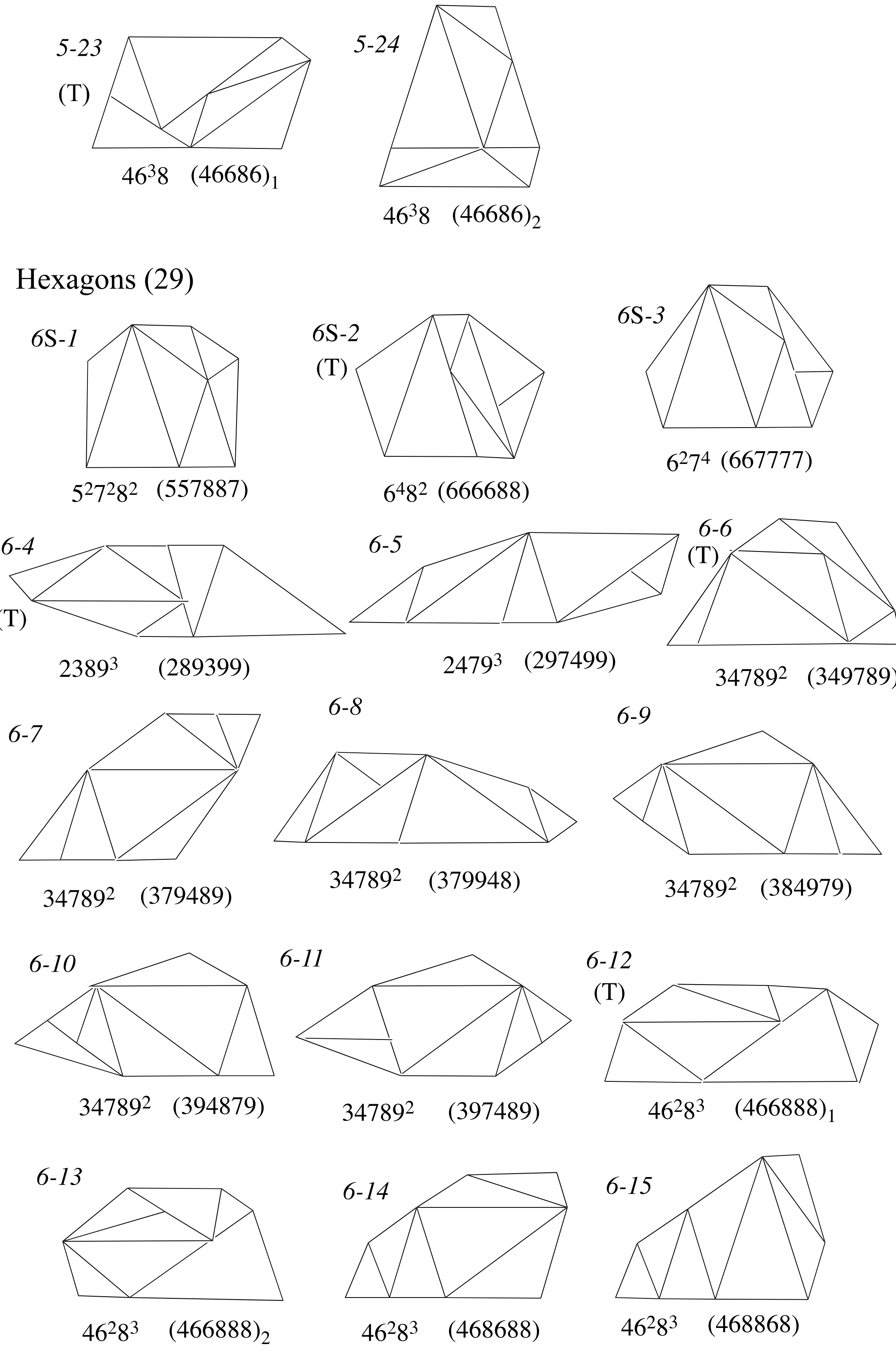


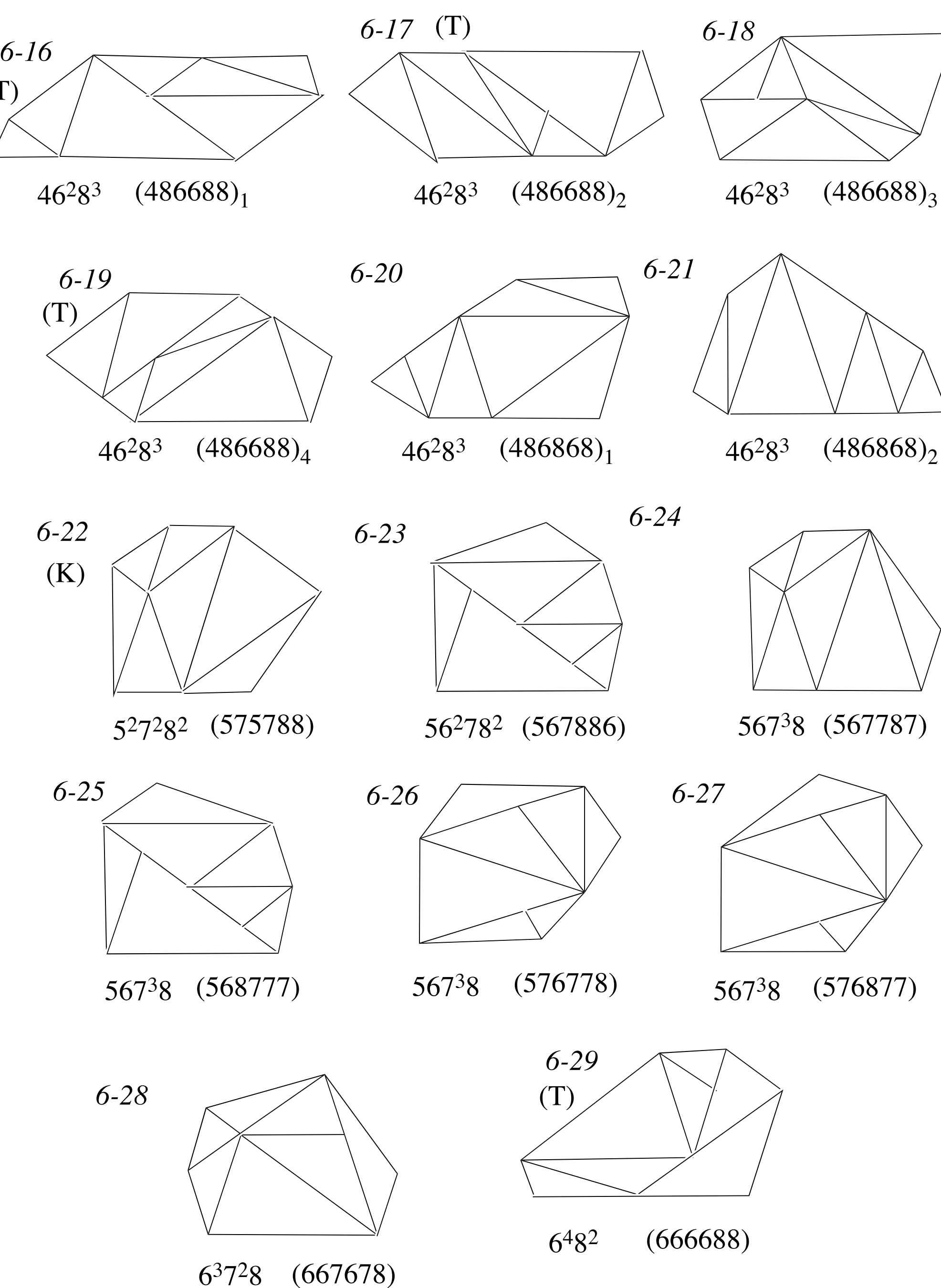


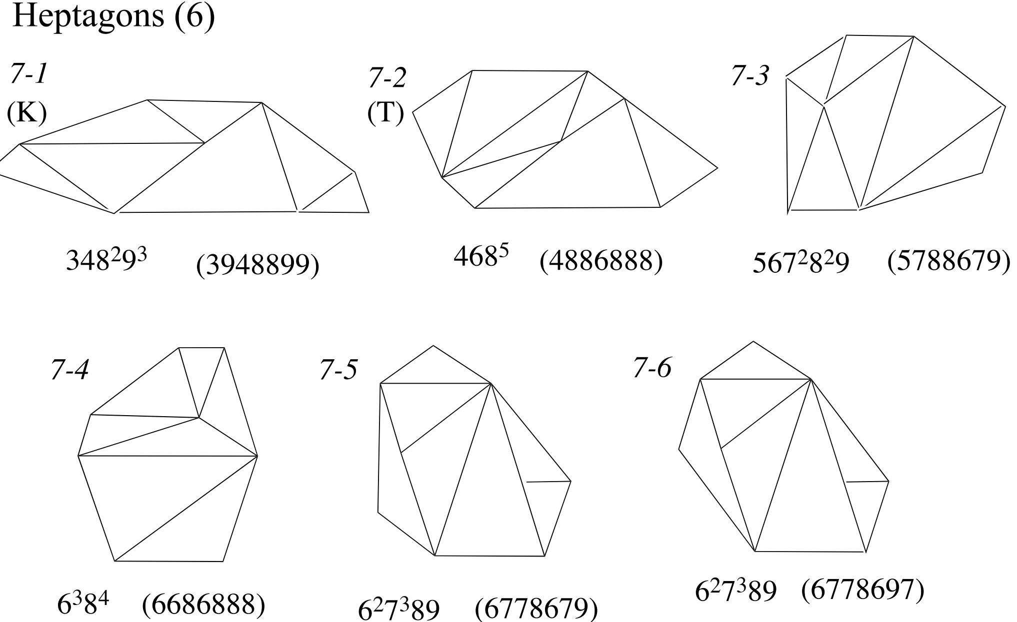
Pentagons (24)




5-6
248^3 $(28488)_1$
5-7
248^3 $(28488)_2$
5-8
26^28^2 $(26688)_1$
5-9
26^28^2 $(26688)_2$
5-10
(T)
26^28^2 (28668)
5-11
347^29 (34779)
5-12
347^29 (37749)
5-13
3478^2 (34878)
5-14
(T)
3478^2 (38478)
5-15
36^278 (37668)
5-16
4^268^2 (44688)
5-17
4^268^2 (44868)
5-18
4^268^2 (46488)
5-19
4^268^2 $(46848)_1$
5-20
4^268^2 $(46848)_2$
5-21 (T)
46^38 $(46668)_1$
5-22
46^38 $(46668)_2$


5-23
(T)
$46^{3}8$ $(46686)_1$
5-24
$46^{3}8$ $(46686)_2$
Hexagons (29)
6S-1
$5^{2}7^{2}8^{2}$ (557887)
6S-2
(T)
$6^{4}8^{2}$ (666688)
6S-3
$6^{2}7^{4}$ (667777)
6-4
(T)
2389^{3} (289399)
6-5
2479^{3} (297499)
6-6
(T)
34789^{2} (349789)
6-7
34789^{2} (379489)
6-8
34789^{2} (379948)
6-9
34789^{2} (384979)
6-10
34789^{2} (394879)
6-11
34789^{2} (397489)
6-12
(T)
$46^{2}8^{3}$ $(466888)_1$
6-13
$46^{2}8^{3}$ $(466888)_2$
6-14
$46^{2}8^{3}$ (468688)
6-15
$46^{2}8^{3}$ (468868)


6-16
(T)
$46^2 8^3$ $(486688)_1$
6-17 (T)
$46^2 8^3$ $(486688)_2$
6-18
$46^2 8^3$ $(486688)_3$
6-19
(T)
$46^2 8^3$ $(486688)_4$
6-20
$46^2 8^3$ $(486868)_1$
6-21
$46^2 8^3$ $(486868)_2$
6-22
(K)
$5^2 7^2 8^2$ (575788)
6-23
$56^2 78^2$ (567886)
6-24
$567^3 8$ (567787)
6-25
$567^3 8$ (568777)
6-26
$567^3 8$ (576778)
6-27
$567^3 8$ (576877)
6-28
$6^3 7^2 8$ (667678)
6-29
(T)
$6^4 8^2$ (666688)



In total: 62 convex polygons (Snug golds)

cf. Tangram: 13
Seisho-nagon: 16

How to Solve the Cake-Cutting Problem in Sublinear Time

Hiro Ito[1] **and Takahiro Ueda**[2]

1 **School of Informatics and Engineering, The University of Electro-Communications (UEC)/Tokyo, Japan; and CREST, JST/Tokyo, Japan**
itohiro@uec.ac.jp

2 **Komatsu Ltd., Tokyo, Japan**
mx.u.2147483647@gmail.com

Abstract

The cake-cutting problem refers to the issue of dividing a cake into pieces and distributing them to players who have different value measures related to the cake, and who feel that their portions should be "fair." The fairness criterion specifies that in situations where n is the number of players, each player should receive his/her portion with at least $1/n$ of the cake value in his/her measure. In this paper, we show algorithms for solving the cake-cutting problem in sublinear-time. More specifically, we preassign fair portions to $o(n)$ players in $o(n)$-time, and minimize the damage to the rest of the players. All currently known algorithms require $\Omega(n)$-time, even when assigning a portion to just one player, and it is nontrivial to revise these algorithms to run in $o(n)$-time since many of the remaining players, who have not been asked any queries, may not be satisfied with the remaining cake. To challenge this problem, we begin by providing a framework for solving the cake-cutting problem in sublinear-time. Generally speaking, solving a problem in sublinear-time requires the use of approximations. However, in our framework, we introduce the concept of "ϵn-victims," which means that ϵn players (victims) may not get fair portions, where $0 < \epsilon \leq 1$ is an arbitrary constant. In our framework, an algorithm consists of the following two parts: In the first (Preassigning) part, it distributes fair portions to $r < n$ players in $o(n)$-time. In the second (Completion) part, it distributes fair portions to the remaining $n-r$ players except for the ϵn victims in poly(n)-time. There are two variations on the r players in the first part. Specifically, whether they can or cannot be designated. We will then present algorithms in this framework. In particular, an $O(r/\epsilon)$-time algorithm for $r \leq \epsilon n/127$ undesignated players with ϵn-victims, and an $\tilde{O}(r^2/\epsilon)$-time algorithm for $r \leq \epsilon e^{\sqrt{\ln n}/7}$ designated players and $\epsilon \leq 1/e$ with ϵn-victims are presented.

1998 ACM Subject Classification G.2.1 [Combinatorics] Combinatorial algorithms, G.3 [Probability and Statistics] Probabilistic algorithms, I.1.2 [Algorithms] Analysis of algorithms

Keywords and phrases sublinear-time algorithms, cake-cutting problem, simple fair, preassign, approximation

Digital Object Identifier 10.4230/LIPIcs.FUN.2016.21

1 Introduction

1.1 What is a sublinear-time algorithm for the cake-cutting problem?

This paper reports the first results on sublinear-time algorithms for solving the cake-cutting problem, in which it is necessary to divide a given cake into pieces and to distribute those pieces to players in a way that ensures that all players are "satisfied," more specifically, in

© Hiro Ito and Takahiro Ueda;
licensed under Creative Commons License CC-BY

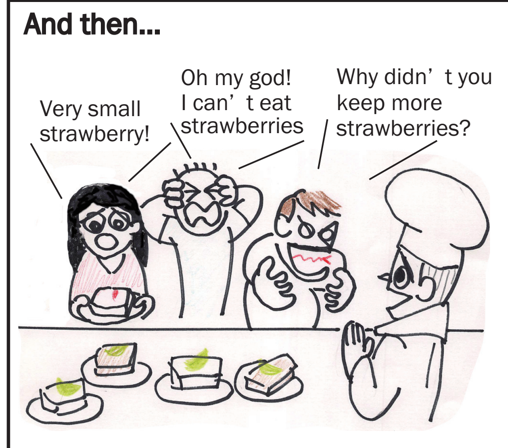


Figure 1 Participants who came late should be satisfied also.

a way that ensures every player believes that his/her portion has at least $1/n$ value of the whole cake, where n is the number of the players. It has previously been known that an $O(n \log n)$-time algorithm [5] and that a $\Theta(n \log n)$-time lower-bound exists for deterministic algorithms [3]. For approximation, a linear-time $O(1)$ approximation algorithm has been given [3].

However, when we divide a property and assign portions to persons, sometimes situations arise in which it is impossible to meet the requirements of all interested persons simultaneously. In such cases, it may be necessary to assign acceptable portions to some persons without asking for approval from all other interested persons. For example, in a reception party of a conference, a people who comes early takes a piece of cake that he likes, but it’s better to leave “fair” pieces for participants who will come later (Fig. 1). This is the motivation behind our desire to develop algorithms for solving the cake-cutting problem in sublinear time.

Herein, we consider ways to *preassign portions to a number of* $(r = o(n))$ *players in* $o(n)$*-time*. However, since all the known algorithms need $\Omega(n)$-time, even for assigning a portion to just one player, the problem is nontrivial. In fact, it is a difficult matter to satisfy just one player. Moreover, even if r players can be satisfied, if the other $n - r$ players are unsatisfied, the solution is clearly suboptimal in many cases. Thus, it is better to be able to satisfy the remaining players by distributing the remaining cake appropriately. However, it is often very hard (or even impossible) to completely satisfy all the other players since we have already assigned a portion of the cake after giving queries to only a sublinear number of players. Thus, we need to make some approximations. With this in mind, we hereby introduce the concept of “ϵn-victims,” which means that *we can give up trying to satisfy at most* ϵn *players (victims)*.

Recently, it has been learned that many problems can be approximated in sublinear time [1, 6, 7, 8, 9, 10, 11, 12, 13, 16]. To solve a problem in sublinear time, it is necessary to introduce some approximations by using a parameter $0 < \epsilon \leq 1$. There are two types of approximations. In the first, which is for decision problems, an edit distance between an instance and an objective property is defined and algorithms distinguish between instances satisfying the property and those ϵ-far from the property with high probability. The second type, which is for optimizing problems, provides ϵn-approximation solutions for objective functions [6].

All known sublinear-time algorithms use one of these two approximation types. Our approximation, ϵn-victims, can be seen as the latter approximation type. Thus, if we stipulate that the goal when solving the cake-cutting problem is to maximize the number of satisfied players, a solution with ϵn-victims is an ϵn-approximation solution.

The results of this paper can be summarized as follows:

- Presenting a framework for solving the cake-cutting problem in sublinear-time.
- Presenting sublinear-time algorithms under this framework.

The framework presented here is as follows:

The proposed sublinear-time cake-cutting framework

- **First (Preassigning) Part:** First, we preassign portions to $r = o(n)$ players in $o(n)$-time.
- **Second (Completion) Part:** Next, we assign portions to the remaining $n - r$ players except for the ϵn victims in poly(n)-time.

Note that it is impossible to do the second part in sublinear-time, since it is necessary to assign one portion to each of the remaining $\Omega(n)$ players (except for the ϵn victims).

Next, we will consider sublinear-time cake cutting algorithms that obey this framework. These algorithms can be divided into two types: one in which **the preassigned players cannot be designated**, and the other in which **they can be designated**. We will then present algorithms for both types. More specifically, for the first (undesignated) type, we can preassign portions to the $r \leq \epsilon n/127$ undesignated players in $O(tr/\epsilon)$-time and set the success probability to at least $1 - (\frac{1}{64})^{t/\epsilon} - \frac{8}{(2t-3)^2 r}$. After that, we can assign portions to the remaining players except for the ϵn victims in $O(n \log n)$-time, where $t \geq 1$ is an arbitrary real number. For the latter (designated) type, for any $0 < \epsilon \leq 1/e$, we can preassign portions to $r \leq \epsilon e^{\sqrt{\ln n}/7}$ designated players in $\tilde{O}(tr^2/\epsilon)$-time and set the success probability to at least $1 - (\epsilon/r)^t \geq 1 - e^{-t}$. After that, we can then assign portions to the remaining players except for the ϵn victims in $O(rn \log rn)$-time, where $t \geq 1$ is an arbitrary real number.

1.2 Definition of the cake-cutting problem

Let P be the set of n players. We assume that every algorithm for solving the cake-cutting problem knows n (which is the number of players)[1].

The cake is represented by the unit interval $C = [0, 1]$. The portion of each player is a set of disjoint subintervals of C. Every player $p \in P$ has his/her subjective nonnegative value function $\mu_p : 2^C \to [0, 1]$, which is defined on every measurable subset of C. Furthermore, μ_p is additive. In other words, the value of the portion of a player is the sum of the subinterval values of his/her portion. The value function is normalized, i.e., $\mu_p(C) = 1$ for every $p \in P$.

A portion $C_p \subseteq C$ of a player $p \in P$ is deemed to be *fair* if $\mu_p(C_p) \geq 1/n$. For any positive real $c \geq 1$, C_p is deemed to be *c-fair* if $\mu_p(C_p) \geq 1/cn$.

When evaluating cake-cutting algorithms, the Robertson-Webb model [14] is generally used. In this model, the two query types listed below are allowed, and the complexity of an algorithm is evaluated by the *query complexity*. In other words, the number of these queries made by the algorithm.

- *Cut query*: For a continuous piece of a cake $D = [a, b] \subseteq C$ $(0 \leq a < b \leq 1)$, a player $p \in P$, and a positive real number $0 \leq \alpha \leq 1$, a query $\textsc{Cut}(D, p, \alpha)$ returns the smallest

[1] Although this assumption may seem trivial, it is an important consideration in sublinear-time algorithms since the algorithm cannot count the number of players in sublinear-time. Such an assumption is generally introduced when sublinear-time algorithms are investigated.

value $x \in [a, b]$ such that $\mu_p([a, x]) = \alpha$. If there is no such x (i.e., $\mu_p(D) < \alpha$), then it returns some predefined message.

- *Evaluation query*: For a continuous piece of a cake $D = [a, b] \subseteq C$ $(0 \leq a < b \leq 1)$, a player $p \in P$, and a positive real number $x \in [a, b]$ (= a point on D), a query $\textsc{Eval}(D, p, x)$ returns the value $0 \leq \alpha \leq 1$ such that $\mu_p([a, x]) = \alpha$. If $x \geq b$ (i.e., asking to evaluate whole of D), then $\textsc{Eval}(D, p, x)$ is simply expressed as $\textsc{Eval}(D, p)$.

1.3 Previous work

One well known method of handing the cake-cutting problem, involves an $O(n \log n)$-time algorithm using the divide-and-conquer concept [5]. This algorithm divides a cake into two pieces and assigns players into two half-sized subsets. It begins by assigning one of the pieces to one of the subgroups, and the other piece to the other subgroup. Then, it recursively applies this separation until every subgroup becomes a singleton. Note that this algorithm requires $\Theta(n)$ queries even if it only assigns a portion to one player. We refer to this algorithm as $\mathrm{DC}(P, C)$.

For the lower-bound results, it is known that the time-complexity is $\Theta(n \log n)$ for a deterministic algorithm [3]. In the same paper, they showed that, with some restrictions, this bound can be also applied to a randomized case. For approximations, Edmonds and Pruhs [4] showed that for c-fair division with $c > 32$, there is an $O(n)$-time randomized algorithm.

The success probability of this algorithm is at least

$$1 - \frac{2^{13}}{c^2(c-32)} - \frac{1024}{c^3} - \frac{128}{c^2} \tag{1}$$

for $c > 32$. We refer to this algorithm as $\textsc{ApproxFair}(P, C, c)$. Our algorithms use these algorithms as subroutines. In addition to these two, numerous other algorithms have been presented [2, 15]. If it is necessary to distribute pieces to $\Omega(n)$ players fairly, clearly we need $\Omega(n)$-time. However, none of the previously known algorithms have considered preassigning portions to $o(n)$ players, and they all need $\Omega(n)$ queries even when assigning a portion to just one player.

1.4 Our results

In this subsection, we will explain the results we have obtained thus far. Throughout this paper, we assume that every player is honest, i.e., that he/she gives correct answers for every query[2]. First we show a preliminary result, which can be obtained as a simple application of [3], as follows:

▶ **Proposition 1.** *For any $t \geq 64$ and any given subset $P_r \subseteq P$ of players with $|P_r| = r \leq n/t$, there is an $O(r)$-time algorithm for assigning fair portions to all players in P_r with success probability at least $1 - 2^9/t^2$.*

The complexity of this algorithm is $O(r)$, and it is clearly the best possible because it matches with the trivial lower bound. Moreover, it also allows us to arbitrarily assign designated r players. However, one obvious flaw of this algorithm is that it may victimize all of the

[2] Even if there is a dishonest player, the honest players will get fair portions, while the dishonest player may not.

remaining players, so efforts should be made to reduce the number of victims. The following algorithm allows a maximum of ϵn victims for any given $0 < \epsilon \leq 1$:

▶ **Theorem 2.** *For any positive real number $0 < \epsilon \leq 1$, any positive integer $r \leq \epsilon n/127$, and any constant real number $t > 3/2$, there is an algorithm for preassigning fair portions to r players in $O(tr/\epsilon)$-time, and then assigning fair portions to the remaining players except for the ϵn players (victims) in $O(n \log n)$-time with success probability at least $1 - \frac{8}{(2t-3)^2 r} - (\frac{1}{64})^{t/\epsilon}$.*

While in the algorithm of Theorem 2, preassigned r members cannot be designated, the following shows an algorithm in which they can:

▶ **Theorem 3.** *For any real numbers $0 < \epsilon \leq 1/e$ and $t \geq 1$, and any set of $r \leq \epsilon e^{\sqrt{\ln n}/7}$ players P_r, there is an algorithm for preassigning fair portions to all players in P_r in $O(\frac{tr^2}{\epsilon}(\log \frac{r}{\epsilon})^3)$-time and then assigning fair portions to remaining players except for the ϵn players (victims) in $O(rn \log(rn))$-time with success probability at least $1 - (\epsilon/r)^t \geq 1 - e^{-t}$.*

1.5 Organization

The remainder of this paper is organized as follows: In Section 2, we present a proof of Proposition 1. In Sections 3 and 4, we examine the undesignated version (Theorem 2) and the designated version (Theorem 3), respectively. In Section 5, we summarize our results and discuss future work.

2 Proof of Proposition 1

In this short section, we show the following proof of Proposition 1.

Proof of Proposition 1. It is sufficient to simply call ApproxFair(r, P_r, C, t). It assigns t-fair portions to all players in P_r. In other words, they feel at least $\frac{1}{tr}$ value in their own portion. From the assumption $r \leq \frac{n}{t}$ it follows that $\frac{1}{tr} \geq \frac{1}{n}$. Therefore, they all get fair portions, and by considering (1) and $t \geq 64$, the probability of failing is at most

$$\frac{2^{13}}{t^2(t-32)} + \frac{2^{10}}{t^3} + \frac{2^7}{t^2} \leq \frac{2^{14}}{t^3} + \frac{2^{10}}{t^3} + \frac{2^7}{t^2} \leq \frac{2^8}{t^2} + \frac{2^4}{t^2} + \frac{2^7}{t^2} \leq \frac{2^9}{t^2}.$$

The time-complexity is clearly $O(r)$. ◀

3 Undesignated r players

In this section, we consider a case where P_r cannot be designated.

3.1 Algorithm for Theorem 2

When preassigned players are not designated, the algorithms can select P_r players arbitrarily. That is, players who feel a relatively high value in a specified part (e.g., the left-side part of the cake) are considered more suitable, such members can be selected at high probability levels by using asking cut-queries to a number ($\lceil tr/\epsilon \rceil$) of other players. Let P' be the set of selected players and let C' be a piece to which these players in P' have assigned high value ($128r/n$). Then, by applying ApproxFair to P' and C' with approximation parameter 128, the players in P' have a high probability of getting fair portions. This summarizes the preassigning part.

For the completion part, it can be expected that a small number of the remaining players will feel that C' (the removed piece) has high value, and that the only way the remaining players can share the rest of the cake ($C - C'$) fairly is by removing the appropriate ϵn players (victims).

Before showing the details of this algorithm, we will first define a subroutine PCUT it uses. The objective of this subroutine is to get a set of $m \in \{0, \ldots, n\}$ players from $Q \subseteq P$ who have a high probability of seeing relatively high value in the left-most part of the piece D.

procedure PCUT(Q, D, α, m)
Input: $Q \subseteq P$, $D \subseteq C$, real value $0 \leq \alpha \leq 1$, integer $0 \leq m \leq n$;
begin
1 **for** $p \in Q$ **do**
2 $x_p :=$ CUT(D, p, α)
3 **enddo**
4 Let Q' be the set of players $p \in Q$ having the 1st, 2nd, ..., and the mth smallest value x_p in Q,
where ties are broken arbitrarily.
5 **Output** Q'
end.

The preassigning part of the algorithm used for proving Theorem 2 is as follows:

procedure PREASSIGNU(P, C, r, ϵ, t)
Input: The set P of n players, The cake $C = [0, 1]$, positive integers r and t, real value $0 < \epsilon \leq 1$;
begin
01 $P_0 := \emptyset$
02 **for** $\lceil \frac{tr}{\epsilon} \rceil$ times **do**
03 Select $p \in P$ UAR and $P_0 := P_0 \cup \{p\}$;
04 **enddo**
05 **if** $|P_0| < r$ **then output** "Failed" and **stop endif**;
06 $P' :=$ PCUT($P_0, C, \frac{128r}{n}, r$)
07 $x := \max_{p \in P'} x_p$
08 $C' := [0, x]$
09 **for** $\lceil \frac{t}{\epsilon} \rceil$ times **do**
10 **call** APPROXFAIR($r, P', C', 128$)
11 **if** above APPROXFAIR succeeds **then**
12 **output** the assignment obtained in Line 10; **stop**;
13 **endif**;
14 **enddo**
15 **comment** all APPROXFAIR in Line 10 failed;
16 **output** "Failed";
end.

By applying PREASSIGNU, providing it does not fail, all players in P' ($|P'| = r$) will have their own portions (which we will later prove are fair). The next important point is ensuring that the remaining players except for ϵn victims will be satisfied. We define the other terms used for treating this problem as follows:

▶ **Definition 4.** Let $Q \subseteq P$ and $D \subseteq C$ be a subset of players and a subset of the cake, respectively. A player $p \in Q$ is called *safe with respect to* (Q, D) if $\mu_p(D) \geq \frac{|Q|}{n}$, or *dangerous*

with respect to (Q, D) otherwise. We may omit "with respect to (Q, D)" if (Q, D) is clear. If all players in Q are safe with respect to (Q, D), we then say that Q is *safe with respect to* D, or *safe* in short, if D is clear. For $m \geq 0$, if there is a subset of $Q' \subset Q$ such that $|Q'| \leq m$ and $Q - Q'$ is safe, then Q is called m*-safe*.

If Q is safe with respect to D, it is clear that all players in Q can get fair portions in D by using arbitrary cake-cutting-algorithms, such as DC(Q, D) (Lemma 6, which will be shown later). Then, for proving the completion part following PreassignU, we should show that $P - P'$ is ϵn-safe with respect to $C - C'$. The algorithm of the completion part is simple: It is sufficient to make a query Eval$(C - C', p)$ for every player p in $P - P'$ and remove the lowest evaluating ϵn players. Pseudo code of this algorithm is shown below:

procedure Completion$(P - P', C - C', n, \epsilon)$
begin
1 $Q :=$ Victimize$(P - P', C - C', \lfloor \epsilon n \rfloor)$
2 **call** DC$(Q, C - C')$
end.

procedure Victimize(P'', D, m)
Input: Subset $P'' \subseteq P$ of players, subset $D \subseteq C$ of the cake, integer $m \geq 0$;
begin
1 **for** $p \in P''$ **do** $x_p :=$ Eval(D, p) **enddo**
2 Let $Q_{\text{vict}} \subseteq P''$ be the set of m players having the 1st, 2nd, ..., mth smallest values of x_p,
where ties are broken arbitrarily;
3 **output** $Q := P'' - Q_{\text{vict}}$;
end.

3.2 Proof of Theorem 2

We prepare the following lemmas for showing the proof of Theorem 2.

▶ **Lemma 5.** *Let N be $\{1, 2, \ldots, n\}$ and S be an $\lfloor \epsilon n \rfloor$ size subset of N for $0 < \epsilon \leq 1$. For real numbers $s, t > 1$ such that $(s-1)(t-1) > 1$ and a positive integer r such that $r \leq \epsilon n / s$, if we choose at least tr/ϵ elements from N uniformly at random (UAR), we then get at least r different elements in S with probability at least $1 - \frac{s^2}{((s-1)(t-1)-1)^2 r}$.*

Proof. Let X be the random variable of the number of chosen elements until we get r different elements in S from N. Further, let X_i be the random variable of the number of chosen elements until we get i-th different elements in S after $i-1$ different elements were chosen from S. Clearly, $X = \sum_{i=1}^{r} X_i$.

Let p_i be the probability that we get a new element from S after we have gotten $i-1$ different elements from S. The following inequalities hold:

$$p_i = \frac{\lfloor \epsilon n \rfloor - (i-1)}{n} \geq \frac{\lfloor \epsilon n \rfloor - (r-1)}{n} > \frac{\epsilon n - r}{n} \geq \frac{\epsilon n - \frac{\epsilon n}{s}}{n} = \frac{(s-1)\epsilon}{s}.$$

Since the random variable X_i follows a geometric distribution, the expected value $E[X_i]$ and the variance $V[X_i]$ satisfy $E[X_i] = 1/p_i$ and $V[X_i] = (1-p_i)/p_i^2$, respectively. By the linearity of expected value,

$$E[X] = \sum_{i=1}^{r} E[X_i] \leq \sum_{i=1}^{r} \frac{s}{(s-1)\epsilon} = \frac{sr}{(s-1)\epsilon}.$$

Since each X_i is independent, the variance satisfies linearity, and thus

$$V[X] = \sum_{i=1}^{r} V[X_i] \leq \sum_{I=1}^{r} \frac{1 - \frac{(s-1)\epsilon}{s}}{\left(\frac{(s-1)\epsilon}{s}\right)^2} = \frac{(s - \epsilon s + \epsilon)sr}{(s-1)^2\epsilon^2}.$$

We compute the probability that we do not get at least r different elements in S when we choose tr/ϵ elements from N uniformly, at random, as follows:

$$\begin{aligned}
\Pr\left[X > \frac{tr}{\epsilon}\right] &\leq \Pr\left[X \geq \frac{tr}{\epsilon}\right] \leq \Pr\left[\left|X - \frac{sr}{(s-1)\epsilon}\right| \geq \frac{tr}{\epsilon} - \frac{sr}{(s-1)\epsilon}\right] \\
&= \Pr\left[\left|X - \frac{sr}{(s-1)\epsilon}\right| \geq \frac{sr}{(s-1)\epsilon}\left(t\frac{s-1}{s} - 1\right)\right] \\
&\leq \Pr\left[|X - E[X]| \geq \frac{sr}{(s-1)\epsilon}\left(t\frac{s-1}{s} - 1\right)\right]. \\
&\quad \text{(From Chebyshev bound, } \Pr[|X - E[X]| \geq a] \leq \tfrac{V[X]}{a^2},\ \forall a > 0) \\
&\leq \frac{V[X]}{\left(\frac{sr}{(s-1)\epsilon}(t\frac{s-1}{s} - 1)\right)^2} \\
&\leq \frac{\frac{(s-\epsilon s+\epsilon)sr}{(s-1)^2\epsilon^2}}{\left(\frac{sr}{(s-1)\epsilon}(t\frac{s-1}{s} - 1)\right)^2} = \frac{s - (s-1)\epsilon}{(t\frac{s-1}{s} - 1)^2 rs} \\
&\leq \frac{s}{\left(t\frac{s-1}{s} - 1\right)^2 rs} = \frac{s^2}{(st - s - t)^2 r} = \frac{s^2}{((s-1)(t-1) - 1)^2 r}.
\end{aligned}$$

The desired inequality is obtained. ◀

▶ **Lemma 6.** *For any $Q \subseteq P$ and $D \subseteq C$, if Q is safe with respect to D, then all players in Q can get fair portions in D by using arbitrary cake-cutting algorithms.*

Proof. By applying a cake-cutting algorithm, every player $p \in Q$ obtains a portion with value at least $\mu_p(D)/|Q|$. From that, Q is safe with respect to D, $\mu_p(D) \geq |Q|/n$ for $\forall p \in Q$. Thus, the value of the cake obtained by $\forall p \in Q$ is

$$\frac{\mu_p(D)}{|Q|} \geq \frac{1}{|Q|} \cdot \frac{|Q|}{n} = \frac{1}{n}.$$

◀

Proof of Theorem 2. We will show the following facts:

(i) All players in P' get fair portions with probability at least $1 - \frac{8}{(2t-3)^2 r} - \left(\frac{1}{64}\right)^{t/\epsilon}$ after calling ApproxFair in line 10 of PreassignU.

(ii) Q (the output of Victimize$(P - P', C - C', \lfloor \epsilon n \rfloor)$ in line 01 of Completion$(P - P', C - C', n, \epsilon)$ is safe with respect to $C - C'$.

In what follows, we show proofs for the above items.

(i) First, we assume that $|P_0| \geq r$ in line 05 of PreassignU and that at least one call of ApproxFair in line 10 of PreassignU succeeds. Let C_p be the portion that player $p \in P'$ gets by this ApproxFair when it succeeds. From the property of ApproxFair,

C_p is at least 128-fair, i.e., $\mu_p(C_p) \geq \mu_p(C')/128r$. From the operations in lines 06–08 of PreassignU, $\mu_p(C') \geq 128r/n$. It then follows that

$$\mu_p(C_p) \geq \frac{1}{128r} \cdot \frac{128r}{n} = \frac{1}{n},$$

i.e., each player in P' gets a fair portion.
Next, we estimate the probability that $|P_0| \geq r$ in line 05 of PreassignU and that at least one call of ApproxFair in line 10 of PreassignU succeeds. From Lemma 5 with regarding P and $\text{Pcut}(P, C, \frac{128r}{n}, \lfloor \epsilon n \rfloor)$ as N and S, respectively [3] and by letting $s = 127$, it follows that the probability that $|P_0 \cap \text{Pcut}(P, C, \frac{128r}{n}, \lfloor \epsilon n \rfloor)| < r$ occurs is at most $\frac{127^2}{(126t-127)^2 r}$. From the assumption of $t > 3/2$, this probability becomes

$$\frac{127^2}{(126t-127)^2 r} < \frac{(127/126)^2}{(t-3/2)^2 r} < \frac{8}{(2t-3)^2 r}.$$

$|P_0 \cap \text{Pcut}(P, C, \frac{128r}{n}, \lfloor \epsilon n \rfloor)| \geq r$ includes $|P_0| \geq r$ and $|P'| = r$.
From (1), the probability that one call of ApproxFair in line 10 of PreassignU succeeds is at least

$$1 - \frac{2^{13}}{2^{14}(128-32)} - \frac{1024}{128^3} - \frac{128}{128^2} = 1 - \frac{83}{6144} > 1 - \frac{1}{64}.$$

Thus, the probability that all the calls of ApproxFair fail is at most $64^{-t/\epsilon}$.
Therefore, the success probability of this algorithm is at least

$$1 - \frac{8}{(2t-3)^2 r} - \left(\frac{1}{64}\right)^{t/\epsilon}.$$

(ii) Assume that $|P_0 \cap \text{Pcut}(P, C, \frac{128r}{n}, \lfloor \epsilon n \rfloor)| \geq r$. From this, $P' \subseteq \text{Pcut}(P, C, \frac{128r}{n}, \lfloor \epsilon n \rfloor)$ follows. This means that for every player $p \in P - \text{Pcut}(P, C, \frac{128r}{n}, \lfloor \epsilon n \rfloor)$, $\mu_p(C - C') \geq \frac{|P|-128r}{|P|}$. From the assumption of $r \leq \lfloor \epsilon n/127 \rfloor$ ($\because r$ is an integer),

$$\mu_p(C - C') \geq \frac{|P| - 128r}{|P|} > \frac{|P| - \lfloor \epsilon n \rfloor - r}{|P|}.$$

It follows that $Q \subseteq P - \text{Pcut}(P, C, \frac{128r}{n}, \epsilon n)$ and $|Q| = n - \epsilon n - r$. Therefore, Q is safe with respect to $C - C'$.

From (ii) and Lemma 6, DC in line 02 of Completion assigns fair portions to all players in $P - P'$. The query complexity of PreassignU is clearly $O(tr/\epsilon)$. The query complexity of Completion is $O(n \log n)$, since DC can be done in $O(n \log n)$. ◀

4 Designated r players

4.1 Algorithm for Theorem 3

In this section, we consider the case where P_r is given. The key to solving this problem is to find a piece C_p that a player $p \in P_r$ prefers. After finding C_p for all $p \in P_r$, if all C_p are disjoint, we then assign C_p to p. Otherwise, i.e., when some $C_{p_1}, \ldots, C_{p_k}$ are "connected"

[3] The reason that $\text{Pcut}(P, C, \frac{128r}{n}, \lfloor \epsilon n \rfloor)$ is considered here is explained in (ii).

(the definition is given later), we allot $C_{p_1} \cup \ldots \cup C_{p_k}$ to $\{p_1, \ldots, p_k\}$ by using a suitable cake-cutting algorithm, e.g., DC.

The basic strategy used to find C_p is as follows. In the beginning, $C_p := C$ (of course, it will be trimmed). We ask a randomly chosen constant number of players (let P_p be the set of chosen players) to evaluate C_p. If a small number of players evaluate it as high, then C_p is fixed. Otherwise (in the first iteration, this case must occur since $C_p = C$), we divide C_p into two pieces such that the half of players in P_p prefer one of the half pieces and the other players prefer the other piece, and let C_p be the half piece that p prefers. By iteratively applying the above operations some fixed number of times, we have a high probability of getting an appropriate C_p.

To show the details of the first (preassigning) part, we use the following concept. Let $\mathcal{C} = \{C_1, \ldots, C_{|\mathcal{C}|}\}$ be a family of cake subsets. We define the *relation graph* $G_{\mathcal{C}} = (\mathcal{C}, E_{\mathcal{C}})$ with respect to $\mathcal{C}$ as $(C_i, C_j) \in E_{\mathcal{C}}$ iff $C_i \cap C_j \neq \emptyset$ for $i, j \in \{1, \ldots, |\mathcal{C}|\}$ and $i \neq j$.

procedure PREASSIGNS(P, C, P_r, ϵ, t)
Input: The set P of n players, The cake $C = [0, 1]$, a subset of r players $P_r \subseteq P$, positive integer t, real value $0 < \epsilon \leq 1$;
begin
01 **for all** $p \in P_r$ **do**
02 $C_p :=$ DEPOSIT$(p, P, C, \epsilon/r, t)$
03 **enddo**
04 Construct the relation graph $G_{\mathcal{C}}$ with respect to $\mathcal{C} := \{C_p \mid p \in P_r\}$.
05 **for all** connected components $\mathcal{C}'$ of $G_{\mathcal{C}}$ **do**
06 Let $C_{p_1}, \ldots, C_{p_k}$ be the vertices (cake subsets) in $\mathcal{C}'$;
07 **call** DC$(\{p_1, \ldots, p_k\}, C_{p_1} \cup \cdots \cup C_{p_k})$
08 Let $C^*_{p_i}$ be the piece assigned by DC in Line 07 for $i = 1, \ldots, k$;
09 **enddo**
10 **output** C^*_p for every $p \in P_r$;
end.

procedure DEPOSIT(p, P, C, ϵ', t)
begin
01 $C' := C$, $h := \left\lceil \frac{2^{10} t}{\epsilon'} \ln \frac{1}{\epsilon'} \right\rceil$
02 **from** $j = 1$ **to** $54 \left(\ln \frac{1}{\epsilon'}\right)^2$ **do**
03 Choose a player from P UAR h times and let P_0 be the multiset of the chosen players;
04 **for all** $q \in P_0$ **do**
05 $\alpha_q :=$ EVAL(C', q)
06 **enddo**
07 Let P' be the multiset of the players $q \in P_0$ such that $\alpha_q \geq \epsilon'$;
08 **if** $|P'| < 2^9 t \ln \frac{1}{\epsilon'}$ **then**
09 **output** C'; **return**
10 **endif**
11 **call** CONDENSE(p, P', C')
12 **enddo**
13 **return**
end.

```
procedure CONDENSE(p, P', C' = [a, b])
begin
01    x_L := a, x_R := b
02    for all q ∈ P' do
03      β_q := EVAL([x_L, x_R], q)
04      x_q := CUT([x_L, x_R], q, β_p/2)
05    enddo
06    Let q_0 be the player such that x_{q_0} is the median of multiset {x_q | q ∈ P'};
07    α_L := EVAL([x_L, x_{q_0}], p)
08    α_R := EVAL([x_{q_0}, x_R], p)
09    if α_L > α_R then
10      C' := [x_L, x_{q_0}]
11    else
12      C' := [x_{q_0}, x_R]
13    endif
14    return
end.
```

$\widehat{C} := \cup_{p \in P_r} C_p$. The completion part of the algorithm for Theorem 3 is simply applying COMPLETION$(P - P_r, C - \widehat{C}, n, \epsilon)$.

4.2 Proof of Theorem 3

Before showing the proof of Theorem 3, we show some lemmas, whose proofs are in Appendix. For $D \subseteq C$ and $0 \leq \alpha \leq 1$, we denote the set of players $p \in P$ such that EVAL$(D, p) \geq \alpha$ by $P(\alpha, D)$.

▶ **Lemma 7.** *For $p \in P$, $D \subseteq C$, and real numbers $0 < \epsilon < 1/e$ and $t \geq 1$, we choose players from $P(\epsilon, D)$ uniformly at random at least $2^9 t/\epsilon$ times and let Q be the multiset of the chosen players. Let D' denote the output D of* CONDENSE(p, Q, D). *Then the following two conditions hold:*

- EVAL$(D', p) \geq$ EVAL$(D, p)/2$, *and*
- EVAL$(D', q) \leq$ EVAL$(D, q)/2$ *for at least $|P(\epsilon, D)|/3$ players $q \in P(\epsilon, D)$ with probability at least $1 - \epsilon^{16t}$.*

Proof. The first item (EVAL$(D', p) \geq$ EVAL$(D, p)/2$) is clear from the operations in Lines 09-13 of CONDENSE. Then, we prove the second item. Let $|P(\epsilon, D)| = m$. Define $P_L =$ PCUT$(P(\epsilon, D), D, \text{EVAL}(D, q)/2, m/3)$ and $P_R = P -$PCUT$(P(\epsilon, D), D, \text{EVAL}(D, q)/2, 2m/3)$. Let Y_i^L (resp, Y_i^R) be a random variable such that it is 1 when the ith element of Q is included in P_L (rest., P_R) and 0 otherwise. $Y^L := \sum_{i=1}^{|Q|} Y_i^L$ and $Y^R := \sum_{i=1}^{|Q|} Y_i^R$. Clearly $E[Y^L] = E[Y^R] = |Q|/3 \geq 2^9 t/3\epsilon$. Every Y_i^L and Y_i^R is an independent Bernoulli trial, and thus from Chernoff bound ($\Pr[X \geq (1+\delta)E[X]] \leq e^{-\delta^2 E[X]/3}$), it follows that

$$\Pr\left[Y^L \geq \frac{|Q|}{2}\right] = \Pr\left[Y^L \geq \frac{3}{2}E[Y^L]\right] \leq e^{-E[Y^L]/12} \leq e^{-128t/9\epsilon}.$$

Here, by considering that for all real number x,

$$x \ln \frac{1}{x} \leq \frac{1}{e} \leq \frac{4}{9},$$

we get

$$\Pr\left[Y^L \geq \frac{|Q|}{2}\right] \leq e^{-32t\ln(1/\epsilon)} = \epsilon^{32t}.$$

Similarly, we also get $\Pr\left[Y^R \geq |Q|/2\right] \leq \epsilon^{32t}$. Let q_0 be the player in line 06 of CONDENSE(p, Q, D). Then, $\Pr[q_0 \in P_L \cup P_R] \leq 2\epsilon^{32t} \leq \epsilon^{16t}$. Therefore, for at least $m/3$ players q (i.e., players in P_L), $\text{EVAL}([x_{q_0}, x_R], q) \leq \text{EVAL}(D, q)/2$ and for at least $m/3$ players q' (i.e., players in P_R), $\text{EVAL}([x_L, x_{q_0}], q') \leq \text{EVAL}(D, q')/2$ with probability at least $1 - \epsilon^{16t}$. ◀

▶ **Lemma 8.** *If $P(\epsilon', C') \geq \epsilon' n$ when an operation of Line 08 of* DEPOSIT(p, P, C, ϵ', t) *is done, then the probability that $|P'| < 2^9 t \ln \frac{1}{\epsilon'}$ occurs is at most ϵ'^{128t}.*

Proof. Let $P_0 = \{q_1, \ldots, q_h\}$ be P_0 constructed in Line 03 of DEPOSIT. Let X_i $(i = 1, \ldots, h)$ be the random variable such that $X_i = 1$ if $q_i \in P(\epsilon', C')$ and $X_i = 0$ otherwise. Let X be the random variable representing $|P_0 \cap P(\epsilon', C')|$. Clearly, $X = \sum_{i=1}^{h} X_i$ and

$$E[X] = \frac{|P(\epsilon', C')|}{n} \cdot \frac{2^{10} t}{\epsilon'} \ln \frac{1}{\epsilon'} \geq 2^{10} t \ln \frac{1}{\epsilon'}.$$

From the Chernoff bound,

$$\begin{aligned}
\Pr\left[X \leq 2^9 t \ln \frac{1}{\epsilon'}\right] &= \Pr\left[X \leq \left(1 - \frac{1}{2}\right) 2^{10} t \ln \frac{1}{\epsilon'}\right] \\
&\leq \Pr\left[X \leq \left(1 - \frac{1}{2}\right) E[X]\right] \\
&\leq e^{-E[X]/8} \\
&\leq e^{2^7 t} \ln \frac{1}{\epsilon'} \\
&= \epsilon'^{128t}.
\end{aligned}$$

◀

In our algorithm, we call CONDENSE(p, P', C') iteratively. Then, for distinguishing C's in different calls, we number them such as $C^{(1)}$, $C^{(2)}$, ...: $C^{(1)}$ is C' of the first call of CONDENSE(p, P', C') (i.e., $C^{(1)} = C$), and the output of CONDENSE$(p, P', C^{(i)})$ is $C^{(i+1)}$ for $i \in \{1, 2, \ldots\}$. We say a call CONDENSE$(p, P', C^{(i)})$ is *good* if for at least $|P(\epsilon', C^{(i)})|/3$ players $q \in P(\epsilon', C^{(i)})$,

$$\text{EVAL}(C^{(i+1)}, q) \leq \text{EVAL}(C^{(i)}, q)/2. \tag{2}$$

From Lemma 7, a call CONDENSE$(p, P', C^{(i)})$ is good with probability at least $1 - \epsilon'^{16t}$.

▶ **Lemma 9.** *Assume that $C^{(j)}$ is obtained from $C^{(i)}$ after at least $\frac{9}{2}(\ln_{1/2} \epsilon' + 1)$ good calls. Then $|P(\epsilon', C^{(j)})| \leq \frac{2}{3}|P(\epsilon', C^{(i)})|$.*

Proof. Assume that $|P(\epsilon', C^{(j)})| > \frac{2}{3}|P(\epsilon', C^{(i)})|$. It is clear that $C^{(j)} \subseteq C^{(j-1)} \subseteq \cdots \subseteq C^{(i)}$. Let $m = |P(\epsilon', C^{(i)})|$. Then, for every $C^{(k)}$ $(k \in \{i, i+1, \ldots, j\})$,

$$|P(\epsilon', C^{(k)})| > \frac{2}{3} m. \tag{3}$$

Here, assume that if (2) occurs for a player $q \in P(\epsilon', C^{(i)})$, then q gets a "stone." If a player gets $\log_{1/2} \epsilon' + 1$ stones, then $\text{EVAL}(C', q) \leq \epsilon'$ and q is removed from $P(\epsilon', C')$. If CONDENSE$(p, P', C^{(i)})$ is good, at least $|P(\epsilon', C^{(i)})|/3$ stones are distributed. By considering (3), after $\frac{9}{2}(\ln_{1/2} \epsilon' + 1)$ good calls, at least $\frac{2}{3} m \cdot \frac{1}{3} \cdot \frac{9}{2}(\ln_{1/2} \epsilon' + 1) = m(\ln_{1/2} \epsilon' + 1)$ stones are distributed. Since one player can get $\ln_{1/2} \epsilon' + 1$ stones at most, every player gets $\ln_{1/2} \epsilon' + 1$ stones and has been removed from the $P(\epsilon', C^{(j)})$, contradiction. ◀

▶ **Lemma 10.** *Let C_p be the output of* DEPOSIT(p, P, C, ϵ', t)*. Assume that $\epsilon' \le 1/e$. Then*
(i) $\mu_p(C_p) \ge \left(\frac{1}{2}\right)^{54(\ln(1/\epsilon'))^2}$*, and*
(ii) $|P(\epsilon', C_p)| \le \epsilon' n$ *with a probability of at least* $1 - \epsilon'^{2t}$*.*

Proof. From Lemma 7, (i) is clear. Consider line 08 of DEPOSIT(p, P, C, ϵ', t). Assume that $|P(\epsilon', C')| \ge \epsilon' n$. Then, from Lemma 8, CONDENSE(p, P', C') is called in probability at least $1 - \epsilon'^{128t}$. From Lemma 7, CONDENSE(p, P', C') is good with probability at least $1 - \epsilon'^{16t}$

From Lemma 9, by the following number of good calls, we get $|P(\epsilon', C')| \le \epsilon' n$.

$$
\begin{aligned}
& \frac{9}{2}(\ln_{1/2} \epsilon' + 1)(\ln_{2/3} \epsilon' + 1) = \frac{9}{2}\left(\frac{\ln \frac{1}{\epsilon'}}{\ln 2} + 1\right)\left(\frac{\ln \frac{1}{\epsilon'}}{\ln \frac{3}{2}} + 1\right) \\
< \; & \frac{9}{2}\left(2 \ln \frac{1}{\epsilon'} + 1\right)\left(3 \ln \frac{1}{\epsilon'} + 1\right) \; (\because \ln 2 > 1/2 \text{ and } \ln 3/2 > 1/3) \\
< \; & \frac{9}{2}\left(3 \ln \frac{1}{\epsilon'}\right)\left(4 \ln \frac{1}{\epsilon'}\right) \; (\because \text{ from } \epsilon' \le 1/e,\ \ln \tfrac{1}{\epsilon'} \ge 1) \\
= \; & 54 \left(\ln \frac{1}{\epsilon'}\right)^2.
\end{aligned}
$$

The probability that "CONDENSE(p, P', C') is called and the call is good" $54\left(\ln \frac{1}{\epsilon'}\right)^2$ times in a row is at least

$$
\begin{aligned}
& 1 - \left(\epsilon'^{128t} + \epsilon'^{16t}\right) \cdot 54 \left(\ln 1/\epsilon'\right)^2 \\
\ge \; & 1 - \left(\epsilon'^{8t}\right)(1/\epsilon')^4 (1/\epsilon')^2 \;\; (\because 54 < e^4 \le (1/\epsilon')^4 \text{ and } \ln 1/\epsilon' \le 1/\epsilon') \\
\ge \; & 1 - \epsilon'^{2t}.
\end{aligned}
$$

Therefore (ii) is obtained. ◀

Proof of Theorem 3. We will show the following facts:
(i) Each player in P_r gets a fair portion by PREASSIGNS(P, C, P_r, ϵ, t).
(ii)] Q (the output of VICTIMIZE$(P - P_r, C - \widehat{C}, \lfloor \epsilon n \rfloor)$ in line 01 of COMPLETIONU$(P - P_r, C - \widehat{C}, n, \epsilon)$ is safe with respect to $C - \widehat{C}$ with probability at least $1 - (\epsilon/r)^t \ge 1 - e^{-t}$.

In what follows, we show proofs of the above items. Note that it is sufficient to consider the case that $\epsilon' = \epsilon/r$ in Lemmas 8, 9, and 10.

(i) From Lemma 10, $\mu_p(C_p) \ge (1/2)^{54(\ln(r/\epsilon))^2}$ for every player $p \in P_r$. Thus by PREASSIGNS, every player finally gets a portion having at least $(1/2)^{54(\ln(r/\epsilon))^2}/r$ value. We will show

$$(1/2)^{54(\ln(r/\epsilon))^2}/r \ge 1/n. \tag{4}$$

This inequality can be transformed as $\ln r + 54 \ln 2 \cdot \left(\ln\left(\frac{r}{\epsilon}\right)\right)^2 \le \ln n$. Here, from $\ln r \le \ln(r/\epsilon) \le (\ln(r/\epsilon))^2$ $(\because 1/\epsilon \ge e)$, the following inequalities hold:

$$\ln r + 54 \ln 2 \cdot \left(\ln \frac{r}{\epsilon}\right)^2 \le (1 + 54 \ln 2)\left(\ln \frac{r}{\epsilon}\right)^2 \le \left(7 \ln \frac{r}{\epsilon}\right)^2$$

Thus, if $(7 \ln(r/\epsilon))^2 \le \ln n$, then (4) holds. This is equivalent to $r \le \epsilon e^{\sqrt{\ln n}/7}$. That is, (4) holds.

(ii) For $p \in P_r$, if $|P(\epsilon/r, C_p)| \le (\epsilon/r)n$, then we say that p is *polite*. From Lemma 10, the probability that $p \in P_r$ in not polite is at most $(\epsilon/r)^{2t}$. Thus, the probability that at least one $p \in P_r$ in not polite is at most $r(\epsilon/r)^{2t} \le (\epsilon/r)^t \le e^{-t}$ (since $\epsilon/r \le \epsilon \le 1/e$). If all players in P_r are polite, then

$$|P(\epsilon, \widehat{C})| \le \sum_{p \in P_r} |P(\epsilon/r, C_p)| \le r \cdot \frac{\epsilon}{r} n = \epsilon n.$$

Since $|P(\epsilon, \widehat{C})|$ is an integer, $|P(\epsilon, \widehat{C})| \leq \lfloor \epsilon n \rfloor$. Thus, all players in $P(\epsilon, \widehat{C})$ are removed by VICTIMIZE with probability at least $1 - (\epsilon/r)^t \geq 1 - e^{-t}$.

It remains necessary to calculate the query complexity. In PREASSIGNS, DEPOSIT is called r times and needs

$$O\left(r \cdot \frac{rt}{\epsilon} \ln \frac{r}{\epsilon} \cdot \left(\ln \frac{r}{\epsilon}\right)^2\right) = O\left(\frac{r^2 t}{\epsilon} \left(\log \frac{r}{\epsilon}\right)^3\right)$$

time. DC for k players can be done in $O(k \log k)$-time if a cake is continuous. However, $\mathrm{DC}(\{p_1, \ldots, p_k\}, C_{p_1} \cup \cdots \cup C_{p_k})$ in line 07 of PREASSIGNS treats $C_{p_1} \cup \cdots \cup C_{p_k}$, which may be separated into at most r continuous pieces. One query on a cake consisting of k continuous pieces is simulated by k queries on the continuous parts. Hence, the query complexity of this DC is $O(r^2 \log(r^2)) = O(r^2 \log r)$. Therefore, the time-complexity of PREASSIGNS is $O((r^2 t/\epsilon)(\log(r/\epsilon))^3 + r^2 \log r) = O((r^2 t/\epsilon)(\log(r/\epsilon))^3)$.

For the completion part, $\mathrm{DC}(Q, C - \widehat{C})$ in COMPLETION is dominant. $C - \widehat{C}$ may be separated into at most $r + 1$ continuous parts. Thus, the query complexity of DC (and the completion part) is $O(rn \log(rn))$. ◀

5 Summary

Herein, we considered a way to solve the cake-cutting problem in sublinear time. For this purpose, we introduced the concept of "ϵn victims," and presented the following framework. In the first (preassigning) part, we preassign portions to $r = o(n)$ players in $o(n)$ time. Then, in the second (completion) part, we assign portions to the remaining $n - r$ players except for the ϵn victims in polynomial-time. (Note that the second part clearly requires $\Omega(n)$-time.) Within this framework, we presented two types of algorithms. In the first, the preassigned players cannot be designated, while in the second, they can be.

For our future work, it remains necessary to show nontrivial lower-bounds. For example, we have not yet proven that only one victim is needed to preassign sublinear players in sublinear-time. Since numerous variations may be considered in our framework, the ability to make extended algorithms is also an attractive subject.

Acknowledgement. We would like to thank Assistant Professor Yuichi Yoshida of the National Institute of Informatics for his valuable advice. We are also grateful for the "Algorithms on Big Data" project (ABD14) of CREST, JST, the ELC project (MEXT KAKENHI Grant Number 24106003), and JSPS KAKENHI Grant Numbers 24650006 and 15K11985, through which this work was partially supported.

References

1 I. Benjamini, O. Schramm, and A. Shapira: Every minor-closed property of sparse graphs is testable, Proc. STOC 2008, ACM, 2008, pp. 393–402.

2 S. J. Brams and A. D. Taylor: An envy-free cake division protocol, American Mathematical Monthly .

3 J. Edmonds and K. Pruhs: Cake cutting really isn't a piece of cake, Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms, Vol. 7 (2006).

4 J. Edmonds and K. Pruhs: Balanced Allocations of Cake, Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science, pp. 623–634 (2006).

5 S. Even and A. Paz: A note on cake cutting, Discrete Applied Mathematics, Vol. 7, pp. 285–296 (1984).

6 O. Goldreich (Ed.): Property Testing – Current Research and Surveys, LNSC 6390, 2010.
7 O. Goldreich and D. Ron: Property testing in bounded degree graphs: Proc. STOC 1997, 1997, pp. 406–415.
8 O. Goldreich, S. Goldwasser, and D. Ron: Property testing and its connection to learning and approximation: Journal of the ACM, Vol. 45, No. 4, July, 1998, pp. 653–750.
9 A. Hassidim, J. A. Kelner, H. N. Nguyen, and K. Onak: Local graph partitions for approximation and testing, Proc. FOCS 2009, IEEE, pp. 22–31.
10 H.Ito, S. Kiyoshima, and Y. Yoshida: Constant-time approximation algorithms for the knapsack problem, Proceedings of the 9th Annual Conference on TAMC, pp. 131–142 (2012).
11 H. Ito, S. Tanigawa, and Y. Yoshida: Constant-Time Algorithms for Sparsity Matroids, Proc. ICALP (1), LNCS 7391, 2012, pp. 498–509.
12 I. Newman and C. Sohler: Every property of hyperfinite graphs is testable, Proc. STOC 2011, ACM, 2011, pp. 675–784.
13 R. Levi and D. Ron: A quasi-polynomial time partition oracle for graphs with an excluded minor, Proc. ICALP 2013 (1), LNCS, 7965, Springer, 2013, pp. 709–720.
14 J. Robertson and W. Webb: Cake-Cutting Algorithms: Be Fair If You Can, A. K. Peters (1998).
15 H. Steinhaus: The Problem of fair division, Econometrica, Vol. 16, pp. 101–104 (1948).
16 Y. Yuichi: A characterization of locally testable affine-invariant properties via decomposition theorems, Proc. STOC 2014, ACM, 2014, pp. 154–163.

Threes!, Fives, 1024!, and 2048 are Hard

Stefan Langerman*[1] and Yushi Uno[2]

1 **Département d'informatique, Université Libre de Bruxelles, ULB CP 212, avenue F.D. Roosevelt 50, 1050 Bruxelles, Belgium**
`stefan.langerman@ulb.ac.be`

2 **Department of Mathematics and Information Sciences, 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

We analyze the computational complexity of the popular computer games Threes!, 1024!, 2048 and many of their variants. For most known versions expanded to an $m \times n$ board, we show that it is NP-hard to decide whether a given starting position can be played to reach a specific (constant) tile value.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, G.2 Discrete Mathematics , F.1.2 Modes of Computation

Keywords and phrases algorithmic combinatorial game theory

Digital Object Identifier 10.4230/LIPIcs.FUN.2016.22

1 Introduction

Threes! [13] is a popular puzzle game created by Asher Vollmer, Greg Wohlwend, and Jimmy Hinson (music), and released by Sirvo for iOS on January 23, 2014. The game received considerable attention from players, game critics and game designers. Only a few weeks after its release, an Android clone *Fives* appeared, and then an iOS clone *1024!* with slightly modified rules. Shortly after, two open source web game versions, both called *2048* were released on github on the same day, one by Saming [12], the other by Gabriele Cirulli [5]. Since then over a hundred new variant have been catalogued [9]. In December 2014, Threes! received the Apple Game of the Year and the Apple Design award.

2048 We first describe Cirulli's 2048 [5] (or just 2048 for short) which has a slightly simpler set of rules (see Fig. 2). The game is played on a 4×4 square grid *board*, consisting of 16 *cells*. During the game, each cell is either empty or contains a *tile* bearing a *value* which is a power of two. When the game begins, a (random) starting *configuration* of tiles is placed on the board. Then, in every turn, one plays a move by indicating one of four directions, up, down, left or right, and then each numbered tile moves in that direction, either to the boundary of the board (a *wall*) or until it hits another tile. When two tiles of value K hit, they merge to become a single tile of value $2K$. If three or more tiles with the same value hit, they merge two by two, starting with the two closest to the wall in the direction of the move. If no tile can move in some direction (e.g., all tiles touch the wall), then that move is *invalid*. After each turn, a new tile of value 2 or 4 appears in

* Directeur de Recherches du F.R.S.-FNRS.

© Stefan Langerman and Yushi Uno;
licensed under Creative Commons License CC-BY
8th International Conference on Fun with Algorithms (FUN 2016).
Editors: Erik D. Demaine and Fabrizio Grandoni; Article No. 22; pp. 22:1–22:14
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

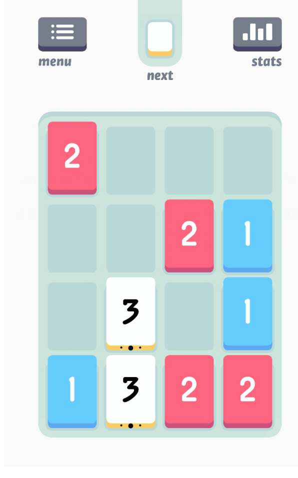


Figure 1 Threes!

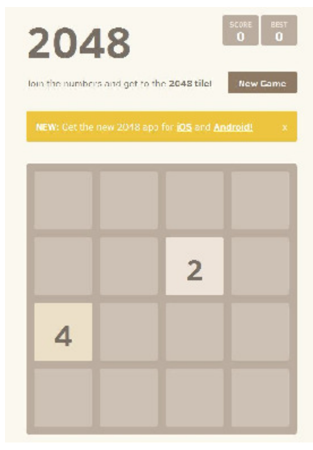


Figure 2 The game 2048: a board and one of its initial configuration.

Figure 3 A forbidden (game over) configuration.

a random empty cell. The objective of the game is to make a tile of value 2048, and/or to maximize the *score* defined as the sum of all new tiles created by merges during the game. If during the game, the board is completely filled and no move is valid, then the game is over and the player loses. We call such a configuration *forbidden* (Fig. 3).

Threes! The tiles in Threes! have values 1, 2, and $3 \cdot 2^i, i \geq 0$. The tiles 1 and 2 combine to form tile 3, and tiles of value $K \geq 3$ combine to form tile $2K$. Another important difference is that when performing a move, all tiles move in the corresponding direction by at most one cell instead of moving as far as possible. For example during a left move, a tile next to the left wall (if any) doesn't move (we say it is *blocked*). Then looking at the tiles from left to right, tiles immediately to the right of one that is blocked will either be blocked as well or will move left to merge with the blocked one if they can combine. A tile next to one that moves or next to an empty space will move one cell to the left. New tiles appear according to an unknown rule, which seems to change depending on the version of the game. It seems to be always of low value and on a cell next to a wall. At the end of the game, the score is computed by totalling 3^{i+1} points for each tile of value $3 \cdot 2^i$ on the board.

Fives was the first clone of Threes! for Android devices. Its rules are nearly identical to Threes!, except that the base tiles have values 2 and 3 which combine to form tile 5, all other tiles have values $5 \cdot 2^i, i \geq 0$.

1024! The main difference with 2048 is that there is a fixed block in the middle of the board that doesn't move during the game. A new tile appears after each turn at a random location on the board (not necessarily next to a wall). The goal is to reach 1024.

Saming's 2048 Just as in Cirulli's 2048, tiles are powers of two, however the tiles move according to slightly different rules. During a left move for example, tiles are considered in each row from right to left. A tile t will only move if its left neighboring cell is empty or of identical value. If there is no tile left of t, then t is moved to a cell adjacent to the left wall. Otherwise let s be the next tile left of t. If t and s are of identical values, the two cells are merged (and the value doubled), and the merged tile does not move this turn. Otherwise, the tile t stops just to the right of s. A new tile of value 2 or 4 is inserted in a random empty cell at the end of the move.

Figure 4 Deterministic 2048.

Det2048 This version is identical to 2048 except that a new tile always appears in the first empty cell (leftmost, then topmost) and its value is always 2 . In its initial configuration, only a single tile 2 is placed in the upper left cell (Fig. 4).

Fibonacci In this popular version, the tiles have values from the Fibonacci sequence, and only tiles of successive values in the sequence are combined.

Other than those, the most natural variants use larger boards, and set higher goal tile values.

The goal of this paper is to determine the computational complexity of Threes!, 2048 and many of their variants. We follow the usual offline deterministic model introduced by Demaine et al. for the videogame Tetris [3]. Given an initial configuration of tiles in an $m \times n$ board, we assume the player has full knowledge of the new pieces that will be added to the board after each move[1]. We prove that even with offline deterministic knowledge (and in all variants listed above), it is NP-hard to optimize several natural objectives of the game:

- maximizing the largest tile created (Max-Tile),
- maximizing the total score (Max-Score), and
- maximizing the number of moves before losing the game (Max-Move).

We show in fact that all three problems are inapproximable. The decision problem for Max-Tile is already NP-hard for a constant tile value, where the constant depends on the variant of the game considered. On the other hand, Max-Move is clearly fixed-parameter tractable (FPT) [7], that is, determining if k moves can be performed without losing takes only $O(4^k mn)$ time (since there are only 4 moves possible at every step). Likewise, every merge increases the score by at least 4, and so the number of moves to achieve score x is at most $x/4$, therefore determining if score x can be achieved takes only $O(4^{x/4}mn)$ and Max-Score is also FPT.

Related works. The tractability of computer games falls under the larger field of *Algorithmic Combinatorial Game Theory* which has received considerable interest over the past decade. See Demaine and Hearn [6] for a recent survey. Block pushing and sliding puzzles are probably the most similar to the games studied here, a notable difference being that here (i) new tiles are (randomly) inserted after each move and (ii) a merging mechanism reduces the number of tiles at each step. Without these differences, the game would be nearly identical to the *Fifteen puzzle* and its generalizations, which interestingly can be solved very efficiently.

[1] We ignore for now the issue of representing the position of the new tiles, which might depend on which cells of the board are empty, and thus on previous moves. As we will see, this will have little influence on the main results.

In a blog post [4], Christopher Chen proved that 2048 is in NP, but for a variant where no new tile is inserted after each move. More recently, two articles have appeared on arXiv with the aim of analyzing the complexity of Threes! and 2048. The first one [8] notes 2048 is FPT(as discussed above) and claims PSPACE-hardness of 2048 by reduction from Nondeterministic Constraint Logic. However Abdelkader et al. [1] noted several issues with that reduction. For instance, in order for their gadgets to function properly, they need to modify the way tiles are moved and inserted during the game. In particular, they allow tiles to be inserted by the game at specific places in the middle of rows and columns in order to maintain an invariant base pattern. Furthermore, the goal tile value in the reduction is a (rather large) function of the input size. In an attempt to resolve these issues, Abdelkader et al. [1, 2] analyze the variant studied by Chen, in which no new tiles are generated during the game. For that case, they show that for 2048, it is NP-complete to decide if a specific (constant) tile value can be reached.

In the present paper, we analyze Threes! and 2048, where new tiles appear and tiles move and merge exactly as they do in the original games. Our proofs are easily extended to most existing variants of the game. We also prove inapproximability results for all these games, and show most of them are in NP.

2 Definitions

The simplest version of these games to describe is probably Det2048, where after each move a tile 2 appears in the first empty cell of the board (in lexicographic order, leftmost, then topmost).

> MAKE-T-DET2048
> Instance: An $m \times n$ board with an initial configuration of tiles, each of which has for value a power of 2, and a number T, where $T = 2^c$ with some constant c. At the end of every turn, a new tile 2 appears in the first empty cell in lexicographic order.
> Question: Can one make T from the given configuration by a sequence moves (up, down, left and right)?

However in the original game, both tiles 2 and 4 can appear, at a location determined by the game. Since we consider an offline model, the value and location of the new tiles should be provided in the input. But while encoding the value of the tile is easy, its location does not have such a natural representation, because the new tile can only be inserted in an empty cell of the board, and the set of empty cells depends on the previous moves in the game. One could conceive several reasonable encodings (e.g., for each new tile, coordinates (x, y) such that the new tile should be placed in the closest/lexicagraphically first empty cell from cell (x, y)). To make our results as general as possible, we just assume the *location* information is encoded in constant space, and the game uses that information to place the new tile.

> MAKE-T
> Instance: An $m \times n$ board with an initial configuration of tiles, each of which has for value a power of 2, a number T, where $T = 2^c$ with some constant c, and the sequence of values (2 or 4) and location of the new tiles to be placed by the game at the end of every turn.
> Question: Can one make T from the given configuration by a sequence moves (up, down, left and right)?

In this setting, the original game 2048 is as MAKE- T with $m = n = 4$ and $T = 2048 = 2^{11}$ ($c = 11$).

It will be useful to denote some of the variants of MAKE- T by appending qualifiers to their name. For example, in the variant MAKE- T ONLY- 2 , only tiles 2 appear after each move. In the DETERMINISTIC variant, new tiles always appear in the lexicographically first empty cell, and so MAKE- T ONLY- 2 DETERMINISTIC is exactly MAKE- T -DET2048.

For the purpose of analyzing the complexity of the game, one might argue that the random nature of the original game might make the game more (or less) tractable. To up the ante, some variants of the game, such as *Evil2048* [11], use a heuristic to guess the worst possible location and value for the new tile at the end of every move. On the other hand, for our hardness proofs it might make sense to define a MAKE- T -ANGEL version, where the player can decide the value and location of the new tile after every move. However, as we will see, none of this makes the game significantly easier, as our NP-hardness proofs and inapproximability results hold for all variants mentioned, including ANGEL version.

We will also define optimization problems MAX-TILE, MAX-SCORE, and MAX-MOVES, whose objective is to maximize the value of the maximum tile created in the game, the total score of the game, defined as the sum of the values of all tiles created by merges, and the number of moves played before losing the game (reaching a forbidden configuration). These will be discussed in more detail in the section on inapproximabililty.

Finally, variants using different merging and movement rules, such as Threes! or Fibonacci will be defined and discussed after the main NP-hardness proof.

3 NP

In a blog post [4] Christopher Chen showed that 2048 is in NP. However to simplify the proof, they assume no new piece gets added to the board after a move. It turns out the proof for the regular game (ONLY- 2 version) is not much harder.

The complexity analysis of every problem depends on a reasonable representation of the input. We here assume the input is provided in the form of b, the size of the board, and a list L of tiles present on the board at the beginning of the game. Thus the input size is $\log b + |L|$.

▶ **Lemma 1.** *For any constant value T,* MAKE- T ONLY- 2 *is in* NP.

Proof. If the board is of size $b \times b$, then the maximum total value of all the tiles on the board without ever reaching T is $\leq Tb^2/2$. Since every move adds a tile with value 2 , the total number of moves without reaching T is $\leq Tb^2/4$. If the number of tiles in the starting configuration is $\geq b$, then so is the input size, and the maximum number of moves reaching T, that is, the size of any yes certificate, is polynomial in the input size. If the number of tiles in the starting configuration is $< b$, then by the pigeonhole principle there is an empty row. Therefore playing down repeatedly will eventually, and repeatedly add new 2 tiles in that row which will accumulate to a single tile of value $2^b \geq T$ for b large enough. ◀

For the more general version without the ONLY- 2 restriction, it is plausible that a similar strategy would work. An interesting question is whether the problem is still in NP if T is not a constant. The difficulty here is that the size of the input could be as small as $\log T$, and so number of moves would be exponential in that. A good first step would be to settle the question of what is the maximum tile value achievable on a $b \times b$ board in the game Det2048. For now, the highest value, even on a 4×4 board is unknown, the highest value was found using a heuristic algorithm [10].

4 NP-hardness

We will show NP-hardness of Make-T by reduction from 3SAT.

> 3SAT
> Instance: Set U of variables, collection C of clauses over U such that each clause $c \in C$ has $|c| = 3$.
> Question: Is there a satisfying truth assignment for C?

▶ **Theorem 2.** Make-T *is* NP*-hard for any fixed* T *greater than 2048.*

Proof. Reduction from 3SAT. From an arbitrary instance of 3SAT with n variables and m clauses, we construct a Make-T instance. The construction will ensure that only tiles of value 2 may be merged (into 4 tiles) except for two tiles of value $T/2$ that can be combined to obtain the target value T at the end of the game if and only if the 3SAT instance is satisfiable.

In order to facilitate the analysis of the game, the instances produced by the reduction will start with all cells of the board filled with tiles, and maintain this invariant after each move (hereafter named *fullness* invariant). In order to achieve this, we ensure that at most one pair of 2 tiles is adjacent on the board at all times (*one-move* invariant) and no other pair of identical tiles ever become adjacent during the game until the very last move. This forces the player to make a binary choice: left/right or up/down. In this manner, at the end of each move, only one cell next to a wall is freed, and so the game (in every known variant) will have to place a new tile in that exact cell.

We explain our construction by specifying the locations where tiles 2 are placed. Since we always construct gadgets by putting tiles 2 in pairs, we denote this by a pair of two 2D points $(\cdot, \cdot)$. In the subsequent figures, they will be represented by black dots on a 2D lattice plane. The rest of the board will be filled with a pattern of tiles that will prevent any accidental merges.

Sketch. The construction has three parts: the *variable* gadgets, the *literal* gadgets, and the *clause checking* gadgets. We place each variable gadget in a rectangle below the x axis in distinct rows and columns (we use negative y coordinates for ease of notation). The clauses will each take up 12 rows above the x axis, 4 for each literal. Each literal will lie in 4 of the rows of its clause and 3 of the columns of its variable. The variable gadgets will cause vertical (down) shifts in their columns, which will be transformed into horizontal (left) shifts in the rows of corresponding clauses by the literal gadgets. Finally, the clause checking gadgets to the right of the board will check, for each clause, that at least one of its rows has been shifted.

Base Pattern. We start the construction by filling the board with the repeating *base pattern* shown in Fig. 5. Assuming the bottom left cell is numbered $(0, 0)$, cell (i, j) of the base pattern contains the tile $2^{3(i \mod 3)+(j \mod 3)+3}$. See Fig. 5.

The gadgets replace some of the cells by the tile 2. This will cause some shifts in rows and columns during the game as those tiles become adjacent. But because of the one-move invariant, at most one row or column shifts in each move. In order to avoid accidental merges between tiles of the base pattern, gadgets will be constructed in such a way that no row or column will ever be shifted more than once (avoiding unwanted merges between new tiles appearing in the same cell). Second, we will adjust the size of the board so that gadgets are

64	128	256	64	128	256	64	128
8	16	32	8	16	32	8	16
512	1024	2048	512	1024	2048	512	1024
64	128	256	64	128	256	64	128
8	16	32	8	16	32	8	16
512	1024	2048	512	1024	2048	512	1024
64	128	256	64	128	256	64	128
8	16	32	8	16	32	8	16

Figure 5 Base pattern.

at a distance of at least 3 from the walls, to avoid new tiles to interfere with the 2 tiles of the gadgets. Furthermore, we will place gadgets in such a way that no two adjacent columns will ever be shifted. Likewise, we will ensure that no two adjacent rows will ever be shifted, except in one place in the clause checking gadget where the one-move invariant will have to be argued more carefully.

Ignoring this last case for now, notice that a tile can only be offset by one position horizontally and one position vertically during the entire game. Two tiles of same value are adjacent if the difference between their coordinates are $(0, 1)$ or $(1, 0)$, and any identical pair of tiles that are not on the same row or the same column start with coordinate difference at least $(3, 3)$, and thus can never become adjacent. If two identical tiles are on the same column, their vertical distance, starting at 3, would have to be reduced twice (using two opposite vertical shifts) in order for them to be at distance 1. However, since adjacent columns cannot be shifted in one game, this can only happen if they are at horizontal distance 2 at some point during the game, but since they started at horizontal distance 0, making that happen would already spend the 2 horizontal moves for those two tiles, making it impossible to bring them back close enough to become adjacent. The same argument applies to two identical tiles on the same row at distance 3.

Finally, consider the special case occurring in the clause checking gadget, where two adjacent rows are shifted. A similar case analysis will show that identical tiles on the same row could become adjacent, but only if the last move is vertical. A careful inspection of the gadget will reveal that no vertical move will occur within the shifted portion of those rows after the adjacent horizontal shift occurs.

Variables. For each variable gadget, we reserve 6 rows of the board, and 3 columns for each clause in which that variable appears. Assume, without loss of generality, that every variable appears at least once in the positive form (otherwise negate it).

Let k_i^+ and k_i^- be the numbers of clauses containing x_i and $\overline{x_i}$, respectively. We define the offset coordinates for each variable by

$$\begin{cases} X_0^{\mathrm{V}} = 0, \\ X_i^{\mathrm{V}} = X_{i-1}^{\mathrm{V}} + 3(k_i^+ + k_i^-) + 7 \quad (1 \le i \le n) \end{cases} \quad \text{and} \quad Y_i^{\mathrm{V}} = -6i \quad (0 \le i \le n-1).$$

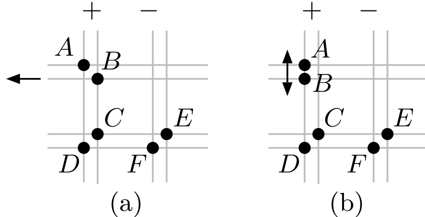


Figure 6 Variable gadget.

Now for each variable x_i ($i = 1, \dots, n$) we construct a gadget as follows. For a choice of true or false, we put a pair of tiles 2 at

$$(A_i^{\mathrm{V}}, B_i^{\mathrm{V}}) = \big((X_{i-1}^{\mathrm{V}} + 1, Y_{i-1}^{\mathrm{V}} + 1), (X_{i-1}^{\mathrm{V}} + 2, Y_{i-1}^{\mathrm{V}})\big) .$$

For the false part we place two pairs of tiles 2 at coordinates

$$(C_i^{\mathrm{V}}, D_i^{\mathrm{V}}) = \big((X_{i-1}^{\mathrm{V}} + 2, Y_{i-1}^{\mathrm{V}} - 2), (X_{i-1}^{\mathrm{V}} + 1, Y_{i-1}^{\mathrm{V}} - 3)\big) ,$$
$$(E_i^{\mathrm{V}}, F_i^{\mathrm{V}}) = \big((X_{i-1}^{\mathrm{V}} + 3k_i^+ + 5, Y_{i-1}^{\mathrm{V}} - 2), (X_{i-1}^{\mathrm{V}} + 3k_i^+ + 4, Y_{i-1}^{\mathrm{V}} - 3)\big) .$$

So in the variable gadget for x_i, six tiles of value 2 are placed as shown in Fig. 6(a) in general. The gadget is *activated* by pulling the row of B_i left (i.e., by merging a pair of adjacent 2 on the row of B_i to the left). Now (see Fig. 6(b)), the tiles A_i and B_i become adjacent and on the same column. At this point the player has the choice to move this column containing A_i down (positive assignment to x_i and to make each clause containing literal x_i true), or up (negative assignment to make each clause containing literal $\overline{x_i}$ true). If the player chooses to move up, then D_i moves up one cell and becomes adjacent to C_i, allowing the player to move left, causing E_i to move one cell left. The tile E_i is now adjacent and on the same column as F_i and the column of F_i can be moved down. Because there are no 2 tiles below the variable gadget or to its left, any sequence of moves other than this one will cause the game to end.

Thus the x_i variable gadget has for effect to move down either the column of A_i (true), or that of F_i (false). This choice will be propagated through each corresponding literal gadgets.

Literals. For literals in clauses we introduce the following coordinates:

$$\begin{cases} X_i^{\mathrm{L}} = X_i^{\mathrm{V}} \quad (0 \le i \le n), \\ X_{n+j}^{\mathrm{L}} = X_{n+j-1}^{\mathrm{L}} + 25 \quad (1 \le j \le m) \end{cases} \quad \text{and} \quad Y_j^{\mathrm{L}} = 12(j-1) + 4 \quad (1 \le j \le m).$$

For variable x_i, suppose its positive literals x_i appear in the p_k-th position of the j_k-th clause ($p_k \in \{0, 1, 2\}$; $k = 1, \dots, k_i^+$; $1 \le j_1 < \cdots < j_{k_i^+} \le m$). Remember that setting x_i to true will shift column $X_{i-1}^V + 1$ down. The gadget for the first positive literal x_i will receiving this vertical activation, shift one of its rows left and propagate the down move to activate the next literal. For this, we put two pairs of tiles 2 at

$$(A_{j_k,p_k}^{\mathrm{L}}, B_{j_k,p_k}^{\mathrm{L}}) = \big((X_{i-1}^{\mathrm{L}} + 3(k-1) + 1, Y_{j_k}^{\mathrm{L}} + 4p_k + 1), (X_{i-1}^{\mathrm{L}} + 3(k-1) + 2, Y_{j_k}^{\mathrm{L}} + 4p_k)\big) ,$$
$$(C_{j_k,p_k}^{\mathrm{L}}, D_{j_k,p_k}^{\mathrm{L}}) = \big((X_{i-1}^{\mathrm{L}} + 3(k-1) + 4, Y_{j_k}^{\mathrm{L}} + 4p_k - 1), (X_{i-1}^{\mathrm{L}} + 3(k-1) + 5, Y_{j_k}^{\mathrm{L}} + 4p_k)\big) .$$

Likewise, when $k_i^- > 0$, negative literals $\overline{x_i}$ appear in the p_k-th position of the j_k-th clause ($p_k \in \{0, 1, 2\}$; $k = 1, \dots, k_i^-$; $1 \le j_1 < \cdots < j_{k_i^-} \le m$). For receiving and propagating

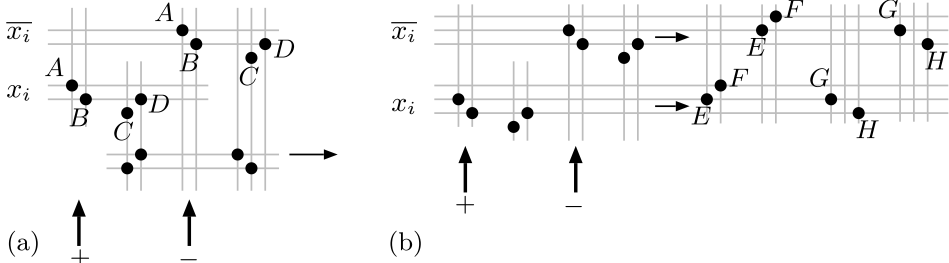


Figure 7 Literal gadget.

vertical activations we put two pairs of tiles 2 at

$$(A^{\rm L}_{j_k,p_k}, B^{\rm L}_{j_k,p_k}) = \big((X^{\rm L}_{i-1}+3(k^+_i+k)+1, Y^{\rm L}_{j_k}+4p_k+1), (X^{\rm L}_{i-1}+3(k^+_i+k)+2, Y^{\rm L}_{j_k}+4p_k)\big)\ ,$$
$$(C^{\rm L}_{j_k,p_k}, D^{\rm L}_{j_k,p_k}) = \big((X^{\rm L}_{i-1}+3(k^+_i+k)+4, Y^{\rm L}_{j_k}+4p_k-1), (X^{\rm L}_{i-1}+3(k^+_i+k)+5, Y^{\rm L}_{j_k}+4p_k)\big)\ .$$

See Fig. 7(a).

The horizontal (right) shifts for both positive and negative literals has for effect to move 2 tiles placed to the right of the board, for use in the clause checking gadgets. We add those pairs of tiles 2 at

$$(E^{\rm L}_{j_k,p_k}, F^{\rm L}_{j_k,p_k}) = \big((X^{\rm L}_{n+j_k-1}+6p_k+1, Y^{\rm L}_{j_k}+4p_k+1), (X^{\rm L}_{n+j_k-1}+6p_k+2, Y^{\rm L}_{j_k}+4p_k+2)\big)\ ,$$
$$(G^{\rm L}_{j_k,p_k}, H^{\rm L}_{j_k,p_k}) = \big((X^{\rm L}_{n+j_k-1}+3p_k+16, Y^{\rm L}_{j_k}+4p_k+1), (X^{\rm L}_{n+j_k-1}+3p_k+18, Y^{\rm L}_{j_k}+4p_k)\big)\ .$$

See Fig. 7(b).

The final appearance of literals x_i and $\overline{x_i}$ will also be represented by two pairs of tiles like above, but the second pair will cause a vertical shift up which will be propagated to activate the next variable gadget or to activate the clause checking gadgets as shown in Fig. 7(b). This process is described next.

Activate. The first variable gadget is activated by a pair of 2 placed at

$$((-3,0),(-2,0))$$

causing a horizontal shift left for $B^{\rm V}_1$. For subsequent variables, assigning the truth value to the final literal x_i or $\overline{x_i}$ $(1 \le i \le n-1)$ will cause a vertical shift up at column $X^{\rm L}_{i-1}+3(k^+_i-1)+4$ or $X^{\rm L}_{i-1}+3(k^+_i+k^-_i)+4$. Note that if $k^-_i=0$, then it is E^V_i and F^V_i which will cause the vertical shift at that position. We propagate that shift into a horizontal left shift activating variable x_{i+1} using two pairs of tiles 2 at

$$\big((X^{\rm L}_{i-1}+3(k^+_i-1)+4, Y^{\rm V}_{i-1}-7), ((X^{\rm L}_{i-1}+3(k^+_i-1)+5, Y^{\rm V}_{i-1}-6)\big)\ ,$$
$$\big((X^{\rm L}_{i-1}+3(k^+_i+k^-_i)+3, Y^{\rm V}_{i-1}-6), (X^{\rm L}_{i-1}+3(k^+_i+k^-_i)+4, Y^{\rm V}_{i-1}-7)\big)\ .$$

See the bottom four tiles of Fig. 7(a).

After the truth assignments of the literals of the last variable x_n or $\overline{x_n}$, one of the same columns is shifted, but this time down and that shift is propagated to activate the clause checking gadgets using two pairs of tiles 2 at

$$\big((X^{\rm L}_{n-1}+3(k^+_n-1)+4, 12m+5), (X^{\rm L}_{n-1}+3(k^+_n-1)+5, 12m+4)\big)\ ,$$
$$\big((X^{\rm L}_{n-1}+3(k^+_n+k^-_n)+3, 12m+4), (X^{\rm L}_{n-1}+3(k^+_n+k^-_n)+4, 12m+5)\big)\ .$$

Figure 8 Clause checking gadget.

Checking Clauses. For clause checking gadgets we take coordinates as follows:

$$X_j^{\mathrm{T}} = X_{n+j}^{\mathrm{L}} \quad (0 \le j \le m) \quad \text{and} \quad \begin{cases} Y_1^{\mathrm{T}} = 12m + 12, \\ Y_j^{\mathrm{T}} = Y_{j-1}^{\mathrm{T}} + 15 \quad (1 < j \le m). \end{cases}$$

For each clause C_j $(j = 1, \ldots, m)$ the corresponding gadget has for purpose to check that at least one literal of that clause has been set to true. To choose which of the three literals will be checked, we we place the following five pairs of tiles 2 at

$$(A_{j,1}^{\mathrm{T}}, B_{j,1}^{\mathrm{T}}) = \big((X_{j-1}^{\mathrm{T}}+17, Y_j^{\mathrm{T}}-7), (X_{j-1}^{\mathrm{T}}+18, Y_j^{\mathrm{T}}-8)\big)\,,$$

$$\big((X_{j-1}^{\mathrm{T}}+17, Y_j^{\mathrm{T}}+2), (X_{j-1}^{\mathrm{T}}+18, Y_j^{\mathrm{T}}+1)\big)\,,$$

$$(A_{j,2}^{\mathrm{T}}, B_{j,2}^{\mathrm{T}}) = \big((X_{j-1}^{\mathrm{T}} + 20, Y_j^{\mathrm{T}} + 2), (X_{j-1}^{\mathrm{T}} + 21, Y_j^{\mathrm{T}} + 1)\big)\,,$$

$$\big((X_{j-1}^{\mathrm{T}} + 20, Y_j^{\mathrm{T}} + 5), (X_{j-1}^{\mathrm{T}} + 21, Y_j^{\mathrm{T}} + 4)\big)\,,$$

$$(A_{j,3}^{\mathrm{T}}, B_{j,3}^{\mathrm{T}}) = \big((X_{j-1}^{\mathrm{T}} + 23, Y_j^{\mathrm{T}} + 5), (X_{j-1}^{\mathrm{T}} + 24, Y_j^{\mathrm{T}} + 4)\big)\,.$$

The pair $(A_{j,1}^{\mathrm{T}}, B_{j,1}^{\mathrm{T}})$ is activated by shifting the row $Y_j^{\mathrm{T}} - 8$ left. Then either the move up shifts the column $X_{j-1}^{\mathrm{T}} + 17$ up, or the sequence down, left, up shifts the column $X_{j-1}^{\mathrm{T}} + 20$ up, or the sequence down, left, down, left, up shifts column $X_{j-1}^{\mathrm{T}} + 23$ up. Any other sequence of moves ends the game. Note that the column of $A_{j,p}^{\mathrm{T}}$, $p = 1, 2,$ or 3 being shifted up is exactly one column left of the one containing the 2 at $H_{j,p}^{\mathrm{L}}$ at the beginning of the game. If the corresponding literal was set to true in the literal gadget, then the 2 had been shifted left and is now shifted up, bringing $G_{j,p}^{\mathrm{L}}$ and $H_{j,p}^{\mathrm{L}}$ next to each other. The row of $G_{j,p}^{\mathrm{L}}$ can now be shifted left, activating the pair $(E_{j,p}^{\mathrm{L}}, F_{j,p}^{\mathrm{L}})$, and the column of $F_{j,p}^{\mathrm{L}}$ can now be shifted down.

To collect the down shift in the column of $F_{j,p}^{\mathrm{L}}$ for the chosen $p = 1, 2,$ or 3, we place seven pairs of tiles 2 at coordinates

$$(C_{j,1}^{\mathrm{T}}, D_{j,1}^{\mathrm{T}}) = \big((X_{j-1}^{\mathrm{T}} + 1, Y_j^{\mathrm{T}} + 13), (X_{j-1}^{\mathrm{T}} + 2, Y_j^{\mathrm{T}} + 14)\big)\,,$$

$$\big((X_{j-1}^{\mathrm{T}} + 4, Y_j^{\mathrm{T}} + 14), (X_{j-1}^{\mathrm{T}} + 5, Y_j^{\mathrm{T}} + 13)\big)\,, \big((X_{j-1}^{\mathrm{T}} + 4, Y_j^{\mathrm{T}} + 9), (X_{j-1}^{\mathrm{T}} + 5, Y_j^{\mathrm{T}} + 10)\big)\,,$$

$$(C^{\mathrm{T}}_{j,2}, D^{\mathrm{T}}_{j,2}) = \left((X^{\mathrm{T}}_{j-1}+7, Y^{\mathrm{T}}_{j}+10), (X^{\mathrm{T}}_{j-1}+8, Y^{\mathrm{T}}_{j}+11)\right),$$

$$\left((X^{\mathrm{T}}_{j-1}+10, Y^{\mathrm{T}}_{j}+11), (X^{\mathrm{T}}_{j-1}+11, Y^{\mathrm{T}}_{j}+10)\right), \left((X^{\mathrm{T}}_{j-1}+10, Y^{\mathrm{T}}_{j}+6), (X^{\mathrm{T}}_{j-1}+11, Y^{\mathrm{T}}_{j}+7)\right),$$

$$(C^{\mathrm{T}}_{j,3}, D^{\mathrm{T}}_{j,3}) = \left((X^{\mathrm{T}}_{j-1}+13, Y^{\mathrm{T}}_{j}+7), (X^{\mathrm{T}}_{j-1}+14, Y^{\mathrm{T}}_{j}+8)\right).$$

Now the vertical shift of column $F^{\mathrm{L}}_{j,p}$ aligns the 2 tiles of $C^{\mathrm{T}}_{j,p}$ and $D^{\mathrm{T}}_{j,p}$, and the rest of the 2 can be used to propagate the horizontal shift until the row of $C^{\mathrm{T}}_{j,3}$ is shifted left. This is the same row as $B^{\mathrm{T}}_{j+1,1}$ and so activates the next clause checking gadget for $j < m$.

Goal. To make a target number X (> 2048), we place a pair of tiles of value $X/2$ at $\left((X^{\mathrm{T}}_{n+m}+1, Y^{\mathrm{T}}_{m}+8), (X^{\mathrm{T}}_{n+m}+2, Y^{\mathrm{T}}_{m}+7)\right)$. This pair will become adjacent when the last clause checking gadget is successfully played and shifts the row of $C^{\mathrm{T}}_{m,3}$ left.

From the construction, it is clear that the tile T can be created if and only if the given 3SAT formula is satisfiable. The size of the board is $\Theta((n+m)^2)$, and the size of the sequence of new tiles is $\Theta(m+n)$. So this reduction takes polynomial space and polynomial time with respect to the input size $n+m$ of the 3SAT instance. ◀

We illustrate a complete example of our reduction in Fig. 9, where the formula for 3SAT is $f = (x_1 \vee \overline{x_2} \vee x_3) \wedge (x_2 \vee x_3 \vee \overline{x_4}) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_4)$. In this figure, two goal tiles $T/2$ are represented by squares.

5 Inapproximability

It is fairly easy to extend the construction from the previous section to show it is NP-hard to approximate Max-Tile, Max-Score or Max-Moves. For Max-Tile and Max-Score, it would be enough to change the value of the goal pair of tiles to an arbitrarily high number. However one might want to impose that the tiles of the input configuration be all of small value. In that case, we can still show inapproximability by using the *pot of gold* technique.

Note that in the previous construction, if the formula is satisfiable, then the goal tiles will be adjacent in column $X^{\mathrm{T}}_{n+m}+1$. We add tiles:

$$(A^{\mathrm{A}}, B^{\mathrm{A}}) = \left((X^{\mathrm{T}}_{n+m}+1, Y^{\mathrm{T}}_{m}+21), (X^{\mathrm{T}}_{n+m}+2, Y^{\mathrm{T}}_{m}+20)\right).$$

We then extend the board to the left of the first variable gadget by $K = 2^p$ columns, and place tiles of alternating values 8 and 16 on row $Y^{\mathrm{T}}_{m}+20$ at negative x coordinates. On the row $Y^{\mathrm{T}}_{m}+19$, we place tiles of alternating values 32 and 8 (so the tiles 8 are just below the tiles 16 and can't merge.

If the formula is satisfiable (and only then), the player can solve the game as before until she shifts column $X^{\mathrm{T}}_{n+m}+1$ down. One can then shift the row of B^{A} left which aligns all the 8 of that row with the 8 of the row below. A move up now merges all those 8 s into 16 s, and repeatedly shifting right $p = \log K$ times will merge all those tiles into one tile of value $16K$. We can then continue the sequence with $S = 2^q$ 2 appearing in the leftmost cell of row $Y^{\mathrm{T}}_{m}+20$, with $q < K$. The total score is then $\Theta(m+n+K+S)$ and the maximum tile is $\Theta(\max(K,S))$.

The input size in this game is the entire size of the board plus the length of the tile sequence and all tiles are of constant value. The original board size is $\Theta((n+m)^2)$ so the augmented board is of size $\Theta((n+m)(n+m+K))$. The number of moves is $\Theta(n+m+\log K+S)$.

So in the standard game:

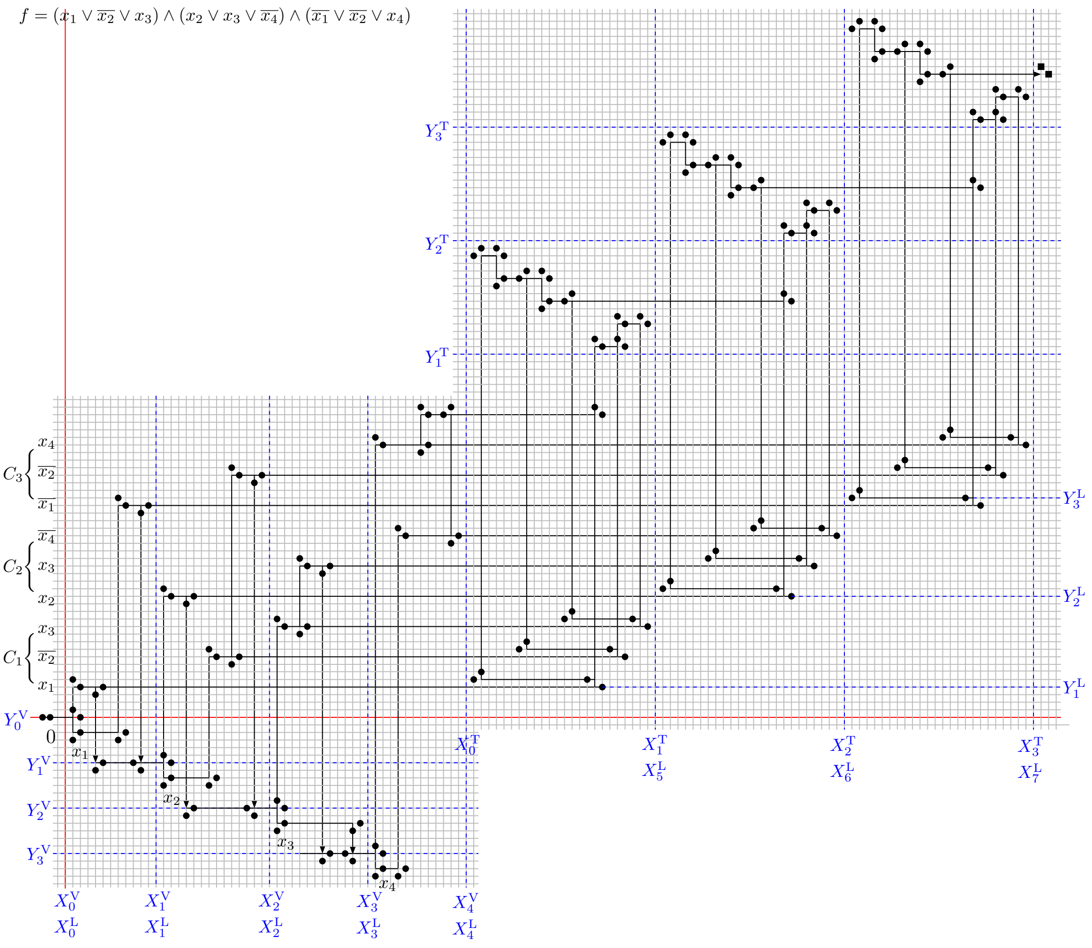


Figure 9 An example of NP-hardness reduction from 3SAT to MAKE-T.

- Taking $K = n + m$, $S = K^3$, the input size is $N = K^3 + \Theta(K^2)$. The maximum tile value is $S = N - O(N^{2/3})$ if the formula is satisfiable, 2048 otherwise. So, it is NP-hard to approximate MAX-TILE within a factor N/c for some constant c.
- Taking $K = n + m$, $S = 2^K$, the input size is $N = 2^K + \Theta(K^2)$. The maximum score is $S = N - O(\log^2 N)$ if the formula is satisfiable, $O(K) = O(\log N)$ otherwise. So it is NP-hard to approximate MAX-SCORE within a factor $o(N/\log N)$.
- Using the same parameters as above, the maximum number of moves is at least $S = N - O(\log^2 N)$ if the formula is satisfiable, $O(K) = O(\log N)$ otherwise. So it is NP-hard to approximate MAX-MOVES within a factor $o(N/\log N)$.

Note the importance of S in the input size N for the standard version of the game. In the game DET2048, however, the sequence of new tiles is implicit and not part of the input. The inapproximability results are then strengthened: all three problems are inapproximable within a factor $o(2^N)$ or $o(2^N/N)$.

6 Variants

Only minor modifications are required to make the NP-hardness reduction work for most known variants of Threes! and 2048. We just describe them for the original game Threes!,

and for the Fibonacci version mentioned in the introduction. The extension of these results to other variants such as Fives, 1024! and Saming's 2048 are immediate and are left as an exercise to the reader.

6.1 Threes!

The reduction for Threes! is nearly identical as for 2048. The easiest way to repeat the proof would be to replace every tile of value 2^a by a tile of value $3 \cdot 2^{a-1}$. A slightly better bound can be obtained in the following manner. Every occurence of tile 2 is replaced by a 3 . The base pattern uses only tiles 1 , and the new tiles added after each move are 1 . Since a 1 can only merge with a 2 , this will ensure the base pattern never causes an unwanted merge. The goal tiles are replaced by two 6 .

Recall that tiles in Threes! move according to slightly different rules (most importantly, every tile stays in place, shifts to an adjacent cell or merges with an adjacent tile in every move). However, because of the fullness and one-move invariants, the result of a move will be the same in Threes! as it was for 2048.

Therefore, an identical proof shows that it is NP-hard to decide if it is possible to achieve tile 12 in Threes!. The inapproximability results for MAX-TILE and MAX-MOVES extend as well. For MAX-SCORE, the situation is even worse, as following the same reduction ending it with a sequence of $S = 2^q$ tiles 3 , we would produce a tile of value $3 \cdot 2^q$ which will produce a score of $3^{q+1} = 3S^{\log_2 3} = \Omega(N^{\log_2 3})$ if the formula is satisfiable, and $O(K) = O(\log N)$ otherwise. Therefore, it is NP-hard to approximate MAX-SCORE in Threes! within a factor $o(N^{\log_2 3}/\log N)$.

6.2 Fibonacci

Denote the i-th Fibonacci number by F_i, that is, $F_1 = F_2 = 1$, and $F_{i+2} = F_{i+1} + F_i$. In the Fibonacci version, tiles merge only if they are adjacent in the Fibonacci sequence.

We modify the reduction so that every occurrence of tile 2 is replaced by a 1 (since $F_1 = F_2 = 1$, they can merge into a 2 when adjacent). The base pattern uses only tiles of value 5, and the new tiles added after each move are 1. Since a 5 can only merge with a 3 or 8, this will ensure the base pattern never causes an unwanted merge. The goal tiles are replaced by 13 and 21. Therefore is NP-hard to decide if it is possible to achieve tile 34. Inapproximability results extend as well.

References

1. Ahmed Abdelkader, Aditya Acharya, and Philip Dasler. On the complexity of slide-and-merge games. *CoRR*, abs/1501.03837, 2015. URL: `http://arxiv.org/abs/1501.03837`, `arXiv:1501.03837`.
2. Ahmed Abdelkader, Aditya Acharya, and Philip Dasler. 2048 without new tiles is still hard. *Proceedings of the 8th International Conference on Fun with Algorithms*, LIPICS volume 49, 2016.
3. Ron Breukelaar, Erik D. Demaine, Susan Hohenberger, Hendrik Jan Hoogeboom, Walter A. Kosters, and David Liben-Nowell. Tetris is hard, even to approximate. *International Journal of Computational Geometry and Applications*, 14(1–2):41–68, 2004.
4. Christopher Chen. 2048 is in NP. `http://blog.openendings.net/2014/03/2048-is-in-np.html`, March 2014.
5. Gabriele Cirulli. 2048. `http://gabrielecirulli.github.io/2048/`, March 2014.

6. Erik D. Demaine and Robert A. Hearn. Playing games with algorithms: Algorithmic combinatorial game theory. In Michael H. Albert and Richard J. Nowakowski, editors, *Games of No Chance 3*, volume 56 of *Mathematical Sciences Research Institute Publications*, pages 3–56. Cambridge University Press, 2009. `arXiv:cs.CC/0106019`.
7. Rodney G. Downey and Michael R. Fellows. *Parameterized complexity*. Springer Heidelberg, 1999.
8. Rahul Mehta. 2048 is (PSPACE) hard, but sometimes easy. *CoRR*, abs/1408.6315, 2014. URL: `http://arxiv.org/abs/1408.6315`, `arXiv:1408.6315`.
9. Phenomist. 2048 variants. `http://phenomist.wordpress.com/2048-variants/`, 2014.
10. QuadmasterXLII. Solve a deterministic version of 2048 using the fewest bytes. `http://codegolf.stackexchange.com/questions/24885/solve-a-deterministic-version-of-2048-using-the-fewest-bytes`, 2014.
11. A.J. Richardson. Evil 2048. `http://aj-r.github.io/Evil-2048/`, March 2014.
12. Saming. 2048. `http://saming.fr/p/2048/`, March 2014.
13. Asher Vollmer, Greg Wohlwend, and Jimmy Hinson. Threes! `http://asherv.com/threes/`, January 2014.

An Arithmetic for Rooted Trees*

Fabrizio Luccio

Dipartimento di Informatica, University of Pisa, Italy
luccio@di.unipi.it

Abstract

We propose a new arithmetic for non-empty rooted unordered trees simply called trees. After discussing tree representation and enumeration, we define the operations of tree addition, multiplication, and stretch, prove their properties, and show that all trees can be generated from a starting tree of one vertex. We then show how a given tree can be obtained as the sum or product of two trees, thus defining *prime trees* with respect to addition and multiplication. In both cases we show how primality can be decided in time polynomial in the number of vertices and prove that factorization is unique. We then define negative trees and suggest dealing with tree equations, giving some preliminary examples. Finally we comment on how our arithmetic might be useful, and discuss preceding studies that have some relations with ours. The parts of this work that do not concur to an immediate illustration of our proposal, including formal proofs, are reported in the Appendix.

To the best of our knowledge our proposal is completely new and can be largely modified in cooperation with the readers. To the ones of his age the author suggests that "many roads must be walked down before we call it a theory".

1998 ACM Subject Classification E.1 Data Structures, G.2.0 [Discrete Mathematics] General, G.2.2 [Discrete Mathematics] Graph Theory

Keywords and phrases Arithmetic, Rooted tree, Prime tree, Tree equation

Digital Object Identifier 10.4230/LIPIcs.FUN.2016.23

1 Basic properties and notation

- We refer to **rooted unordered trees** simply called trees. Our trees are non empty. **1** denotes the tree containing exactly one vertex, and is the basic element of our theory.
- In a tree T, $\boldsymbol{r}(T)$ denotes the root of T; $x \in T$ denotes any of its vertices; n_T and e_T respectively denote the numbers of vertices and leaves. A subtree is the tree composed of a vertex x and all its descendants in T. The subtrees routed at the children of x are called subtrees of x. s_T denotes the number of subtrees of $\boldsymbol{r}(T)$.
- A tree T can be represented as a binary sequences S_T (the original reference for ordered trees is [11]). In our scheme T is traversed in left to right preorder inserting 1 in the sequence for each vertex encountered, and inserting 0 for each move backwards. Then S_T is composed of $2n$ bits as shown in Figure 1, and has a balanced parenthesis recursive structure $1\ S_1\ .\ .\ .\ S_k\ 0$ where the S_i are the sequences representing the subtrees of $\boldsymbol{r}(T)$. The sequences for tree **1** is 10. Note that all the prefixes of S_T have more 1's than 0's except for the whole sequence that has as many 1's as 0's.

Since T is unordered, the order in which the subsequences S_i appear in S_T is immaterial (i.e., in general many different sequences represent T). However a *canonical form* for trees is

* This work was partially supported by MIUR of Italy under PRIN 2012C4E3KT national research project AMANDA – Algorithmics for Massive and Networked DAta.

© Fabrizio Luccio;
licensed under Creative Commons License CC-BY

T

$S_T = 1\ \underbrace{1\ 0}_{S_1}\ \underbrace{1\ 1\ 0\ 1\ 0\ 0}_{S_2}\ \underbrace{1\ 1\ 0\ 0}_{S_3}\ 0$

Figure 1 Tree representation as a binary sequence. S_1, S_2, S_3 represent the subtrees of the root of T.

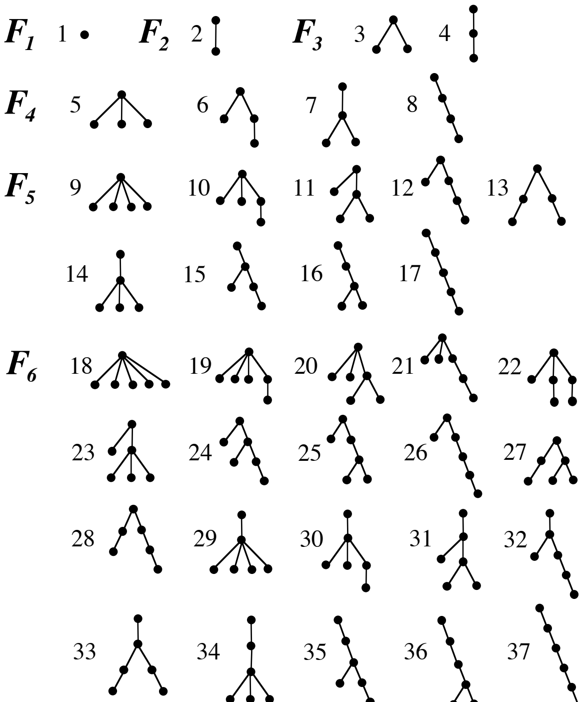


Figure 2 The canonical families of trees $\mathcal{F}_1$ to $\mathcal{F}_6$ and the corresponding tree enumeration.

established so that their sequences will be uniquely determined, and will result to be ordered for increasing values if they are interpreted as binary numbers. To this end the trees are grouped into consecutive families $\mathcal{F}_1, \mathcal{F}_2, \ldots$ as shown in Figure 2, where $\mathcal{F}_i$ contains the trees of i vertices. So the trees are ordered for increasing number of vertices, and inside each family the ordering is determined by the canonical form as follows. Trees and sequences are then numbered with increasing natural numbers (see Appendix A for the sequences of the trees of $\mathcal{F}_1$ to $\mathcal{F}_6$).

- If the sequences are interpreted as binary numbers, for two trees U, T with $n_U < n_T$ we have $S_U < S_T$ because the initial character of each sequence is 1 and S_U is shorter than S_T. This is consistent with the property that the trees of $\mathcal{F}_{n_U}$ precede the trees of $\mathcal{F}_{n_T}$ in the ordering.
- The families $\mathcal{F}_1, \mathcal{F}_2$ contain one tree each numbered 1, 2.
- The ordering of the trees in $\mathcal{F}_{n>2}$ is based on the ordering of the preceding families. Consider the multisets of positive integers whose sum is $n-1$. E.g., for $n = 6$ these multisets are: 1,1,1,1,1 - 1,1,1,2 - 1,1,3 - 1,2,2 - 1,4 - 2,3 - 5 ordered for non-decreasing value of the digits left to right. Each multiset corresponds to a group of consecutive trees in $\mathcal{F}_n$, where the digits in the multiset indicate the number of vertices of the subtrees of the root. For $\mathcal{F}_6$ in Figure 2, multiset 1,1,1,1,1 refers to tree 18; multiset 1,1,1,2 refers to

tree 19; multiset 1,1,3 refers to trees 20 and 21 that have the two trees of $\mathcal{F}_3$ as third subtree, following the ordering in $\mathcal{F}_3$; ...; multiset 2,3 refers to trees 27 and 28; the last multiset 5 refers to trees 29 to 37 whose roots have only one child.

So the first tree in $\mathcal{F}_n$ is the one of height 2 with $n-1$ subtrees of the root of one vertex each and sequence 1 1 0 1 0 1 0 . . . 1 0 0; and the last tree is the "chain" of n vertices and sequence 1 1 . . . 1 0 0 . . . 0. As said the binary sequences representing the trees in $\mathcal{F}_n$ are ordered for increasing values, see Appendix A.

Many of these trees (not necessarily all) of each family $\mathcal{F}_n$ can be generated from the ones in $\mathcal{F}_{n-1}$ using the following:

Doubling Rule DR. From each tree T in $\mathcal{F}_{n-1}$ build two trees T_1, T_2 in $\mathcal{F}_n$ by adding a new vertex as the leftmost child of $\boldsymbol{r}(T)$, or adding a new root and appending T to it as a unique subtree.

For example the four trees of $\mathcal{F}_4$ in Figure 2 can be built by **DR** from the two trees of $\mathcal{F}_3$. The nine trees of $\mathcal{F}_5$ can be built by **DR** from the four trees of $\mathcal{F}_4$, with the exception of tree 13. The twenty trees of $\mathcal{F}_6$ can be built by **DR** from the nine trees of $\mathcal{F}_5$, with the exception of trees 27 and 28. In fact the number of extra trees that cannot be built with **DR** increases sharply with n. Letting f_n denote the number of trees in $\mathcal{F}_n$ we immediately have $f_n \geq 2^{n-2}$ for $n \geq 2$. But a deep analysis [3, 8] has shown that the asymptotic value of this function is much higher, and can be approximated as:

$$f_n \sim 0.44 \cdot 2.96^n \cdot n^{-3/2}. \tag{1}$$

Then the minimum length of the sequences representing the trees of $\mathcal{F}_n$ is given approximately by:

$$\log_2(0.44 \cdot 2.96^n \cdot n^{-3/2}) \sim 1.57\,n - 1.5\log_2 n - 1.19$$

much less than the $2n$ bits of our proposal. We only note that for $n \geq 2$ all the binary sequences representing our trees begin with two 1's and end with two 0's (see the listing in the Appendix A), then these four digits could be removed, leaving a sequence of $2n-4$ bits to represent a tree. We shall see that our representation is amenable at working easily on the trees, so we maintain it, leaving the construction of a shorter efficient coding as a challenging open problem.

An arbitrary tree T can be transformed into its canonical form with Algorithm CF of Figure 3. An elementary analysis shows that the algorithm is correct and each of its steps **1**, **2** can be executed in total $O(n^2)$ time. The algorithm may possibly be improved, however our present aim is just showing that the problem can be solved in polynomial time.

2 Operators and tree generation

Our basic operations are addition (symbol $+$) and multiplication (symbol $\cdot$, or simple concatenation) defined as follows. Referring to Figure 4, let A, B be two arbitrary trees:

Addition. $T = A + B$ is built by merging the two roots $\boldsymbol{r}(A)$, $\boldsymbol{r}(B)$ into a new root $\boldsymbol{r}(T)$. That is the subtrees of A and B (if any) become the subtrees of $\boldsymbol{r}(T)$. We have $A + \mathbf{1} = \mathbf{1} + A = A$.

algorithm CF(T)

1. forany vertex $x \in T$
  count the number of vertices $n_1, \dots n_k$ of its subtrees;
  reorder these subtrees for non decreasing values of the n_i;
  let $G_1, \dots, G_r$ be the groups of subtrees with the same number $g_1, \dots, g_r$
    of vertices, with all $g_i > 2$;
 // reordering is necessary but not sufficient for having T in canonical form
 // the trees in all G_i must be be arranged in canonical order

2. forany $x \in T$, down-top from the vertices closest to the leaves
  forany group $G_i = \{T_1, \dots, T_s\}$
    compute the representing sequences $S_1, \dots, S_s$;
    order $S_1, \dots, S_s$ for increasing binary value;
    permute $T_1, \dots, T_s$ accordingly.

Figure 3 Structure of Algorithm CF for transforming an arbitrary tree T of n vertices in canonical form. CF requires polynomial time in n.

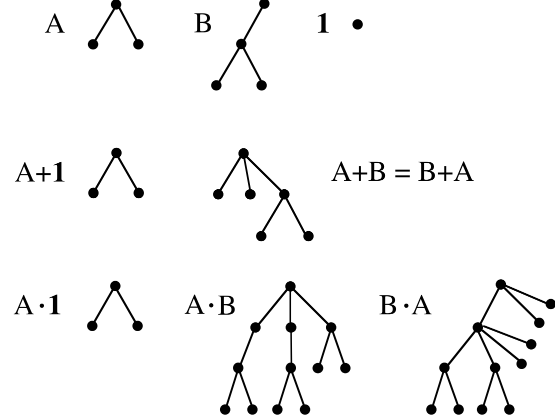


Figure 4 Examples of addition and multiplication.

Multiplication. $T = A \cdot B$ is built by merging $\boldsymbol{r}(B)$ with each vertex $x \in A$ so that all the subtrees of $\boldsymbol{r}(B)$ become new subtrees of x. We have $A \cdot \mathbf{1} = \mathbf{1} \cdot A = A$.

In both operations it is immaterial in which order the subtrees are attached to the new parents. We also define the operation stretch (symbol over-bar) whose interest will be made clear in the following:

Stretch. $T = \bar{A}$ consists of a new root $\boldsymbol{r}(T)$ with A attached as a subtree.

In the notation stretch has precedence over multiplication, and multiplication has precedence over addition. Two propositions immediately follow:

▶ **Proposition 1.** *For $T = A + B$ we have $n_T = n_A + n_B - 1$. For $T = A \cdot B$ we have $n_T = n_A n_B$. For $T = \bar{A}$ we have $n_T = n_A + 1$.*

▶ **Proposition 2.** *Addition is commutative and associative. That is $A + B = B + A$ and $(A + B) + C = A + (B + C)$.*

For a positive integer $k > 1$ and a tree A we can define the product $T = kA$ (not to be confused with the product of trees) as the sum of k copies of A. Due to Propositions 2 and 1, the k copies of A can be combined in any order and we have $n_T = k\, n_A - k + 1$. For any given k, the trees of n_T vertices obtained as a product kA are only f_{n_A}, that is they constitute an exponentially small fraction of all the trees in $\mathcal{F}_{n_T}$. For example the "even" trees (k even) are a small minority among all the trees with the same number of vertices. Similarly we can define the stretch-product $U = k\bar{A}$ as A stretched k times, and we have $n_U = n_A + k$. Again for any given k, the trees of n_U vertices obtained as a stretch-product $k\bar{A}$ are only f_{n_A} and constitute an exponentially small fraction of all the trees in $\mathcal{F}_{n_U}$.

Studying associativity and commutativity in tree multiplication is more complicated. From the definition of multiplication we have with simple reasoning:

▶ **Proposition 3.** *Multiplication is associative.*

That is $(A \cdot B) \cdot C = A \cdot (B \cdot C)$. For a positive integer $k > 1$ and a tree A we can define the power $T = A^k$ as the product of k copies of A. Due to Propositions 3 and 1 the multiplications can be done in any order and we have $n_T = n_A^k$. Again, for any given k, the different trees of n_T vertices obtained as $T = A^k$ are only f_{n_A}.

Multiplication is generally not commutative. For a product $A \cdot B$ we consider two cases $n_A = n_B$ and $n_A > n_B$ ($n_B > n_A$ is symmetric), for which we pose the conditions below. Recall that, for any tree X, e_X and s_X respectively denote the number of leaves of X and the number of subtrees of $\boldsymbol{r}(X)$. For $n_A > n_B$ our conditions are only necessary.

▶ **Proposition 4** (Proof in the Appendix B)**.** *For $n_A = n_B$ we have $A \cdot B = B \cdot A$ if and only if $A = B$.*

▶ **Proposition 5** (Proof in the Appendix B)**.** *For $n_A > n_B$ we have $A \cdot B = B \cdot A$ only if the following conditions are all verified:*

(i) *$n_a/e_A = n_B/e_B$;*
(ii) *B is a proper subtree of A;*
(iii) *if $s_A \geq s_B$ all the subtrees of $\boldsymbol{r}(B)$ must be equal to some subtrees of $\boldsymbol{r}(A)$.*

The trees $3 = A$ and $2 = B$ of Figure 2 do not comply with conditions (i) and (ii) of Proposition 5 and we have $A \cdot B = 22$ different from $B \cdot A = 20$. Commutative products are in fact quite rare. An example with $A \cdot B = B \cdot A$ is shown in Figure 5 where the three conditions of Proposition 5 are verified. In this particular case we have $A = B^2$ hence $A \cdot B = B^3$. Finally multiplication is generally not distributive over addition. From Proposition 1 we can immediately prove:

▶ **Proposition 6.** $(A + B) \cdot C = A \cdot C + B \cdot C$ *if and only if* $C = \mathbf{1}$.

A basic fact about our arithmetic is that all trees can be generated by the single *generator* **1** using addition and stretch.[1] Namely:

- Tree **1** is the generator of itself.
- Assuming inductively that each of the trees in $\mathcal{F}_i$ with $1 \leq i \leq n-1$ can be generated by the trees of the preceding families, then each tree T in $\mathcal{F}_n$ can also be generated. In fact if $\boldsymbol{r}(T)$ has one subtree T_1 then T can be generated as $\bar{T}_1$; if $\boldsymbol{r}(T)$ has $k \geq 2$ subtrees

[1] Stretch been included in the operation set to allow the construction of all trees starting from a finite set of generators. The reader may check that addition and multiplication, or stretch and multiplication, are not sufficient for this purpose. It may be noted that a similar need has arisen in tree algebras (not arithmetic) studied in language theory, e.g. see [7].

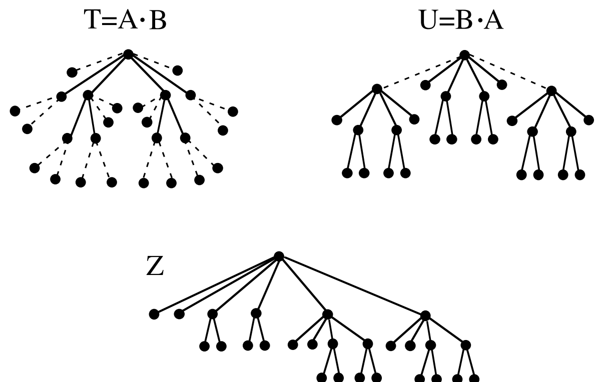


Figure 5 An example of commutative product $A \cdot B = B \cdot A$ for B subtree of A. The two trees are shown in solid and dashed lines, respectively. Z is $A \cdot B$ in canonical form.

$T_1, T_2, \ldots, T_k$ then T can be generated as $U + V$ where U is T deprived of T_k and V is T deprived of $T_1, T_2, \ldots, T_{k-1}$.

3 Prime trees

In the arithmetic of natural numbers the basic operations are addition and multiplication, with $x + 0 = x$ and $x \cdot 1 = x$. Prime numbers under addition have no sense, since all x greater than 1 can be constructed as the sum of two smaller terms other than 0 and x. In our arithmetic for trees, instead, primality occurs in relation with addition and multiplication. In this whole section we refer to trees T with $n_T > 1$. We pose:

▶ **Definition 7.**

(i) T is *prime under addition (*shortly *add-prime)* if can be generated by addition only if the terms are **1** and T (tree **1** has a companion role of integer 0 in $I\!N$).

(ii) T is *prime under multiplication (*shortly *mult-prime)* if can be generated by multiplication only if the factors are **1** and T.

The definition of mult-primality is the natural counterpart of the one of primality in $I\!N$. As it may be expected its consequences are not easy to study. For add-primality, instead, the situation is quite simple. We have:

▶ **Proposition 8** (Proof in the Appendix B). *T is add-prime if and only if $\mathbf{r}(T)$ has only one subtree.*

As a consequence of Proposition 8 deciding if a tree is add-prime is computationally “easy". From Proposition 8 , and from the construction given in the **DR** rule we have:

▶ **Proposition 9.** *For $n \geq 2$ the number of add-prime trees is f_{n-1}.*

From Equation (1) we have: $f_{n-1}/f_n \to \sim 0.34$ for $n \to \infty$, that is the add-prime trees in $\mathcal{F}_n$ are asymptotically about one third of the total. Each of the remaining *add-composite* (i.e., non add-prime) trees T can be uniquely factorized in s_T factors.

For mult-primality we start with two immediate statements respectively derived from Proposition 1, and from the definition of multiplication for trees with at least two vertices:

▶ **Proposition 10.** *If n is a prime number all the trees with n vertices are mult-prime.*

▶ **Proposition 11.** *If $\boldsymbol{r}(T)$ has only one subtree then T is mult-prime.*

The converse of Propositions 10 and 11 do not hold in our arithmetic. That is if n_T is a composite number or $\boldsymbol{r}(T)$ has more than one subtree, tree T may still be mult-prime. In a sense mult-prime trees are more numerous than primes in $I\!N$. For example out of the twenty trees in $\mathcal{F}_6$ (see Figure 2) only trees 20, 22, 24, and 28 are *mult-composite* (i.e. non mult-prime), as they can be built as $2 \cdot 3$, $3 \cdot 2$, $4 \cdot 2$, and $2 \cdot 4$, respectively.

Since if n_T is prime T is mult-prime, and the problem of deciding if n_T is prime is polynomial in $\log n_T$, deciding if T is mult-prime is straightforward for n_T prime. However the problem is difficult for n_T composite because T may be mult-prime or mult-composite. An algorithm for n_T composite may consist of building all the products $A \cdot B$ and $B \cdot A$ of two trees A, B of a, b vertices respectively for all the factorizations of n_T as $a \cdot b$, and comparing T with these products looking for a match. However this method is impracticable unless n_T is very smal, then we must find a different way to decide mult-primality. To this end consider a property of product trees based on the observation that, if $T = A \cdot B$, all the subtrees of $\boldsymbol{r}(B)$ are also subtrees of $\boldsymbol{r}(T)$. Namely:

▶ **Proposition 12** (Proof in the Appendix B)**.** *Let $T = A \cdot B$ with $A, B \neq \mathbf{1}$, and let Y be a subtree of $\boldsymbol{r}(B)$ with maximum number n_Y of vertices. Then the subtrees of $\boldsymbol{r}(B)$ are exactly the subtrees of $\boldsymbol{r}(T)$ with at most n_Y vertices.*

In the mult-composite tree Z of Figure 5, if the first subtree of $\boldsymbol{r}(Z)$ (containing one vertex) is a subtree of maximal cardinallity of one of the factors, B in this case, then B consists of a root plus the first two subtrees of $\boldsymbol{r}(Z)$. Similarly, if the third subtree of $\boldsymbol{r}(Z)$ is a subtree of maximal cardinality of one of the factors, A in this case, then A consists of a root plus the first four subtrees of $\boldsymbol{r}(Z)$. We pose:

▶ **Notation 13.** *For an arbitrary tree T:* (i) *$G_1, \ldots, G_r$ are the groups of subtrees of $\boldsymbol{r}(T)$ with the same number $g_1, \ldots, g_r$ of vertices, $g_1 < g_2 < \cdots < g_r$;* (ii) *$H_i = \bigcup_{j=1}^{i} G_j$, $1 \leq i \leq r$, i.e. each H_i is the group of subtrees of $\boldsymbol{r}(T)$ with up to g_i vertices.*

Based on Propositions 12 and Notation 131 we can build the primality Algorithm MP of Figure 6 that requires polynomial time in the number of vertices. Since all trees with a prime number n of vertices are mult-prime, MP is intended for testing trees with n composite. However MP works for all trees and can always be applied to avoid a preliminary test for the primality of n.

▶ **Proposition 14.** *Mult-primality of any tree T can be decided in time polynomial in n_T.*

Proof. Proof in the Appendix B, based on the analysis of algorithm MP. ◀

Note that if T is mult-composite Algorithm MP allows to find a pair of factors A, B at no extra cost, with B mult-prime. In fact, if a cycle i of step **3** is completed, the algorithm is interrupted on the **return** statement and the group H_i contains exactly the subtrees of $\boldsymbol{r}(B)$, while the tree Z is reduced to A. In particular B is the last factor of a product of mult-prime trees, with $T = T_1 \cdot T_2 \cdot \cdots \cdot T_k \cdot B$. If Algorithm MP is not interrupted with the **return** statement, all these factors can be detected. As a consequence we have:

▶ **Proposition 15** (Proof in the Appendix B)**.** *Mult-factorization of any tree T is unique.*

Finally note that counting the number of add-prime trees is simple (Proposition 9), but an even approximate count for mult-prime trees is much more difficult. We pose:

▶ **Open Problem.** *For a composite integer n determine the number of mult-prime trees of n vertices.*

```
algorithm MP(T)
1. CF(T);
     // transform T in canonical form with Algorithm CF of Figure 4
2. let H_1, ..., H_r be the groups of subtrees of r(T) as in Notation 1;
3. for 1 ≤ i ≤ r − 1
       copy T into Z;
       traverse Z in preorder
           forany vertex x encountered in the traversal
               if x has all the subtrees of H_i erase these subtrees in Z
               else exit from the i-th cycle;
           return MULT-COMPOSITE;
4. return MULT-PRIME.
```

Figure 6 Structure of Algorithm MP for deciding if a tree T of n vertices is mult-prime.

4 Negative trees, with a window on tree equations

Once addition and multiplication are known, it is natural to define the inverse operations.

We define the **subtraction** $A = T - B$ if and only if all the subtrees of $\boldsymbol{r}(B)$ are also subtrees of $\boldsymbol{r}(T)$. Then A equals T deprived of such subtrees. This is the inverse of the addition $T = A + B$, with $n_A = n_T - n_B + 1$. We have $T - \mathbf{1} = T$.

We define the **division** $A = T/B$ if and only if there exists a subset Ψ of the vertices of T such that each $v \in \Psi$ has exactly the subtrees of $\boldsymbol{r}(B)$, and the tree T' obtained as T deprived of such subtrees has exactly the vertices of Ψ. Then $A = T'$. This is the inverse of the multiplication $T = A \cdot B$, with $n_A = n_T/n_B$. We have $T\ /\ \mathbf{1} = T$.

Also the operation of stretch has an inverse. We define the **un-stretch** (symbol underline) if and only if $\boldsymbol{r}(A)$ has exactly one subtree T, and we pose $\underline{A} = T$. In the notation un-stretch has precedence over multiplication and stretch has precedence over un-stretch.

As negative numbers arose from subtraction in integer arithmetic, the more intriguing concept of negative trees arises here from tree subtractions. We propose the following definition. All the vertices of a tree T are either **positive** (then T is positive) or **negative** (then T is negative), except for the root that is **neutral**. Positive and negative vertices are respectively indicated with a black dot or an empty circlet. The root is also indicated with a black dot. Changing the sign of a tree amounts to changing the nature of all its vertices except for the root. Tree $\mathbf{1}$ is neutral and we have $\mathbf{1} = -\ \mathbf{1}$.

Addition and subtraction between A and B keep their definition with the additional condition that if A is positive and B is negative all the subtrees of $\boldsymbol{r}(B)$ are also subtrees of $\boldsymbol{r}(A)$ or vice-versa, and positive and negative subtrees with identical shape cancel each other out in the result (See Figure 7). Multiplication and division between A and B also keep their definition with the additional condition that if A and B are both positive or both negative the result is positive, otherwise is negative.

At this point we may open a window on **tree equations** whose terms have all the nature of a tree, but integers may appear as multiplicative coefficients or exponents. In a sense they are companions of the Diophantine equations with integers, but the solutions are now required to be trees. We may consider equations of different degrees with different number of variables, ask questions on the existence and on the number of solutions, study

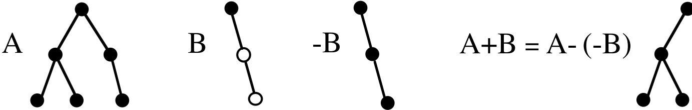


Figure 7 Addition between a positive tree A and a negative tree B.

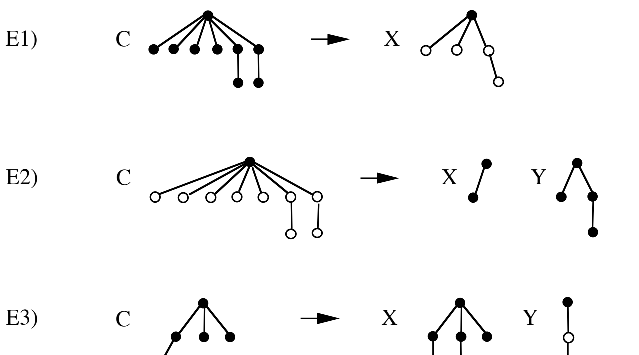


Figure 8 Solution of the tree equations: **E1**: $2X + C = \mathbf{1}$. **E2**: $3X + 2Y + C = \mathbf{1}$. **E3**: $2X + 3Y + C = \mathbf{1}$.

the computational complexity of finding them. In fact we give only some examples, leaving the field completely open.

Denote trees and integers with capital and lower case letters respectively. The simplest equation is linear and has only one unknown X. We put:

$$aX + C = \mathbf{1}, \qquad i.e. aX = -C\,. \tag{2}$$

Equation (2) admits exactly one solution if and only if the s_C subtrees of $\boldsymbol{r}(C)$ can be divided in $g \geq 1$ groups $G_1, \ldots, G_g$ of identical subtrees, where each G_i has cardinality $k_i a$ for $k_i \geq 1$, see example E1 in Figure 8. In this case X has s_C/a subtrees that can be divided in g groups of k_i subtrees identical to the ones of G_i. This solution can be easily built in time polynomial in n_C starting with the transformation of C in canonical form. Note that X and C have opposite sign.

A standard linear tree equation in two unknowns X, Y can be expressed as:

$$aX + bY + C = \mathbf{1}, \qquad i.e. aX + bY = -C\,. \tag{3}$$

This equation is companion of the diophantine equation $ax + by = c$ widely used in modular algebra, that admits an integer solution if and only if c is divided by $gcd(a, b)$. In general equation (3) admits a solution if and only if one of two non trivial conditions 1 and 2 holds, corresponding respectively to trees X, Y of equal sign or of opposite sign as in the examples E2, E3 of Figure 8. These conditions are reported in Appendix C. In both cases the solution can be built in time polynomial in n_C. Referring to the number of vertices, a necessary condition for the solution is that the integer equation $a\,n_X + b\,n_Y = n_C + a + b - 1$ (Case 1), or $a\,n_X - b\,n_Y = n_C + a - b - 1$ (Case 2), has an integer solution in n_X, n_Y, that is $n_C + a + b - 1$, or $n_C + a - b - 1$, is divided by $gcd(a, b)$ as in the examples above.

Higher degree equations are more difficult to handle. For the quadratic tree equation:

$$aX^2 + bY + C = \mathbf{1}, \qquad i.e. aX^2 + bY = -C \tag{4}$$

a necessary condition for the solution is the existence of two integers n_X, n_Y satisfying the algebraic equation $a\,n_X^2 + b\,n_Y = n_C + a + b - 1$ for Y positive, or $a\,n_X^2 - b\,n_Y = n_C + a - b - 1$ for Y negative, a well known NP-complete problem. To find a reasonably interesting approach for deciding whether equation (4) has a solution is left as an open problem.

A "more ambitious" problem can be expressed as:

$$X^n + Y^n = Z^n \tag{5}$$

with the question of deciding wether equation (5) has a tree solution X, Y, Z for any $n \geq 2$. In fact even for $n = 2$ the problem is not simple. Due to Proposition 1 we have the necessary condition $n_X^2 + n_Y^2 - 1 = n_Z^2$ for its solution, i.e. the existence of a "quasi-Pythagorean" triple of integers. In fact such triples exist, as for example $\{4, 7, 8\}$, but the existence of Pythagorean trees with such numbers of vertices is left as an open problem.

5 Possible applications and extensions

While the main purpose of the present study is the one of defining arithmetic concepts outside the realm of numbers, let us briefly discuss what the role of our proposal in applications might be.

Essentially all trees used in computer algorithms are rooted, and different families have been defined among them to deal with particular problems. We do not put any restriction on the tree structure. The trees considered here simply correspond to nested sets as for example hierarchical structures in computer science; or department plans in business organization; or phylogenetic trees in biology, etc. Note that the subtrees are essentially unordered at any vertex, although they must be stored in some standard form to be represented, e.g. following an alphanumeric label order of similar. Or, of course, our canonical order.

Some actions generally required in a hierarchical structure are the following. (i) Join two independent trees A, B to form a new tree T by merging the roots of A, B: e.g. merging two XML files. (ii) Add a new subtree B to the root of a tree T: e.g. adding a new task to a public authority. (iii) Join two independent trees A, B to form a new tree T with A, B subtrees of the root: e.g. joining two phylogenetic trees under a common ancestor. In our arithmetic action (i) is directly represented as $T = A + B$; action (ii) is represented as $T = T + \bar{B}$; and action (iii) is represented as $T = \bar{A} + \bar{B}$. These actions can be respectively undone as: (i') $A = T - B, B = T - A$; (ii') $T = T - \bar{B}$; and (iii') $A = \underline{T - \bar{B}}$, $B = \underline{T - \bar{A}}$. These inverse actions, for example, are basic tools for scheduling multithreaded computations [4].

An important extension of action (ii) is inserting a new subtree A at a given vertex v of T. This is obtained by an iterative operation along the path $\pi = (v_0, v_1, \ldots, v_k)$, from $\boldsymbol{r}(T) = v_0$ to $v = v_k$. Letting $T_0, \ldots, T_k$ be the subtrees rooted at vertices $v_0, \ldots, v_k$, hence $T = T_0$, we set $S_i = T_i - \bar{T}_{i+1}$ for $i = 0, 1, \ldots, k-1$; then we set $T_k = T_k + \bar{A}$; then we set $T_{i-1} = S_{i-1} + \bar{T}_i$ for $i = k, k-1, \ldots, 1$, where $T_0 = T$ gives the transformed tree. A similar operation is required to extract a subtree A at vertex v. Propositions 2 and 3 hold for the subtrees rooted at v, with obvious effects on the whole tree. Other operations can be considered and their representation investigated along the lines above. In particular multiplication may be performed on subtrees only, and even be limited at leaves.

We have cast a glance at tree equations as an invitation to look into this new field. A possible application is in data compression where the form $a_1X_1 + a_2X_2 + \cdots + a_kX_k = C$ can be the basis for representing C through the representation of $X_1, \cdots, X_k$, $a_1, \cdots, a_k$, thereby reducing the storage space from $\Theta(n_C)$ to $\Theta(n_{X_1} + \cdots + n_{X_k} + \log a_1 + \cdots + \log a_k)$: a substantial saving if $a_1, \cdots, a_k$ are large. Also multiplication may be useful in data

compression because the information contained in a product $A \cdot B$ is fully present in its factors, so the storage space needed for the product can be reduced from $\Theta(n_A \cdot n_B)$ to $\Theta(n_A + n_B)$. So the concept of primality may be of practical interest in the reverse-engineering operation of deciding if a tree has been generated as a sum or a product.

6 Other studies on tree arithmetic

Up to now only one major line of research, that we call LBY, has been directed to defining arithmetic on trees. Opened by J.L. Loday *et al* in connection with dendriform algebras [5], it was then developed by J.L. Loday himself who gave a full description of arithmetic operations on binary trees and their properties, showing an embedding of $\mathbb{N}$ in the subsets of all binary trees of n vertices [6]. A. Bruno and D. Yasaki worked on Loday's theory introducing primality and counting properties on subsets of trees in [2]. LBY is limited to binary trees, which carries simpler consequences than in our general case. A non-commutative tree addition is defined in LBY attaching the second addend to a deepest leaf of the first one, and this operation is given in two versions for building any tree from one generator (as in our proposal two different operations are needed). From this construction stems a definition of tree multiplication to produce trees different from our products. Several interesting properties are derived, including some counting arguments on the different families of trees built. The most relevant extension done by Bruno and Yasaki over Loday's theory is the definition and treatment of prime trees under multiplication. Aside from proceeding with similar purposes, none of the definitions and results of LBY applies to our theory, or vice-versa.

Another study on tree arithmetic, due to R. Sainudiin, is aimed at using binary trees for treating mapped partitions of a special class of intervals [10], and has nothing to share with LBY and with our theory. None of these works deals with aspect of computational complexity related to the operations on trees.

Along an independent line of research several papers have been directed to define graph multiplication, from the seminal work of G. Sibidussi [9] to the one of B. Zmazek and J. Zerownik [12]. In this context prime graphs and graph factorization have been considered under various operations of multiplication, see [1]. Again, if applied to trees as special graphs, all the definitions and results on tree multiplication are unrelated to ours.

Acknowledgements. Many thanks are due to Federico Poloni, Mahdi Amani, and to the conference reviewers, for their comments and suggestions.

References

1. N. Bray and E.W. Weisstein. Graph Product. *Math World* - A Wolfram Web Resource. *http://mathworld.wolfram.com/GraphProduct.html*
2. A. Bruno and D. Yasaki. The Arithmetic of Trees. *Involve* 4 (1) (2011), pp. 1–11.
3. S. Finch. Two Asymptotic Series. `http://www.people.fas.harvard.edu/~sfinch/`
4. C.E. Leiserson, T.B. Schardl, and W. Suksompong. Upper Bounds on Number of Steals in Rooted Trees. *Theory of Computing Systems* 58 (2016), pp. 223–240.
5. J.L. Loday, A. Frabetti, F. Chapoton, and F. Goichot. Dialgebras and related operands. *Lecture Notes in Mathematics* 1763, Springer-Verlag, Berlin (2001), pp. 7–66.
6. J.L. Loday. Arithmetree. *J. Algebra* 258 (1) (2002), pp. 275–309.
7. C. Pair and A. Quère. Définition et étude des bilangages réguliers. *Information and Control* 13 (1968), pp. 565–593.

8 J.M. Plitkin and J.W. Rosenthal. How to obtain an asymptotic expansion of a sequence from an analytic identity satisfied by its generating function. *J. Australian Math Soc.* (Ser. A) 56 (1994), pp. 131–143.
9 G. Sabidussi. Graph Multiplication. *Mathematische Zeitschrift* 72 (1960), pp. 446–457.
10 R. Sainudiin. Algebra and Arithmetic of Plane Binary Trees: Theory & Applications of Mapped Regular Pavings. `http://www.math.canterbury.ac.nz/r.sainudiin/talks/MRP_UCPrimer2014.pdf`
11 S. Zaks. Lexicographic Generation of Ordered Trees. *Theoretical Computer Science* 10 (1980), pp. 63–82.
12 B. Zmazek and J. Zerownik. Weak Reconstruction of Small Product Graphs. *Discrete Mathematics* 307 (2007), pp. 641–649.

A List of sequences

The binary sequences representing the trees of the first six canonical families.

1	10		
		18	110101010100
2	1100	19	110101011000
		20	110101101000
3	110100	21	110101110000
4	111000	22	110110011000
		23	110110101000
5	11010100	24	110110110000
6	11011000	25	110111010000
7	11101000	26	110111100000
8	11110000	27	111001101000
		28	111001110000
9	1101010100	29	111010101000
10	1101011000	30	111010110000
11	1101101000	31	111011010000
12	1101110000	32	111011100000
13	1110011000	33	111100110000
14	1110101000	34	111101010000
15	1110110000	35	111101100000
16	1111010000	36	111110100000
17	1111100000	37	111111000000

B Proofs of propositions 4, 5, 8, 12, 14, 15

Proof of Proposition 4. The if part is immediate. For the only if part let $T = A \cdot B$ and $U = B \cdot A$. From the construction of the two products we immediately have $e_T = n_A e_B$ and $e_U = n_B e_A$. If $T = U$ we have $e_T = e_U$ then $n_A e_B = n_B e_A$, then $e_A = e_B$ since $n_A = n_B$. Note that T and U contain $e_A = e_B$ subtrees rooted in the former leaves of A and B respectively, each coinciding with B and A respectively. Each of these subtrees contains $n_B = n_A$ vertices, while all the other subtrees of T, U contain a different number of vertices. Then for having $T = U$ the former two groups of subtrees should be identical, that is each subtree coinciding with B in T must be equal to a subtree coinciding with A in U. That is $A = B$. ◀

Proof of Proposition 5. Let $T = A \cdot B$ and $U = B \cdot A$.

Condition (i) Immediate from the observation that $T = U$ implies $e_T = e_U$ (see the proof of Proposition 4).

Condition (ii) As in the proof of Proposition 4, consider the subtrees of T, U respectively attached to the former leaves of A in T and of B in U. Since $n_A e_B = n_B e_A$ (see the proof above) and $n_A > n_B$ we have $e_A > e_B$. In T there are e_A such subtrees of n_B vertices and in U there are e_B such subtrees of n_A vertices. For having $T = U$ the above subtrees of T (all coinciding with B) should be present also in U where, by the construction of $B \cdot A$, they must appear as subtrees of the copies of A in U.

Condition (iii) By construction the s_B subtrees of $\boldsymbol{r}(B)$ appear also in T as subtrees of $\boldsymbol{r}(T)$ where they are the ones with fewer vertices because all the others have at least n_B vertices. And the s_A subtrees of $\boldsymbol{r}(A)$ appear also in U as subtrees of $\boldsymbol{r}(U)$ where they are the ones with fewer vertices because all the others have at least n_A vertices. Note that all these other subtrees of $\boldsymbol{r}(U)$ have more vertices than the subtrees of $\boldsymbol{r}(B)$ since $n_A > n_B$. For having $T = U$ the s_B subtrees of $\boldsymbol{r}(B)$ that appear as subtrees of $\boldsymbol{r}(T)$ must be equal to s_B subtrees of $\boldsymbol{r}(U)$ and, for what just seen about these subtrees, they must be equal to s_B subtrees among the ones with fewer vertices, i.e. with subtrees of $\boldsymbol{r}(A)$. This also implies that if $s_A = s_B$ then $A = B$. ◀

Proof of Proposition 8. By contradiction. *If part*: for an arbitrary tree $X = A + B$ with $A, B \neq \mathbf{1}$, $\boldsymbol{r}(X)$ has at least two subtrees, then $T \neq X$ for any pair $A, B \neq \mathbf{1}$. *Only if part*: if $\boldsymbol{r}(T)$ has $k > 1$ subtrees $T_1, \ldots, T_k$ then $T = U + V$, where for example U is equal to T deprived of T_k and V is equal to T deprived of $T_1, \ldots, T_{k-1}$. ◀

Proof of Proposition 12. Since $T = A \cdot B$, the subtree Y has been inserted at $\boldsymbol{r}(T)$ as the largest subtree of $\boldsymbol{r}(B)$. Then also the subtrees of $\boldsymbol{r}(T)$ with at most n_Y vertices must have been inserted at $\boldsymbol{r}(T)$ as subtrees of $\boldsymbol{r}(B)$ since they have too few vertices for deriving from former subtrees of $\boldsymbol{r}(A)$ whose vertices are merged with B in T. Furthermore the remaining subtrees of $\boldsymbol{r}(T)$ cannot be subtrees of $\boldsymbol{r}(B)$ since they have too many vertices by the hypothesis that Y is a largest subtree of $\boldsymbol{r}(B)$. ◀

Proof of Proposition 14. Refer to Algorithm MP.

Correctness. Only step **3** requires an analysis. Z is the changing version of T and is restored at each i-th cycle. If one of the groups H_i of subtrees can be erased from Z at all vertices encountered in the traversal, the cycle is completed and the algorithm terminates declaring that T is mult-composite. In fact tree B, whose root has the subtrees in H_i, is one of the factors of T (see Proposition 12). If none of the i-cycles can be completed, that is no H_i can be found as being the group of subtrees of x in all vertices x of Z, the tree T is mult-prime as declared in step **4**.

Complexity. A superficial analysis of the algorithm is the following. Step **1** requires $O(n^2)$ time as discussed for algorithm CF. Step **2** is executed with a linear time scan because the tree is now in canonical form and the number of vertices in each subtree of the root has been computed by algorithm CF in step **1**. Step **3** requires $O(n)$ copy operations of T into Z in $O(n^2)$ time, and $O(n)$ traversals each composed of $O(n)$ steps, for a total of $O(n^2)$ steps. At each step at vertex x the subtrees in H_i must be compared with the subtrees of x with the same cardinality; this can be done by representing such subtrees with their binary sequences S and comparing these sequences. In the worst case vertex x has $O(n)$ subtrees of length $O(n)$, so that building and comparing all the sequences takes time $O(n^2)$, and the total

time required by step **3** is $O(n^4)$. Note that this analysis is very rough because the number of vertices of T decreases during the traversal, so the stated bound $O(n^4)$ is exceedingly high. ◀

Proof of Proposition 15. By contradiction assume that T has two different factorizations $T_1 \cdot T_2 \cdots T_k$ and $S_1 \cdot S_2 \cdots S_h$ in multi-prime factors. Tracing back from k and h, let T_i and S_j be the first pair of factors encountered with $T_i \neq S_j$. Then we have $T_1 \cdot T_2 \cdots T_i = S_1 \cdot S_2 \cdots S_j$. By Proposition 12 T_i must contain S_j as one of its factors (or vice-versa), against the hypothesis that T_i is mult-prime. ◀

C Solution of linear tree equations in two unknowns

The linear tree equation:

$$aX + bY + C = \mathbf{1}, \qquad i.e. \quad aX + bY = -C$$

admits a solution if and only if one of the following two conditions holds, corresponding respectively to trees X, Y of equal sign or of opposite sign.

1. The s_C subtrees of $\boldsymbol{r}(C)$ can be divided in $g \geq 1$ groups $G_1, \ldots, G_g$ and $h \geq 1$ groups $H_1, \ldots, H_h$ of identical subtrees, where each G_i has cardinality $g_i a$ for $g_i \geq 1$ and each H_i has cardinality $h_i b$ for $h_i \geq 1$. In this case X has $\sum_{i=1}^{g} g_i$ subtrees divided in g groups of g_i subtrees identical to the ones of G_i; and Y has $\sum_{i=1}^{h} h_i$ subtrees divided in h groups of h_i subtrees identical to the ones of H_i. This solution can be built in time polynomial in n_C. Note that X and Y have the same sign, and C has opposite sign. See Equation E2 in Figure 8.
2. Let the unknown trees X and Y have opposite sign. W.l.o.g. let the subtrees of $\boldsymbol{r}(X)$ be divided in $k+h$ groups $G_1, \ldots, G_{k+h}$ of identical subtrees, and the subtrees of $\boldsymbol{r}(Y)$ be divided in k groups $H_1, \ldots, H_k$ of identical subtrees, with $k \geq 1$ and $h \geq 0$. And let the subtrees of $\boldsymbol{r}(C)$ be divided in $k+h$ groups $C_1, \ldots, C_{k+h}$ of identical subtrees. x_i, y_i, c_i respectively denote the cardinalities of G_i, H_i, C_i.
   To allow the addition $aX + bY$ the subtrees in H_i must be identical to the ones in G_i for $1 \leq i \leq k$; the subtrees in C_i must be identical to the ones in G_i for $1 \leq i \leq k+h$; and we have the system of diophantine equations:

$$ax_i - by_i = c_i \qquad \text{for} \qquad 1 \leq i \leq k \tag{6}$$

$$ax_i = c_i \qquad \text{for} \qquad k+1 \leq i \leq k+h \tag{7}$$

   whose integer solutions (if any) state that the a copies of the subtrees of G_i suffice to elide the b copies of the subtrees of H_i in C, for $i \leq k$; and a copies of the subtrees in G_i appear as subtrees of C_i, for $i > k$. The system can be solved under the conditions:

$$c_i/gcd(a, b) \text{ integer} \qquad \text{for} \qquad 1 \leq i \leq k \tag{8}$$

$$c_i/a \text{ integer} \qquad \text{for} \qquad k+1 \leq i \leq k+h \tag{9}$$

   for a value of k established as the minimum value for which condition (9) holds (this fixes also the value of h). Then if all conditions (8) hold the system is solved in time polynomial in n_C and two trees X, Y satisfying equation (3) are immediately built from the values of x_i, y_i, out a potentially infinite number of solutions. In particular note that, for all i, the values x_i, y_i must be both positive to represent subset cardinalities. If this does not happen, an alternative positive solution is built from the other by standard methods. See equation E3 in Figure 8.

Two Dots is NP-Complete

Neeldhara Misra

Indian Institute of Technology, Gandhinagar, India
mail@neeldhara.com

Abstract

Two Dots is a popular single-player puzzle video game for iOS and Android. In its simplest form, the game consists of a board with dots of different colors, and a valid move consists of connecting a sequence of adjacent dots of the same color. We say that dots engaged in a move are "hit" by the player. After every move, the connected dots disappear, and the void is filled by new dots (the entire board shifts downwards and new dots appear on top). Typically the game provides a limited number of moves and varying goals (such as hitting a required number of dots of a particular color). We show that the perfect information version of the game (where the sequence of incoming dots is known) is NP-complete, even for fairly restricted goal types.

1998 ACM Subject Classification F.2 Analysis Of Algorithms And Problem Complexity

Keywords and phrases combinatorial game theory, NP-complete, perfect information, puzzle

Digital Object Identifier 10.4230/LIPIcs.FUN.2016.24

1 Introduction

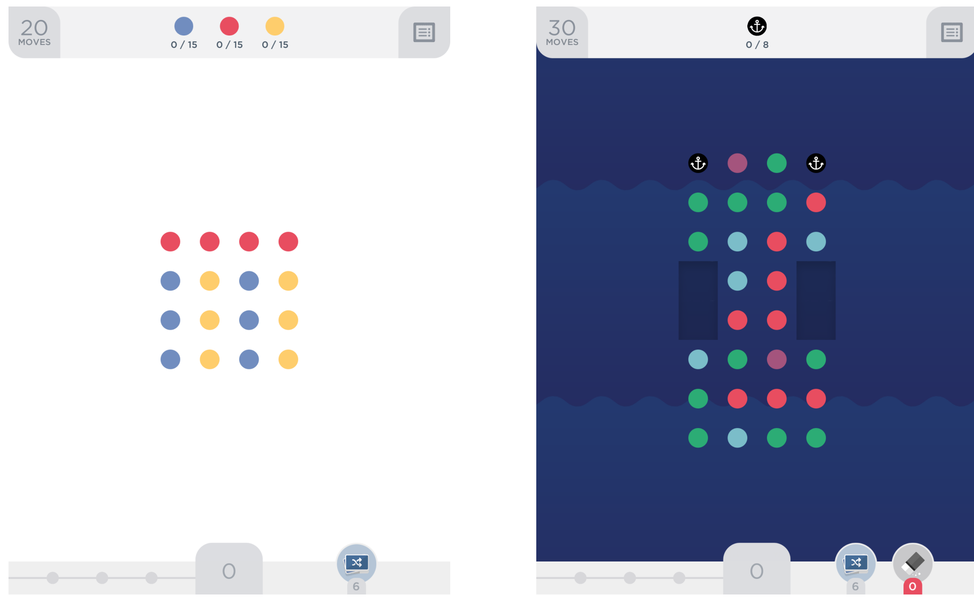
Two Dots® (http://weplaydots.com/twodots.html) is a puzzle game that came out in May 2014 on the iOS and Android platforms. Having surpassed 30 million downloads after just about a year of launch, it is a popular game that is widely accepted as addictive and frustrating in roughly equal measure. In this work, we set out to investigate the computational complexity of playing this puzzle. In its present form, it is a single-player game, and we will not encounter any strategic questions.

The game arena consists of a grid, and at each location there is a colored dot. Dots of the same color can be "connected" by the player, as long as they are adjacent horizontally or vertically (but never diagonally). When dots are connected, they disappear, and the remaining dots fall down as if influenced by gravity. The voids on top are filled with fresh dots. The game provides a certain number of moves, and demands that certain goals should be met (which are typically of the form of collecting at least so many dots of such and such a color, where a dot of a particular color is collected whenever the player connects dots of that color).

An interesting move is the square, wherein, if there are four dots of the same color arranged in a square like configuration (with no gaps, see Figure 3), then swiping across the square causes all dots of said color to disappear. It turns out that this move is frequently a game-changer, and plays an important role in our results too. It is clearly a popular heuristic, and the official Two Dots tutorial even offers the helpful quip: *"When in doubt, make squares"*.

Having spent several frustrating hours with the game of Two Dots, and making limited progress through the increasingly challenging levels, the author was compelled to ask if finding a winning sequence of moves that meets all the goals is an easy problem, even in the unrealistic but optimistic perfect-information setting, where we know exactly what dots are coming up. We discover that the dots from the future are quite irrelevant, and the gameplay

© Neeldhara Misra;
licensed under Creative Commons License CC-BY
8th International Conference on Fun with Algorithms (FUN 2016).
Editors: Erik D. Demaine and Fabrizio Grandoni; Article No. 24; pp. 24:1–24:12
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



(a) Level One of Two Dots.

(b) A more advanced level of Two Dots.

Figure 1 Screenshots from the Two Dots game. The top-left corner shows the total number of moves allowed at the start of the game, and the top-middle shows the target goals. The bottom-middle tracks the score after every move, an aspect that will not be relevant to our present study.

even on just the starting board can get quite intricate, especially since the newly arriving dots can be designed to be of no help to the stated goals. After setting up the formal decision version of the Two Dots game, we establish the following:

- The game is NP-complete, even when the board has only four rows. (Theorem 2.)
- The game is W[1]-hard when parameterized by the number of moves. (Corollary 4.)
- The game is NP-complete even when played without the square move. (Theorem 5.)
- The game remains NP-complete when there is only one goal of collecting two dots of a particular color, even when there is no restriction on the number of moves. (Theorem 6.)

The main objective of this work is to provide reassurance to those Two Dots addicts who might occasionally wonder if the game is truly hard. Of course, we note that the literature is rich in the analysis of the algorithmic aspects of combinatorial games and puzzles. More specifically, games that are popularized by mobile platforms have drawn a lot of attention in the recent past. This work is inspired by, and is in the spirit of some of these developments, which include showing the hardness of popular games like CandyCrush [9], Flow [2], Trainyard [3], 1024, 2048, Threes! [8], 2048 (even without new tiles) [1].

Organization of this Paper. After setting up the formulation of Two Dots as a decision problem in the next section, we dedicate Section 3 to a proof of Theorem 2, and Section 4 to showing hardness in the more restricted scenarios.

2 Preliminaries

The game of Two Dots gets more challenging as the levels progress, largely due to the addition of new goal types and harder obstructions. Players of the game will recognize issues like ice-breaking, containing fire, dealing with monsters, anchors, slime, bombs and so forth.

However, in the interest of keeping the exposition straightforward, we will only deal with the minimalistic version of the game, which we will formulate in this section. This approach also leads us to the conclusion that Two Dots is hard even in its more rudimentary versions. We will also ignore the scoring systems and pose the game as a decision problem.

2.1 A Game of Two Dots

An instance of Two Dots consists of the following[1]:
1. A $(m \times n)$ grid $\mathcal{B}$, a set of colors $\mathcal{C}$, and a mapping f from $\mathcal{B}$ to $\mathcal{C}$.
2. A natural number k, specifying the number of moves allowed in the game.
3. A set of goals $\mathcal{G}$. Every element of $\mathcal{G}$ is a pair (c, ℓ), where $c \in \mathcal{C}$ and $\ell \in \mathbb{N}$.
4. A sequence of colors σ, representing the dots that will fill the voids created by the moves.

We note that the map f in (1) above need not be injective, in other words, multiple locations on the board may be mapped to the same color. The semantics of the pairs given by (3) is that ℓ dots of color c need to be "hit", a notion that we will define in a moment. Summarizing the above, an instance of Two Dots is fully specified by the following tuple:

$$\mathfrak{D} := \langle f : [m] \times [n] \longrightarrow \mathcal{C}, k, \mathcal{G} = \{(c_1, \ell_1), \cdots, (c_g, \ell_g)\}, \sigma \rangle,$$

where n, m, k and the ℓ_i's are natural numbers, $\mathcal{C}$ is a finite set, the c_i's belong to $\mathcal{C}$ and σ is a sequence whose entries are from $\mathcal{C}$. Following tradition, we will say that every location on the board is occupied by a *dot*, and the colors of these dots is given by the function f. Also, our array indexing starts at one, and rows are counted from bottom to top, and columns from left to right.

Intuitively, a player has a winning strategy in an instance of Two Dots if all the goals can be achieved with k moves, given (or despite) the dots that join the board according to σ. To formalize this, we need to first define moves, and the notion of dots being hit.

There are two types of moves in Two Dots: *regular moves* and *square moves*. We first describe the regular moves, which essentially involve removing paths in the grid occupied by the same color. To define this formally, we will need the notion of adjacent locations and valid sequences.

▶ **Definition 1.** Two dots at locations (a, b) and (c, d) are said to be *adjacent* if either $a = c$ and $b = d \pm 1$ or $b = d$ and $a = c \pm 1$, that is, if one of them is to the top, right, left, or bottom of the other. A sequence of locations $\langle t_1, \ldots, t_s \rangle$ is said to be *valid* if t_i is adjacent to t_{i+1} for all $1 \leq i \leq s - 1$, and further, $f(t_i) = f(t_{i+1})$ for all $1 \leq i \leq s - 1$.

Note that dots aligned diagonally are *not* considered adjacent. A player can use any valid sequence of locations towards a regular move. If a valid sequence had s locations and these locations were occupied by dots of color c, then using this sequence causes the color c to be *hit* s times. Some readers may prefer imagining the player collecting s dots of color c when the sequence is committed. This also creates voids in the locations corresponding to the sequence. The dots "fall down" to fill out the voids — it is useful to think of the board as a vertically oriented object, and the dots therein following the natural laws of gravity. Of course, this simply pushes the voids to the top, which are the filled out with new dots, which are colored according to the first s colors given by the sequence σ. We refer the reader to Figure 2 for an illustration.

[1] Readers unfamiliar with the game may find the definitions in this section either simpler or unnecessary after playing it for some time!

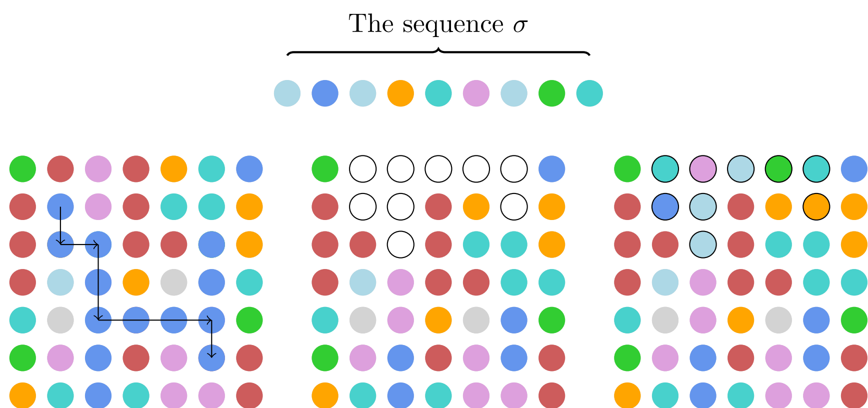


Figure 2 A depiction of the regular move. The first panel shows the set of locations that are committed to the move, the second panel shows the voids created, and the third panel shows how the voids are filled in accordance with σ.

Summarizing the above more formally, a regular move given by a valid sequence of locations $\langle t_1, \ldots, t_s \rangle$ causes the following changes:

1. Let c be the color of the dots in the given sequence. Suppose $\mathcal{G}$ contains a pair (c, ℓ). If $s \geq \ell$, the pair (c, ℓ) is purged from $\mathcal{G}$, else the pair is updated to $(c, \ell - s)$.
2. The function f is updated as follows. For every location $t_i = (a_i, b_i)$, we reset $f(a_i, b_i) = f(a_i + h, b_i)$, where h is the largest number such that every location in:

   $$\langle (a_{i+1}, b_i), \ldots (a_{i+j}, b_i), \ldots (a_{i+h-1}, b_i) \rangle$$

   is a part of the player's sequence.
3. The updated version of f from the previous step is not well-defined for s locations. These locations are assigned to the first s colors from σ. The voids are filled up by σ in a bottom-to-top, left-to-right order. The sequence σ is reset to the subsequence $\sigma[s+1, :]$.[2]

We now turn to a square move (see Figure 3). If a player identifies four adjacent locations in the form of a square, for instance:

$$\langle (i, j), (i+1, j), (i, j+1), (i+1, j+1) \rangle,$$

such that the said locations are all occupied by dots of the same color, say c, then committing the square causes all dots of color c on the board to be hit. In other words, committing a square replaces every location that has a dot colored c with a void, and these voids are filled just as with a regular move. The color c is hit as many times as it appears on the board, and the goal corresponding to c (if there is one) is updated in the same way as with the regular move. In practice, square moves are handy — they tend to clear out large parts of the board and are often necessary for certain goal types. This move turns out to be crucial to our reductions as well.

At the end of k moves, when the game is over, we say that the player has won if the set of goals is empty. The computational problem at the heart of our discussions is the following:

---

[2] This is Pythonic notation, we simply mean here that the first s elements of σ are "chopped off" from the sequence.

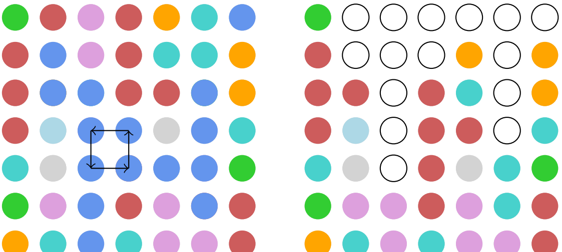


Figure 3 A depiction of the square move, which has the effect of eliminating all the blue dots from the board. The example is rather similar to the above, but note the difference in the number of voids created. The process for filling up the voids is identical, and is therefore omitted.

Two Dots
Given an instance of Two Dots, determine if there exists a sequence of at most k moves that meets all the goals.

▶ Remark. The sequence σ is important to maintain the invariant of a full board, and the reader may legitimately worry about whether σ is long enough to sustain the voids that can be created by k moves. We assume that if σ falls short, then all future voids are filled by dots of distinct colors and colors that are disjoint from the ones of the board (it is useful to think of these as "dummy colors"). Indeed, in our reductions the "dots from the future" will not be important, and thus we will simply provide a null sequence, and it will be apparent that the new dots will have no impact on gameplay.

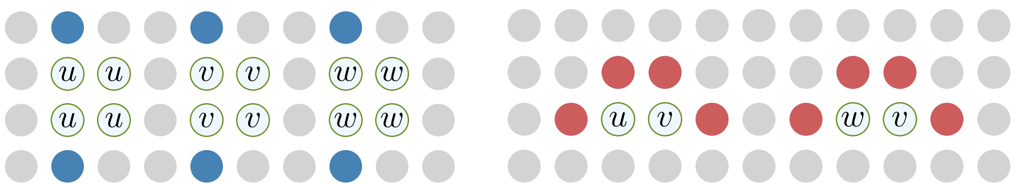
2.2 Other Definitions

In this subsection we briefly recall the definitions of the problems that we will be using to show NP-hardness. We refer the reader to [7] for an introduction to reductions, and to [5] for a survey of complexity-theoretic studies of combinatorial games.

Satisfiability. A *literal* is a propositional variable x or a negated variable $\overline{x}$. A *clause* is a collection of literals. A propositional formula in conjunctive normal form, or *CNF formula* for short, is a set of clauses. A CNF formula F is *satisfiable* if there is some truth assignment to the variables that satisfies all the clauses, where a clause is satisfied if at least one of its literals evaluates to true. The special case of 3-SAT, where every clause has at most three literals, is also well-known to be NP-complete.

Exact Cover. An instance of Exact Cover by 3-Sets, abbreviated X3C, consists of an universe $\mathcal{U} = \{u_1, \ldots, u_n\}$ and a family of sets $\mathcal{F} = \{S_1, \ldots, S_m\}$, where each set has three elements. The question is if there is a set cover of size $(n/3)$, that is, a subcollection $\mathcal{G} \subseteq \mathcal{F}$ such that each element of $\mathcal{U}$ appears exactly once among the sets of $\mathcal{G}$. The problem is well-known to be NP-complete.

Clique. An instance of Clique consists of a graph $G = (V, E)$ and a positive integer k. The question is if there exists a subset S of at least k vertices that form a clique, that is, for every pair of vertices (u, v) such that u and v belong to S, we have that $(u, v) \in E$. This is also a classic NP-complete problem.



(a) The Vertex Gadgets. **(b)** A Tester Gadget for the edges (u, v) and (w, v).

Figure 4 The vertex and tester gadgets used in the reduction from CLIQUE. The colors α and β are depicted, respectively, by blue and red.

3 Hardness in the basic setting

In this section we show the NP-hardness of TWO DOTS, by a reduction from CLIQUE.

Overview of the Reduction. If n is the number of vertices in the instance of CLIQUE, then we are going to have $(n + 2)$ colors for the dots (apart from several more dummy colors). The instance of TWO DOTS that we generate has two main components:

1. For every $1 \leq c \leq n$, we will have a square on dots of color c, as shown in Figure 4a. These will be the *vertex selector* gadgets.
2. For every edge e, we will have a gadget as shown in Figure 4b, which lapses into something useful only when both vertices involved in the edge e are picked for square moves. These are our *tester gadgets*, together with a carefully defined goal they ensure that the colors chosen for square moves correspond to vertices that form a clique in G.

We allow for $2k + \binom{k}{2}$ moves and set a goal of hitting at least $2k$ dots of color $(n + 1)$ (which is the color used in the vertex gadgets), and $4\binom{k}{2}$ dots of color $(n + 2)$ (which is the color used in the tester gadgets). It turns out that the only way to achieve these goals is to apply square moves on colors corresponding to the vertices of a clique, and regular moves on the tester gadgets corresponding to the edges of the clique. We now turn to a more formal description of the construction, and the correctness.

The Construction. Let (G, k) be the instance of CLIQUE, and suppose $G = (V, E)$ where $V = \{v_1, \ldots, v_n\}$, and $E = \{e_1, \ldots, e_m\}$. We now describe the instance of TWO DOTS. The board has four rows and $(3n + 5m)$ columns. The set of colors is as follows.

- We have a set of $4(3n + 5m)$ dummy colors, one for every location of the board.
- We introduce the color c_i corresponding to the vertex v_i, for all $1 \leq i \leq n$.
- We introduce two additional special colors, denoted by α and β.

The mapping f is given by the following, wherein we place squares corresponding to vertices in the first $3n$ columns and the tester gadgets in the remaining columns, all next to each other.

- On the second and third rows, columns $(3i - 1)$ and $3i$ are occupied by dots colored c_i. On the first and fourth rows, column $(3i - 1)$ is occupied by the color α.
- Suppose the edge $e_j = (v_p, v_q)$. On the second row, the two columns $(3n + 5j - 2), (3n + 5j - 3)$ are occupied by dots colored c_p and c_q, respectively.
- Further, on the second row, the columns $(3n + 5j - 4), (3n + 5j)$ are occupied by dots colored β. On the third row, the two columns $(3n + 5j - 2), (3n + 5j - 3)$ are occupied by dots colored β.
- On any location (i, j) that is not accounted for by the above, we use a dot with the dummy color corresponding to that location.

We conclude the description of the instance by fixing the number of moves and the goals. The number of moves is given by $2k + \binom{k}{2}$, and we specify two goals, namely (α, k) and $(\beta, 4\binom{k}{2})$. The sequence σ is the null sequence (voids are filled by fresh and distinct colors). This completes the construction.

In the discussion that follows, we say that a tester gadget corresponding to the edge $e = (v_p, v_q)$ has *collapsed* if square moves were executed corresponding to *both* c_p and c_q. Note that once a tester gadget has collapsed, we can hit four dots colored β with one regular move.

The Forward Direction. Suppose S is a clique of size k in G. We execute square moves corresponding to all vertices in S, using up k moves. This makes k pairs of dots of color α adjacent, and we spend the next k moves in eliminating these. Since S was a clique, the square moves also cause $\binom{k}{2}$ of the tester gadgets to collapse (one corresponding to each edge of the clique). The remaining $\binom{k}{2}$ moves are regular moves on the collapsed gadgets, where each move hits four dots colored β, thereby fulfilling the given goal.

The Reverse Direction. The following is immediate from the two goals:

- At least $2k$ moves must be spent in eliminating $2k$ dots of color α, since the configuration of the board forces dots of color α can only be eliminated in pairs, and the only way to make a pair adjacent is to use up a square move on some c_i, $1 \leq k \leq n$.
- At least $\binom{k}{2}$ of the tester gadgets must collapse because of the square moves, otherwise it is not possible to achieve the goal corresponding to β in $\binom{k}{2}$ moves.

Let S be the set of those $1 \leq i \leq n$ for which a square move was executed on color c_i. It is clear that $|S| \geq k$, since anything less will fail the goal with respect to color α. We claim that the set of vertices corresponding to S forms a clique in G. Indeed, note that only those tester gadget collapse that correspond to edges with both endpoints in S. Any missing edge in S will imply that fewer than $\binom{k}{2}$ tester gadgets collapsed, which contradicts the goal with respect to the color β.

This completes the proof of correctness and leads us to our first theorem.

▶ **Theorem 2.** Two Dots *is NP-complete.*

We remark that membership in NP is easy to show, since a sequence can be used as a certificate. We now turn to some implications of Theorem 2. The first one follows from the fact that our reduction used only four rows. We note that there is no obvious way of “flipping” the construction to derive an analogous result on the number of columns, since the gameplay is inherently asymmetric. For instance, if the tester gadgets corresponding to edges were to be “stacked” one on top of another, then a single square move could cause some unintentional collapses.

▶ **Corollary 3.** Two Dots *remains NP-complete even when the number of rows is a constant.*

Since Clique is W[1]-hard when parameterized by solution size, we also have the following comment on the parameterized complexity of Two Dots when parameterized by the number of moves. We refer the reader to [4] for terminology related to parameterized complexity.

▶ **Corollary 4.** Two Dots *is* W[1]*-hard when parameterized by the number of moves.*

There are (at least) two aspects of this reduction that the reader might find troublesome, or to propose a euphemism, two things that might be identified as providing the source of

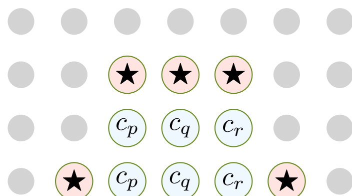


Figure 5 The gadget corresponding to a set $S = \{u_p, u_q, u_r\}$, from an instance of X3C.

hardness. One is the convenience of controlling far-out parts of the board via the square move, and the other is the fact that both goals specify a target that is a function of the input size, rather than, say, a constant. Addressing the above, in the next section we show that the hardness persists even if only regular moves were available to the player, and, in a separate result, we establish NP-completeness with only one constant-sized goal.

4 Some Restricted Scenarios

In this section, we describe reductions showing hardness in more restricted scenarios: the first is when the square move is not available to the player, and the second involves only one constant-sized goal. We point out that some charms are lost in both cases: with only regular moves, we encounter a non-constant number of goals of non-constant size, and in the case of the constant sized goal, we are faced with a rather large board. This renders all the three reductions mutually non-subsuming.

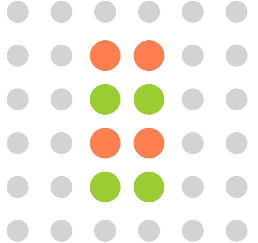
4.1 Hardness without the Square Move

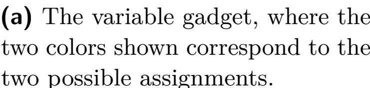
Here we propose a simple reduction from X3C, which we recall is the problem of Exact Cover by 3-Sets. We begin with the construction. Let $(\mathcal{U}, \mathcal{F})$ be an instance of X3C, and further, suppose $\mathcal{U} = \{u_1, \ldots, u_n\}$ and $\mathcal{F} = \{S_1, \ldots, S_m\}$. For our instance of Two Dots, we will have a board with three rows and $6m$ columns, and $(n+1) + 18m$ colors. We use $c_1, \ldots, c_n$ and $\bigstar$ to denote the first $(n+1)$ colors, and the remaining are simply identified as dummy colors (one associated with each location of the board).

We now turn to the map f. We have one block of six columns for every $S_i \in \mathcal{F}$, as seen in Figure 5. More formally, we have:

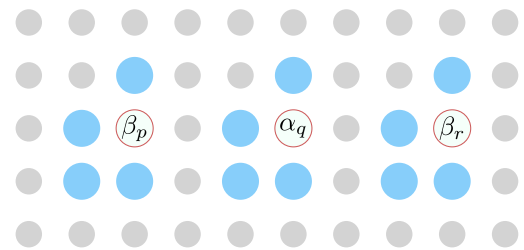
- For columns $6i - 4$ and $6i$, for all $1 \leq i \leq m$, in the first row we have dots colored $\bigstar$. For the columns $6i - 3, 6i - 2$ and $6i - 1$, on the third row, we have dots colored $\bigstar$.
- Suppose the set S_i consists of the elements u_p, u_q, u_r. Then on the first and second rows of columns $6i - 3, 6i - 2$ and $6i - 1$, we have dots colored c_p, c_q and c_r, respectively (see Figure 5).
- On any location (i, j) that is not accounted for by the above, we use a dot with the dummy color corresponding to that location.

We now set the following goals. For every $1 \leq i \leq n$, we have the goal $(c_i, 2)$, and additionally, we have the goal $(\bigstar, 5(n/3))$. Also, the number of moves allowed is $n + n/3$, and σ is the null string. This completes the description of the instance, and we now turn to the proof of correctness. For ease of discussion, we say that the gadget corresponding to a set $S = \{u_p, u_q, u_r\}$ has collapsed if the dots colored c_p, c_q, c_r belonging to that gadget were hit by regular moves.





(a) The variable gadget, where the two colors shown correspond to the two possible assignments.



(b) A Clause gadget corresponding to the clause $(\overline{x_p}, x_q, \overline{x_r})$. The color corresponding to the clause is depicted in Blue.

Figure 6 The variable and clause gadgets used in the reduction from 3-SAT.

The Forward Direction. Suppose $\mathcal{G} \subseteq \mathcal{F}$ is a collection of $(n/3)$ sets that covers each element of the universe exactly once. For each $1 \leq i \leq n$, let $g(i)$ denote the set from $\mathcal{G}$ that contains i. We hit the dots c_i in the gadget corresponding to the set $g(i)$. Note that so far we have used n moves, satisfied the goals corresponding to c_i and exactly $(n/3)$ set gadgets (corresponding to the ones in $\mathcal{G}$) have now collapsed. We use the remaining $(n/3)$ moves to clear out the $\bigstar$ rows in the collapsed gadgets, thereby meeting the remaining goal.

The Reverse Direction. Note that to meet the goals corresponding to the colors c_i, $1 \leq i \leq n$, we must use at least n moves. Since there are only $(n/3)$ moves remaining for the goal corresponding to $\bigstar$, we must collapse exactly $(n/3)$ set gadgets. It is easily seen that the collapsed gadgets correspond to to an exact cover, as desired.

These arguments culminate in the following result.

▶ **Theorem 5.** Two Dots *is NP-complete, even when played only with regular moves.*

4.2 Hardness with Constant Goals

We now turn to our quest of showing hardness when the game is not demanding when it comes to goals. This is our final reduction, and here we reduce from 3-SAT.

Overview of the Reduction. We present an informal sketch of the construction first. The idea is to have two dots of a special color, denoted by $\bigstar$, separated by a stack of dots corresponding to the clauses of the SAT instance. In particular, we will have a color corresponding to every variable and every clause, and the only way to destroy the column of dots separating the two dots colored $\bigstar$ is to use square moves on every color corresponding to a clause. However, these squares are not directly available on the original board, and they can be created by collapses, as in the previous reductions. As the reader as perhaps guessed by now, these collapses are in correspondence with variables being set appropriately with respect to the clauses.

One difference from the previous reductions is the additional concern that we cannot permit square moves corresponding to contradictory literals such as x and $\overline{x}$. Therefore, the squares corresponding to variables, which trigger the whole sequence are also not directly available — in the variable gadget, a potential square corresponding to a negated (positive) literal must be "sacrificed" in order for the square for the positive (negated) literal to manifest. We refer the reader to Figure 6a for the exact configuration.

The Construction. Let ϕ be an instance of 3-SAT with clauses $C_1, \ldots, C_m$ over the variables $x_1, \ldots, x_n$. We have a board with $(m+11)$ rows, and $(3n+9m+1)$ columns. We introduce the following colors.

- We have two colors α_i and β_i corresponding to each variable x_i and its negation, respectively.
- We introduce the color c_j corresponding to each clause C_j, and an additional color $\bigstar$.
- We have as many dummy colors as there are locations on the board.

  Now we turn to the configuration of the board, which is as follows (see Figure 7).

- The first $(m+2)$ rows of the first column have dots with the colors $\bigstar, c_1, \ldots, c_m, \bigstar$, appearing in that order (from bottom to top).
- The first $3n$ columns starting from the second column, and the rows $(m+4), (m+5), (m+6), (m+7)$ are reserved for the variable gadgets (see Figure 6a). In particular, the rows $(m+4)$ and $(m+6)$ on columns $3i$ and $(3i+1)$ contain dots with the color α_i while the rows $(m+5)$ and $(m+7)$ on the same columns contain dots with the color β_i.
- The last $9m$ columns, and the rows $(m+9), (m+10)$ and $(m+11)$ are reserved for the clause gadgets (see Figure 6b). In particular, we have the following.
  - If the clause C_j consists of the literals (ℓ_p, ℓ_q, ℓ_r), then we have dots with the colors corresponding to these literals occupying the locations at columns $(3n+1)+(9j-6)$, $(3n+1)+(9j-3)$ and $(3n+1)+9j$ on row $(m+10)$, respectively.
  - On row $(m+9)$ and $(m+11)$, at columns $(3n+1)+(9j-6)$, $(3n+1)+(9j-3)$ and $(3n+1)+9j$, we have dots with the color c_j.
  - On row $(m+9)$ and $(m+10)$, at columns $(3n+1)+(9j-7)$, $(3n+1)+(9j-4)$ and $(3n+1)+(9j-1)$, we have dots with the color c_j.
- On any location (i, j) that is not accounted for by the above, we use a dot with the dummy color corresponding to that location.

Our instance allows for $(2n+m+1)$ moves, the goal is given by $(\bigstar, 2)$, and σ is the null string. This completes our description of the instance, and we are now ready to show the correctness.

As usual, to enable the discussion of the correctness, we introduce some terminology. We say that the gadget corresponding to the clause C_j collapses if a square move is executed on a color corresponding to some literal in C_j. Notice that if C_j collapses, then it creates at least one square of color c_j.

The Forward Direction. In the forward direction, let $\tau : \{x_1, \ldots, x_n\} \rightarrow \{0, 1\}$ be a satisfying assignment for ϕ. For each variable x_i for which $\tau(x_i) = 1$, we execute a regular move on the pair of dots colored β_i such that a square on dots of color α_i is created, and then execute a square move on α_i. Similarly, for each variable x_i for which $\tau(x_i) = 0$, we execute a regular move on the pair of dots colored α_i such that a square on dots of color β_i is created, and as before, we execute the square move on β_i.

Note that we have used up $2n$ moves so far. Since τ is a satisfying assignment, every clause gadget collapses, and we use the next m moves to execute square moves on each color c_j, $1 \leq j \leq m$. At the end of this, the two dots colored $\star$ become adjacent and we use a regular move on them in the last available move to finish the game.

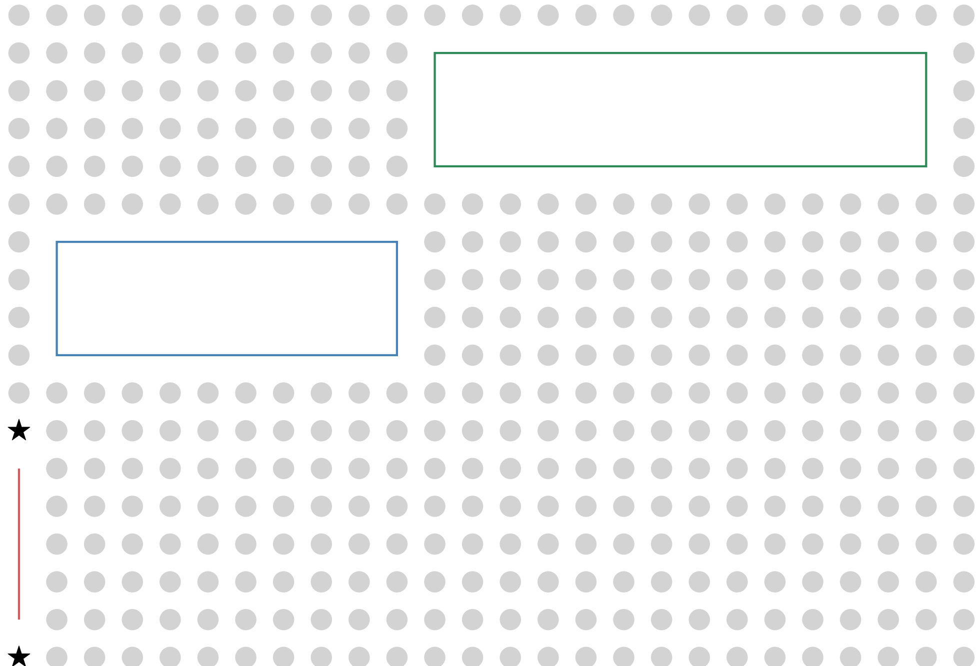


Figure 7 The overall schematic of our reduction from SAT. The bottom-left line in red is a cartoon for the list of clauses, the blue rectangle in the center hosts the variable gadgets from left to right, and the green rectangle in the top right contains the clause gadgets, again from left to right

The Reverse Direction. In the reverse direction, we first collect some immediate observations. Note that it is imperative for every dot in the first column and rows $2, 3, \ldots, m+1$ to be hit, for the goal to be met. The only way for this to happen is for the player to execute square moves on each c_j, $1 \leq j \leq m$. This, in turn, is made possible only when every clause collapses. A clause collapse can only be caused by a square move on the α_i's and β_i's. Further, by the structure of the variable gadget, it is clear that for each i, a square move is executed on either α_i or β_i, but never both.

So we let X^+ be the set of variables for which square moves were executed on the corresponding α_i, and analogously, we let X^- be the set of variables for which square moves were executed on the corresponding β_i. It follows from the last comment in the previous paragraph that these sets of variables are disjoint. Therefore, consider the well-defined assignment τ that sets everything in X^+ to one and everything in X^- to 0, and any remaining variables arbitrarily. We claim that τ is a satisfying assignment.

Indeed, if not, there is some clause C_j that is not satisfied by τ, and the corresponding clause in the game does not collapse by construction. This means, in turn, that there was no way for the dot colored c_j on the first column to be eliminated, which implies that the two dots colored ★ did not become adjacent, contradicting our assumption that we started with a winning sequence. This completes the argument, leading us to the following.

▶ **Theorem 6.** Two Dots *is NP-complete when there is one goal demanding two hits.*

▶ Remark. The reduction above could have also been executed with, say, Dominating Set. However, an interesting aspect of this reduction is that it is easily seen to work even if there is no upper bound specified on the number of moves. This was one of the reasons to parameterize by the number of moves, to see if this contained the hardness in the parameterized setting,

although Corollary 4 answers this in the negative. In this context, we note that an $n^{\mathcal{O}(k)}$ algorithm in the number of moves is easily obtained, by guessing the move at each step, and simulating the game: this search tree has polynomially many branches and depth k.

5 Concluding Remarks

We have shown the NP-completeness of Two Dots in some fairly restricted settings. Somewhat unusually for combinatorial puzzles, these reductions turned out to be rather simple. The question of whether combining restrictions from all the scenarios finally brings us to a tractable setting is the most pertinent one. Other musings include the question of the complexity of the game when the number of columns is a constant, and what happens if we parameterize by the number of columns, or the number of colors. Also, in the version of the game where locations catch fire, possibly there are parallels with the Firefighting problem [6].

Acknowledgements. The author acknowledges support by the INSPIRE Faculty Scheme, DST India (project IFA12-ENG-31). Thanks also to Dots for the game of Two Dots!

References

1. Ahmed Abdelrazek, Aditya Acharya, and Philip Dasler. 2048 without new tiles is still hard. In *Proceedings of the 8th International Conference on Fun with Algorithms (To Appear)*, 2016.
2. Aaron B. Adcock, Erik D. Demaine, Martin L. Demaine, Michael P. O'Brien, Felix Reidl, Fernando Sanchez Villaamil, and Blair D. Sullivan. Zig-zag numberlink is NP-complete. *JIP*, 23(3):239–245, 2015.
3. Matteo Almanza, Stefano Leucci, and Alessandro Panconesi. Trainyard is NP-hard. In *Proceedings of the 8th International Conference on Fun with Algorithms (To Appear)*, 2016.
4. Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015.
5. Demaine. Playing games with algorithms: Algorithmic combinatorial game theory. In *MFCS: Symposium on Mathematical Foundations of Computer Science*, 2001.
6. Fedor V. Fomin, Pinar Heggernes, and Erik Jan van Leeuwen. Making life easier for firefighters. In *Proceedings of the 6th International Conference on Fun with Algorithms*, volume 7288, pages 177–188. Springer, 2012.
7. M. R. Garey and D. S. Johnson. *Computers and Intractability*. Freeman, San Francisco, 1979.
8. Stefan Langerman and Yushi Uno. Threes!, Fives, 1024!, and 2048 are hard. In *Proceedings of the 8th International Conference on Fun with Algorithms (To Appear)*, 2016.
9. Toby Walsh. Candy crush is NP-hard. *CoRR*, abs/1403.1911, 2014. URL: `http://arxiv.org/abs/1403.1911`.

This House Proves That Debating is Harder Than Soccer

Stefan Neumann[1] and Andreas Wiese[2]

1 **University of Vienna, Faculty of Computer Science, Vienna, Austria**
stefan.neumann@univie.ac.at
2 **Max Planck Institute for Computer Science, Saarbrücken, Germany**
awiese@mpi-inf.mpg.de

Abstract

During the last twenty years, a lot of research was conducted on the sport elimination problem: Given a sports league and its remaining matches, we have to decide whether a given team can still possibly win the competition, i.e., place first in the league at the end. Previously, the computational complexity of this problem was investigated only for games with two participating teams per game. In this paper we consider Debating Tournaments and Debating Leagues in the British Parliamentary format, where four teams are participating in each game. We prove that it is NP-hard to decide whether a given team can win a Debating League, even if at most two matches are remaining for each team. This contrasts settings like football where two teams play in each game since there this case is still polynomial time solvable. We prove our result even for a fictitious restricted setting with only three teams per game. On the other hand, for the common setting of Debating Tournaments we show that this problem is fixed parameter tractable if the parameter is the number of remaining rounds k. This also holds for the practically very important question of whether a team can still qualify for the knock-out phase of the tournament and the combined parameter $k+b$ where b denotes the threshold rank for qualifying. Finally, we show that the latter problem is polynomial time solvable for any constant k and arbitrary values b that are part of the input.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases complexity, elimination games, soccer, debating

Digital Object Identifier 10.4230/LIPIcs.FUN.2016.25

1 Introduction

Debating and soccer are deeply rooted in our society. Debating dates back to the times of the ancient greek when already in 460 BC the citizens of Athens were meeting in one of the first parliaments of the world for discussions and votings [4]. This gave rise to the fine art of rhetoric, the skill to speak in a public debate in a convincing manner, to give a solid argumentation for the provided claims, and to win the support of the audience for the own case. Since the ancient Greece the art of debating has developed, and great speeches became milestones of history such as the famous speech delivered by Martin Luther King on August 28, 1963 containing the dictum “I have a dream” [12]. Nowadays, all over the world there are debating societies at universities and outside academia that are devoted to debates and public speaking. This has a long tradition, for instance, the Cambridge Union Society was founded in 1815 and has been run continuously for more than 200 years now [3]. Important for this paper is that there are debating competitions: teams of debaters meet and argue for and against the case of a previously specified motion. The roles (pro and

© Stefan Neumann and Andreas Wiese;
licensed under Creative Commons License CC-BY
8th International Conference on Fun with Algorithms (FUN 2016).
Editors: Erik D. Demaine and Fabrizio Grandoni; Article No. 25; pp. 25:1–25:14
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

contra) are assigned randomly and thus the debaters do not necessarily argue for the side that they personally support.

Like debating, soccer is an integral part of the contemporary societies in many countries. It is played by 250 million players in more than 200 countries which makes it the world's most popular sport [7]. Even more people are passionate for watching the matches and supporting their favorite teams. For instance, the final of the last world cup 2014 was watched by more than one billion people world wide [9].

It is clear that debating and soccer play a significant role in modern societies. However, one question has remained open: what is harder, debating or soccer? Empirically there are only very few indications. There are quotes by soccer players such as "We lost because we didn't win." (Ronaldo [20]), "I also told him that verbally." (Mario Basler [1]), "It doesn't matter if it is Milano or Madrid as long as it is Italy." (Andreas Möller [16]), or "I can see the carrot at the end of the tunnel." (Stuart Pearce [19]) which suggest that excelling rhetorically might be harder than playing soccer. On the other hand, the political careers of heads of states typically surpass their soccer careers by orders of magnitude. For instance, Gerhard Schröder, the former chancellor of Germany, played only in the Bezirksliga [22] which is nowadays the 7th level of the soccer league system in Germany. For the current German chancellor Angela Merkel we are not aware of any non-trivial soccer abilities. However, she is known to occasionally frequent the German national team's changing room after important matches [17].

From a scientific point of view it is difficult to compare debating and soccer since they have only few intersection points that allow a scientifically accurate comparison. One of the few is the following: consider a league in which soccer/debating teams play matches against each other according to a pre-defined schedule that indicates on which match days which respective teams play each other. Consider your favorite team t_1. The question is: are there outcomes for all remaining matches such that t_1 wins the championship?

In soccer, this question is polynomial time solvable if there are at most two remaining matches per team and NP-hard for at most three matches per team under the three-point rule [2, 13]. The latter is nowadays ubiquitous in soccer leagues and tournaments (such as in all FIFA world cups since 1994, in most national soccer leagues since 1995, and in some of them even much earlier [6]). It specifies that if a team wins a match it scores three points for the league ranking and the losing team scores zero points, if the match is a draw then both teams score one point.

For debating, we focus in this paper on the British parliamentary style format that enjoys great popularity world wide and is played for instance in the world universities debating championships [24]. In this format, four teams are playing in each game and the winning team scores three points, the second team scores two points, the third team scores one point, and the fourth team scores zero points. If in the final ranking multiple teams have the same number of points, then a tiebreaker is used. For simplicity, in this paper we assume that this tie-breaker is the total number of FUN papers written by members of the team and that the team t_1 has written the most FUN papers among all participating teams. Thus, t_1 wins the championship if there is no team with more points than t_1 and the corresponding problem is called DebatingLeague.

1.1 Our contribution

In this paper we prove that DebatingLeague is NP-hard, even if there are only two remaining matches to play for each team. This shows that debating is computationally harder than soccer in two ways: first, if there are only two remaining matches to play for each team then

in soccer we can decide in polynomial time whether a given team can still win [2]. Secondly, for an arbitrary number of remaining matches soccer is easy under the two-point rule [2], i.e., the winning team scores two points, rather than three. The two-point rule has the important feature that for each match there is a given number of points (two) that are completely distributed among the participating teams. This is also the case in debating: in each match there are six points available and they are all distributed. While with this feature soccer is easy, DebatingLeague is NP-hard despite of this which underlines the complexity of the latter problem. To the best of our knowledge, this is the first time that the elimination problem has been studied for games with more than two teams per match. In fact, we prove that our hardness result even holds in a fictitious setting in which only three teams participate in a game and they score two, one, and zero points, respectively.

While DebatingLeague is NP-hard if only two matchdays are remaining, we can show something different for the system that is typically played in debating *tournaments*. There, the matches of the teams are defined in a similar way as in Swiss-system tournaments [8] (which are for instance common in chess): after each round the teams are ordered according to the number of points they scored so far. Then the teams ranked 1st-4th play one match, the teams ranked 5th-8th play the second match, and so on. Since the pairings in each round depend on the initial ranking and the outcomes of the previous rounds, the above hardness result for DebatingLeague does not apply. In practice debating tournaments have a first phase organized as above and a second phase that is played as a knock-out tournament. There is a threshold b specifying that the first b teams of the final ranking after the first phase qualify for the knock-out phase, denoted as *breaking*. A key question that a team typically asks itself during a tournament is whether it can still break. Formally, we denote by DebatingTournament the problem of deciding whether t_1 can finish on place b or better with k rounds left in the tournament.

We show that DebatingTournament can be solved in time $O(f(k+b)\cdot n)$, i.e., the problem is fixed parameter tractable for the combined parameter $k+b$. In particular, this implies that for any constant k it is polynomial time solvable to decide whether t_1 can win the tournament, while for $k=2$ DebatingLeague is NP-hard. For our algorithm we first prove that if initially the team t_1 is "too far behind", i.e., has a too large initial rank depending on k and b, then it cannot break anymore. For the remaining case we provide an algorithm with a running time of $O(f(k+b)\cdot n)$ for a suitable function f. Additionally, we show that for constant k the problem is polynomial time solvable (for an arbitrary value of b that is part of the input). Thus, even for arbitrary b the case that $k=2$ is in P, in contrast to DebatingLeague.

1.2 Other related work

In 1966, Schwartz [21] proved that using flow networks it can be decided in polynomial time whether a baseball team can still win a baseball league. In baseball the winner of a game wins a single point and the looser gets zero points, there is no tie. McCormick [15] generalized this result by giving a polynomial time algorithm which allowed to fix a number of losses for the team that is supposed to win the league. Wayne [23] characterised all teams of a baseball league which can still win the league by giving a threshold value for the number of points and the number of matches a team must have to be able to win the league. He further gave a polynomial time algorithm to compute this threshold. This result was later improved by Gusfield and Martel [11] who gave thresholds for a bigger set of possible outcomes of the matches. For baseball leagues they gave a faster algorithm to determine the threshold and further allowed leagues with multiple divisions and wild-cards [11].

A major difference between baseball and soccer leagues is which outcomes are possible in a single game. For soccer leagues with the three-point-rule it was proven by [13] and [2] independently that it is NP-hard to determine whether a team can win the league. Pálvölgyi [18] proved that when we are given the table of a soccer league and a list of games that were played so far without their outcomes, it is NP-hard to decide whether this table is valid, i.e., whether the distribution of points to the teams can be achieved by real outcomes of games. In [5], the authors construct a hypergraph representing the teams and their remaining matches. Depending on certain properties of this graph they prove multiple hardness results for the question whether a certain team can still win the competition.

In [14], Kern and Paulusma consider games with two teams, but allow a game to have many different outcomes. They prove that it can be decided in polynomial time whether a team can still win the competition if and only if in each match exactly m points can be distributed arbitrarily to both teams (for any positive integer m).

2 Debating League

In this section we prove that DebatingLeague is NP-hard, even if each team has at most two remaining matches to play. First, let us define the problem formally. Let $T = \{t_1, \dots, t_n\}$ be the set of teams participating in the debating league. We denote the set of remaining matches by $M \subset T^4$, i.e., we have $(t_i, t_j, t_k, t_l) \in M$ iff the teams t_i, t_j, t_k and t_l still have to play against each other in a match. We assume that each possible match occurs at most once; further, throughout the whole section the game schedule of remaining matches is fixed. The winner of each match scores 3 points, the second placed team scores 2 points, the third placed team scores 1 point and the loosing team does not get any point. We are given a *score vector* $s \in \mathbb{R}^n$ with an entry s_i for each team t_i that indicates how many points team t_i already obtained before playing the remaining matches. Notice that the tuple (T, M, s) encodes all information we need about the competition. In the DebatingLeague problem we want to find out whether team t_1 can still win the competition.

▶ **Definition 1.** In the DebatingLeague problem we are given a tuple (T, M, s) and we want to answer the question whether there are outcomes for all matches M, such that at the end there is no team that has more points than team t_1.

We will prove that this problem is already NP-hard when each team has at most two remaining matches. We prove this first for a variant of DebatingLeague where we have only 3 teams per match and each team has at most two matches left to play. In a game the winner gets 2 points, the second placed team gets 1 point and the looser gets 0 points. We still want to decide whether team t_1 can win the competition. We denote this problem ThreeTeamDebating. It can also be characterised by a tuple (T, M, s) similarly to above.

▶ **Theorem 2.** *The ThreeTeamDebating problem is NP-hard even when each team has at most two remaining matches to play.*

Before we start giving the proof of Theorem 2, we introduce a way to visualize instances of ThreeTeamDebating as graphs. Suppose we are given an instance (T, M, s) of ThreeTeam-Debating in which each team plays at most two matches. We visualize its matches via a *game graph* $G = (V, E)$ in the following way: For each game $g \in M$, we introduce a *game vertex* $v_g \in V$. For each team t_i that participates in two matches g, g', i.e. if $t_i \in g$ and $t_i \in g'$, we introduce an edge e_i connecting v_g and $v_{g'}$. Such an edge will be called a *team edge*. Each edge will receive a weight w_i which encodes how many points team t_i can still get

without obtaining more points than team t_1. If a team has only one game remaining, we do not introduce an edge for it. Notice that later team t_1 will not be part of the game graph as we can assume w.l.o.g. that it wins all of its remaining games and has no games left.

We prove Theorem 2 via a reduction from 3-Bounded-3-SAT [10] to ThreeTeamDebating. Let φ be a 3-Bounded-3-SAT formula with variables $x_1, \ldots, x_n$ and clauses $C_1, \ldots, C_m$. We can assume that each variable occurs in two or three different clauses and that it occurs at least once positively and at least once negatively. We can further assume that each clause has two or three literals.

We construct an instance (T, M, s) of ThreeTeamDebating. First, we describe gadgets out of which our construction is composed and prove some of their properties. Afterwards, we describe how to combine the gadgets to the final instance. In the sequel, we will prove some properties about our construction. We will use the term "We can assume that ..." for the claim that team t_1 can still win the championship if and only if it can still win the championship for outcomes of the matches where the respective following statement is true. In our construction, t_1 has no remaining game to play. We distinguish the other teams into *two-game teams* and *one-game teams*, where the former type has two remaining games to play and the latter type has one remaining game to play. For each team, we will define how many points it can still score without getting more points than t_1. We will not exactly specify how many points each team has initially since it matters only how many points it can still get without overtaking t_1.

For each variable x in φ we introduce a *ring gadget*. Assume that x occurs in the three clauses C_i, C_j, C_k. The ring gadget for a variable x consists of the six games given by the set $G_x := \{g^1_{x,C_i}, g^2_{x,C_i}, g^1_{x,C_j}, g^2_{x,C_j}, g^1_{x,C_k}, g^2_{x,C_k}\}$ and six teams two-game teams as specified by $T_x := \{t^1_{x,C_i}, t^2_{x,C_i}, t^1_{x,C_j}, t^2_{x,C_j}, t^1_{x,C_k}, t^2_{x,C_k}\}$. If x appears in only two clauses C_i, C_j we use the same setup for a fictitious clause C_k.

The games of the teams in T_x are visualized in Figure 1. Ignoring teams which are not from the set T_x and which we will introduce later, the game g^1_{x,C_i} is played by the teams t^2_{x,C_k}, t^1_{x,C_i}, the game g^2_{x,C_i} is played by the teams t^1_{x,C_i}, t^2_{x,C_i}, the game g^1_{x,C_j} is played by the teams t^2_{x,C_i}, t^1_{x,C_j}, the game g^2_{x,C_j} is played by the teams t^1_{x,C_j}, t^2_{x,C_j}, the game g^1_{x,C_k} is played by the teams t^2_{x,C_j}, t^1_{x,C_k}, and the game g^2_{x,C_k} is played by the teams t^1_{x,C_k}, t^2_{x,C_k}. Thus, when visualizing the games in G_x and the teams in T_x they form a cycle. Each team g^1_{x,C_ℓ} with $\ell \in \{i, j, k\}$ is allowed to get 2 points and each team g^2_{x,C_ℓ} with $\ell \in \{i, j, k\}$ can get 3 points. The other teams participating in the games G_x (to be defined later) will only be able to score exactly 1 point and hence they will not be able to win a game. Hence, we can assume that in each game $g \in G_x$ one team in T_x that plays in g must score 2 points. Furthermore, each team in T_x can win at most one game and since there are six games in G_x and six teams in T_x, each team in T_x must win exactly one game.

One way to visualize the outcome of the circle games is to orient each edge in the game graph. The team edge of a team $t \in T_x$ points towards the unique game in which t scores 2 points. In this viewpoint, the following lemma implies that we can assume that all edges of the cycle are either oriented clockwise or counter-clockwise.

▶ **Proposition 3.** *We can assume that either the ring gadget is* oriented clockwise*, i.e. game g^z_{x,C_ℓ} is won by team t^z_{x,C_ℓ} for $\ell \in \{i, j, k\}$ and $z \in \{1, 2\}$, or the ring gadget is* oriented counter-clockwise*, i.e. game g^2_{x,C_ℓ} for $\ell \in \{i, j, k\}$ is won by team t^1_{x,C_ℓ} and the games g^1_{x,C_i},g^1_{x,C_j},g^1_{x,C_k} have winners t^2_{x,C_k},t^2_{x,C_i},t^2_{x,C_j}, respectively.*

Later, the two possible orientations of the ring gadget for variable x will correspond to setting the variable x to true or to false. Next, we introduce a *clause game* g_C for each clause C in φ. Let C be a clause with variables x, y, z. We introduce three two-game teams

$t^4_{x,C}, t^4_{y,C}, t^4_{z,C}$ that play g_C and each of them will play in another game that we will define later. Each of them can still score 2 points. Intuitively, the team among them that scores 2 points in g_C will correspond to the variable that satisfies the clause C in a satisfying assignment. Note that for the names of the teams we do not distinguish whether a variable x occurs positively or negatively in C.

We describe now how we connect the clause games with the ring gadgets, see Figure 1. Let x be a variable that occurs in a clause C. For this occurrence, we introduced the team $t^4_{x,C}$ above. We now introduce a game $g^3_{x,C}$, a two-game team $t^3_{x,C}$, and a one-game team $t^3_{x,C,d}$. The team $t^3_{x,C}$ can still get 1 point and the team $t^3_{x,C,d}$ can still get 2 points. We define that $g^3_{x,C}$ is the second game of $t^4_{x,C}$, the only game of $t^3_{x,C,d}$, and one of the two games that $t^3_{x,C}$ plays. The intuition behind this construction is that if $t^3_{x,C}$ gets 0 points in its second game (that we have not specified yet) then the team $t^4_{x,C}$ can score up to 2 points in game g_C (without getting more points in total than t_1). On the other hand, if $t^3_{x,C}$ gets 1 point in its other game, then $t^4_{x,C}$ can score only up to 1 point in g_C and in particular, it cannot score 2 points in g_C anymore. Later, the first case will correspond to the case that x satisfies C whereas the second case will correspond to the case that x does not satisfy C.

▶ **Proposition 4.** *Let x be a variable appearing in a clause C. We can assume that*

- *if $t^3_{x,C}$ scores 1 point in a game different than $g^3_{x,C}$ that it plays, then $t^4_{x,C}$ scores at most 1 point in game g_C, and*
- *if $t^3_{x,C}$ scores 0 points in a game different than $g^3_{x,C}$ that it plays, then $t^4_{x,C}$ can score up to 2 points in the game g_C.*

We specify the second game for the team $t^3_{x,C}$ (i.e., the game different than $g^3_{x,C}$ that it plays). If x appears positively in clause C then this second game is defined to be $g^2_{x,C}$, otherwise, this second game is defined to be $g^1_{x,C}$. If the variable x appears in three clauses C_i, C_j, C_k then three games from the G_x are still missing one team, exactly one per clause. For these games we add the one-game teams $T^d_x := \{t_{x,d_i}, t_{x,d_j}, t_{x,d_k}\}$, each of them playing the game with the corresponding clause in the subscript and each of them allowed to score 1 point. If x appears in only two clauses then we similarly add two one-game teams such that each of them is allowed to score 1 point and by this we ensure that each game in G_x has three teams. This completes the definition of the instance.

▶ **Lemma 5.** *If φ is satisfiable then there is an outcome of the defined instance of ThreeTeam-Debating such that no team gets more points than t_1.*

Proof. Suppose we are given a satisfying assignment to the variables in φ. From this satisfying assignment we will construct outcomes of the games, such that team t_1 wins the championship. Intuitively, we will want to assign all points as depicted in Figure 1. We will now describe this formally.

Let x be a variable. If x is true, then we orient the ring gadget of x counter-clockwise according to the left image in Figure 1; formally, the winners of the games G_x are assigned as defined in Proposition 3. If x is false, then the ring gadget of x is oriented clockwise according to the right image in Figure 1. All one-game teams T^d_x will place second in their games and thus obtain a single point each.

Consider a clause C with variables x, y, z. In the satisfying assignment one of them must satisfy C. Assume w.l.o.g. that x satisfies C. Then we let team $t^4_{x,C}$ score 2 points in the game g_C and we let an arbitrary team among $t^4_{y,C}, t^4_{z,C}$ score 1 point and the other one 0 points. For the game $g^3_{x,C}$ we let the one-game team score 2 points and team $t^3_{x,C}$ score 1 point, team $t^4_{x,C}$ obtains no additional point from this game. Team $t^3_{x,C}$ further scores 0

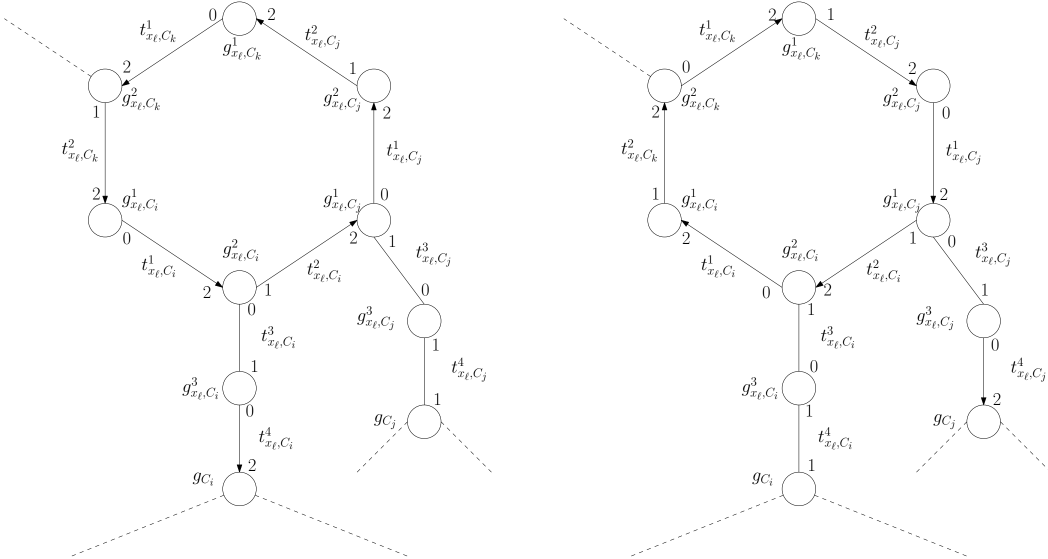


Figure 1 An excerpt of the game graph for variable x_ℓ which occurs in clauses C_i and C_k positively and in C_j negatively. In the left image the outcomes of the games for $x_\ell = \mathit{true}$ are visualised, in the right image we have $x_\ell = \mathit{false}$. The edges are directed towards the game that was won by the corresponding team; the numbers close to the game vertices show how many points the associated two-game teams win in this game.

points in its remaining game (game $g^1_{x,C}$ or $g^2_{x,C}$, depending on whether x appears negatively or positively in C) and the remaining point of this game goes to the team which can obtain 3 points in total. In the game $g^3_{y,C}$ we let the team $t^4_{y,C}$ score 1 point and team $t^3_{y,C}$ scores 1 point from its game in G_y. For the teams and games for variable z we use the same distribution of points as for y. We define the outcomes of the games in the same way for each clause C. See Figure 1 for a sketch of the outcomes described above.

All games distribute all of their points: In the above assignment, for each clause game g_C we have distributed all points by construction. For all x, C, the games $g^3_{x,C}$ have a one-game team as a winner and by construction the second place goes to either $t^3_{x,C}$ or $t^4_{x,C}$. It is left to to argue about the games from the G_x. For each $g^z_{x,C} \in G_x$ with $z \in \{1, 2\}$ we must have a winner since we assigned the winners as defined in Proposition 3. If $g^z_{x,C}$ has a one-game team participating then this team can place second in our construction and hence all points are distributed. If $g^z_{x,C}$ has only two-game teams, then we constructed our variable assignment such that $g^z_{x,C}$ gives its last point to $t^3_{x,C}$, if x was not used to satisfy C. If x was used to satisfy C, then 1 point goes to its participating team which can still get 3 points (by the orientation for the ring gadget we picked, this team cannot have won $g_{x,C}$). Finally, by construction there is no team that scores more points than we had specified, i.e., there is no team that scores more points than t_1. ◀

▶ **Lemma 6.** *If there is an outcome of the games in the defined instance of ThreeTeamDebating such that no team gets more points than t_1 then the formula φ is satisfiable.*

Proof. Suppose there is an outcome of the games such that no team gets more points than t_1. We construct an assignment to the variables in φ that satisfies the formula. Let x be a variable. Consider the ring gadget for x. Due to Proposition 3 for the scores of the teams in G_x there are two possibilities. We set x to be true if its ring gadget (as presented in Figure 1) is oriented counter-clockwise in the sense of Proposition 3, otherwise, we set x to false.

We prove that this variable assignment satisfies φ. Consider a clause C with three variables x, y, z. Assume w.l.o.g. that $t^4_{x,C}$ scores 2 points in game g_C. We claim that then x satisfies C. Proposition 4 implies that since $t^4_{x,C}$ scores 2 points in game g_C, team $t^3_{x,C}$ cannot get any points in game $g^3_{x,C}$. Hence, the other game g of $t^3_{x,C}$ must be won by a team which can get 2 points and have a second placed team which can achieve 3 points.

If x appears positively in C, then by construction we have $g = g^2_{x,C}$. But then with the previous observation and Proposition 3, the ring gadget is oriented counter-clockwise and thus we have set x to true. Thus, x must satisfy C. On the other hand, if x appears negatively in C, then we have $g = g^1_{x,C}$. This implies that the ring gadget is oriented clockwise and thus we have set x to false and hence x satisfies C. ◀

Finally, we observe that in the above construction each team has at most two remaining matches. This completes the proof of Theorem 2. Now we can show that DebatingLeague is NP-hard.

▶ **Theorem 7.** *The DebatingLeague problem is NP-hard even if each team has at most two remaining matches to play.*

Proof. Let (T', M', s') be an instance of ThreeTeamDebating. We modify it to an instance of DebatingLeague. We begin by letting team t_1 win all of its remaining matches in the ThreeTeamDebating instance and updating the score vector accordingly for all teams. In each of the games won by team t_1, we replace t_1 by a dummy team that plays exactly one match and can still score two points. Now we update the instance to have four teams per match: For each game g, we add a dummy team that plays only in g and that can still score three points. Let (T, M, s) denote the resulting instance of DebatingLeague. Observe that in this instance t_1 is not participating in any game.

If $(T', M', s') \in$ ThreeTeamDebating, then we can copy the outcomes of all games to (T, M, s) and then assign 3 points to each dummy team. This gives a solution for Debating-League.

On the other hand, consider an outcome of (T, M, s) where t_1 wins the championship. Then the newly added dummy teams do not necessarily have to win their respective games. However, we can resolve this in the following way: For each game won by a non-dummy team, we change the outcome of the match such that the newly added dummy team and the winning non-dummy team change positions. Hence, a newly added dummy team obtains 3 points and the other teams just get fewer points than before. Thus, all newly added dummy teams win their respective games. We do a similar manipulation to make sure that the dummy teams that replaced t_1 score exactly 2 points and we replace them by t_1. This implies that the outcomes of the matches disregarding the dummy teams give a solution for (T', M', s'). ◀

We would like to point out that the above construction can easily be adapted to show that it is also NP-hard to decide whether t_1 can finish among the b best teams for any constant b (and thus in particular if b is part of the input). This can be achieved by simply adding $b - 1$ dummy teams that do not participate in any game and initially have more points than t_1.

3 Debating Tournaments

In this section we will consider the DebatingTournament problem: We are given a set of teams $T = \{t_1, \ldots, t_n\}$, where n is a multiple of 4, and a vector $s \in \mathbb{R}^n$, where entry s_i specifies how many points team t_i has scored so far. We further get a parameter k which indicates

how many rounds (i.e., match days) are left to play. Contrary to the league setting from the previous section, the fixtures are not determined beforehand. At each match day the teams with ranks $4r+1, 4r+2, 4r+3$, and $4r+4$ for each $r \in \mathbb{N}_0$ play a game. The points for winning the games are distributed as in the DebatingLeague setting. Additionally, we are given a parameter b. We want to decide whether there are outcomes for all remaining matches such that at the end there are at most $b-1$ teams with more points than t_1. Since we assume that in case of ties t_1 is always preferred, this means that t_1 finishes among the b best teams. This is an interesting question since in debating tournaments it is common to have several rounds in the above format, after which only the best b teams are promoted to the playoffs in which a knock-out elimination mode is played. Teams who manage to finish among the b best teams are said to *break*. Note that for $b=1$ this problem is identical to the question whether team t_1 can still place first. We prove that the problem is fixed parameter tractable (FPT) if both k and b are taken as parameters by giving an algorithm with a running time of $O(f(k+b) \cdot n)$.

Recall the assumption that in tie-breaking t_1 is always preferred. For the other teams, we assume w.l.o.g. that we have a fixed total order for the teams that specifies how to break ties if two teams have exactly the same number of points. The next lemma states a necessary condition for when t_1 can still break: t_1 has to be among the best $4^k b$ teams in the initial ranking s. For our algorithm, we use this lemma to output "no" if t_1 is not among the first $4^k b$ teams in s.

▶ **Lemma 8.** *Let t be a team that is among the best $4^\ell b$ teams when there are $\ell \in \{0, ..., k\}$ rounds left to be played. Then it has to be among the best $4^{\ell+1} b$ teams when there are $\ell+1$ rounds left to be played. If a team is among the best b teams at the end of the tournament then it must be among the best $4^k b$ teams when there are k rounds left to be played.*

Proof. We start with the first claim. Assume for contradiction that team t is at a position larger than $4^{\ell+1} b$ when there are $\ell+1$ rounds left to be played and it is among the best $4^\ell b$ teams when there are ℓ rounds left to be played. Observe that in the round when $\ell+1$ games are left, the $4^{\ell+1} b$ best placed teams will play $4^\ell b$ matches. Each of these games must have a winner and among the participating teams $4^\ell b$ teams must win their respective match (i.e., score 3 points) and thus will have more points than t when ℓ rounds are left, even if t wins its match. Hence, with ℓ rounds left to play, team t must have a position worse than $4^\ell b$.

The second claim can be shown by induction using the first claim as the inductive step: If a team t is among the best b teams when $\ell = 0$ rounds are left to be played then it must be among the best $4b$ teams before the last round, among the best $4^2 b$ teams before the last two rounds, ..., and among the best $4^k b$ teams when there are k rounds left. ◀

Now we describe a recursive FPT algorithm with parameters b and k, that solves a given instance of DebatingTournament. We define two sets $S_{>t_1} := \{t_i | s_i > s_1\}$ and $S_{\leq t_1} := \{t_i | s_i \leq s_1\}$. Both sets can be constructed in time $O(n)$. If $|S_{>t_1}| > 4^k b$, then the algorithm stops as team t_1 cannot break anymore by Lemma 8. Otherwise, the algorithm finds the best $4^k b$ teams by taking team t_1, all teams from $S_{>t_1}$ and filling the remaining $4^k b - |S_{>t_1}| - 1$ slots with teams from $S_{\leq t_1}$ in descending order of points. This step can be implemented in time $O(4^k b \cdot n)$: we iterate over all elements of $S_{\leq t_1}$ and keep track of the best team that was not yet added. When the iteration finished, we add the best team we found and mark it as added. We have one iteration over $O(n)$ elements for each free slot of of the $O(4^k b)$ teams, and thus we need a running time of $O(4^k b \cdot n)$. Denote by $T^{(k)}$ the obtained set of teams.

The teams in $T^{(k)}$ play $4^{k-1}b$ matches. We guess the outcomes of all these matches that still allow t_1 to be among the best b teams at the end. For each match there are 4! possible outcomes and thus there are $(4!)^{4^{k-1}b}$ possible game outcomes to enumerate. We update the scores of the teams accordingly. Denote by $T^{(k-1)}$ the first $4^{k-1}b$ teams in the resulting ranking. Lemma 8 implies that in any outcome of all matches of the n given teams all teams in $T^{(k-1)}$ must also be in $T^{(k)}$. This justifies that we enumerate only the matches for the teams in $T^{(k)}$, rather than the matches for all n given teams. Then we guess the outcome of the $4^{k-2}b$ matches for the teams in $T^{(k-1)}$ that allows t_1 to break eventually. We continue recursively for all remaining rounds. For each guess of the outcomes of a round, e.g., when there are only ℓ rounds remaining and we have $4^{\ell}b$ teams left to consider, we make one recursive call to our routine with $\ell - 1$ remaining rounds and $4^{\ell-1}b$ remaining teams.

To evaluate the complexity of the algorithm let us observe that for a single matchday there are at most $(4!)^{4^{k-1}b}$ possible outcomes, since each match has 4! possible outcomes and during a single round of the tournament there are at most $4^{k-1}b$ games to be played. The recursion depth is k which yields an overall running time of $\left((4!)^{4^{k-1}b}\right)^k = 2^{(2b)^{O(k)}}$ of our algorithm.

In total, we need time $O(4^k b \cdot n)$ for the first phase of the algorithm in which we determine the best $4^k b$ teams. For the simulation of all possible outcomes we need time $2^{(2b)^{O(k)}}$. Note that if we set $b = 1$, the algorithm decides in time $n \cdot 2^{2^{O(k)}}$ whether t_1 can place first in a tournament without playoffs. Thus, this problem is FPT for parameter k.

▶ **Theorem 9.** *If there are k remaining rounds to be played in a debating tournament, there is an algorithm that decides in time $n \cdot 2^{(2b)^{O(k)}}$ whether t_1 can place among the first b teams at the end of the tournament.*

3.1 Constant number of rounds

We present an algorithm that decides in time $n^{O(k^4)}$ whether a team can still break if there are k more rounds to play. In particular, this implies that for any constant k the problem is polynomial time solvable, in contrast to DebatingLeague.

As before, suppose we are given a ranking with n teams where for each team t_i we are given a value s_i that denotes how many points team i has scored so far. Again, assume that after the last round the first b teams in the ranking break (and thus participate in the play-offs). Also, we are given a value k that denotes the number of remaining rounds and we want to decide whether t_1 can still break. Consider a round such that including this round there are only $\ell \leq k$ more rounds to play. For each team t_i let s_i^{ℓ} denote its score at the beginning of the round. We distinguish three types of teams: teams t_i with $s_i^{\ell} > s_1^{\ell} + 3\ell$, teams t_i with $s_1^{\ell} - 3\ell \leq s_i^{\ell} \leq s_1^{\ell} + 3\ell$, and teams t_i with $s_i^{\ell} < s_1^{\ell} - 3\ell$. Denote those teams by T_T^{ℓ}, T_M^{ℓ}, and T_B^{ℓ}, respectively (for top, middle, and bottom). At the end of the tournament, the final score for each team t_i will be in $\{s_i^{\ell}, ..., s_i^{\ell} + 3\ell\}$. Thus, during the last k rounds team t_1 cannot overtake any of the teams in T_T^k and none of the teams in T_B^k can overtake t_1. Thus, intuitively, only the exact scores teams in T_M^k are relevant when deciding whether t_1 can still break. Our algorithm enumerates all possible remaining outcomes of the remaining matches but in doing so, it does not keep track of the scores of the teams in $T_T^k \cup T_B^k$. For the initial scores of the teams in T_M^k there are only $O(k)$ possibilities and during k rounds a team can score at most $O(k)$ points. Thus there are also only $O(k)$ possibilities for the scores of teams in T_M^k during the last k rounds. In order to describe the ranking for those teams, up to permutations it sufficies to keep track of the total number of teams with each

of the $O(k)$ possible scores. This yields $n^{O(k)}$ many possibilities in total which allows us to solve the problem via a dynamic program.

Formally, we will pretend that all teams in T_T^k have exactly the same number of points initially and that the same is true for all teams in T_T^k. This is justified by the following lemma.

▶ **Lemma 10.** *Assume that there are only ℓ rounds left to play. Consider an initial ranking given by a number of points s_i^ℓ for each team t_i. Then t_1 can still break if and only if it can still break in any initial ranking given by a number of points $\bar{s}_i^\ell$ for each team t_i such that*

- $s_1^\ell = \bar{s}_1^\ell$,
- *there is a bijection $f : T_M^\ell \to \bar{T}_M^\ell := \{t_i | \bar{s}_i^\ell - 3\ell \leq \bar{s}_1^\ell \leq \bar{s}_i^\ell + 3\ell\}$ such that for each $t_i \in T_M^\ell$ we have that in s^ℓ and $\bar{s}^\ell$ the teams t_i and $f(t_i)$ have the same rank and the same scores and $f(t_1) = t_1$,*
- $|T_T^\ell| = |\bar{T}_T^\ell| := |\{t_i | \bar{s}_i^\ell > \bar{s}_1^\ell + 3\ell\}|$ *and* $|T_B^\ell| = |\bar{T}_B^\ell| := |\{t_i | \bar{s}_i^\ell < \bar{s}_1^\ell - 3\ell\}|$.

Proof. We prove the claim by induction. For $\ell = 0$ it is immediate since t_1 can still break if and only if $|T_T^0| < b$. Suppose now the claim is true for some value ℓ and we want to prove it for $\ell + 1$. It is immediate that t_1 can break in the initial ranking $s^{\ell+1}$ if it can break in any initial ranking $\bar{s}^{\ell+1}$ with the above properties since $s^{\ell+1}$ satisfies these properties.

Now suppose that t_1 can break in the initial ranking $s^{\ell+1}$ and consider an initial ranking $\bar{s}^{\ell+1}$ with the above properties. Consider the outcome of the games in the current round for the ranking $s^{\ell+1}$ such that t_1 breaks after the last round. We construct an outcome of the games of the current round for the initial ranking $\bar{s}^{\ell+1}$. Consider a game $\bar{g}$ in which the teams $\{\bar{t}^{(1)}, \bar{t}^{(2)}, \bar{t}^{(3)}, \bar{t}^{(4)}\}$ participate. There is a corresponding game g, played by team $\{t^{(1)}, t^{(2)}, t^{(3)}, t^{(4)}\}$ according to the initial ranking $s^{\ell+1}$ such that for each $j \in \{1, 2, 3, 4\}$ we have that

- if $\bar{t}^{(j)} \in \bar{T}_M^\ell$ then $t^{(j)} = f^{-1}(\bar{t}^{(j)}) \in T_M^\ell$ and thus in $s^{\ell+1}$ and $\bar{s}^{\ell+1}$ the teams $\bar{t}^{(j)}$ and $t^{(j)}$ have exactly the same rank and the same score,
- if $\bar{t}^{(j)} \in \bar{T}_T^\ell$ then $t^{(j)} \in T_T^\ell$, and
- if $\bar{t}^{(j)} \in \bar{T}_B^\ell$ then $t^{(j)} \in T_B^\ell$.

Note that the first property implies that if $\bar{t}^{(j)} = t_1$ then $t^{(j)} = t_1$. For defining the outcome of $\bar{g}$ we simply the take of the outcome of game g from the known outcomes for all remaining matches that let t_1 break eventually. For each $j \in \{1, 2, 3, 4\}$ we assign the team $\bar{t}^{(j)}$ exactly the same score as team $t^{(j)}$ in those outcomes. We do this operation with all games $\bar{g}$. Denote by $\bar{s}^\ell$ the resulting ranking and by s^ℓ the ranking resulting if we apply those outcomes to $s^{\ell+1}$. Based on the induction hypothesis, we claim that t_1 can still break in $\bar{s}^\ell$. First, it is clear that $s_1^\ell = \bar{s}_1^\ell$ since $f(t_1) = t_1$. Consider a team t_i. If $t_i \in \bar{T}_T^{\ell-1}$ then $t_i \in \bar{T}_T^\ell$ and also if $t_i \in T_T^{\ell-1}$ then $t_i \in T_T^\ell$. Similarly, if $t_i \in \bar{T}_B^{\ell-1}$ then $t_i \in \bar{T}_B^\ell$ and also if $t_i \in T_B^{\ell-1}$ then $t_i \in T_B^\ell$. Finally, if $t_i \in \bar{T}_M^{\ell-1}$ then

- $t_i \in \bar{T}_M^\ell$ if and only if $f^{-1}(t_i) \in T_M^\ell$ and then t_i and $f^{-1}(t_i)$ have the same score in s^ℓ and $\bar{s}^\ell$
- $t_i \in \bar{T}_T^\ell$ if and only if $f^{-1}(t_i) \in T_T^\ell$, and
- $t_i \in \bar{T}_B^\ell$ if and only if $f^{-1}(t_i) \in T_B^\ell$.

Therefore, $|T_T^\ell| = |\bar{T}_T^\ell|$ and $|T_B^\ell| = |\bar{T}_B^\ell|$ and also there is a bijection $f : T_M^\ell \to \bar{T}_M^\ell$ with the properties required by the induction hypothesis. Thus, the induction hypothesis implies that t_1 can still break when starting with the initial ranking $\bar{s}^\ell$. ◀

We use Lemma 10 to justify that we can work with a new initial ranking s' instead of s. Note that the sets $T_B^k \dot\cup T_M^k \dot\cup T_T^k$ form a partition of the participating teams. For each team

$t_i \in T_B^k$ we define $s_i' := 0$. For each team $t_i \in T_M^k$ we define $s_i' := s_i$. For each team $t_i \in T_T^k$ we define $s_i' := s_1 + 3k + 1$. The next proposition follows immediately from Lemma 10.

▶ **Proposition 11.** *The team t_1 can break with the initial ranking s' if and only if it can break with the initial ranking s.*

In our algorithm, we use a dynamic program in order to enumerate all possible outcomes of the remaining k rounds when starting with the initial ranking s'. Key is that there are only $O(k)$ different scores that a team can have during these k rounds since there are only $O(k)$ different initial scores and each team can score at most $3k$ many points. We call two score vectors $\tilde{s}, \tilde{s}'$ *equivalent* if $\tilde{s}_1 = \tilde{s}_1'$ and if for each value x the number of teams with exactly x points is the same in $\tilde{s}$ and $\tilde{s}'$. The team t_1 can clearly break for an initial score vector $\tilde{s}$ if and only if it can still break in any equivalent initial score vector $\tilde{s}'$.

▶ **Lemma 12.** *When starting with the score vector s', there are only $n^{O(k)}$ equivalence classes for the score vectors arising during the last k rounds.*

Proof. For the number of points of t_1 there are only $O(k)$ possibilities. The other teams there can have at most $O(k)$ different scores. Thus, in order to describe an equivalence class it suffices to specify the points of t_1 and how many teams there are with each of the $O(k)$ possible different scores. This gives only $n^{O(k)}$ different possibilities in total. ◀

Our dynamic program works as follows: we have a DP-table entry (ℓ, C) for each $\ell \in \{0, ..., k\}$ and each equivalence class C of the possibly arising score vectors. We store either "yes" or "no" in this cell, corresponding to whether or not t_1 can still break if there are ℓ more rounds to play and we start with a score vector that is equivalent to C.

▶ **Lemma 13.** *Let $\ell \in \{0, ..., k\}$. Suppose we have computed the entry of the cell (ℓ, C') for each equivalence class C'. Then in time $n^{O(k^4)}$ we can compute the entry for a cell $(\ell + 1, C)$.*

Proof. Consider a score vector corresponding to C. We distinguish the different types of the games arising in the current round. We say that two games with teams $\{t^{(1)}, t^{(2)}, t^{(3)}, t^{(4)}\}$ and $\{\bar{t}^{(1)}, \bar{t}^{(2)}, \bar{t}^{(3)}, \bar{t}^{(4)}\}$, respectively, are of the same *type* if there exists a bijection $g : \{t^{(1)}, t^{(2)}, t^{(3)}, t^{(4)}\} \to \{\bar{t}^{(1)}, \bar{t}^{(2)}, \bar{t}^{(3)}, \bar{t}^{(4)}\}$ such that $t^{(j)}$ and $g(t^{(j)})$ have exactly the same score for each $j \in \{1, 2, 3, 4\}$. There are only $O(k^4)$ types of games at only 4! different outcomes for each game. Thus, in order to enumerate all possible outcomes of all games it suffice to guess how many games of each type have which of the 4! possible outcomes. Finally, there are 4! possible outcomes for the game that t_1 participates in. This gives $n^{O(k^4)}$ possibilities in total and for each possibility we obtain a cell (ℓ, C') for some equivalence class C'. ◀

Thus, in time $n^{O(k^4)}$ we can fill the entries of all DP-cells. There is one cell (k, C) such that C corresponds to the equivalence class that contains s'. The entry of this cell is "yes" if and only if t_1 can still break.

▶ **Theorem 14.** *There is an algorithm with running time $n^{O(k^4)}$ that decides whether a given team t_1 can still break if there are at most k remaining rounds to play in a tournament, for an arbitrary breaking threshold b that is part of the input.*

Acknowledgments. The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement no. 340506. Additionally, this house would like to thank Nicolas McLardy for hosting Stefan in Berlin during the time when this paper was written.

References

1 Mario Basler. Das habe ich ihm dann auch verbal .... Fussballzitate.com. Accessed: 2016-04-22. URL: http://www.fussballzitate.com/zitate/das-habe-ich-ihm-dann-auch-verbal-gesagt.html.

2 Thorsten Bernholt, Alexander Gülich, Thomas Hofmeister, and Niels Schmitt. Football elimination is hard to decide under the 3-point-rule. In *Proceedings of the 24th International Symposium on Mathematical Foundations of Computer Science*, MFCS'99, pages 410–418, London, UK, UK, 1999. Springer-Verlag. doi:10.1007/3-540-48340-3_37.

3 Celebrating 200 years of free speech and the art of debating. The Cambridge Union Society. Accessed: 2015-12-18. URL: https://cus.org/.

4 Mark Cartwright. Athenian democracy. Ancient History Encyclopedia. Accessed: 2015-12-18. URL: http://www.ancient.eu/Athenian_Democracy.

5 Katarína Cechlárová, Eva Potpinková, and Ildikó Schlotter. Refining the complexity of the sports elimination problem. *Discrete Applied Mathematics*, 2015.

6 Nick Cholst. Why 'three points for a win' is a loss for football – a closer look into one of the most important rule changes in football history. Cafè Futebol. Accessed: 2015-12-18. URL: http://www.cafefutebol.net/2013/09/11/why-three-points-for-a-win-is-a-loss-for-football-a-closer-look-into-one-of-the-most-important-rules-in-football-history/.

7 Kelly Phillips Erb. Numerous FIFA Officials Arrested in Massive Corruption Scheme Tied to World Cup, Other Tournaments. Forbes. Accessed: 2015-12-18. URL: http://www.forbes.com/sites/kellyphillipserb/2015/05/27/numerous-fifa-officials-arrested-in-massive-corruption-scheme-tied-world-cup-other-tournaments/.

8 Handbook. FIDE. Accessed: 2015-12-18. URL: https://www.fide.com/fide/handbook.html?id=18&view=category.

9 2014 FIFA World Cup reached 3.2 billion viewers, one billion watched final. Fédération Internationale de Football Association. Accessed: 2015-12-18. URL: http://www.fifa.com/worldcup/news/y=2015/m=12/news=2014-fifa-world-cuptm-reached-3-2-billion-viewers-one-billion-watched--2745519.html.

10 Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

11 Gusfield and Martel. The structure and complexity of sports elimination numbers. *Algorithmica*, 32(1):73–86, 2002. doi:10.1007/s00453-001-0074-y.

12 Marting Luther King Jr. Martin Luther King I Have a Dream Speech. American Rhetoric. Accessed: 2015-12-18. URL: http://www.americanrhetoric.com/speeches/mlkihaveadream.htm.

13 Walter Kern and Daniël Paulusma. The new FIFA rules are hard: complexity aspects of sports competitions. *Discrete Applied Mathematics*, 108(3):317–323, 2001. doi:10.1016/S0166-218X(00)00241-9.

14 Walter Kern and Daniël Paulusma. The computational complexity of the elimination problem in generalized sports competitions. *Discrete Optimization*, 1(2):205–214, 2004. doi:10.1016/j.disopt.2003.12.003.

15 S. Thomas McCormick. Fast algorithms for parametric scheduling come from extensions to parametric maximum flow. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC'96, pages 319–328, New York, NY, USA, 1996. ACM. doi:10.1145/237814.237978.

16 Andreas Möller. "Hauptsache Italien!" Legendäre Fußball-Sprüche. tz München. Accessed: 2016-04-22. URL: http://www.tz.de/sport/fussball/hauptsache-italien-legendaere-fussball-sprueche-fotostrecke-zr-3286394.html.

17 Philip Oltermann. Angela Merkel the mascot: German chancellor's relationship with the team. The Guardian. Accessed: 2015-12-18. URL: http://www.theguardian.com/world/2014/jul/14/angela-merkel-chancellor-german-team-relationship.

18 Dömötör Pálvölgyi. Deciding soccer scores and partial orientations of graphs. *Acta Univ. Sapientiae, Math*, 1(1):35–42, 2009.

19 Stuart Pearce. Stuart Pearce quotes. Think Exist. Accessed: 2016-04-22. URL: http://thinkexist.com/quotation/i-can-see-the-carrot-at-the-end-of-the-tunnel/539152.html.

20 Ronaldo. Ronaldo quotes. Think Exist. Accessed: 2016-04-22. URL: http://thinkexist.com/quotation/we-lost-because-we-didn-t-win/539102.html.

21 B. L. Schwartz. Possible winners in partially completed tournaments. *SIAM Review*, 8(3):302–308, 1966. doi:10.1137/1008062.

22 Jürgen Voges. Bundeskanzler: Als Gerhard Schröder noch den Rasen pflügte. Stern. Accessed: 2015-12-18. URL: http://www.stern.de/politik/deutschland/bundeskanzler-als-gerhard-schroeder-noch-den-rasen-pfluegte-3067112.html.

23 Kevin D. Wayne. A new property and a faster algorithm for baseball elimination. *SIAM Journal on Discrete Mathematics*, 14(2):223–229, 2001. doi:10.1137/S0895480198348847.

24 Rules. World Debating News. Accessed: 2015-12-18. URL: http://worlddebating.blogspot.ie/p/rules.html.